Model-Based Exploration of Parallelism in Context of Automotive Multi-Processor Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

der Universität Osnabrück

an der Fakultät für Mathematik und Informatik

von

Robert Martin Höttger

Bochum

November 2020

Verteidigung:	22.04.2021
Dekan / Dekanin:	Prof. Dr. Tim Römer
Gutachter / Gutachterinnen:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr. Burkhard Igel
	Prof. Dr.-Ing. Peter Ulbrich

Ut sementem feceris, ita metes.
(Marcus Tullius Cicero)

Acknowledgement

I would like to thank all people that supported my research and helped me during my time as a PhD student. First and foremost, I would like to thank my wife Juliane and my two sons Janes and Leard. You gave me your unconditional support and love through all this long process. In challenging times and straining days, your love is all it takes to keep me on track.

I also would like to express my appreciation to my advisory committee: Prof. Dr.-Ing. Olaf Spinczyk and Prof. Dr. Burkhard Igel. Thank you for giving me the opportunity to be part of various research projects, for your time, patience, understanding, advises, and lead throughout my studies. I highly value your and Prof. Dr.-Ing. Peter Ulbrich's time and thoughts on my research.

My gratitude further goes to various people of the IDiAL research institute. Many thanks to Uwe Lauschner, who I spent 13 years of studies with, and I am very grateful for having such a close friend all along. Special thanks to Philipp Heisig, who supported me in proofreading this document, writing project outlines, full project proposals, deliverables, reports, and in many other matters, which has significantly biased the time I could spend on my research positively. Thank you, Andreas Grosche, for our pleasant discussions and for helping me improving this thesis in many ways. Furthermore, I am grateful for fruitful discussions of any matters with Rene Ukrig, Philipp Tendyra, Daniel Fruhner, and Lukas Krawczyk. Another person I want to thank is Prof. Dr. Carsten Wolff. His widespread efforts and diligence continuously takes the research institute to the next level. I highly value Carsten's concise statements, his humorous bias, and all his efforts towards a promising future of IDiAL but also for students, lecturers, and employees.

Luckily, various students supported my research by writing their research projects or theses under my supervision. To keep this list short, I only mention Mustafa Özcelikörs, Pedro Cuadra, The Bao Bui, and Junhyung Ki, who I am very grateful for having published papers with. Due to my research being involved in some international research projects, I also want to thank the people behind Amalthea4public and Appstacle, who not only spent many efforts on various technologies I learned so much about, but also let me enjoy exceptionally great international cooperation.

Finally, I would like to thank my parents and siblings for inspiration, advice, and financial support. I will always keep enjoying the atmosphere with all of you.

Abstract

This dissertation entitled 'Model-Based Exploration of Parallelism in the Context of **Automotive** Multi-Core Systems' deals with the analytical investigation of different temporal relationships for automotive multi-processor systems subject to critical, embedded, real-time, distributed, and heterogeneous domain requirements. Vehicle innovation increasingly demands high-performance platforms in terms of, e.g., highly assisted or autonomous driving such that established software development processes must be examined, revised, and advanced. The goal is not to develop application software itself, but instead to improve the **model-based** development process, subject to numerous constraints and requirements. Model-based software development is, for example, an established process that allows systems to be analyzed and simulated in an abstracted, standardized, modular, isolated, or integrated manner. The **verification of real-time behavior** taking into account various **constraints and modern architectures**, which include graphics and heterogeneous processors as well as dedicated hardware accelerators, is one of many challenges in the real-time and automotive community. The **software distribution** across hardware entities and the identification of software that can be executed in parallel are crucial in the development process. Since these processes usually optimize one or more properties, they belong to the category of problems that can only be solved in polynomial time using non-deterministic methods and thus make use of (meta) heuristics for being solved. Such (meta) heuristics require sophisticated implementation and configuration, due to the properties to be optimized are usually subject to many different analyses.

With the results of this dissertation, various development processes can be adjusted to modern architectures by using new and extended processes that enable future and computationally intensive vehicle applications on the one hand and improve existing processes in terms of efficiency and effectiveness on the other hand. These processes include runnable partitioning, task mapping, data allocation, and timing verification, which are addressed with the help of constraint programming, genetic algorithms, and heuristics.

Zusammenfassung

Diese Dissertation mit dem Thema 'Model-Based Exploration of Parallelism in Context of **Automotive** Multi-Core Systems' befasst sich mit der Erforschung verschiedener zeitlicher Zusammenhänge von kritischen und eingebetteten Echtzeitsystemen im automobilen Multicore-Prozessor-Kontext. Es werden etablierte Prozesse untersucht und Herausforderungen gelöst, die bei der Weiterentwicklung von Fahrzeugsoftware entstehen, so beispielsweise in Bezug auf autonomes Fahren. Dabei geht es nicht darum, Applikationssoftware selbst zu entwickeln, sondern stattdessen den Entwicklungsprozess, der an eine Vielzahl an Rahmenbedingungen geknüpft ist, zu verbessern. Die **modellbasierte Softwareentwicklung** ist zum Beispiel ein etablierter Prozess, mit dem sich übergreifend Systeme modular, isoliert oder integriert analysieren sowie simulieren lassen. Die **Verifikation von Echtzeitverhalten** unter Berücksichtigung verschiedener **Rahmenbedingungen und moderner Architekturen**, die beispielsweise Grafik- und heterogene Prozessoren sowie dedizierte Hardwarebeschleuniger beinhalten, ist eine von vielen Problemstellungen in der Echtzeit- und Automotive Community. Ebenso spielt die **Softwareverteilung** auf die Hardware und die Identifikation von Softwareteilen, die parallel ausgeführt werden können, eine maßgebliche Rolle im Entwicklungsprozess. Darüber hinaus müssen etwaige Parallelisierungsprozesse in den meisten Fällen eine oder mehrere Zielfunktionen optimieren. Da sich Allokationsprobleme in Kombination mit Optimierung meist nur mit nicht-deterministischen Verfahren in Polynomialzeit lösen lassen, müssen geeignete (Meta-) Heuristiken genutzt werden, um Lösungen effizient berechnen zu können. Zudem erfordern diese (Meta-) Heuristiken systematische Implementierung und Konfiguration, da die zu optimierenden Eigenschaften in der Regel eine Vielzahl an Analysen beinhalten.

Mit den Ergebnissen dieser Dissertation können verschiedene Entwicklungsprozesse an moderne Architekturen angepasst werden, um einerseits zukünftige rechenintensive Fahrzeugapplikationen zu ermöglichen und andererseits existierende Prozesse in Bezug auf Effizienz und Effektivität zu verbessern. Hierzu gehören neue und erweiterte Prozesse wie die Partitionierung von Runnables zu Tasks, das Verteilen von Tasks auf Prozessoren sowie die Verifikation von Echtzeit, die in Kombination und mit Hilfe von Constraint-Programmierung, genetischen Algorithmen und Heuristiken gelöst werden.

This dissertation was supported by the following projects and organizations:

- ITEA3 Amalthea, BMBF funded under 01IS11020A 2012-2014
- ITEA3 Amalthea4public, BMBF funded under 01|S14029K 2015-2017
- ITEA3 Appstacle, BMBF funded under 01IS14029A 2017-2019
- ITEA3 Panorama, BMBF funded under 01IS18047D 2019-2022
- Fachhochschule Dortmund - Institute for the Digital Transformation of Application and Living Domains (IDiAL)
- Fachhochschule Dortmund - Promotionskolleg

Index terms

Amalthea, Amalthea4public, App4mc, Partitioning, Mapping, Software Distribution, Timing Verification, Response Time Analysis, Autosar, Mixed-Critical Systems, Embedded Systems, Real-Time Systems, Automotive, Constraints

Publications

The following lists peer-reviewed publications the author of this thesis was involved with during his research studies ordered by year and month chronologically. Publications used for this dissertation are highlighted **bold**.

Publications until 2017

[1] Robert Höttger, Burkhard Igel, and Erik Kamsties. "A Novel Partitioning and Tracing Approach for Distributed Systems Based on Vector-Clocks." In: *7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems*. Vol. 02. IDAACS'13. IEEE, 2013, pp. 670–675. DOI: 10.1109/IDAACS.2013.6663010.

[2] Robert Höttger, Burkhard Igel, and Erik Kamsties. "Vector-Clock Tracing and Model-based Partitioning for Distributed Embedded Systems." In: *International Journal of Computing* 12.4 (2013). ISSN 1727-6209, Available online: https://bit.ly/3iJjMNl, pp. 324–332.

[3] Robert Höttger and Burkhard Igel. "A Concept of Vector Clock Utilization in an Iterative Tracing Approach for Distributed Embedded Systems." In: *18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES'15. ACM, June 2015. DOI: 10.1145/2764967.2764969.

[4] **Robert Höttger, Lukas Krawczyk, and Burkhard Igel. "Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems." In: *International Conference on Parallel, Distributed Systems and Software Engineering*. Vol. 2. ICPDSSE'15. eISSN: 1307-6892, Available online: https://bit.ly/3iOBHT6. World Academy of Science, Engineering and Technology, Jan. 2015, pp. 2643–2649.**

[5] Robert Höttger, Phil Närdemann, and Lukas Krawczyk. "Comprehensive Utilization of the AMALTHEA Tool Platform – A Use-Case along with the Parallax Activity Bot." In: *International Research Conference Dortmund*. June 2015.

[6] Lukas Krawczyk, Daniel Fruhner, Robert Höttger, Carsten Wolff, and Christopher Brink. "AMALTHEA - Eine durchgängige Entwicklungsplattform für die modellgetriebene Entwicklung automobiler eingebetteter Systeme." In: *10. Paderborner Workshop: Entwurf mechatronischer Systeme, Heinz Nixdorf MuseumsForum, Paderborn*. Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn, Apr. 2015.

[7] Carsten Wolff, Robert Höttger, Burkhard Igel, Erik Kamsties, Uwe Lauschner, and Christopher Brink. "Automotive Software Development with AMALTHEA." In: *4th International Scientific Conference on Project Management in the Baltic Countries.* Apr. 2015.

[8] Robert Höttger, Uwe Lauschner, Phil Närdemann, Philipp Heisig, Carsten Wolff, Erik Kamsties, and Burkhard Igel. "Teaching Distributed and Parallel Systems with APP4MC." In: *International Symposium on Embedded Systems and Trends in Teaching Engineering.* DesIRE '16. Available online: https://bit.ly/3c8jD3J. Sept. 2016, pp. 126–134. ISBN: 978-80- 558-1041-6.

Publications 2017

[9] **Robert Höttger, Burkhard Igel, and Olaf Spinczyk. "On Reducing Busy Waiting in AUTOSAR via Task-Release-Delta-based Runnable Reordering." In: *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition.* DATE '17. IEEE, Mar. 2017, pp. 1510–1515. DOI: 10.23919/DATE.2017.7927230.**

[10] Robert Höttger, Mustafa Özcelikörs, Philipp Heisig, Lukas Krawczyk, Carsten Wolff, and Burkhard Igel. "Constrained Mixed-Critical Parallelization for Distributed Heterogeneous Systems." In: *Proceedings of the 2017 Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.* IDAACS '17. IEEE, Sept. 2017. DOI: 10.1109/IDAACS.2017.8095077.

[11] Pedro Cuadra, Lukas Krawczyk, Robert Höttger, Philipp Heisig, and Carsten Wolff. "Automated Scheduling for Tightly-Coupled Embedded Multi-Core Systems using Hybrid Genetic Algorithms." In: *Proceedings of the 2017 International Conference on Information and Software Technologies.* ICIST '17. Springer International Publishing, Oct. 2017, pp. 362–373. DOI: 10.1007/978-3-319-67642-5_30.

[12] Robert Höttger, Harald Mackamul, Andreas Sailer, Jan-Philipp Steghöfer, and Jörg Tessmer. "APP4MC: Application Platform Project for Multi- and Many-core Systems." In: *IT - Information Technology. Methods and Applications of Informatics and Information Technology.* Vol. 59, Issue 5. De Gruyter Oldenbourg, Nov. 2017. DOI: 10.1515/itit-2017-0019.

Publications 2018

[13] Johannes Geismann, Robert Höttger, Lukas Krawczyk, Uwe Pohlmann, and David Schmelter. "Automated Synthesis of a Real-time Scheduling for Cyber-physical Multi-core Systems." In: *Communications in Computer and Information Science: Model-Driven Engineering and Software Development (MODELSWARD)* (2018). DOI: 10.1007/978-3-319-94764-8_4.

[14] Robert Höttger, Mustafa Özcelikörs, Philipp Heisig, Lukas Krawczyk, Pedro Cuadra, and Carsten Wolff. "Combining Eclipse IoT Technologies for a RPI3-Rover along with Eclipse Kuksa." In: Available at: http://ceur-ws.org/Vol-2066/. CEUR Workshop Proceedings, 2018, pp. 101–106.

[15] Robert Höttger and Jörg Tessmer. "1st Workshop on Software Engineering for Applied Embedded Real-Time Systems (SEERTS)." In: *Software Engineering und Software Management 2018.* Vol. P-279. Available online: https://dl.gi.de/

handle/20.500.12116/21179. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik e.V. (GI), Mar. 2018, pp. 43–45.

[16] Robert Höttger. *Why Open Source is Driving the Future Connected Vehicle.* Mobility in a Globalised World Conference 2018. Sept. 2018. DOI: 10.20378/irbo-54827.

Publications 2019

[17] **Robert Höttger, Lukas Krawczyk, Burkhard Igel, and Olaf Spinczyk. "Memory Mapping Analysis for Automotive Systems." In: *Work in Progress Paper, 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2019), Montreal, Canada* (Apr. 2019). Available at https://bit.ly/2QFWtXt.**

[18] Philipp Heisig, Sven Erik Jeroschewski, Johannes Kristan, Robert Höttger, and Ahmad Banijamali. "Bridging the Gap between SUMO & Kuksa: Using A Traffic Simulator for Testing Cloud-based Connected Vehicle Services." In: *SUMO Conference 2019* (May 2019). DOI: 10.29007/9kkv.

[19] **Robert Höttger, Junhyung Ki, The Bao Bui, Burkhard Igel, and Olaf Spinczyk. "CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems." In: *10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS '19), co-located with the 31st Euromicro Conference on Real-Time Systems (ECRTS '19)* (July 2019). Available online: https://bit.ly/33EnhhP.**

[20] **Robert Höttger, Burkhard Igel, and Olaf Spinczyk. "Constrained Software Distribution for Automotive Systems." In: *Proceedings of the 2019 International Conference on Information and Software Technologies*. 25th ICIST '19. Springer International Publishing, Oct. 2019. DOI: 10.1007/978-3-030-30275-7_44.**

Contents

1 Introduction

During recent decades, software has been increasingly developed with the intention of being executed on multiple Processing Units (PUs)[1]. The general purpose computing sector uses multi-core technologies since the 1990s to overcome high power and heat dissipation as well as computational limits of single-chip processors due to frequency scaling and physical chip manufacturing constraints. Experts predicted that five nano meter transistor size could be the end of Moore's law [21] due to quantum tunneling through the gate oxide layer [22]. Although research is still concerned with tackling the minimal transistor size constraint, multi-processor technologies have already pervaded the embedded, real-time, and automotive computing domains. The foundations of this transition were formed by Foster [23], who initially illustrated the Partitioning Communication Aggloremation Mapping (PCAM) approach shown in Figure 1.1 that (a) forms partitions from an initial problem (software), (b) derives dependencies between the partitions, (c) agglomerates the partitions into groups (tasks), and finally (d) maps and schedules those partitions to PUs. Since then, lots of research has been invested in each of the four areas for an immense variety

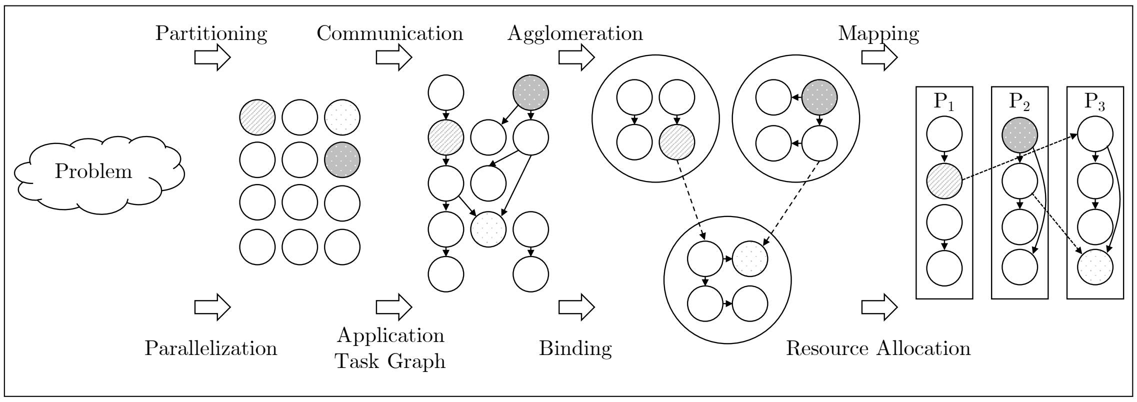


Figure 1.1: Foster's PCAM approach: Partitioning, Communication, Agglomeration, and Mapping [23], also known as parallelization, application task graph, binding, and resource allocation [24]

of applications and domains. The PCAM approach still applies to modern challenges as

[1]A PU is often referred to as *processor* or *core* in existing literature

stated in a recent survey [24] that investigates related research just under a slightly different terminology namely *parallelization, application task graph, binding*, and *resource allocation* for PCAM, respectively. Parallelization tools such as MAPS [25] or MNEMEE [26] use the PCAM approach, identify data dependencies as well as most atomic executable entities, and target similar optimization goals compared with partitioning and task to PU mapping presented in this thesis via reducing inter-PU communication. However, these tools are neither model-based nor cover Automotive Open System Architecture (AUTOSAR) and AMALTHEA[2] specific constraints crucial in the automotive industry.

With modern demands of highly assisted driving applications and the evolution towards autonomous driving, multi- and many-core technologies have not yet reached their full potentials due to the vast amount of various heterogeneous constraints in the automotive domain. High-end cars from 2010 were already constructed of around 100 Electronic Control Units (ECUs) [27], and sophisticated architectures are required to overcome the diverse application domains across various safety levels (cf. Automotive Safety Integrity Level (ASIL)) from very high, e.g., a brake actuator, to relatively low, e.g., an infotainment feature.

Applying Foster's approach during software development activities in the automotive industry is not straightforward, especially when facing different application domains, heterogeneous hardware and software, as well as Original Equipment Manufacturers (OEMs), tier suppliers, and tool vendors involved in the development chain. Consequently, this thesis makes use of the Model Based Software Development (MBSD) paradigm by using AMALTHEA [2], which is described in Section 3.1.

1.1 Thesis Statement and Scope

Research presented in this thesis solves challenges emerging from the progression to heterogeneous multi-core architectures in the automotive domain. Real-time aspects, constraints, requirements, specifications, and protocols of respective applications are outlined and incorporated along with the proposed technologies facilitating the PCAM process. The comprehensive Design Space Explorations (DSEs) for concurrently executing software across diverse development phases with varying limitations, the assessment of results, and the provision of new technologies make this thesis unique compared to related work. Typical scheduling approaches are outlined, and existing research for, e.g., Worst Case Response Time (WCRT)-Analysis, is used to support this thesis' technologies. Given task and PU sets are investigated to identify useful or optimal distinct mappings of the former to the latter, which consider various properties such as periods, execution times, priorities, deadlines, scheduling policies, frequencies, memory accesses, memory types, Central Processing Unit (CPU) and Graphics Processing Unit (GPU) performance, network interfaces, and more. The analyses use different assessment metrics such as schedulability, resource utilization, speedup, runtime, and parallelism (cf. Chapter 7).

[2] AMALTHEA meta-model information can be found at the Application Platform Project for Multi Core (APP4MC) online documentation `https://www.eclipse.org/app4mc/documentation/`, visited 11.2020

Thesis Statement

The holistic concern of constraints and requirements across the many-fold heterogeneous domains in the automotive industry in terms of parallelism exploitation requires new and appropriate design space exploration, optimization, and timing verification approaches. These approaches have to account for, e.g., hardware accelerators, GPUs, different function domains, broader network connectivity, task chain latency, or specific communication paradigms, which are necessary for vehicle evolution towards, e.g., highly assisted and autonomous driving.

The contributions outlined in the following Section 1.2 solve modern WCRT-Analysis, Partitioning, and Mapping challenges, provide advanced solutions as well as formally verified techniques to consider various constraints and optimize goals affected by the intertwined nature of six-fold domain-crossing automotive systems (i.e., embedded, real-time, mixed-critical, concurrent, heterogeneous, and distributed; cf. Chapter 2), and finally give insights into DSE in context of parallelism exploration for automotive multi-core systems.

1.2 Contributions

The following sections list the different research this thesis includes. Each topic is briefly described and put into context. Basic research components and relationships are outlined in Figure 1.2, whereas numbers in the lower right corners of components reflect the Section which addresses the corresponding component. The main component

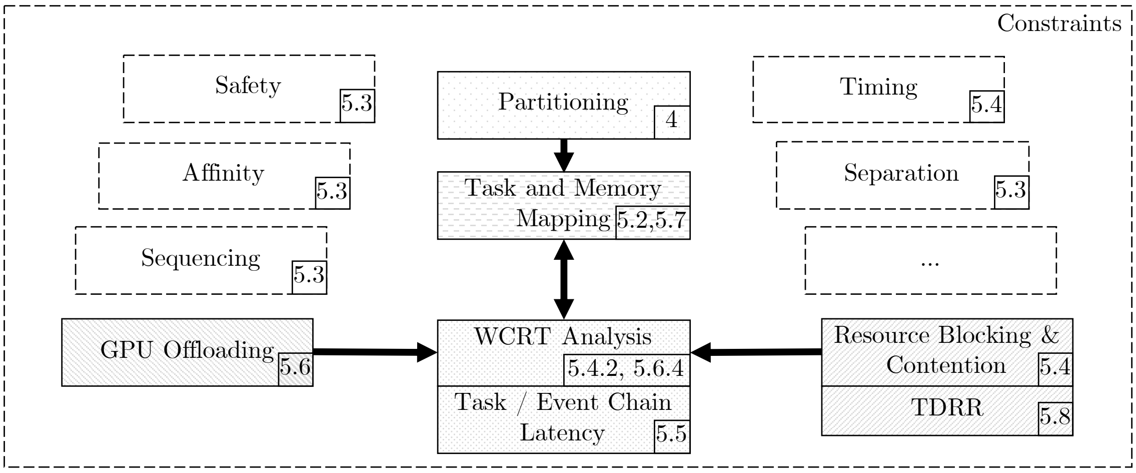

Constraints
Safety 5.3
Affinity 5.3
Sequencing 5.3
Partitioning 4
Task and Memory Mapping 5.2,5.7
Timing 5.4
Separation 5.3
...
GPU Offloading 5.6
WCRT Analysis 5.4.2, 5.6.4
Task / Event Chain Latency 5.5
Resource Blocking & Contention 5.4
TDRR 5.8

Figure 1.2: Thesis' research components and relationships. Task-release Delta-based Runnable Reordering (TDRR) is a novel approach to reduce busy waiting outlined in Section 5.8.1.

is WCRT-Analysis, which is influenced by all its surrounding boxes. In fact, response times are (i) the major optimization goal for the mapping process (cf. Figure 1.2 dashed background box) and (ii) influenced by not only various constraints but also GPU offloading situations (cf. Figure 1.2 diagonal down background box) as well as resource blocking and contention (cf. Figure 1.2 diagonal up background box). Constraints, indicated by boxes with dashed borders, pervade all essential boxes, and just a constraint subset is shown in Figure 1.2. They typically reflect requirements but also define properties as well as necessities systems need to fulfill and consider to ensure correctness, compliance, verification, and validation. Constraints can be of different types and describe, e.g., affinities (e.g., required peripheral access), separations (e.g., Freedom From Interference (FFI)), timing properties (e.g., deadlines tasks have to meet), sequences (e.g., ensuring causally correct execution orders), communication semantics (e.g., Logical Execution Time (LET), implicit or explicit communication, synchronization), safety levels (cf. ASIL), and more. This thesis considers such constraints for valid, effective, and efficient solutions of the partitioning, mapping, and WCRT-analysis challenges. Model-checking, simulation, model variability investigation, or tracing is out of scope.

1.2.1 Model-based Software Partitioning

The partitioning contribution of this thesis is the application of graph algorithms to a set of atomic execution units (runnables[3]) to form tasks that potentially can run concurrently on a target system. The partitioning process considers dependencies, activation rates, instruction costs, and constraints such as safety groups, separations, component tags, and communication overheads. The partitioning technologies neither cover specific scheduling algorithms nor hardware properties such as processing speed, memory architecture, accelerators, or similar. These properties are used during the subsequent mapping contribution (cf. next Section 1.2.2). Following Foster's PCAM approach [23], partitioning in this thesis includes communication and agglomeration. The initial partitioning research [4] has been further extended as outlined in Section 4.2.

1.2.2 Constrained Software to Processing Unit Mapping

Constrained software distribution in this thesis' context covers the component deployment problem for distributing tasks to multiple heterogeneous PUs on either a single system or a network of systems or ECUs. The mapping terminology is used in this thesis for defining static assignments, respectively allocations of tasks to PU, but also labels to memory, or runnables to tasks (cf. terminology in Section 2.1, notations in Section 3.2).

This thesis's mapping technologies consider an extensive amount of constraints typically employed in the automotive industry. Since the mapping problem is known to be NP-hard [28, 29][4], various DSEs meta-heuristics such as Genetic Algorithm (GA), Constraint Satisfaction Problem (CSP), Simmulated Annealing (SA), Integer Linear Programming (ILP), and greedy heuristics are investigated and compared according to efficiency, usability, and result quality. Garey and Johnson showed in [28] that problems faced in this thesis are a combination of NP-complete problems, i.e., multiprocessor scheduling, precedence constrained scheduling, resource-constrained scheduling, scheduling with individual deadlines, preemptive scheduling, and scheduling to minimize weighted completion time. The task mapping problem with optimization is, in general, perceived as being NP-hard in the strong sense [29]. Additionally, State-of-the-Art (SotA) challenges such as GPU-offloading (cf. Section 1.2.5) are incorporated into optimization goals of the DSEs to find not only feasible but also optimized solutions, which consider modern architectures, requirements, as well as performance and application demands. Various

[3]Terminology explained in Section 2.1

[4]No algorithm is known that solves NP-hard problems in deterministic polynomial time

constraints are formally analyzed and applied to industrial models. The mapping research was published at [20] and is further put into context and aligned with other research of this thesis.

1.2.3 Constrained Data to Memory Mapping

Based on data affinities, access rates, data and memory size, memory types, and communication channels, the data to memory mapping challenge is solved in this thesis, forming the third significant contribution. As modern automotive systems tend to centralize multiple ECUs into more powerful high-performance computers to cope with highly assisted or even autonomous driving challenges, sophisticated memory architectures have to be faced by developers and system designers. The effect of data to memory mapping usually manifests in memory stalls, contention- and blocking delays, or different protocol overheads, which need to be mitigated or even cleared in the best case. Such mitigation approaches are formally investigated and further incorporated into optimization goals of the mapping DSE implementation approaches. Corresponding ideas and initial approaches were published at [17] and finalized in this thesis along with Section 5.7.

1.2.4 Task-Release-Delta-based Runnable Reordering

The TDRR approach addresses the reduction of busy waiting periods on a multi-PU system based on task release times and scheduling tables calculated offline during the system design process. The mechanism is based on **resource blocking** analysis for spinlocks, which are used in AUTOSAR to protect globally shared resources. This research was published at [9] and forms a new approach to mitigate busy waiting and consequently reduce task execution and response times.

1.2.5 CPU-GPU Response Time Analysis

Considering the process of offloading computation from a CPU to a GPU for Response Time Analysis (RTA) approaches is another recent challenge that requires appropriate formal verification within the automotive industry [30]. GPUs are increasingly used in the automotive domain due to the high parallel computation capabilities, which are significantly useful for highly concurrent processes, such as image processing. The latter is crucial for highly assisted or even autonomous driving. Analytical solutions to the formal verification challenges of response times across CPUs and GPUs were published at [19] and form another contribution to this thesis. Due to the absence of solutions for such specific environments, this contribution forms a novel approach for the automotive and real-time community.

1.2.6 Task-Chain-Latency Analyses

In addition to CPU-GPU response time analysis, Task Chain Latency (TCL) and Event Chain Latency (ECL) investigation raised many questions in the automotive domain recently [31]. Especially the consideration of different communication paradigms, such as explicit, implicit, and LET communication, influence the latency of task chains and requires new analytical techniques. Various solutions have been proposed such as [32], [33], or [34], but none has yet combined TCL/ECL with CPU-GPU RTA. As a consequence, this thesis proposes novel TCL/ECL analyses in a heterogeneous CPU-GPU environment by combining SotA research with results of CPU-GPU RTA.

1.3 Prior Work

Contributions presented in this thesis are based on the AMALTHEA model and most technologies have been implemented along with the open-source Eclipse APP4MC platform[5] using Java. AMALTHEA is AUTOSAR[6] compliant and both models are introduced along with dedicated Sections 3.1 and 2.3.1, respectively. AMALTHEA is Eclipse Modling Framework (EMF)-based, has been developed since 2011, and still evolves in regular releases since 2014. AMALTHEA model entities are aligned to formal notations in Section 3.2. TDRR was initially developed using AMALTHEA *v0.8.3* and the mapping DSEs use AMALTHEA *v0.9.8*, which is the latest AMALTHEA version technologies of this thesis are migrated to. The model has its origins in the AMALTHEA ITEA research project that was conducted between 2011 and 2014. Its ITEA successor AMALTHEA4PUBLIC ran between 2014 and 2017. Official project documents are published at the ITEA online representation, i.e., ITEA AMALTHEA[7] and ITEA AMALTHEA4PUBLIC[8], respectively. AMALTHEA is also majorly used in industrial projects as well as recent research projects such as AramisII[9] or Panorama[10]. Apart from AramisII, the author of this thesis took part in all the research projects mentioned above. AMALTHEA terminology, semantic descriptions, and notations used throughout this thesis are outlined in Chapter 3.

Significant research technologies this thesis implements are listed below and again referenced to and put into context in the respective sections.

- Formal blocking time analysis of spinlocks in AUTOSAR from [35]
- Formal Weighted Round Robin (WRR) WCRT analysis from [36]
- Formal Fixed Priority Preemptive Scheduling (FPPS) / Rate Monotonic Scheduling (RMS) analysis based on [37] and [38]
- Formal offset-based RTA using transaction based on [39]
- Formal mixed-Preemptive RTA using preemption thresholds based on [40]
- Mapping technologies from [41], EPL2.0 licensed.
- The Jenetics library [42], Apache-2.0 licensed.
- The JGraphtT library [43], dual licensed under LGPL and EPL2.0
- The Choco library [44], BSD-licensed.
- The AMALTHEA model [45] and its APP4MC platform, EPL2.0 licensed.

[5]https://www.eclipse.org/app4mc, visited 11.2020
[6]https://www.autosar.org, visited 11.2020
[7]https://itea3.org/project/amalthea.html, visited 11.2020
[8]https://itea3.org/project/amalthea4public.html, visited 11.2020
[9]https://www.aramis2.org, visited 11.2020
[10]https://itea3.org/project/panorama.html, visited 11.2020

1.4 Organization

This thesis is organized as follows. Chapter 2 provides fundamental basics for each of this thesis' contribution and gives insights into its technological environments. The statement and scope of this thesis (cf. Section 1.1) is summarized and reflected in Section 2.9.

AMALTHEA terminology (the system model) and notations are presented in Chapter 3.

Subsequently, Chapter 4 outlines the partitioning process, i.e., forming tasks that potentially can run in parallel. Just as for other technological contributions, a dedicated section for related work can be found at the beginning of the chapter.

As a consecutive step to the partitioning, the mapping technologies and algorithms are described in Chapter 5. The mapping processes are extended by technologies to cover a broader software distribution across not only PUs, but also ECU networks on the one hand, and to address various constraints across timing, safety, criticality, and reliability on the other hand. These extensions also include timing verification along with RTA methods across CPUs and also GPUs as well as sophisticated task chain latency elaborations. Since AUTOSAR relies on the Priority Ceiling Protocol (PCP) protocol for local resources but uses simple spin-locks for globally shared resources, the novel TDRR approach is presented in Section 5.8 to improve WCRTs based on AMALTHEA via reducing busy waiting periods.

To evaluate the contributions, Chapter 6 presents seven case study models the approaches of Chapter 4 and 5 are applied to. Various details, properties, and contexts of both industrial and hypothetical models are presented and compared.

Metrics, benchmarks, and various measurements are then presented in Chapter 7 by applying the contributions under different configurations to the case study models. Results are evaluated and discussed.

Finally, Chapter 8 forms a conclusion for this thesis and outlines the contribution's benefits for the research community in terms of AUTOSAR as well as software parallelization and distribution for automotive systems. Research worth being further investigated in future work accompanies the conclusion in Section 8.2.

2 Background and Fundamentals

This chapter introduces various domains automotive systems employ and then systematically outlines necessary background information and related work to this thesis's contributions.

Compared to most other industries, automotive systems are subject to requirements and constraints from a higher amount of application domains. Consequently, automotive systems

1. employ multi-processor architectures to meet performance and energy demands (**concurrency** domain),
2. form a combination of hard- and software that serves a certain need across a larger mechanical or electrical environment to produce outputs based on given inputs such as data or events from peripherals (**embedded** domain),
3. are designed to produce outputs within predefined timing constraints (bounds) in contrast to general purpose or mobile computing applications (**real-time** domain),
4. have to serve tasks of different criticality levels due to safety (cf. ASIL and ISO26262[11]) demands and less critical applications (e.g. infotainment related) being executed in the same environment (**mixed-critical** domain),
5. are constructed by a huge amount of ECUs connected through various networks such as Controller Area Network (CAN), Local Interconnect Network (LIN), Automotive Ethernet, Media Oriented Systems Transport (MOST), and others (**distributed** domain),
6. and consist of heterogeneous PUs, due to the above mentioned domain diversity (**heterogeneous** domain).

These domains make automotive systems highly-constrained and demand sophisticated timing analyses and development tools. A dedicated community and domain-related research usually investigate each of such domains. Automotive systems combine, extend,

[11] International Organization for Standardization (ISO)

and integrate related technologies across domains and even add more approaches to meet the diverse requirements.

The major focus of this thesis is on multiprocessor technologies. The two most obvious reasons for applying multiprocessor technologies to automotive systems are (i) increasing computing performance and (ii) reducing energy consumption. A program can be calculated entirely concurrently in an ideal environment, and the system performance is then proportional to the number of PUs. The formal and coarse outline of both program performance and energy consumption is given in the Appendix H.1. In such an ideal environment, twice the number of PUs increases the performance by two accordingly. Though, in reality, software often does not have an ideal structure for being executed coherently concurrently and parts may require running solely sequential. Furthermore, overheads over single-processor execution stemming from, e.g., synchronization, scheduling, memory management, multiprocessor protocols, and others result in the proportionality coefficient between performance and the number of PUs being smaller than 1. Instead of increasing the number of PUs, increasing the PU frequency could result in a similar performance gain. Frequency scaling is often not possible though, due to physical limits in transistor manufacturing and energy consumption depending on temperature and transistor types and amounts [46], performance gain is not proportional to the operating voltage but approximately with its square. As a consequence, doubling a system's frequency instead of the number of PUs results in eight times the power consumption when assuming linear frequency-voltage relation, which is four times as costly as increasing the number of PU. As a consequence, using multiprocessor technologies is the more reasonable approach for (i) increasing performance and (ii) reducing energy consumption compared with increasing frequencies or decreasing transistor size, even though additional PUs require more complex system design and add various overheads.

New programming models and languages were introduced for multi-processor architectures in the automotive domain due to challenges that were not present with single-processor platforms. Additionally, migrating legacy software requires careful analysis and tools to meet parallelization, synchronization, or memory demands. According to the latter, a PU's local on-chip level 1 cache, shared on-chip level 2 cache, and off-chip memory require appropriate technologies to reduce cache interference, address real-time constraints, provision memory mapping (layouts), reduce contention effects, and more. A classic, but yet highly discussed, challenge specific to the automotive domain is the software distribution across ECUs of heterogeneous multi-processor structures, which is also addressed in this thesis.

2.1 Basic Terms and Concepts

This section describes terms and concepts used by the subsequent chapters and sections. Runnables, Tasks, Labels, Stimuli, and other AMALTHEA meta-model entities are described in Chapter 3 and similar and related models are outlined in Section 2.3. General information on partitioning and mapping problems is briefly given in Section 2.5, but more details and specific solutions are presented in the Chapters 4 and 5.

2.1.1 Scheduling Generics

Since timing verification requires to consider **scheduling**, related terminology is outlined in this section. Four main scheduling categories shown in Figure 2.1 can be used to distinguish the basic scheduling concepts.

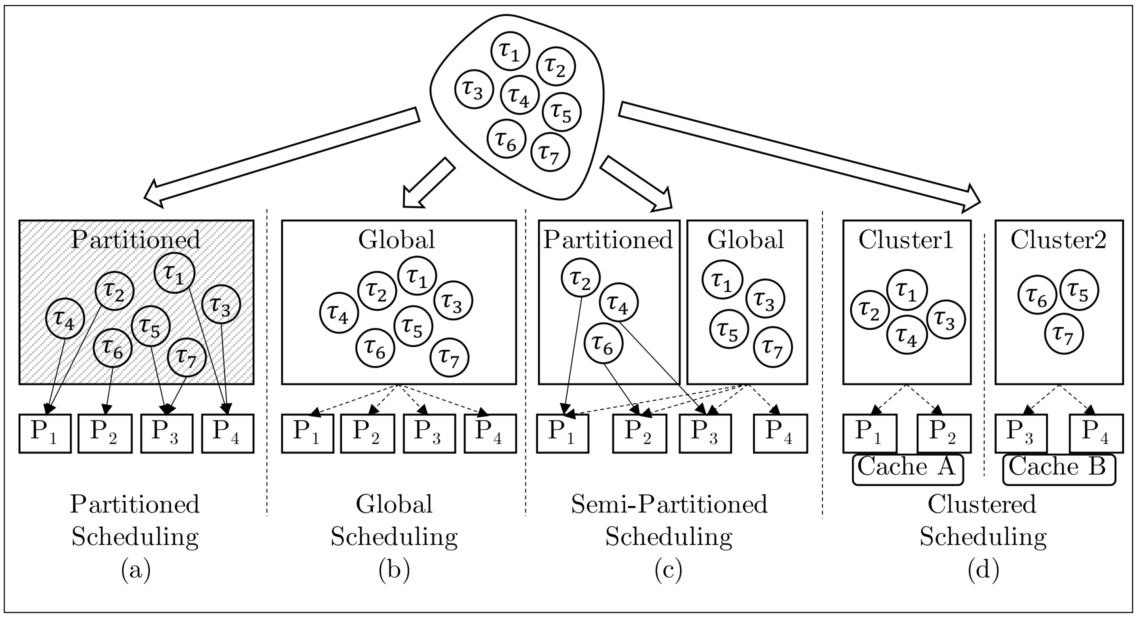


Figure 2.1: Basic concepts of (a) partitioned, (b) global, (c) semi-partitioned, and (d) clustered scheduling

Partitioned scheduling (a) is used for automotive hard real-time systems, and tasks are statically mapped to PUs based on sophisticated offline analyses provided by the mapping process. Each PU then employs its own scheduler that schedules tasks locally according to static priorities and rules provided by, e.g., RMS (preemptive, explained later in this Section), which is still preferred in AUTOSAR. In general, fixed-priority Deadline Monotonic Scheduling (DMS), dynamic-priority schedulers such as Earliest Deadline First (EDF), or WRR and others can be used at the same time locally on different PUs under partitioned scheduling depending on the required guarantees the scheduler provides and the scheduled task sets require.

In **global** scheduling, tasks are dynamically assigned to PUs during runtime. This approach (employed by Global-Earliest Deadline First (GEDF), for instance) is usually not applied in automotive systems due to complex timing verification, primarily when deriving tight response time bounds for dynamic mapping and the **migration** of tasks across PUs (see Section 2.3.1). Migration costs occur for global scheduling in case a task is mapped to a different PU either (i) at task instance (job) bounds or (ii) at any time. In both cases, migration costs must be taken into account for timing verification, and resuming a task on a different PU after it was preempted usually involves higher migration costs due to not only taking into account the inputs of the task, but also the system state it was conceiving during preemption. Migration costs typically include caches, registers, pointers, or even shared memory blocks. Consequently, the highly intertwined coherency of cause-effect chains, states, modes, and constraints manifest in too many uncertainties, high overheads, and complex timing verification such that AUTOSAR prohibits task migration [47]. Nevertheless, AMALTHEA still supports global scheduling, as shown in Section 3.1.4, which can be utilized by, e.g., AUTOSAR Runtime for adaptive Applications (ARA), which is though out of this thesis' scope.

Semi-partitioned scheduling (c) has been introduced in [48], bounds the number of task migrations, and hence reduces migration costs over global scheduling. Therefore, some (usually just a few) tasks are allowed to migrate to other (soft real-time) PUs, and other tasks are scheduled in partitioned fashion. The original semi-partitioned scheduling only covers soft real-time constraints for tasks, which are allowed to migrate, such that their application can at most be employed by ARA. Extensions to [48], which target at deriving hard real-time scheduling bounds and have been studied in [49], are out of scope here too, just as ARA. The same holds for **clustered** scheduling (d), which employs separated global scheduling for clusters. Migration beyond clusters is not allowed. Instead of dividing tasks into partitioned and globally scheduled task sets, which is required for semi-partitioned scheduling, clustered scheduling necessitates splitting a task set into disjoint sets, each of which is then scheduled across a cluster of PUs (semi-globally). According to [50], task and PU clusters are constructed based on the use of cache memory.

In addition to scheduler categories, specific scheduling scheme properties must be presented, such as whether a scheduler allows **preemption** or not. For critical software that relies on the highest priority tasks being executed as quickly as possible, preemption is necessary. Preemption allows critical tasks to preempt, i.e., interrupt, lower priority tasks such that the low priority tasks stop being executed, and the critical tasks take over and occupy the PU resource. Under AUTOSAR, an executable entity in the preempted state is denoted to be *ready*. An executable entity can be terminated to enter the suspended state, from where it can be activated towards the preempted state. From being preempted, it can resume towards the running state, from where the *wait* and *preempt* transitions reach the waiting and preempted states, respectively. This state machine is further shown in Figure 2.2 and part of the AUTOSAR Runtime Environment (RTE) specification in [51, page 149].

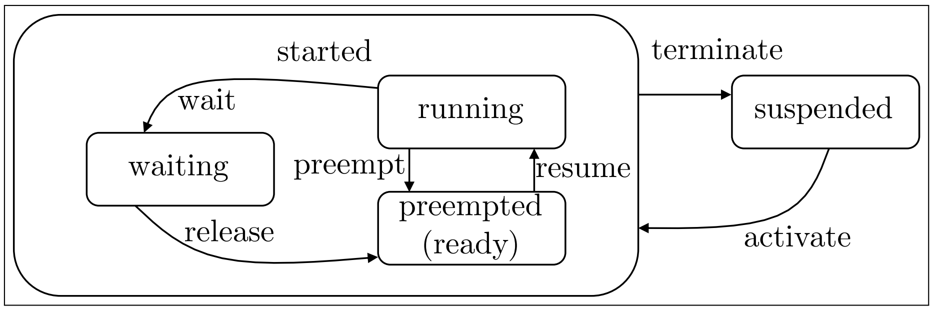


Figure 2.2: State machine for executable entities in AUTOSAR

In terms of AMALTHEA, the states are further extended by distinguishing between task, runnable, and component event entities, which are part of the `Events` meta-model. The amount of event types (state machine transitions) for tasks, for instance, is increased to 12 to cover six main states *polling*, *running*, *waiting*, *parking*, *ready*, and *active* as well as *terminated* and *not initialized*. More information on the AMALTHEA events model is given in [45]. Preemptive tasks increase the system's responsiveness, whereas preemption can be immediate, ignored, or at specific points during a task's execution in AUTOSAR. The latter applies to cooperative tasks, which can be preempted immediately from higher priority preemptive tasks or at runnable bounds [51, Chapter 1.4, page 33] from other higher priority cooperative tasks, which is after the task's current runnable finished its execution. Operating System (OS) tasks are usually non-preemptive, and other tasks are fully preemptive, i.e., they can be preempted immediately. AUTOSAR scheduling is consequently mixed-preemptive. RTA (cf. Section 2.6.2) for fixed-priority preemptive tasks (FPPS) has been studied in many research articles such as [38, 52–55], and more, whereas RTA for non-preemptive tasks (Fixed Priority Non-Preemptive Scheduling (FPNS)) [56] and limited preemption (e.g., Fixed Priority Deferred Preemptive Scheduling (FPDS) [57]), has gained fewer attention. Fixed preemption point (cooperative) scheduling has its origins in [54], is one type of limited preemption scheduling, and must be distinguished from preemption threshold and deferred preemption approaches. The former is investigated in [58] and only allows tasks beyond a specific priority level to be preempted, and the latter postpones preemption by a specific amount of time and has been investigated in [57]. These analyses are compared and put into context by a survey in [59]. Recent research also tackles the mapping problem for minimizing the shared resource unit number (PUs) under limited-preemptive scheduling in [60]. However, in this thesis, the optimization goal is primarily minimizing response times for a given hardware model rather than optimizing the hardware itself.

With automotive systems demanding a precise timing analysis for predictability and determinism, an adequate timing model must cover assessing the various scheduling situations. Figure 2.3 defines the typical **timing parameters** used throughout this thesis.

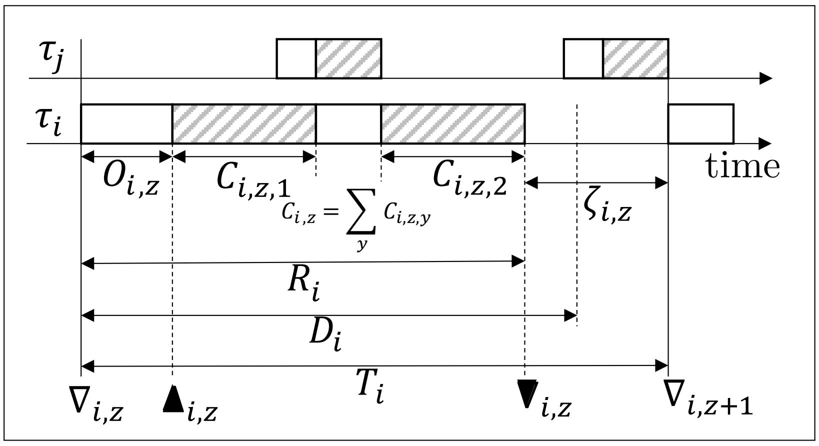


Figure 2.3: Typical timing properties of a task

The arrival time $\triangledown_{i,z}$ of a task τ_i's $z-th$ instance (job), is a specific point in time the task intends to be scheduled, which often repeats periodically with period T_i. Due to blocking or higher priority tasks, a task may have to wait until it is scheduled at a PU at release time $\blacktriangle_{i,z}$. This waiting time is typically outlined as initial pending time, phasing [38], or offset and denoted with $O_{i,z}$ here. According to the Burns notation [61], the offset is also known as release jitter and can vary across jobs, i.e., task instances. For preemptive scheduling, the execution of a task may be interrupted several y times, such that the tasks gross execution time may consist of several core execution times $C_{i,z,y}$. Hence, the total gross execution time of a task instance (job) is defined by $C_{i,z} = \sum_y C_{i,z,y}$. The gross execution time can vary at different instances z, due to varying Worst Case Execution Times (WCETs) C_i^+ and Best Case Execution Times (BCETs) C_i^-. The response time R_i is the sum of the gross execution time, all preemption times in between, and the initial pending time. WCRT-analysis is further presented in Section 2.6. The average slack time $\overline{\zeta_i} = \frac{\sum_z \zeta_{i,z}}{|z|}$ can give insights about the task's responsiveness, especially when it is compared with the task's response time. For instance, Chapter 7 uses the system's slackness $\overline{\zeta} = \frac{\sum_i \zeta_i}{n}$ (with n denoting the number of tasks) as a metric to compare the responsiveness for different partitioning and mapping results. The notations above are further consolidated in Section 3.2.

A task's **deadline** D_i represents a strict relative point in time, to which all of the task's instructions must have been executed concerning its arrival $\triangledown_{i,z}$. Hard real-time systems demand zero tardiness such that no deadline is allowed to be missed. If a task is still being executed or waiting for being executed at its deadline, its timing constraint is violated, and safe or correct system behavior can not be guaranteed. A deadline violation result depends on its type, either being hard, soft, or firm. According to safety standards (cf. Section. 2.4), a hard timing constraint (deadline) violation can result in various risks and hazards and ultimately cause human life loss. In contrast, a soft deadline violation may result in a tolerated behavior such as a video frame being lost within the infotainment system. Alternatively, systems can use tasks with firm deadlines, which denote that a sporadic deadline violation is tolerable, but a repeated miss of deadlines is not. Deadlines can be derived implicitly from the task's period (cf. RMS), so that a task must at least finish before its next instance arrives. Lehoczky has shown in [38] that RTA for Fixed-Priority (FP) task sets can be processed for arbitrary deadlines using the *level-i busy period* approach. This approach is used in this thesis, along with various adaptions and extensions outlined in Section 5. Sanudo et al. also used that approach in [62] for tasks with arbitrary deadlines along with the Formal Methods for Timing Verification (FMTV) challenge, but the closely related approach of this thesis further incorporates the various constraints presented in Section 3.1.6.

Task **priorities** are either (a) static, (b) dynamic but fixed within an instance (a task instance is often denoted as 'job' in related work), or (c) fully-dynamic. Examples are RMS (optimal for single PU fixed priorities [53]), EDF (optimal for dynamic priority [53]), and Least Laxity First (LLF), respectively. The former defines static priorities that are implied by task periods and never change later on so that deadlines are implicit. EDF scheduling derives priorities once upon tasks' arrivals so that task priorities are dynamic across task instances but static for a single instance. LLF scheduling defines fully dynamic priorities at arbitrary points in time. If priorities are not derived from task periods, the Audsley method [63] is one of the well-known alternative priority assignment approaches. The approach iteratively assigns priorities, beginning with the lowest, to tasks under the condition that the task is schedulable and all other unassigned tasks have higher priorities. Davis et al. have proven that the Audsley assignment is optimal for a set of scheduling approaches and schedulability tests in [64]. Even though the priority assignment problem has a significant effect on the utilization of resources such as the CAN bus as shown in [64], priorities are assumed to be known a priori for this thesis' technologies.

Schedulability defines whether a task set meets all its deadlines under a particular scheduling algorithm. In contrast, **feasibility** denotes if a task set can meet its deadlines disregarding the scheduling algorithm. An unfeasible task set is never schedulable, and a schedulable task set is always feasible. A feasible schedule can be schedulable for one scheduling algorithm but may not be schedulable under another scheduling algorithm. A good survey of schedulability tests is given in [49]. For instance, RMS exposes the **sufficiency test** of Eq. 2.1 with n denoting the number of tasks.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{2.1}$$

This means, for $n \to \infty$, the sum of all task costs C_i divided by period T_i (i.e. the task utilization values) must be lower or equal to $\ln 2 \sim 0.69$ to be schedulable [53]. This value is denoted as the **utilization bound**. Utilization values up to this bound are schedulable as they pass the sufficiency test. Utilization results beyond the utilization bound require further analysis and might be schedulable until the *exact utilization bound* (necessity test). For instance, under RMS, task sets exposing a utilization between 69% and 100% can be schedulable, if their periods are harmonic, i.e. the periods are integer multiple of each other. For homogeneous multi-PU systems, the optimal exact utilization bound equals the number of PUs. When trying to find the optimal utilization, the relationship of the calculated utilization and the exact utilization bound can be used as a metric to quantify utilization quality. EDF and LLF provide $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$ schedulability tests and hence allow 100% utilization without requiring harmonic task sets compared to RMS. Finally, to ensure hard timing guarantees, this thesis uses WCRT tests that account for not only specific scheduler properties, but also various (worst-case) situations and constraints. The approach is close to exact schedulability tests that include blocking times, interference, and network delays, which also have been investigated in [24, 65–67].

Optimality of EDF and LLF scheduling algorithms for single PUs has been proven in [68] and [69], respectively. Using GEDF scheduling for multiple PUs has been studied a lot, first along with soft real-time constraints [70] and later on for hard deadlines [71]. Although optimal online multi-PU scheduling for sporadic tasks is impossible as proven in [72], various research addresses tightening utilization bounds on GEDF such as [73] or [74]. Alongside GEDF, Proportionate Fairness (PFair) [75] has gained significant research interest, which targets at granting processing resources to tasks proportionally to their utilization factor. Therefore, PFair scheduling uses time quanta as schedule points, is known to be optimal for global fixed-priority scheduling, but though exposes several limitations such as constant execution times or implicit deadlines as stated in [76]. Over the last decade, many advancements have been published for global scheduling to overcome different limitations and general drawbacks such as context-switch and priority determination overheads, or the assumption of constant execution times. However, in terms of classic AUTOSAR with hard real-time constraints, determinism and tight latency bounds as well as a precise analysis of overload conditions, at which low-priority tasks may miss deadlines up to a bounded point, are in favor over dynamic priority scheduling approaches such that partitioned RMS is still the prior choice over, e.g., GEDF scheduling or similar global approaches that involve migration costs. If not stated otherwise, tasks are considered to have hard real-time deadlines in this work.

In context with scheduling, the **starvation** term is used to investigate the influence of high priority tasks to lower priority ones. Starvation needs to be investigated to guarantee that lower priority tasks eventually get processing resources (for being scheduled and executed) and do not starve, i.e., wait an unbounded amount of time. This thesis implicitly ensures freedom from starvation by considering deadlines on a holistic basis, i.e., low priority tasks never starve if the formal analysis proved schedulability, which means that all tasks complete their instructions at latest by their deadline.

Priority inversion defines the situation when a lower priority task is executed, although a higher priority task arrived and demands for being scheduled. In a non-preemptive scheduling situation, the priority inversion time can be as long as an entire task execution, since a higher priority task may have arrived just right after the lower priority task, which was just scheduled instantly. In First In First Out (FIFO) scheduling, the priority inversion can be even much longer than a single task if multiple lower priority tasks arrived just before the higher priority task. Bounding priority inversion is not only of interest for schedulers, but also targeted by various resource sharing protocols for the multiprocessor context, which are outlined in Section 2.8.

For parallel and concurrent systems, **deadlocks and race conditions** are typical challenges developers need to face. Parallel executing tasks can run on a single PU system, which means that tasks make progress over time, since the scheduler assigns processing time in a specific fashion, although the actual execution is sequential. In contrast, the concurrent progress of a system is defined by tasks making progress simultaneously, i.e., tasks are scheduled on multiple PUs. **Mutex** implementations such as a monitor or a semaphore can prevent race conditions by ensuring that resource changes are done mutually exclusive and in a serializable fashion. Sequential resource accesses ensure that a task's result only depends on its input, but never on the order its runnables are executed. Deadlocks usually require a more detailed analysis than race conditions since resource accesses can be nested, and preemption may cause locks being held by preempted tasks.

2.1.2 Partitioned FPMP Scheduling

According to AUTOSAR classic, scheduling is assumed to be partitioned, i.e., each PU employs a single scheduler that is concerned with scheduling a fixed-priority mixed-preemptive task set (Partitioned Fixed-Priority Mixed-Preemptive Scheduling (PFPMPS)) [77]. Global scheduling, i.e., a single scheduler that dynamically releases tasks across multiple PUs, could be employed by ARA, but is not in the scope of this thesis. In PFPMPS, a task set is scheduled in isolation, which results in no migration costs. The only interleaving influence is blocking of shared resources, i.e., buses, shared memory, dedicated hardware, or similar exemplary shown in [78]. Applying RTA solely for each PU is an established method according to existing literature. The scheduling policy can be RMS, for instance, which has been proven to be optimal for single PUs and fixed priorities [53] and hence is optimal for the partitioned scheduling case, too. Its advantage is that complex timing verification, accompanied by the mapping DSE that involves several constrains, can be calculated offline to optimize various metrics. This process is NP-hard and requires appropriate mechanisms (cf. Section 5.9), e.g., polynomial-time approximation algorithms to overcome the intractability. After the mapping process, PFPMPS can be run on a heterogeneous multi-processor system serving all constraints, requirements, and optimization goals. The two prerequisites *(i)* (offline) timing verification and *(ii)* partitioning and mapping form two of the main contributions of this thesis, which are automated and supported by various novel technologies. Although PFPMPS does not provide sharing idle times in a cross-PU manner, ensuing strict timing and safety guarantees is still crucial and SotA for industrial AUTOSAR systems. Therewith, offline, RTA-verified, and holistic constraint considering partitioning and mapping processes have to serve different goals such as temporal and spatial isolation. Thus, RTA is extended in this thesis to cover modern hardware and software requirements, including timing verification subject to partitioning and mapping for PFPMPS.

2.2 The Heterogeneous Era

Heterogeneity gained significant importance in the recent decade, and related research will presumably continue growing in the near future. PANORAMA[10], for instance, forms an international research project for developing many-fold heterogeneous automotive systems. More precisely, architectures in the automotive domain not only consist of many-fold **heterogeneous hardware** platforms, but also involve **heterogeneous function domains** and **diverse parties** in the development process, which imposes new challenges during timing and constraint verification.

In terms of **hardware**, ARM's *big.LITTLE* architecture is an example a of a heterogeneous system consisting of different PU types. ARM's *big.LITTLE* [79] initial system was released in 2013 in the form of two 1.8GHz Cortex-A15 and two 1.2GHz Cortex-A7 cores. Later on in May 2017, *big.LITTLE* was superseded by the DynamIQ architecture that increased the number of cores per cluster to eight and improved voltage and cache management. The WATERS2019 challenge [30] (solved in Section 6.2) addresses the heterogeneous NVIDIA Jetson TX2 platform[12], which consists of two 64-Bit Denver CPUs, four ARM Cortex-A57 MPCores and 256 NVIDIA CUDA cores from the Pascal^{TM} GPU architecture. NVIDIA further targets more powerful high end architectures [80]. The NVIDIA Jetson AGX Xavier is currently under development and will feature new dedicated deep learning and vision accelerators for artificial intelligence in addition to several other advancements to outperform not only the NVIDIA Jetson TX2 but also general-purpose computers. In addition to heterogeneous PUs and accelerators, research also has to face varying memory types, communication paradigms, and software of different criticality levels. For example, data access times can vary between memory types, and having shared memory usually requires mutual exclusion algorithms, synchronization mechanisms, deadlock, and race condition prevention.

As mentioned above, the many-fold function **domains**, e.g., control engineering, stream processing, and cloud or cognition computing, further accompany heterogeneous hardware. Modern Adaptive Cruise Control (ACC) systems, for instance, use control loop algorithms to keep a distance to a vehicle ahead based on front radar sensor values. The Enginge Management System (EMS)'s target speed is adjusted by the ACC regarding either acceleration, keeping the current speed, or, if required, the vehicle brakes are activated. Additionally, image processing is used by speed sign recognition applications to adjust the vehicle speed according to road traffic signs at the same time the ACC runs. The latter application usually adjusts the ACC's target speed. Finally, smart navigation applications react to the latest traffic updates, which can also influence the vehicle speed. Examples can be wrong-way driver notifications, recent road accidents, or dynamic speed limitations on roads. In such situations, involved applications demand domain-related requirements and affinities to, e.g., Digital Signal Processors (DSPs), GPUs, Floating Point Units (FPUs), Field Programmable Gate Arrays (FPGAs), Aplication-*S*pecific Integrated

[12]https://developer.nvidia.com/embedded/jetson-tx2, visited 11.2020

Circuits (ASICs) mobile network gateways, which have to be considered by appropriate mechanisms during the development. Moreover, modern and next generation autonomous vehicles require the domains to increasingly interact with each other.

Finally, the collaborating **parties** involved in developing automotive systems are heterogeneous, too, since OEMs, tier suppliers, tool vendors, and semiconductor companies jointly cooperate to achieve products that are verified to be integrated to cars, often manufactured in series of millions. A hardware manufacturer might be concerned with electromagnetic emissions or vibration isolation, whereas the tier one supplier targets better resource utilization, and the OEM wants to reduce overall costs. Different party goals, approaches, tools, and variants interfere with each other and significantly influence the systems engineering process.

Using Model Based Systems Engineering (MBSE) is an established methodology in the automotive industry and forms a reasonable approach to address challenges of the heterogeneous era. The next Section 2.3 introduces various models and their benefits, especially when addressing the above-outlined heterogeneity challenges. Additionally, the heterogeneous era demands for significant innovation in timing verification (RTA) and DSE tools to cope with the ever-increasing performance demands and at the same time meet the vacillating concerns of real-world scenarios. Therefore, a combination of MBSE, RTA, and DSE is presented in this thesis and applied to real-world benchmarks and models.

2.3 Model-based Automotive Engineering

Models are preferably used in the automotive industry due to the following benefits:

- Models are based on meta-models and hence formally defined.
- They can form an abstraction of entire systems or parts of a system to reduce complexity and ease system analysis.
- Using models allows various partners involved in a product to exchange information without having to exchange actual program code, i.e., their Intellectual Property (IP).
- Models provide the basis for key artifacts, e.g., automated testing, traceability, various analyses for timing, behavior and others, code generation, consistency checking, and simulation.
- The above key artifacts can be used across various system development phases at ease and often create less verbose data / memory footprint.
- Model-based design allows continuous validation and verification along with phases of the V-Model from system design to implementation and testing.

Liebel et al. studied the use, challenges, and shortcomings of industrial MBSE in [81] via a state-of-practice survey, and not only found out that MBSE is a widely established process in the embedded and automotive domains, but also that (i) Eclipse-based tools, e.g., Eclipse App4mc, are most frequently used next to Matlab/Simulink, and (ii) interoperability and usability are common challenges. The latter issue can indeed be tackled by using a common yet open-source basis such as Amalthea outlined in Section 3.1, which is already used along with industrial projects and research such as [32–34, 40, 62, 82–93], and many more.

The following Sections 2.3.1–2.3.6 outline various automotive related models whereas the next Chapter 3 is dedicated to the AMALTHEA model used in this thesis.

2.3.1 AUTOSAR

The AUTOSAR de-facto standard is well established in the industry since it is used by nearly every OEM and tier supplier. "AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers, service providers, and companies from the automotive electronics, semiconductor and software industry"[6]. The initial AUTOSAR version was released in 2003, and the latest release valid for this thesis is 4.4.0 and was released in late 2018. Its main goal is to standardize the software architecture for ECUs to form cross-vehicle and -platform variant interoperability, scalability, modularity, maintainability, and sustainability in line with requirements and standards such as safety and ISO26262, respectively. It defines standardized and highly configurable interfaces and design processes. AUTOSAR specifications are publicly available, but tool support such as Artop[13] is reserved for AUTOSAR members only. AUTOSAR exposes a meta model that has been advanced by timing extensions [94] since version *4.0.1* (12.2009) through the Timing Augmented Description Language (TADL), which forms the outcome of the two international research projects TIMMO and TIMMO2USE[14]. Such extensions cover timing constraints (age, synchronization, offset, order), events, and event chains, which are analyzed in this thesis' Section 5.5. The AUTOSAR layered software architecture is one of AUTOSAR's most known specifications and describes components of and interfaces between AUTOSAR software in the form of Software Components (SWCs) in the application layer and services of the Basic Software (BSW) layer. These two layers are connected through the RTE, and the basic software runs on top of the ECU hardware. SWCs run independently of the infrastructure using the Virtual Functional Bus (VFB) and usually employ a set of runnables. Runnables are triggered by one or several events based on, for instance, the periodic task they are running for, data receive events, notification events of other processing entities, server function calls, and others. Communication between runnables is represented by label accesses as outlined in the notations of Section 3.2. For intra-SWC communication, such data is used as inter-runnable-variables whereas inter-SWC dependencies are realized through port interfaces [95]. SWCs typically represent affinity constraints, which are used for the software distribution technology presented in Chapter 5. Scheduling under AUTOSAR is FPs-based [47, Chapter 7.1.1], partitioned [47, Chapter 7.9.2], and tasks do not migrate [47, Chapter 4.5.3]. As a consequence, this thesis concentrates on analyses for FPPS.

Since 2017, ARA is released, which extends AUTOSAR by a hypervisor and allows POSIX-based OSs. Additionally, to some extent, even Linux OSs are used in the automotive domain [96] that further increase the technology diversity. However, the concepts and technologies provided in this thesis are concerned with AUTOSAR classic.

2.3.2 SHIM

Another model related to multi-processing and the automotive industry is SHIM™[15] (Software-Hardware Interface for Multi-many-core). SHIM provides an interface to

[13]https://www.artop.org, visited 11.2020

[14]http://adt.cs.upb.de/timmo-2-use/, visited 11.2020

[15]https://www.multicore-association.org/workgroup/shim.php, visited 01.2020

exchange multicore hardware platform properties, which affect the software at the architectural design level. SHIM has been developed by a cooperation of academia and industry and is maintained as well as extended by the MULTICORE ASSOCIATION. It supports tools that focus on the architectural design level of multi- & many-core systems. Therefore, various characteristics like communication channels between cores (e.g., routing, message passing protocols), memory (e.g., memory size, access latency, hierarchy, topology, coherency), and other cores or accelerators (e.g., instructions, special execution units, address space) are described in an EXtensible Markup Language (XML) schema. By exchanging the SHIM XML from a hardware vendor for a dedicated hardware platform, tool vendors can use this description as an interface to optimize and analyze system configuration and implementation such as the mapping of software tasks to hardware as well as performance analyses. Software architecture, middleware, runtime environment, OS, or hypervisor related properties like scheduling are though not supported by SHIM.

2.3.3 SysML

SysML [97] is a general-purpose systems modeling language initiated in 2007 and standardized by the Object Management Group (OMG). SysML aims at covering challenges of complex multi-disciplinary systems via using MBSE. In contrast to conventional document-based design specification approaches, SysML explores domain models (or model-based specifications) as the primary format for information exchange between engineers, while directly involving project stakeholders in early development stages. Capturing consistent system requirements, system contexts, use cases, boundary conditions, functions, and their subsystem interfaces from different interacting engineering disciplines (such as software, hardware, engine, control, power electronics, mechanics, etc.) down to the detailed design at the component level are in the scope of SysML. Hence, SysML is trying to address the growing variability of products and the increasing complexity from different domains. Besides, these domains are often mixed up in new, often beforehand, unforeseen operation contexts, which leads to potential interoperability problems. Therefore, an architecture model comprises a structured set (or a graph) of logically interconnected model elements that provide the ability to automatically query a context of any function with corresponding requirements, structure, behavior, parameters, and intended scenarios of its use during any development phase. Traceability, validation, and verification can thus be applied to specifications and requirements, boundary conditions, and interfaces for system components from different domains.

Since SysML does not provide methodologies to guide engineers along with using the MBSE technology, other tools and specifications like INCOSE Object-Oriented Systems Engineering Method (OOSEM [98]), SYSMOD [99], Architecture Analysis and Design Integrated Approach (ARCADIA) [100], IBM Rational Unified Process for System Engineering (RUP-SE) [101], SPES/SPES XT [102], and others can be used along with SysML. Although the interest in SysML and the MBSE methodology has grown in the last decade [103], criticism exists regarding an extensive use of language extendability in terms of Unified Modeling Language (UML) profile extensions across developers from different domains. The resulting interoperability of SysML models further manifests in the standard format for storing SysML metadata information, i.e., OMG XML Metadata Interchange (XMI) [104], which becomes increasingly tool and vendor specific. Due to the generic nature and syntactic and semantic overlap with UML, SysML models lack in precise and unambiguous semantics to provide formal verification or model checking methods.

2.3.4 Real-time Calculus

The Real Time Calculus (RTC) is a deterministic queuing theory model to find hard performance bounds for real-time systems. RTC extends the Network Calculus [105], which also uses the stochastic queuing theories and targets data flows and queuing networks, by real-time concepts. In contrast to this thesis' analytical approaches, RTC uses event streams along with cumulative functions (request and delivery curves) to simulate not only the arrival of tasks but also, e.g., available resources. In fact, AMALTHEA provides an interface to RTC since the Stimulation model includes *arrival curve* descriptions. RTC is closely related to Compositional Performance Analysis (CPA), which has been used along with *pyCPA* to solve recent challenges in automotive systems development (cf. Section 6.2) in [106] for WCRTs analysis and in [33] for data age and reaction latency values derivation under different communication paradigms (cf. Section 5.5). However, together with model checking and discrete event simulation, stochastic and deterministic queuing theory is not in scope of this thesis since much research is already available for these techniques such as [107–112], or [113][16] among others.

2.3.5 Automotive SPICE

Automotive SPICE[17] is a process assessment and reference model developed by a common special interest group of automotive OEMs, the Procurement Forum, and the SPICE User Group. It considers requirements of ISO/IEC 33004 ("Requirements for process reference, process assessment and maturity models" [114]) and defines groups and processes that need to be involved in the development process. For instance, the SPICE reference model defines primary-, organizational-, and supporting life cycle processes. Primary life cycle processes are grouped into acquisition, supply, system engineering, and software engineering process groups, whereas organizational groups are structured into the management process-, reuse process-, and process improvement process groups. The groups are in charge of various processes along the V-model from requirements specification to system qualification test and various processes in between. An assessment rates different processes according to capability levels and process attributes. Automotive SPICE imposes specific requirements parties involved in developing automotive systems need to fulfill. Verification is one of those processes which includes timing analysis. Consequently, this thesis's technologies form mandatory processes required by Automotive SPICE established by OEMs, tier suppliers, and tool vendors.

2.3.6 Other Models

EAST-ADL[18] is an approach to model automotive architectures on an abstract basis and its initial version 2.0 was released in 2008. ASAM MDX (Association for Standardization of Automation and Measuring Systems - Model Data eXchange format) is another model which was initially released in 2006 by German OEMs and suppliers for automotive systems and specifically addresses software modeling. The current format is XML and contains descriptions of function interfaces, parameters, variable data, and scheduling. ASAM works in cooperation with AUTOSAR and ISO such that AUTOSAR specifications, like the SWC-Template, are strongly influenced by ASAM MDX [91]. Further extensions have been

[16]As part of the Synopsys Inc. CoMET System Engineering IDE `http://www.synopsys.com`, visited 11.2020

[17]Automotive SPICE `http://www.automotivespice.com`, visited 11.2020

[18]EAST-ADL `www.east-adl.info`, visited 11.2020

outlined in research such as TADL [115], which covers the logical time and different time bases and units to utilize synchronous languages for timing analysis. Additionally, models of computation and communication exist such as Polygraph that extends Synchronous Data Flow (SDF) with frequency and communication arithmetic [116]. The Polygraph model uses the MARTE[19] modeling language and hence provides interfaces to schedulability analysis tools like Polarsys Time4Sys, MAST [117][20], and PyCPA [106]. In combination with Papyrus, which is the graphical modeling tool, or Capella Tools[21], further interfaces likely exist for using Pegase[22] or Cheddar[23]. More information on schedulability analysis and simulation tools is given in Table 2.1.

TADL [115] has not been added to AUTOSAR, ASAM MDX is neither open-source nor publicly accessible, and EAST-ADL does not include the broad characteristics of AMALTHEA. Polygraph seems to be a promising approach, but as stated in [116], some components related to SDF are still closed source. With the analysis of the above-outlined models, AMALTHEA has been identified as the most appropriate model for investigating partitioning, mapping, and timing verification challenges for the automotive domain. AMALTHEA is further described in Chapter 3.

With SysML trying to mitigate the effect of heterogeneous function domains, automotive SPICE addressing, among others, increasing complexity due to heterogeneous parties involved in the development process, and AMALTHEA, AUTOSAR, EAST-ADL, SHIM, and others covering abstraction, systems engineering, architecture, and system modeling, the technology stack is already getting dense without even taking safety, distribution, parallelization, timing verification, DSE, resource optimization, and other technologies in the scope of this thesis into account. The latter are described and put into context in the following.

2.4 Safety & Criticality Levels

Automotive functional safety is often referred to as ISO26262 due to the international recognized ISO, which has worldwide members of 162 countries. The ISO26262 standard defines requirements and processes to exclude hazards caused by malfunctioning systems. Therefore, electric components from sensors, actuators, networks, over microcontrollers to ECUs must be analyzed according to properties defined in the standard. Such properties must be measured along with safety reviews, audits, and assessments. ISO26262 includes ASILs, which classify a software's relation to potential hazards and risks. ASILs are derived from risk assessment, which is based on severity, frequency, and controllability values, i.e., the possible damage impact, the probability of exposure, and the driver's possibility to react to the malfunction, respectively. ASIL classes, i.e., criticality levels, reach from A to D, whereas D defines the most critical risk or hazard. Considering functional safety during automotive systems development has been investigated in dedicated research, e.g., in [118]. Model-based safety analyses using error models and error propagation across SWCs has been studied in, e.g., [119, 120] and is out of scope here.

Figure 2.4 shows an example of four CPUs with dedicated schedulers for each ASIL.

[19]MARTE `http://www.omg.org/omgmarte/`, visited 11.2020
[20]MAST `http://mast.unican.es/`, visited 08.2020
[21]Polarsys `https://polarsys.org/capella/`, visited 11.2020
[22]Rtaw-Pegase `https://www.realtimeatwork.com/software/rtaw-pegase/`, visited 11.2020
[23]Cheddar `https://bit.ly/3kPzqIq`, visited 11.2020

Typically, partitioned scheduling is preferred for hard real-time software and global scheduling for soft real-time. Criticality levels can but do not necessarily have to respect priority levels, such that the priorities of ASIL-D tasks are higher than priorities of tasks from ASILs C–A.

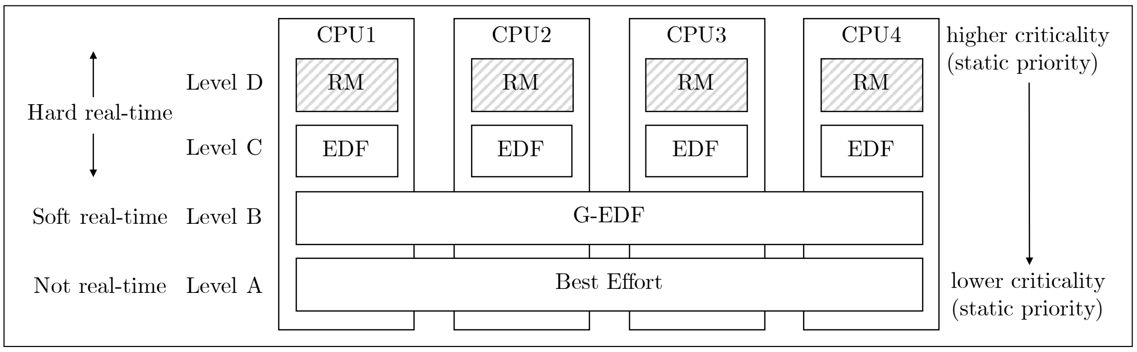


Figure 2.4: Example on schedulers used for different criticality levels based on [121]

In this thesis, ASILs are considered during the partitioning and mapping technologies as outlined accordingly in Chapter 4 and 5, respectively. Thus, to guarantee, e.g., FFI for ASIL-D, engineers can choose isolation from lower ASILs or even any other software, which is decoded into separation constraints. Alternatively, dedicated hardware such as FPGAs can be used for spatially increasing FFI for safety critical tasks, which has been studied in [89] and used along with Amalthea and App4mc. FPGAs therefore require advanced configurations and modeling to form deterministic execution units. Results of [89] show that timing analysis for safety critical tasks can be significantly mitigated and hardware interference can be reduced in a mixed FPGA-CPU environment. The approach can be used in combination with analyses of this thesis via a corresponding model, but the case study models presented in Chapter 6 do not contain such information.

When using separation constraints as the first choice, the provided technologies still guarantee meeting timing constraints (deadlines) by formal verification according to priorities and the scheduling paradigm even if the amount of, e.g., ASIL-D software exceeds hardware capabilities such that FFI can not be guaranteed by isolation. In that case, ASIL-D software is required to have higher priorities than software that is tagged for ASIL levels C–A. Consideration of redundancy, which is also applied to higher ASIL software as stated in [122], is not in scope here due to the generality of this thesis' technologies. However, Amalthea and the MBSD provides to manually model redundancy and priority values at ease, such that a corresponding consideration is, in general, possible.

2.5 Software Distribution

Software distribution for automotive systems gained significant importance in recent years due to the increasing demands of advanced driver assistance systems, autonomous driving, as well as architectural changes towards the centralization and consolidation of functional domains and ECUs [96]. Additionally, standardization, as mentioned in the previous sections (e.g., Autosar, automotive Spice, collaboration across Tier suppliers, OEMs, and various tool vendors, and requirements from legacy applications) necessitate sophisticated approaches when applying software distribution methodologies to the already

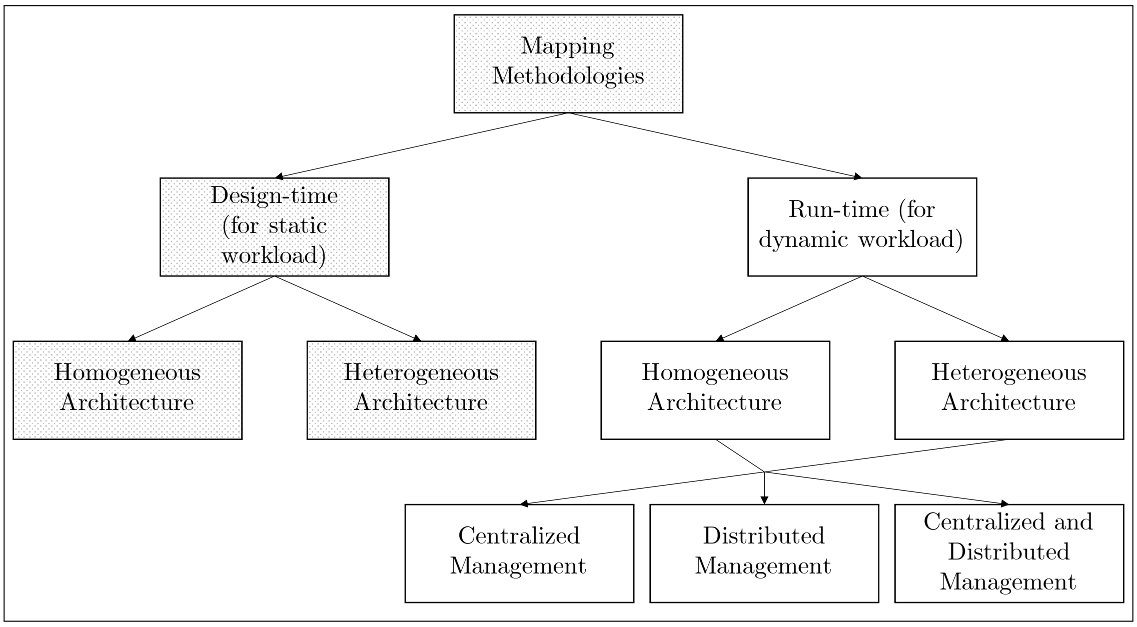

Mapping Methodologies
Design-time (for static workload)
Run-time (for dynamic workload)
Homogeneous Architecture
Heterogeneous Architecture
Homogeneous Architecture
Heterogeneous Architecture
Centralized Management
Distributed Management
Centralized and Distributed Management

Figure 2.5: Mapping taxonomy from [123]

highly constrained domain of automotive systems.

The fundamental concern of software distribution in the AUTOSAR context is the **partitioning** of runnables, i.e., atomic functions, to tasks and the **mapping** of such tasks to PUs across microcontrollers and ECUs. More precisely, given a set of runnables $\mathcal{R} = \{r_1, ...r_p\}$ and a set of PUs $\mathcal{P} = \{P_1, ..., P_u\}$, the goal is to find (a) a distinct runnable to task assignment $M^{\mathcal{T}}_{r_a} = [1, n] \ : \ \forall r_a \in \mathcal{R}, a \leq p; p = |\mathcal{R}|; n = |\mathcal{T}|$ that is calculated by the partitioning process and forms the task set $\mathcal{T}$, and (b) a distinct task to PU assignment $M^{P}_{\tau_i} \in [1, u] \ : \ \forall \tau_i \in \mathcal{T}, i \leq n, u = |\mathcal{P}|$ denoted as mapping. Both technologies are valid if and only if there exists precisely one task assignment for each runnable and hence a single PU mapping for each task. From the formal analysis perspective, minimizing WCRTs under the partitioning and mapping problems (load balancing) is an NP-hard problem [28]. This two-phase approach yields many advantages such as distribution flexibility, level-based pairings, or separations, and the consideration of various constraints described in Section 5.1. While this rather generic perceiving challenge has been studied for decades, the mandatory domain-specific constraints' holistic concern has been either omitted or only partially investigated. By making use of the AUTOSAR compliant AMALTHEA model, this thesis's work applies to a widely established superset of automotive constraints on the one hand (cf. Section 5.1), and further covers industry-driven requirements on the other hand.

Related work stretches across a wide variety of application domains. Figure 2.5 shows a taxonomy of mapping problem methodologies with the highlighted left parts being addressed in this thesis. Partitioning and mapping are investigated during design time for homogeneous and heterogeneous hardware, and workload and communication behavior are assumed to be known and predefined. In this scenario, DSE is typically addressed by dedicated heuristics, Mixed Integer Linear Programming (MILP), GAs, or Constraint Programming (CP) (see Section 2.7). However, automotive systems demand various requirements combined by its six-fold domains introduced in Section 2, which all DSE approaches have to address. For example, processor affinities are beneficial regarding application performance, fault tolerance, or security as stated in [124, 125]. Such PU affinity for tasks is considered via *arithmetical* constraints in Chapter 5, ensuring that a solution must contain given task to PU pairing. Similarities exist compared with robotics, logistic, or avionics domains, but only the latter typically exposes similar criticality and network interface levels. In this context, typical optimization goals reach from execution time, energy consumption, resource utilization to reliability, or solution quality, as stated in a mapping survey in [123].

Typical greedy heuristics to the mapping problem have been outlined in [126] such as First Fit (FF), Best Fit (BF), Worst Fit (WF), decreasing utilization, or decreasing criticality. Although such greedy approaches do not optimize any system parameters, they are often taken as reference mappings. Approaches to software distribution in this thesis assume a predefined PU set. Task allocation strategies that only consider a task set and processing capabilities (instructions per second), but not a predefined number of PUs (those approaches assume to have n number of tasks, theoretically scheduled on n PUs in the worst case) as described in [127] are not in scope.

2.6 Timing Verification

Timing verification is the basis for WCRT analysis to ensure that all system's deadlines are met, and resources are used efficiently under all given properties and circumstances. Approaches for timing verification presented in this thesis use formal schedulability analysis, RTA, and holistic methodologies. Hamann et al. have argued in [128] that holistic approaches require an increasing number of equations and dependencies per component and timed automata scale exponentially. However, holistic approaches can consider global performance effects much easier than compositional analysis, which potentially results in tighter timing bounds [129]. In general, it is common practice that timing verification is covered by measurements and tracing of software on target-hardware, which requires that both are available by the time the measurements are performed. However, if such process is not automated, it is highly error-prone due to manual configuration and setup, and the absence of hardware and software in early development phases further highly aggravate early timing verification and estimation. Consequently, appropriate methodologies and tools have been invented and used for timing verification and performance analysis, exemplary given in Table 2.1. The order of tools is arbitrary within a cell, and the table does not intend to be an exhaustive survey of timing verification tools and performance analyses. Tools are assigned to a methodology based on their primary concern, but they may cover different concerns. For instance, some simulation tools, CPA, model checking, and formal analysis techniques also claim to cover holistic analyses.

Methodology	Short Description	Example Tools
Simulation	Hardware and software models are triggered by discrete event sets (imitation of environment). Behavior and non-functional properties can be investigated by debugging / tracing a simulation scenario. Usually, not every possible situation can be covered.	SystemC, SCoPE [130] INCHRON chronSim[24] Vector TA-Toolsuite[25] RTaW-Pegase [131] Simulink[26]
Compositional Analysis (CPA)	Creating request, delivery, arrival, and service curves to feed a system and propagate events through a model, which consists of several components connected via a network. Schedulability analysis complexity is reduced by calculations done component-wise instead of globally.	RTC (cf. Section 2.3.4) pyCPA [106] Symtavision/Luxoft SymTA/S [132]
Model Checking	Using, e.g., temporal logic to define a correctness property, which is checked against a formal model (e.g. timed automata that includes states, clocks, transitions, actions, etc.)	UPPAAL [133] KRONOS [134] Romeo [135] SPIN[27], DREAM[28], MUNTA[29]
Formal Analysis	Apply schedulability and utilization tests as well RTA and End-to-End (E2E) latency equations to abstract software and hardware models. Mostly addresses worst and best case situations only.	MAST [30] [117] Cheddar[23]
Holistic Analysis	Extends formal analysis (or CPA) by bus communication delays based on various protocols (e.g. Time Division Multiple Access (TDMA)), offsets, arbitrary deadlines, resource sharing, synchronization etc.	*Tindell et al.* [136] *Pop et al.* [137] *Palencia et al.* [138]

Table 2.1: Timing verification and performance analysis methods and tools

The double line in Table 2.1 separates formal and holistic analyses, which are in the primary scope of this thesis, from other timing verification techniques. Italic references in the last row and column of Table 2.1 denote research rather than available tools.

Investigating safety constraints or failure rates in combination with causality analysis is often performed by model checking. This algorithmic technique systematically explores all

[24]chronSIM https://www.inchron.com/tool-suite/chronsim/, visited 11.2020
[25]TA Toolsuite https://bit.ly/34u8Fll, visited 11.2020
[26]Simulink https://www.mathworks.com/products/simulink.html, visited 11.2020
[27]SPIN http://spinroot.com/spin/whatispin.html, visited 11.2020
[28]DREAM http://dre.sourceforge.net/, visited 11.2020
[29]MUNTA https://github.com/wimmers/munta, visited 11.2020
[30]MAST https://mast.unican.es, visited 11.2020

system's states and configuration to discover erroneous or undesired situations. Leitner-Fischer et al. apply model checking in [122] to an automotive scenario (airbag control unit) to analyze FFI along with ISO26262. Their work shows that model checking can be used to verify FFI by ensuring that a safe state is reached under any circumstances, even when injecting a deadlock situation. However, model checking has shown issues such as the state explosion problem [139], which makes alternatives more attractive.

To classify timing verification and performance analysis techniques via covered features and properties, a recent "Survey of Timing Verification Techniques for Multi-Core Real-Time Systems", which is given in [140], provides a good feature set and overview of recent timing verification techniques used in the embedded real-time domain, which also applies to automotive systems. In accordance with [140], this thesis' contributions can be outlined as shown in Table 2.2. The assessment is done by combining publications [4, 9, 17, 19, 20] as well as new content, e.g. from Section 5.5. Abbreviations are outlined in the following list.

Type of Analysis				Timing Model				HW properties						SW properties		
I	W	S	M	E	γ	Mi	ID	Pt	#C	#L	LM	GM	IC	Pr	Ph	Is
X	x	X	X	X		x	X	H,P	N	N	C,S,BM	C,S,BM	B,NoC	X	X	X

Table 2.2: Timing verification features of this thesis based on [140]

Consequently, timing verification mechanisms used in this thesis cover all of the SotA properties outlined by [140] except Cache Related Preemption Delay (CRPD) as described in the following in more detail (colors and notations in line with [140]):

1. I = Interference Analysis: Since tasks are not necessarily executed in isolation, tasks can interfere with each other and influence each other's timing properties. For instance, RMS or WRR scheduling (cf. Sections 5.4.2 and 5.6.2) consider priorities and preemption, which significantly influence response times. Furthermore, the memory mapping approach presented in Section 5.7 considers various network interfaces between hardware modules such as ECUs.

2. W = WCET: This thesis identifies worst-case situations and considers WCET bounds for a set of tasks running on a target platform. WCETs are assumed to be given and not derived from dedicated WCET analysis technologies except the derivation of time from a task's instructions on a specific PU (lower case x denotes minor feature consideration).

3. S = Schedulability Analysis: As part of Section 5.2, this thesis considers utilization and schedulability analyses.

4. M = Mapping Analysis: In addition to schedulability analyses, Section 5.9 solves the task to PU mapping problem and optimizes towards different fitness functions. WCRT optimization, for instance, involves interference I, WCET W, schedulability S, mapping M, and details of the following.

5. E = Execution time bounds (WCET) are assumed to be given with the AMALTHEA model.

6. γ = CRPDs are not considered. Foundations of CRPD (CPMD), i.e., the timing

costs of resuming a task execution after a preemption accompanied by cache affinity loss, have been provided in [141]. Adding CRPD to the WCRT analysis would manifest in assessing the number and size of accessed labels and worst-case preemption numbers per task.

7. Mi = Migration delays are accounted via Inter Process Trigger (IPT) entities for offloading tasks to GPUs. Consequently, migration delays are not considered in the classical sense, since task migration across cores is not allowed in PFPMPS (only applicable to global scheduling). Instead, CPU-GPU migration is covered by Copy Engine (CE) operations to copy data from a triggering tasks, which calls the IPT, to the GPU task and vise versa (indicated by a lower case x in Table 2.2).

8. ID = Interference delay values are considered via local and global blocking delays during copy engine operations, events, and synchronous or asynchronous GPU task offloading.

9. Pt = H,P: Platform properties of hardware are considered in terms of bus arbitration and heterogeneous PUs (H), which may cause different task execution times without necessarily linear relations. Local and global memory accesses across predictable platforms (P) with memory contention, peripheral affinities, and acceleration features are covered, too. Many-core platforms (M) with advanced isolation mechanisms of Network on Chip (NoC)s or Common-Off-The-Shelf platforms (C) are not in the focus of this thesis.

10. #C = The number of PUs is N (cf. Chapter 5). Investigations of Chapter 7 addresses the task mapping problem towards up to 64 cores, which is yet not an upper bound.

11. #L = Number of memory levels is N here, since the AMALTHEA model provides arbitrary memory level hierarchies through `structure` and `connection` entities (cf. UML class diagram of Figure 3.4: [1..*] self composition of `HwStructure` + [1..*] `HwConnection` composition). In fact, `port` connections can span over various levels such that `access latency` (delay constants) values must be summed up and even investigated throughout multiple paths if there are multiple ways to access some memory throughout connection handlers and ports.

12. LM = Local memory delays are considered regarding access latencies based on the local memory type, which can be of type cache (C), scratchpad (S), or banked memory (BM).

13. GM = Global memory is, in the same way, accounted as local memory and is further subject to contention delays.

14. IC = Interconnect interference can be caused by buses (B) or NoCs. Such influence is outlined in Section 5.7.

15. Pr = Precedence graph properties play an important role for Runnable Sequencing Constraints (RSCs), causal relationships, and form the basic requirements for TDRR described in Section 5.8.1. In addition to runnable sequencing, task precedence constraints are covered in Section 4.2, respectively edges e_φ in the edge set $\mathcal{E}$.

16. Ph - Phased execution is addressed by the implicit communication paradigm.

17. Is - Software isolation is covered by considering the LET paradigm as well as separation constraints.

Maiza et al. assign 119 references to the above properties and features in [140], none of which covering the broad characteristics of Table 2.2 nor the AMALTHEA focus.

2.6.1 WCET Analysis

WCET analysis has been investigated especially in [142] and [113] for timing verification and optimization. Commercial tools exist to solely cover WCET analysis like *AbsInt aiT WCET*[31]. The influence of shared caches, interconnection structures, buses, inter PU dependencies, and modes are analyzed and a WCET optimization Evolutionary Algorithm (EA) is presented to reduce WCET by improved resource scheduling in [113]. The latter has a similar idea compared with TDRR presented in this thesis (cf. Section 5.8.1). The mechanism is called "WCET-driven Multi-Core Instruction Scheduling" (*WMIS*), whereas instructions can be mapped to runnables from TDRR. Both mechanisms are compile-time optimizations. The primary difference is that TDRR decides the runnable order based on calculated schedules that address task release delta values and runnable dependency graphs, and *WMIS* considers TDMA bus scheduling for tasks running on machines separated by a TDMA bus. TDRR can be extended for holistic schedule tables also to cover bus arbitration. For multi-processor machines, shared memory accesses are assumed not to require TDMA bus connections and hence achieve fewer overheads. For distributed ECUs connected through buses, TDRR can potentially adapt to network interfaces by considering transmission delays exemplarily shown along with the CAN bus in Section 5.7. In general, WCETs are considered as static given values in this thesis.

2.6.2 FPPS WCRT Analysis

WCRT investigation serves the purpose of guaranteeing schedulability, i.e., the assurance that no task deadlines are ever violated for a given system of scheduler, task sets, and WCETs. Formal WCRT analysis based on FPPS has history. Liu and Layland provide the fundamentals in [53] by proving that the critical instant, i.e., the situation that all tasks arrive at the same time, forms the worst-case situation from which WCRTs can be derived and by proving the RMS / Fixed Priority Scheduling (FPS) utilization bound of $U = n(2^{1/n} - 1)$ with n denotes the number of tasks. This work is extended by Lehoczky in [38] for arbitrary deadlines by introducing the *level-i* busy period, which is also known as the windowing technique. Lehoczky shows in [38] that a window (level-i busy period) has to be investigated from the critical instant to guarantee to find the WCRTs of tasks. The technique is re-evaluated in [143] together with more recent related work. To determine the length of such window, a recursive function is needed that uses the iterative fixed point lookup, i.e., the function stops when the result of two consecutive calculations do not change. Tindell and Clark extend Lehoczky's work in [136] for a holistic RTA. A task's response time is subject to blocking, which is defined by a factor based on the PCP analyzed in [144] and a set of tasks that have a higher priority than the task under investigation. The consideration of communication through buses accompanies holistic RTA. Tindell et al. therefore analyze response times over a generic TDMA bus (mutually exclusive to PUs) based on fixed message priorities and FIFO message queuing.

[31]https://www.absint.com/ait/index.htm, visited 11.2020

Since then, the real-time research community uses these basics for several further analyses, of which the work by Davis et al. [145] should be pointed out due to the focus on holistic (network-wide) non-preemptive scheduling. Their work includes the estimation of CAN transmission and response times based on the message identifier length, the transmission time for a single bit, and the data bytes to be transmitted. Given that CAN messages are non-preemptive, the maximal blocking time is defined by the longest transmission time of all lower priority messages. Finally, the work uses jitter parameters imposed to a message at the time it is released, the blocking delay, and the busy period length to calculate the total worst-case CAN message delay. A detailed presentation of the formulas, how they are applied to AMALTHEA models, and what additions are incorporated for this thesis are outlined in Section 5.7.2.

Partitioned fixed-priority non-preemptive scheduling is further investigated in [146], which uses sporadic Directed Acyclic Graph (DAG) tasks (RTA for DAG tasks covered in [147]) and the self-suspending scheme to overcome non-preemption issues. A task suspends itself when a precedence constraint is not satisfied. Self-suspension-based scheduling has gained lots of research interest [29] in the past, but similar to other scheduling approaches such as GEDF, it is out of this thesis' scope since it has not been applied to AUTOSAR and the automotive domain.

2.7 Design Space Exploration of Intractable Optimization Problems

As mentioned in Section 1.2.2, this thesis primarily deals with solving the **runnable partitioning**, **memory mapping**, and **task mapping** challenges with respect to optimizing, e.g., processor utilization and response times, via applying polynomial-time approximation methods in form of DSE meta-heuristics. Apart from processor utilization and response time optimization, (a) platform minimization, (b) priority assignment, and (c) policy selection belong to typical optimization goals. For the former (a), based on a task and PU set, a task mapping, a scheduling policy, and a locking protocol, the goal is to find the minimal amount or frequency of PUs being used. This goal also often correlates with energy minimization. Concerning policy selection (c), based on a task and PU set, a task mapping, and a scheduling policy, the goal is to identify a locking protocol that keeps the system schedulable while potentially optimizing other criteria. Variations and combinations of the above and similar problems have been investigated in [148] and addressing (a)–(c) are potential optimization goals, which could be tackled in future work beyond this thesis.

In general, DSE approaches can be categorized into meta-heuristics, hybrid, and exact techniques. The following outlines some of the basic properties of such DSE methods and argues, which and why some have been applied to the partitioning and mapping problems.

2.7.1 Local Search

When dealing with optimization problems, a local search algorithm or heuristic starts with finding an arbitrary solution and then iteratively moves to a neighbor solution. Local search is often chosen due to its simplicity. However, implementation often follows a greedy scheme, i.e., choices are made regarding optimal decisions within a specific *stage*, which often is not the optimal choice in a global sense. Moreover, local search often scales bad, since iteration over direct *neighbors* can be very costly and time-consuming. Hence, (greedy-based) local search often misses global optima and valuable parts of the solution space. Furthermore, local search often has a dedicated model or application to work with, and its applicability to different problems is minimal. Consequently, other more appropriate DSE paradigms are chosen to overcome the limitations of local search algorithms to solve partitioning and mapping problems.

2.7.2 Simulated Annealing

The **SA** paradigm is probably the easiest way to overcome local optima trap issues of local search by applying stochastic value changes instead of neighbor investigations only. As a meta-heuristic to approximate a globally optimal solution (not exact as ILP), SA initially finds an arbitrary solution and then compares it with one or more other solution(s). Inspired by annealing in metallurgy, SA decides probabilistically, which solution is more worthy and chosen as the comparator for the next iteration. The probability of this decision process is defined by programmed *temperature* and *energy* attributes. The temperature changes in each iteration via some function and usually meets zero at the end of the search process. As a consequence, worse solutions are also taken into account to overcome local optima. The energy difference, which is the subtraction of the solutions' energy values, denotes their fitness, i.e., quality.

To tweak a SA algorithm, the initial temperature, probability function (annealing schedule), and compared solution set can usually be configured. By the time the temperature meets zero, the selected solution is *good enough*, i.e., it may not be the exact optimum but at least within a close range to it. SA is easy to implement, but identifying useful configuration parameters is often crucial for finding good results in an appropriate amount of time. For this thesis, the SA implementation along with measurements in Chapter 7 were outperformed by the GA approaches and hence SA is excluded from corresponding measurements.

2.7.3 (Mixed-) Integer Linear Programming

ILP and MILP are probably the most common techniques for DSE in terms of software parallelization for heterogeneous MultiProcessor System-on-Chips (MPSoCs) as stated in [142]. They are exact techniques to be applied to optimization problems, which can be described by integer variables. Therefore, a linear objective function is defined that can be optimized towards maximal or minimal values and depends on several constraints, each described as linear functions, too. MILP allows some variables to be of non-integer natures compared with ILP, which only allows integer values. Since there is no polynomial-time algorithm to solve MILPs, relaxation is used for solving MILPs, which results in a lower bound on the optimal solution. These methods are especially famous due to their mathematical ease and application to problems that (a) are subject to a lower number of constraints or (b) require only binary decisions. MILP has scalability issues for large-scale problems as stated in, e.g., [149, 150] but still finds application to recent resource-oriented partitioning or scheduling problems such as [60, 151, 152], and many more.

2.7.4 Genetic Algorithms

GAs are a particular type of EAs and have similarities to SA algorithms, due to both approaches using stochastic search methods during the search space investigation [153].

Though, instead of three parameters to configure a SA process, a GA provides many more configurations and a more sophisticated search space investigation. The GA meta-heuristic is inspired by evolutionary theory and consists of an initial population of phenotypes, genotypes, chromosomes, and genes as well as mutation and crossover operations. From the initial **population** (random solution set), better generations (solutions) are iteratively created by applying mutation and crossover operations to chromosomes (properties) of a selected solution set. Consequently, a new offspring is created and added to the population, and less-fit individuals are dropped. A population's individual (a single solution) is defined by a **fitness** value that is usually taken as the optimization criteria. A **gene** encodes a parameter via its *allele*, which is part of a solution, e.g., a task to PU mapping. A **chromosome** consists of at least one gene. If multiple genes construct a chromosome, they must be of the same type and domain. If genes of different types or different domains are required, multiple chromosomes must be implemented. A **genotype** then consists of a chromosome set, whereas a phenotype is defined by precisely one genotype. Finally, a population if formed by a sequence of **phenotypes** (individuals), i.e., one or more chromosome sets. This thesis makes use of the Jenetics library [42] to encode the above-outlined model. Jenetics then provides various interfaces to define selection and altering operations, i.e., selecting individuals for recombination to produce offsprings and creating genetic diversity by applying mutation, recombination, or crossover operations. A **mutation** defines the probability of a gene (resp. genotype and chromosome) to be altered, i.e., changed. **Recombination**, in contrast to mutation, swaps single or multiple (multi-point crossover) genes between two individuals. In essence, only some of the *fittest* individuals of a generation are stochastically chosen for the mutation process that forms new generations with modified genes. A GA terminates on different properties, which can be the number of generations, a predefined fitness value, the number of generations to which the fitness values do not improve (i.e., steady fitness), resolution time, and others.

Along with publication [11], a GA is combined with SA towards a Hybrid Genetic Algorithm (HGA) by using SA as an evolution stream to undo crossover operations of the GA, if the SA criteria are not fulfilled. This process can potentially improve the GA resolution time and hence convergence speed. Cordes has shown in [142, p. 107] that ILP solvers are not preferred for multi-objective DSE, which though can be tackled by GAs and hence are incorporated in this thesis.

2.7.5 Constraint Programming

In contrast to mathematical programming such as MILP, quadratic programming, genetic programming, and others, constraint programming not only covers most of the mathematical operations but also comes with powerful paradigms to further constrain combinatorial problem spaces and consequently increase exploration efficiency. CP belongs to the hybrid techniques and can thus be used to prove an optimal solution or find solutions close to the optimum. Perron stated in [154] that the usage of CP is beyond MILP for optimizing applications in industrial operations research projects such that CP becomes a viable and promising approach to be in focus for the task to PU mapping outlined in Chapter 5. CP has also increasingly found application in modern design space exploration processes due to its more natural and flexible modeling capabilities [154]. CP can be used as a paradigm to either satisfy constraints of a CSP or to optimize variables for Constraint Optimization Problems (COPs). Therefore, CP requires to define a model that consists of (a) variables, (b) domains, (c) constraints, and optionally (d) optimization goals.

Example 2.1: CP Basics

Given is (a) an integer i that is (b) allowed to take values from 1 to 10, which is hence its initial domain, i.e. $i \in [1, 10] \cap \mathbb{N}$. The first constraint is given by allowing (c) only even numbers, i.e., $i\%2 = 0$. With the domain and the first constraint, the example can already be used for a CSP, which would result in solutions $i = \{2, 4, 6, 8, 10\}$. With an optimization goal (d), such as $\min(i)$, a COP can be set, which results in only one optimal solution, i.e. $i = 2$.

A valid solution to the CSP only exists if all its constraints are satisfied. Constraints can be of logical, arithmetical, set, graph, or real-value nature, for instance. The advantage of CP over MILP is (i) the natural programming of relations between variables (constraints), (ii) the vast flexibility in modeling beyond the scope of integers, and (iii) the search space investigation methodology. For the latter, one can pick between, adjust, and modify various **filtering** algorithms to prune incompatible values for valid solutions from a variable's domain. Then, all constraints related to such domain change are revised because the domain reduction can further affect other variables and deduce their domains, which is called **propagation**. In other words, when the search process runs and inconsistent value combinations regarding the defined constraints and assigned values or domain reduction are identified, these findings are propagated so that infeasible solutions that violate any constraints are removed from the solution space (i.e., backtracking). Custom search heuristics and **backtracking** can be systematically applied to the model to remove inconsistencies that could have been discovered multiple times in regular search procedures. Incremental assignments combined with backtracking search or complete assignments combined with the stochastic search are examples for such search heuristics. In general, CP also allows proving optimal solutions or unsatisfiability of a CSP [155].

In Section 5, a CP model, as well as its application to AMALTHEA is outlined, to consider a broad set of automotive software constraints such as pairings, separations, activations, sequences, ASIL properties and more when distributing software across tasks and PUs of vehicles. Even though additional timing constraints have been formulated in [115], only those applied to AMALTHEA and AUTOSAR are addressed in this thesis. Results are compared with existing DSE approaches such as GAs, dedicated heuristics, and ILP-based solutions.

2.7.6 Other DSE and Mapping Heuristics

A valuable, yet not exhaustive, study is provided by Braun et al. in [156] that investigates Opportunistic Load Balancing (OLB), Minimum Completion Time (MCT), Min-min, Max-min, duplex, GA, SA, Genetic Simulated Annealing (GSA), Tabu search, and A* heuristics. More recently, Wang et al. published in [95] a comparison of different DSE methodologies in terms of the mapping problem across AUTOSAR applications. Both publications [95] and [156] clearly identify GA to consistently give the best results. More DSE methods have been used along with the mapping problem such as Ant Colony Optimization (ACO) [157, 158], Chicken Swarm Optimization (CSO) [159], Particle Swarm

Optimization (PSO) [160], Simulated Annealing and Tabu Search (SAT) [161], Integer Non-Linear Programming (INLP) [162], or Bat Algorithm (BA) [163] (a recent analysis is given in [164]). Nevertheless, those approaches are out of this thesis' scope due to (i) ILP, GA, and CSP satisfy the requirements to cope with the problems addressed in this thesis and (ii) the scarce availability of the former meta-heuristics as open-source libraries, respectively their implementation overheads required if being implemented from scratch.

2.8 Sharing Resources and Blocking Times

Blocking analysis is required for a correct RTA, especially in the multi-processor domain, since shared resources must often be mutually exclusive and hence require appropriate mechanisms to ensure priority inversion bounding as well as deadlock or race condition prevention. Blocking from shared resources is found the major influence on response times for PFPMPS, and a corresponding survey was published by Abel et al. in [78]. For instance, accesses to shared I/O peripherals, global memory, or special purpose units (e.g., FPU) often require mutually exclusive access for deterministic and causally correct system execution. Locks, mutex algorithms, and various protocols support mutual exclusion by managing critical sections, i.e., the program code accessing the shared resource, but also introduce processing overheads that affect the system's responsiveness, i.e., tasks' response times.

The Ph.D. research by Negrean [77] covers a good analysis of blocking times under typical scheduling policies and resource sharing protocols among Autosar. Lock-free and wait-free approaches that use buffers, queues, or heaps, which can be alternatively used for synchronization, are discussed in [165] and are neither reflected in Autosar nor investigated here.

In the situation of a lock being hold already by the time a task wants to lock it, the task can either **spin** over acquiring the lock until it becomes available, which is known as *busy waiting*, or **suspend** under the assumption that the scheduler resumes it as soon as the blocking task holding the lock releases it. In Autosar, the former is employed by *spinlocks* for data shared globally between PUs, and the latter is used along with the OSEK Priority Ceiling Protocol (OPCP) for data shared among tasks locally on a single PU [47, 166], which are described in the following sections.

2.8.1 Sharing Local Resources in Autosar: PCP vs OPCP

The principle of the PCP is similar to the Priority Inheritance Protocol (PIP), i.e., when a shared resource is accessed but already locked, the locking tasks inherits the priority of the accessing task. Task priorities are assumed to be fixed and known, such that resource ceilings can be computed for all resources, which is the highest priority of tasks accessing the resource as shown in Eq. 2.2 with $\pi_{lc}(CS^v)$ denoting the local priority ceiling for the critical section CS^v. In line with related work, a system's highest priority is π_1 and π_2 is one priority level lower than π_1 so that $\pi_1 > \pi_2$.

$$\pi_{lc}(CS^v) = \max_{j:\tau_j \text{ locally accessing } CS^v} \pi_j \tag{2.2}$$

Priority changes induced by the PCP do not influence the fixed priority scheduler process. The *system priority ceiling* is dynamic during the system execution and defined by the maximal allocated resource ceiling priority at a given point in time. It is used to allow higher priority tasks to preempt lower priority tasks that hold a lock until the higher priority task wants to lock the same resource. At that time, the lower priority task inherits the system priority ceiling that was *low* at the time the lower priority task was running in the Critical Section (CS) and then raised to *high* due to the higher priority task trying to lock the CS. In addition to PIP, a task under the PCP only enters a critical section when it is free, and there is no risk of chained blocking or deadlock. As a consequence, PCP is deadlock-free, and blocking can be bounded to at most one critical section length, which is highly beneficial over, e.g., the PIP or the Non Preemption Protocol (NPP). PCP has been introduced and analyzed in [144] according to direct, push-through, and ceiling blocking times.

The OPCP corresponds to the Highest Locker Protocol (HLP) and features the same benefits as PCP and further eases implementation due to no priority inheritance and hence no system ceiling priority tracking. The major difference of OPCP and PCP is that PCP does not suffer from long inheritance related inversions as OPCP [167], which is shown in Figure 2.6. Based on the average response time across tasks, Figure 2.6 a) provides an

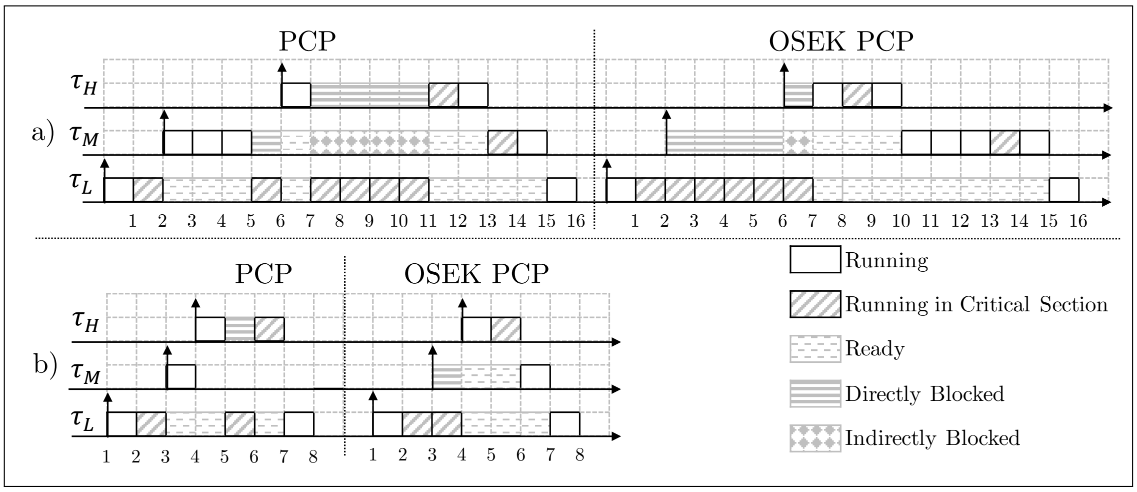


Figure 2.6: PCP vs OPCP in two situations :
a) OPCP and b) PCP feature better response times

example of OPCP performing better than PCP primarily driven by the high priority task's response time, i.e. $R_{\tau_H}^{OPCP} = 4 < 7 = R_{\tau_H}^{PCP}$. Despite that, a different situation shown in Figure 2.6 b) can result in PCP performing better than OPCP, which is mostly caused by the medium priority task's response time $R_{\tau_M}^{PCP} = 1 < 4 = R_{\tau_M}^{OPCP}$. Task arrival times are the same for PCP and OPCP in the respective situations provided in Figure 2.6.

2.8.2 Sharing Global Resources in AUTOSAR

The PCP is migrated to distributed PU systems by Rajkumar et al. for partitioned suspension-based scheduling along with the Distributed Priority Ceiling Protocol (DPCP) [168] and adapted towards shared memory along with the Multiprocessor Priority Ceiling Protocol (MPCP) in [169]. MPCP (a) minimizes remote blocking of global resources shared across PUs and (b) bounds priority inversion via boosting a locking task's priority to a higher priority than every other task across the entire system [170].

Instead of the suspension-based protocols like MPCP or DPCP, Stack Resource Protocol (SRP) (known to be optimal for single PU systems) and Multiprocessor Stack Resource Policy (MSRP) [171] were proposed as spin-based protocols for partitioned scheduling. Furthermore, the Flexible Multiprocessor Locking Protocol (FMLP) is presented by Block et al. in [172] to cover both global and partitioned scheduling. FMLP does not restrict CS nesting or having periodic tasks only. A recent systematic review on multi-processor real-time locking protocols is given in [148] and analyzes the problems and benefits across existing research beyond spinlocks and OPCP used in AUTOSAR. In [148] also ensues prior work by Wieder et al. [35], which already outlined some missing definitions of the AUTOSAR spinlock type.

Although the research community presented the protocols mentioned above for multi-processor environments, the AUTOSAR consortium decided to stick to simple spinlocks to protect global resources without defining FIFO-, priority-ordered, or unordered spinlocks, under the opinion of development ease and flexibility [47].

Section 5.4.1 provides an analysis of local direct blocking, local push-through blocking, ceiling blocking, and global blocking, whereas the latter assumes FIFO queues, which makes global blocking pessimistic due to the imposed blocking delay being proportional to the amount of PUs.

2.9 Summary and Motivation

(I) Automotive systems require a holistic, model-based, and formally verified analysis for exploring parallelism and concurrency along with heterogeneous, distributed (networked), mixed-critical, embedded, concurrent, and real-time domains.

(II) The vast amount of constraints and requirements impose a huge complexity for such analyses.

(III) Partitioning and task mapping go hand in hand with WCET and WCRT minimization and hence require sophisticated algorithms and meta-heuristics to solve the problem of minimizing timely interference, while meeting all timing constraints as well as modern high-performance demands of development activities towards highly assisted and even automated driving applications.

(IV) Existing DSE and timing verification methodologies must be advanced to address modern many-fold heterogeneous CPU-GPU architectures through considering synchronous and asynchronous task to GPU offloading, as well as new blocking, contention, queuing, CE, task chain latency effects under different communication paradigms such es explicit, implicit, and LET.

(V) Open-source models must provide the possibility to (a) exchange automotive system details without real code, (b) cover all typical automotive system details, (c) be industrially used along with a dedicated open-source Integrated Development Environment (IDE) across various Tier suppliers and OEMs, (d) be compliant to AUTOSAR, and (e) provide Application Programming Interfaces (APIs) to simulate, analyze, visualize, and adapt models or even extend the meta-model.

(VI) No work exists that meets the above challenges I–V.

The technologies and contributions of this thesis fill the gap of VI by addressing the challenges I–IV via (I) covering the cross-domain requirements and constraints, (II) using appropriate heuristics and meta-heuristics to cope with the problems' complexity, and (III/IV) providing tooling to cover and formally verify constraints for optimized solutions along with advanced (new) concepts for (i) WCET and WCRT minimization, (ii) task chain latency analysis, as well as (iii) CPU-GPU task offloading and execution.

Just one model is identified of covering requirements of item V, namely AMALTHEA, which is presented in the following Chapter 3.

3 System Model

This chapter outlines and defines not only the semantics of various AMALTHEA entities (sections 3.1–3.1.7) but also their mathematical notation in Section 3.2, used for analyses throughout this thesis. Notations are based on the Burns standard notation [61] and presented in Table 3.1. Some minor notations deviate from the Burns notation, e.g., the task set $\mathcal{T}$ instead of τ due to consistency reasons, i.e., having uppercase calligraphic symbols for all sets. Subscripts are mostly used for identification purposes via an index value, and superscripts are used for type identification. For instance, $B_i^{s,+}$ denotes the worst-case ($^+$) global blocking (^{s}) of task τ_i. The notation Table 3.1 is intended to be used as a reference throughout this document.

3.1 AMALTHEA

The AMALTHEA model [45] is chosen as the primary model for this thesis because it is AUTOSAR compliant, entirely open-source, used by the automotive industry, and easily accessible through the APP4MC platform. It consists of several basis models namely **software**, **hardware**, **stimuli**, **components**, **events**, **operating system**, **constraints**, **mapping**, **custom property**[32], common elements, config, measurements, and property constraints (by early 2020, AMALTHEA *v0.9.8*). The AMALTHEA and AUTOSAR models have many entities in common but also particular distinctions, which have been studied in [173]. Essentially, AMALTHEA covers most aspects available in either AUTOSAR or ASAM MDX and further includes various extension points to be able to define nearly any possible automotive system. Since version *0.9*, AMALTHEA even covers data and ECU network modeling, which has been identified as a deficit in older versions according to [173]. In addition to AUTOSAR, AMALTHEA features model entities of OS runtimes and various **OS resources**, **multiple stimuli**, function domains, complex call graphs, generic limits for metrics, data coherency groups, and **affinity constraints**. Bold highlighted model entities are used in this thesis throughout Chapter 4–7, and outlined in the following Sections 3.1.1–3.1.7. Much more information about the model itself and entities not used in this thesis is available online at [45].

Figure 3.1 shows both the partitioning and the mapping approaches integrated into the

[32]This thesis makes only used of bold highlighted AMALTHEA models.

APP4MC platform.

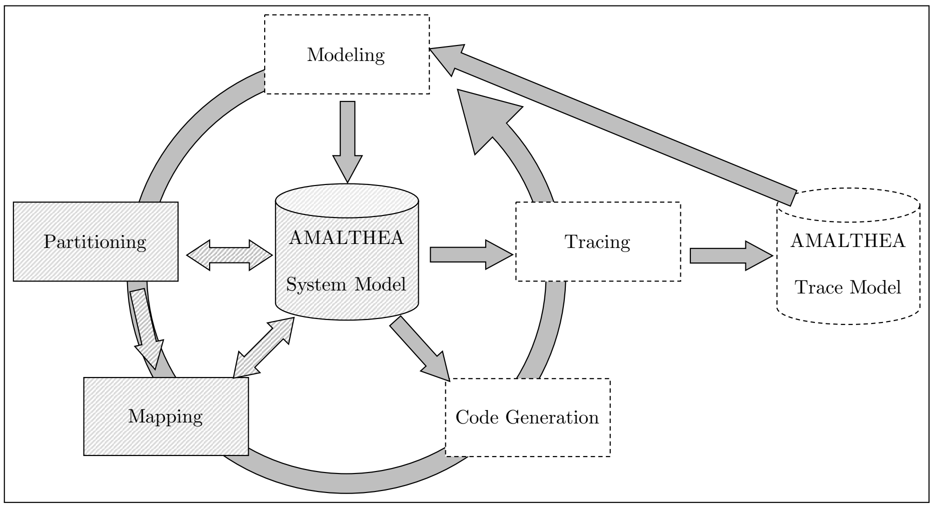


Figure 3.1: AMALTHEA platform: development cycle and features based on [45]

Both partitioning and mapping processes read and extend various AMALTHEA models entities based on the process' input, configuration, and corresponding model analyses. Further topics like *Modeling*, *Code generation* and *Tracing* provide necessary features for comprehensive system engineering, but are not in scope here.

The AMALTHEA model has been developed by various partners of research projects mentioned in Section 1.3, but primarily by the Robert Bosch GmbH company. By taking part in this development, the author of this thesis also took part in some decisions for the AMALTHEA model evolution itself, primarily with the Eclipse APP4MC Committer role, but the main contributions address the partitioning plugin and RTA tools.

3.1.1 AMALTHEA Software Model

The software model forms the primary input for this thesis' implementation and gives information about runnables, labels, tasks, ticks, dependencies, and more, based on AUTOSAR notations.

Definition 3.1: Runnable

A runnable $r_a \in \mathcal{R}$ is a representation of atomic program code. It consumes instructions c_a, may read and write labels $(\uparrow_a, \downarrow_a) \subseteq \mathcal{L}$, and call OS entities such as semaphores, events, other runnables, inter-process activations, or others along with its activity graph. A runnable's instructions is defined by the sum of ticks or execution needs:

$$c_a = \sum_j c_{a,j} \tag{3.1}$$

Here, $c_{a,j}$ represents the $j-th$ constant value derived from an execution needs or ticks entry within the runnable's activity graph.

The runnable Definition 3.1 assumes that instructions are represented as constant values. Whenever other types are existent in the model, worst case or upper bound parameters are taken as a reference to meet timing verification requirements, i.e., guaranteeing various timely properties even at the worst possible (worst case) situation. Switching from WCET/WCRT to BCET/Best Case Response Time (BCRT) analysis is though just a matter of process configuration. Figure 3.2 shows different software meta model entities as screenshots from [174], with root entities in the first column, task / runnable activity graph items in the second, and tick / execution need items in the third column as of AMALTHEA version 0.9.8. These screenshots do not show the properties for different entities, which are partially referred to textually in the following if necessary. A comprehensive property outline goes beyond the scope of this thesis but can be found at [45].

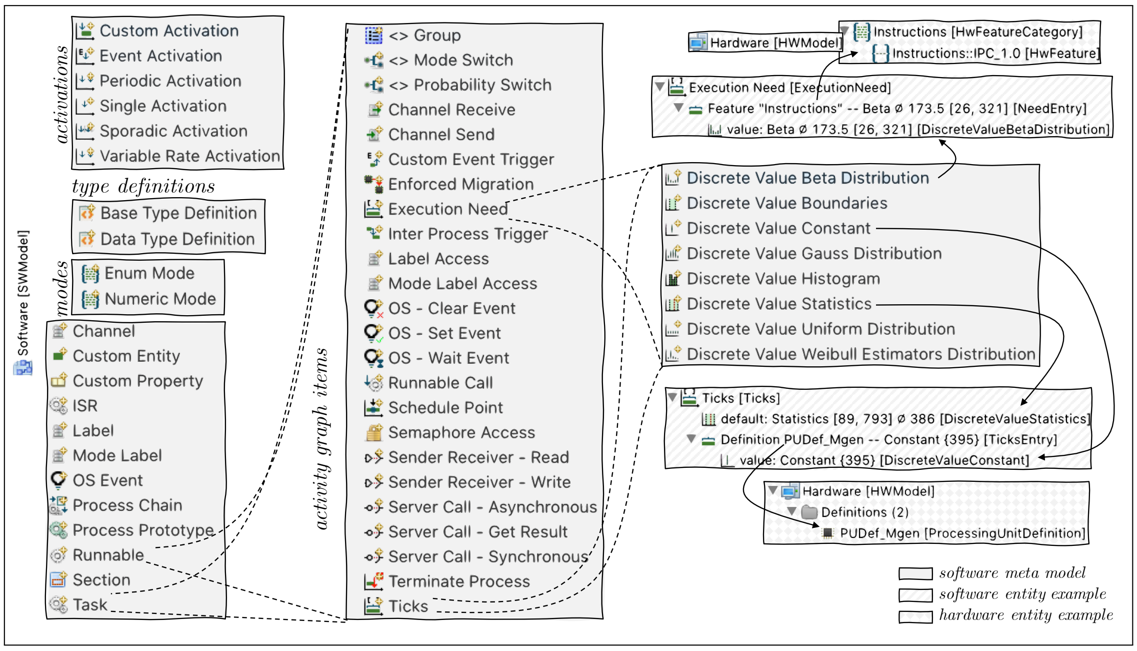


Figure 3.2: AMALTHEA software model v0.9.8 entities excerpt [174]. Dashed lines indicate possible entity containment provided by the AMALTHEA meta-model.

The third column of Figure 3.2 shows some model examples (as indicated by the legend) to provide some insights into how the various instructions and distribution entities are modeled and put in context with features or specific hardware definitions.

A runnable is often derived from a code function that runs sequentially, i.e., it can not be subdivided further. Instructions are represented as `execution needs` for certain features or generic `ticks` in terms of AMALTHEA as shown in Figure 3.2. Both entities can be hardware independent or hardware specific. These values are then required on a PU for being executed and represent, e.g., an arithmetical operation. `Execution needs` or `ticks` are assumed to be known and provided as part of the input models in this thesis.

Definition 3.2: Instructions

Instructions can be represented as beta distribution, boundary, constant, Gauss distribution, histogram, statistic, uniform distribution, or Weibull estimator distribution values. Each representation is further modeled as either (I) execution needs, which can be hardware feature specific, e.g., refer a dedicated instructions per second $\varkappa_x$ parameter, which is again referenced by specific PUs or structures such as an ECU, or (II) ticks, which can be PU definition specific or entirely hardware independent.

To derive the normalized execution time $c^s_{a,x}$ per second for a runnable r_a on a PU P_x, Eq. 3.2 is used for periodicity (T_{r_a}) given in pico seconds.

$$c_{a,x} = \begin{cases} \frac{c_a}{f_x} \cdot \left\lceil \frac{10^{12}}{T_{r_a}} \right\rceil & \text{if } c_a \text{ is ticks constant} \\ \frac{c_a}{f_x \cdot \varkappa_x} \cdot \left\lceil \frac{10^{12}}{T_{r_a}} \right\rceil & \text{if } c_a \text{ is execution need constant} \end{cases} \tag{3.2}$$

A Weibull estimator distribution, which is provided by the Democar model (cf. Section 6.4) for instance, is a continuous probability function consisting of two parameters scale λ and shape k, which define different probability density functions. Just as beta (cf. Figure 6.2) and other distribution parameters, such functions are especially useful when not only verifying WCRTs, but also probabilities, average values and similar through simulation or tracing. As pointed out at the runnable Definition 3.1, instructions are referenced by runnables via a single value c_a, which seems to contradict the instruction Definition 3.2. The former assumption necessary to concentrate on worst-case scenarios for schedulability and safety verification.

Instructions are shown in Figure 3.2's third column along with three examples, namely (i) beta distribution, (ii) constant, and (iii) statistic values. Here, execution needs are used for (i) the beta distribution and (ii) constant values as well as (iii) statistic values make use of ticks. The execution needs of the beta distribution (i) refer to a hardware features category, whereas ticks of the constant (ii) and statistic example (iii) refer to a specific PU definition. Beta distributions are further used for generating models along with Section 6.5 and Figure 6.2. This thesis does not assess timing properties in regard to the distribution or probability of, e.g., response or execution times. Consequently, any reference to such instructions relates to worst, i.e. upper bound, and best, i.e. lower bound values, which are denoted as c^+_a and c^-_a, respectively.

In general, hardware-independent instructions are modeled as `ticks` and a `default` entry. `Extended` entries, which can be modeled in addition to default ticks, must reference a specific hardware definition (`HWDefinition`, see Figure 3.2, third column bottom), such that `extended ticks` are used along with time derivation for a specific set of PUs that belong to the corresponding hardware definition. `Default ticks` can be applied to any PU, which belongs to a hardware definition that is not included in any extended ticks of the same processing entity. Alternatively, `execution needs` can be modeled, which require to reference a hardware feature category (see Figure 3.2, third column top: arrows from the execution need example entity to the hardware feature category entity, outline the coherency reference). As outlined in the AMALTHEA documentation [45, Execution Time

Section], execution times can always be translated into ticks and then ignored for further time evaluation. For this translation, a mapping dependent recipe can be used and allows nearly arbitrary computation prescripts, specifically designed for flexibility. Since hardware definitions can reference one or more feature categories, `execution needs` define a lower instruction modeling level and provide decimal coherences (in contrast to `ticks`, which are always distinct) for features that can be included for multiple hardware definitions (see also class diagram 3.4).

Any runnable instructions can be included multiple times within a runnable's activity graph. In this thesis, it is assumed that c_a is constructed by the sum of all runnable activity graph items (cf. Definition 3.1). By accounting for `extended` and `feature`-specific instruction entities, hardware-dependent heterogeneous execution times are considered. Such instructions are denoted as $c_{a,x}$ with x being the PU index the task, which accommodates the corresponding runnable, is mapped to ($M^P_{\tau_i} = x; M^{\tau}_{r_a} = i \Leftrightarrow M^P_{r_a} = x$, cf. notations of Table 3.1). Further properties of runnables used in this thesis are ASIL levels, activation, and tag references for the partitioning, and the label size property for the label to memory mapping. Execution conditions and other runnable parameters are not used in this thesis. Runnables are grouped into tasks by the partitioning process with a specific ordering, as outlined in Chapter 4.

The following Definition 3.3 outlines the terminology of labels, i.e. notation and semantics, which are derived from the AUTOSAR standard, e.g. from `GetResource()` and `ReleaseResource()` calls.

Definition 3.3: Label

Labels $\mathcal{L} = \{l_1, ...l_q\}$ represent data such as variables, parameters, data structures, or similar. They are accessed by processing entities such as runnables or tasks and consist of a size parameter ls_v.

Data must be allocated to memory, which is provided in the mapping model (Section 3.1.5). Due to various PUs having different access latency values to the varying memories and types, the memory mapping influences task execution and response times. For example, Intel's Nehalem CPU architecture accesses L1-cache memory by four cycles, L2 cache with ten cycles, whereas IBM's power-6 architecture requires four cycles to access L1 cache as well, but 24 cycles to access L2 cache, and in general, more than 100 cycles have been found for modern multi-PU systems and significantly higher cycles, e.g., 240 cycles for accessing Dynamic Random Access Memory (DRAM) in [175]. In practice, the largest observed label sizes from industrial models (see Chapter 6) are in the range of several megabytes. Labels are read and or written as part of runnables' or tasks' activity graphs. For this thesis, label access activities are assumed to be modeled on runnable level only.

Definition 3.4: Task

A task $\tau_i \in \mathcal{T}$ is a tuple of $\{T_i, \mathcal{R}_i, C_i, \pi_i, D_i, \mathcal{L}_i, ag_i, pt_i\}$, respectively a period, an ordered aggregation of runnables referring to the same activation T_i, instructions[a]*, a priority, a deadline, a set of accessed labels (memory demand), an activity graph, as well as a preemption type along with further coherences of the* Amalthea *meta-model. A task's instructions are defined by the sum of its runnables' instructions:*

$$C_i = \sum_a c_a \text{ with } r_a \in \mathcal{R}_i \ : \ M^{\tau}_{r_a} = i \tag{3.3}$$

The preemption type of a task is either preemptive, cooperative (is either preempted immediately from higher priority preemptive tasks or at runnable boundaries from higher priority cooperative tasks), or non-preemptive.

[a] A set of instructions if hardware-specific instructions are available

Since multiple tasks can exist for the same activation, a runnable to task mapping is not distinct and rather forms flexibility in the system design process to optimize various goals such as responsiveness, which manifests in minimizing task response times. For this thesis, a single runnable to task mapping is assumed, such that a runnable must not be assigned to more or less than one task. Tasks can potentially reference multiple activations, but this thesis assumes a single activation pattern for each task. Within an activity graph ag_i, a task can interact with various model entities such as runnables, channels, events, labels, modes, semaphores, server calls, and more or consume instructions while being scheduled on a PU through activity graph items ($ag_i = \{agi_{i,1}, ...\}$). This thesis' most-used entities are runnable calls and inter-process triggers. The former defines $\mathcal{R}_i$, i.e., the runnables called by task τ_i in order of the position in the activity graph ag_i. The latter (inter-process triggers) are used to trigger other processes (e.g., tasks or runnables) that are mapped to, e.g., other PUs. A task's WCET $C^+_{i,x}$ for a PU P_x is either assumed to be known a priori or can be derived from its static instructions C_i (cf. Section 2.6.1), the frequency and optionally the instructions per second of a PU. Commercial tools to retrieve WCET bounds exist such as aiT WCET Analyzer from AbsInt GmbH [176], RapiTime by Rapita Systems Ltd. [177], or Bound-T by Tidorum Ltd. [178], whereas research tools from academia such as Chronos [179], TuBound [180], OTAP [181], CalcWCET167 [182], OTAWA [183], or SWEET [184] have also found their application to real-world scenarios.

A DAG model composed of vertices and edges is not directly given in Amalthea, but the partitioning process (cf. Chapter 4) makes use of the JgraphT library[33] to construct a DAG from tasks $\mathcal{T}$ and edges $\mathcal{E}$ derived from accesses to labels. The derivation of edges based on label accesses is part of the partitioning Chapter 4. The DAG is then used to analyze precedence constraints (dependencies), paths, and structures to group runnables into tasks to minimize communication overheads, balancing load, and maximizing parallelization potential.

Two more Amalthea software model entities should be mentioned, namely `activations` and `process prototypes`. Both are preliminary elements for stimuli and tasks,

[33] JgraphT library https://jgrapht.org, visited 11.2020

respectively. Such preliminaries are dedicated to early system design phases to keep system complexity low and provide easily accessible items for DSEs, such as partitioning. As soon as results of this early DSEs are ready to be used for, e.g., the task mapping process, activation, and Process Prototype (PP) entities are migrated to stimuli and tasks by a one-to-one transformation. After that, tasks and stimuli can be enhanced by further information obtained by tracing, simulation, or requirements analyses such as extended ticks, advanced offset, arrival curves, ASIL levels, and many more.

3.1.2 Hardware Model

The hardware model is designed to be close to the SHIM and AUTOSAR standards but still provides more flexibility to consider typical automotive hardware, such as ECUs or networks consisting of domain-related bus systems. The hardware model is required by the memory (cf. Section 5.7) and task mapping (cf. Chapter 5) processes. When using APP4MC, entities of Figure 3.3 can be used to model hardware information.

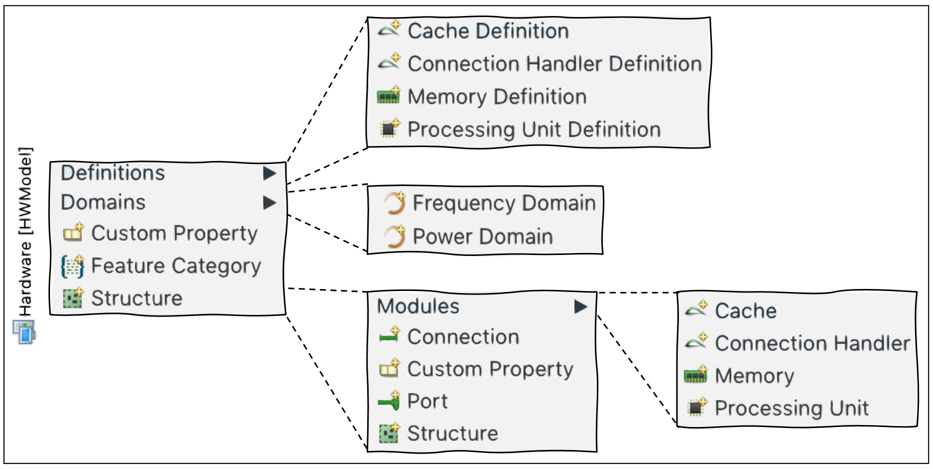


Figure 3.3: AMALTHEA hardware model v0.9.8 entities excerpt [174]

The most relevant information is given in PU, frequency, and memory definitions. The latter is defined by data rate, data size, and access latency values, which can be of various instruction types mentioned in Definition 3.2. Latency values are used by blocking analysis, the memory mapping process, and timing verification. The various entities of Figure 3.3 are further put into context in the meta-model class diagram of Figure 3.4, which also provides the types of features, structures, ports, and port interfaces. The most flexibility is given by the recursive [0...*] relation of `HwStructures`, which can be of various types, as shown in the right middle part of class diagram 3.4. Type definitions of Figure 3.4 provide typical bus interfaces and hardware levels of the automotive domain.

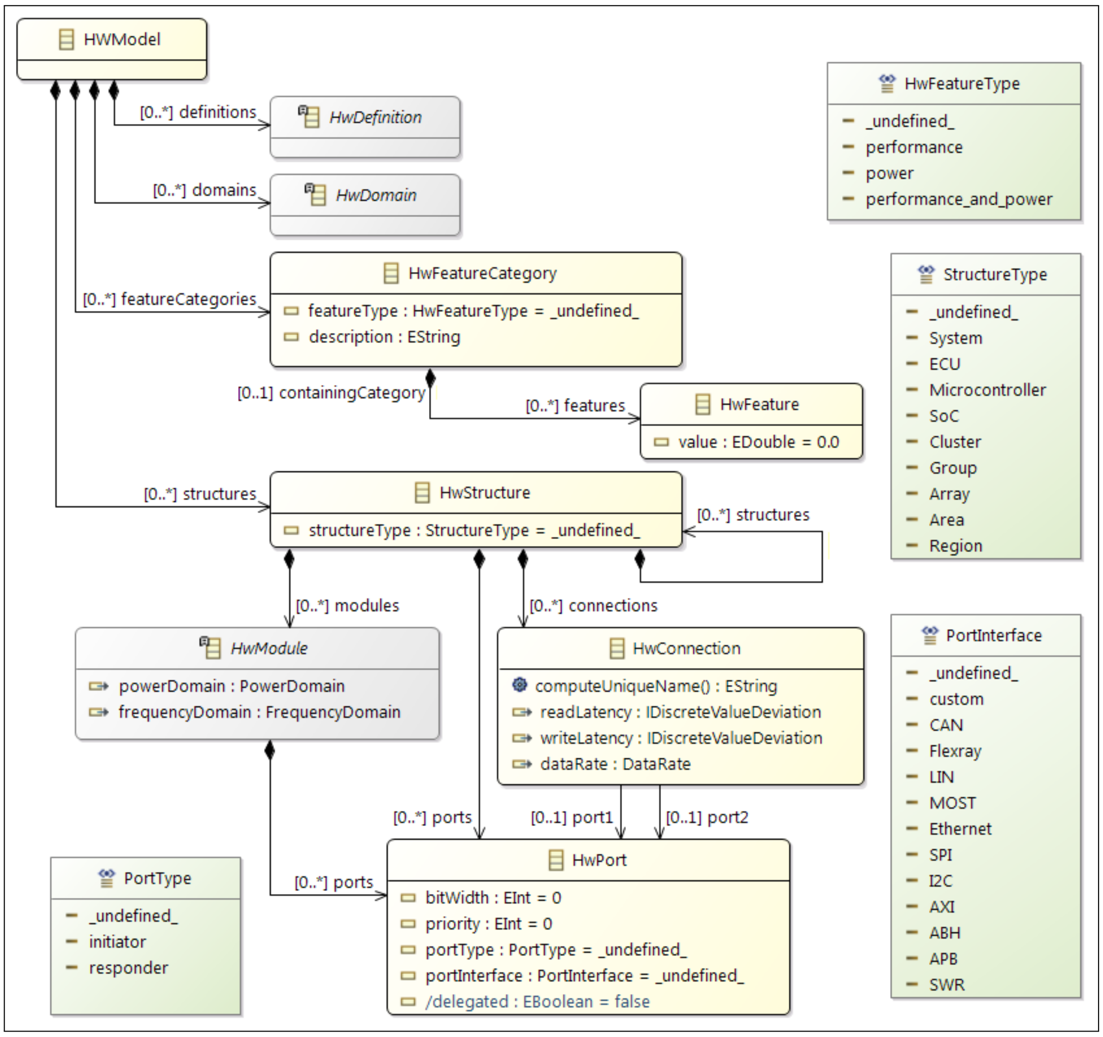


Figure 3.4: AMALTHEA hardware model v0.9.8 main class diagram [45]

In line with software model characteristics shown in Figure 3.2, the `HWFeature` class of Figure 3.4 provides the flexibility of, e.g., scaling instructions for various PUs that reference specific `HWFeatures`. This reference is not shown in Figure 3.4 and is part of the PU definition (cf. Figure 3.3, second column) that is required for every PU instance.

`Port interfaces` are required to model network communication throughout an arbitrary hierarchy of hardware modules of different structure types. `HWDomains` define the frequencies of PUs in terms of the frequency domain or the voltages for the power domain. Such values are used to derive task execution times or power consumption of hardware under the respective domain. `HwModules` are contained in structures across different structure types and define major hardware entities, which inherit from either a PU, `memory`, `cache`, or `connectionHandler` (cf. Figure 3.3). In this thesis, structure entities are used for deriving access latency values along the shortest paths from one module to another through connections, ports, connection handler, module hierarchies, and entire networks.

Definition 3.5: Processing Unit

A processing unit $P_x \in \mathcal{P}$ references a frequency and power domain, of which the former is used to derive time for instructions via its frequency value f_x in Hz. It can be included on arbitrary levels across hardware modules defined by structure types such as systems, ECUs, micro-controller, System on Chips (SoCs), clusters, groups, arrays, areas, or regions. PUs can further contain ports, caches, and access elements. A PU's capacity is defined in Eq. 3.4 and provides the number of instructions the PU can execute in one second, with $\varkappa_x$ denoting the Instructions Per Cycle (IPC) hardware feature.

$$puc_x = f_x \cdot \varkappa_x \tag{3.4}$$

Given Definitions 3.1–3.5 and Eq. 3.1, WCETs can be calculated as shown in the following Example 3.1.

Example 3.1: Execution Time Derivation

Given is a task τ_i requiring $C_i^+ = 2780$ default upper bound execution needs (instructions) for being executed every $T_i = 100ms$. The execution time calculation for τ_i being mapped to a processor P_x, which runs at a frequency $f_x = 200MHz$ and contains a `HWFeature` ‘`IntructionsPerCycle`‘ with value $\varkappa_x = 1.2$, is given in Equation 3.1.

$$C_{i,x}^s = \frac{2780 \cdot 10^{12}}{200 * 10^6 \cdot 1.2 \cdot 100 \cdot 10^9} = 115.83\mu s \text{ per second}$$

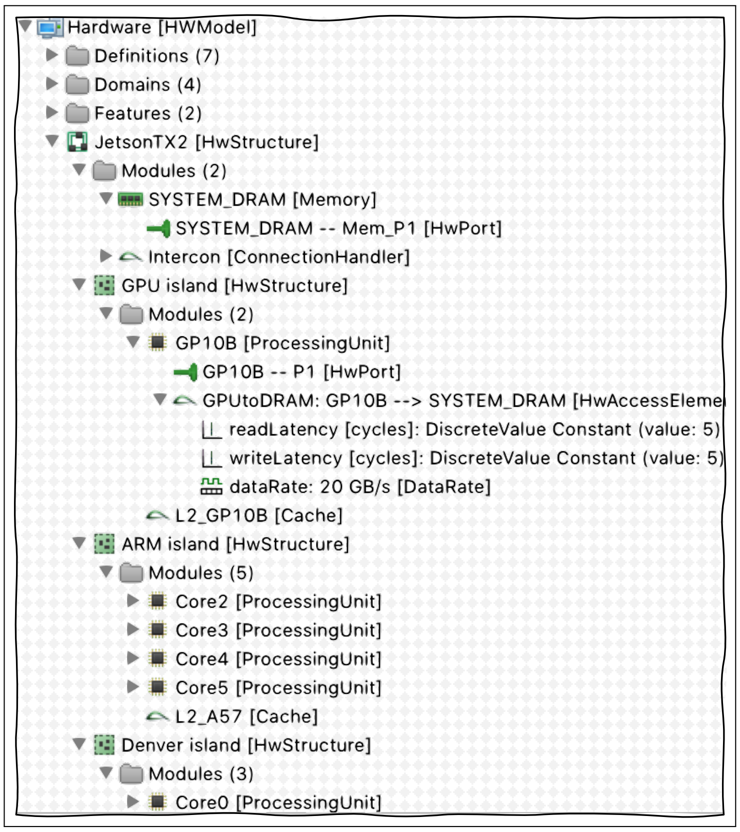
This equation uses normalization towards one second with picosecond scaling. The same task on another processor P_y, which runs at the same frequency but features a $\varkappa_y = 1.0$ value, results in $C_{i,y}^s = 139\mu s$. The same result is calculated for a ticks instruction type with the default value of 2780 that does not refer to a hardware feature category.

Hence, the general WCET calculation, normalized towards one second indicated by superscript ^{s}, is shown in Eq. 3.5 and execution time without normalization is given in Eq. 3.6. With the periodicity given in pico seconds, the latter fraction $\left\lceil \frac{10^{12}}{T_i} \right\rceil$ gives the amount of times the task is executed per second.

$$C_{i,x}^{+,s} = \frac{C_i^+}{f_x \cdot \varkappa_x} \cdot \left\lceil \frac{10^{12}}{T_i} \right\rceil \quad \text{with } T_i \text{ in picoseconds} \tag{3.5}$$

$$C_{i,x}^+ = \frac{C_i^+}{f_x \cdot \varkappa_x} \tag{3.6}$$

An example hardware model is presented in Figure 3.5, which shows some of the Nividia Jetson TX2[12] properties. Another example is later shown in Figure 5.10, which shows a hypothetical ECU network.



```
▼ Hardware [HWModel]
    ▶ Definitions (7)
    ▶ Domains (4)
    ▶ Features (2)
    ▼ JetsonTX2 [HwStructure]
        ▼ Modules (2)
            ▼ SYSTEM_DRAM [Memory]
                SYSTEM_DRAM -- Mem_P1 [HwPort]
            ▶ Intercon [ConnectionHandler]
        ▼ GPU island [HwStructure]
            ▼ Modules (2)
                ▼ GP10B [ProcessingUnit]
                    GP10B -- P1 [HwPort]
                    ▼ GPUtoDRAM: GP10B --> SYSTEM_DRAM [HwAccessEleme
                        readLatency [cycles]: DiscreteValue Constant (value: 5)
                        writeLatency [cycles]: DiscreteValue Constant (value: 5)
                        dataRate: 20 GB/s [DataRate]
                L2_GP10B [Cache]
        ▼ ARM island [HwStructure]
            ▼ Modules (5)
                ▶ Core2 [ProcessingUnit]
                ▶ Core3 [ProcessingUnit]
                ▶ Core4 [ProcessingUnit]
                ▶ Core5 [ProcessingUnit]
                L2_A57 [Cache]
        ▼ Denver island [HwStructure]
            ▼ Modules (3)
                ▶ Core0 [ProcessingUnit]
```

Figure 3.5: Amalthea hardware model v0.9.8 excerpt for the Nvidia Jetson TX2 board

This hardware model of Figure 3.5 is used for the WATERS challenge outlined in Section 6.2 along with the CPU-GPU timing verification methods of Section 5.6.

For every PU and memory pairing, explicit read and write delays can be modeled based on instructions (df. Definition 3.2). This thesis assumes that these instructions are modeled according to accessing a 64 Byte cache line cl_d (just as described in [30]). Therefore, three latency derivation methods can be used that rely on (1) concrete instructions $c_{x,d}$, (2) the communication (port interface) bit with $bw_{x,d}$, or (3) the communication data rate $dr_{x,d}$. Given that the data rate is defined by $dr_{x,d} = f_x \cdot \frac{bw_{x,d}}{8}$, the three above mentioned derivations are shown in Eq. 3.7 exemplary for read access $\uparrow_{x,d}$ between PU P_x and memory m_d, but the same holds for write accesses $\downarrow_{x,d}$.

$$\uparrow_{x,d} = \frac{c_{x,d}}{f_x \cdot \varkappa_x} = \frac{cl_d}{dr_{x,d}} = \frac{cl_d \cdot 8}{f_x \cdot \varkappa_x \cdot bw_{x,d}} \tag{3.7}$$

As a consequence, the cache line access delay in instructions is defined by $c_{x,d} = \frac{cl_d \cdot 8}{bw_{x,d}}$, which can be validated if multiple properties (1)–(3) are given in the same model. The next Example 3.2 shows the derivation of read access delays for either a cache line, a connection bit width, and a data rate.

Example 3.2: Read Access Delay Calculation Methods

Assuming a 2 GHz PU P_x with an IPC value $\varkappa_x = 1$ that requires 8 instructions to read a cache line $cl = 64$ Byte from memory m_d, the read latency is:

$$\uparrow_{x,d}= \frac{8}{2 \cdot 10^9 \cdot 1} = 4 \text{ ns}$$

If the instructions are unknown, but the connection bit width is given as $bw_{x,d} = 64$, the read latency is then (nominator is a cache line in bit):

$$\uparrow_{x,d}= \frac{64 \cdot 8}{2 \cdot 10^9 \cdot 1 \cdot 64} = 4 \text{ ns}$$

Finally, given a read data rate $dr_{x,d} = 16$ GB/s (or bandwidth), the read latency is:

$$\uparrow_{x,d} \frac{64}{16 \cdot 10^9} = 4 \text{ ns}$$

With properties of the software and hardware AMALTHEA models, many time related delays can already be calculated, but stimuli, OS overheads, constraints, and mapping models, which are necessary for timing verification, are still required and hence outlined next.

3.1.3 Stimulation Model

The stimulation model contains activation values referenced by the tasks and runnables of the software model. Periodic, sporadic, and inter-process activation values are in the scope of this thesis. Definition 3.6 outlines the periodic stimulus terminology along with the properties the AMALTHEA model entity provides.

Definition 3.6: Periodic Stimulus

A periodic stimulus is defined by an offset, recurrence (interval), minimal distance value, and optional entities for jitter distribution, execution conditions, and mode value lists. At least a recurrence value is required to derive the period T_i for a task τ_i.

Execution conditions and mode value lists are dedicated specifically for operation modes and hence not used in this thesis just as the jitter distribution, which is omitted due to its focus on simulation and probabilistic analyses. The major offset, recurrence, and jitter properties are shown in Figure 3.6 [45].

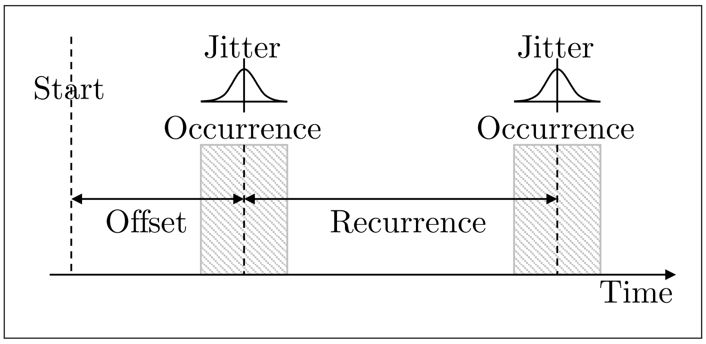


Figure 3.6: Stimulation model: periodic properties [45]

Due to priorities, memory contention, scheduling policies, or other system properties, the arrival time (start) of a task can differ from its release time. The latter defines the actual execution start of a task. Control-loop applications typically employ periodicity to regularly measure sensors and control actuators based on various control engineering loops. Deadlines in this context are required to stabilize closed-loop control as originally published by Liu and Layland in [53]. Periodic offsets are further used along with inter-process activation values and asynchronous GPU offloading, so that (i) the second part of a task τ_i, constituted by activity graph items after the task's GPU calculation trigger event, is activated after the GPU finishes the offloaded calculation[34], and (ii) other tasks can execute during task τ_i is passively waiting for the GPU to finish the offloaded instructions. A more detailed analysis of this situation is presented in Section 5.6.

Sporadic stimuli are defined by a minimal inter-arrival time and consequently form a generalization of periodic stimuli based on [49] and [185]. Consequently, sporadic stimuli can be treated as periodic ones, and hence, they are incorporated into this work's approaches. Periodic bursts, which have been studied in, e.g., [186], are supported in Amalthea, but they are not considered here. This also holds for a-periodic stimuli such as single, arrival curve [77], event, variable rate [55], or intra-sporadic [75] stimuli. Dynamic mode-, state-, or parameter-dependent asynchronous activation patterns can also exist in Autosar or Amalthea, and in correspondence with [187], a deadline of an asynchronously activated task is defined by half of its period. This generalization has been implemented as an intermediate step via migrating corresponding stimuli to entities employing half of the original period, but since mode-dependent task response time analysis has been already studied in [77], a more advanced mode-dependent analysis is omitted here.

3.1.4 Operating System Model

Information of the OS is required to consider scheduler properties such as the type (e.g., RMS) and its overheads, but also for considering resource protection approaches such as semaphores potentially imposing blocking delays. Semaphores have a PCP flag in Amalthea to identify tasks that can have a higher priority than their initially modeled fixed-priority. Semaphores can be of type resource, counting, or spinlock for bounding the number of concurrent accesses to one or more tasks, and to define whether the blocked tasks are either suspended or has to actively poll the semaphore to get access to the shared

[34]The offloaded response time defines the offset value.

resource, respectively. The latter causes wasted CPU resources due to busy waiting, which is further analyzed in Section 5.8.1.

Scheduler and OSs themselves can be hierarchically ordered under AMALTHEA because schedulers can have parent associations with other schedulers and due to OSs providing a recursive [0...*] relationship. The scheduler hierarchy permits, e.g., that a global scheduler is associated with several partitioned schedulers, such that the latter are occupying their executing PU, and scheduling their responsible tasks, only if the former global scheduler grants it. An example for this situation is available at the APP4MC platform [174, Documentation>User Guide>Examples>Scheduler Examples]. Those `Scheduler Associations` can further contain arbitrary parameter extensions or scheduling parameters in the form of key-value pairs, each of which can be associated with min budget, max budget, and replenishment values, which can each take arbitrary time parameters. The APP4MC model entity hierarchy is shown in Figure 3.7.

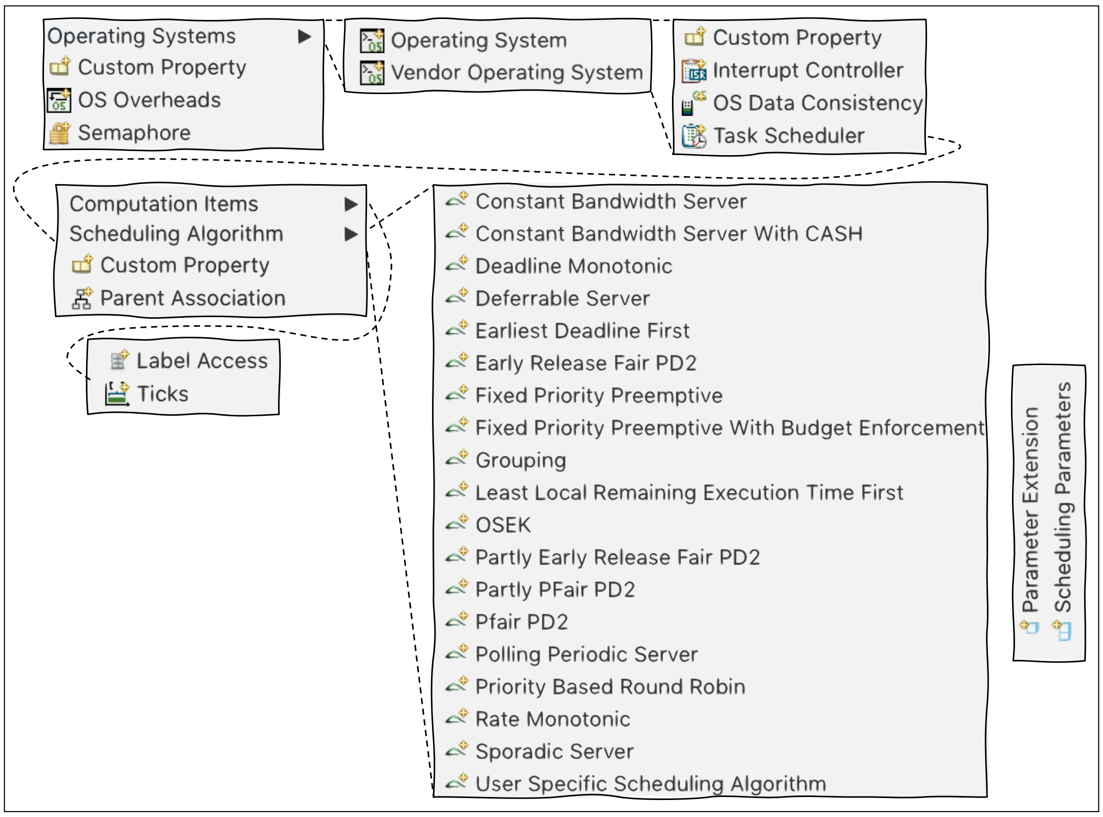


Figure 3.7: APP4MC OS model v0.9.8 entities excerpt [45]

AMALTHEA provides a comprehensive set of fixed-priority, dynamic-priority (see Section 2.1.1), Pfair [75], and reservation-based server scheduler [188][35], which go beyond the scope here and can be found at [174, Documentation>Data Models>OS Model>Scheduler>Scheduler Algorithm]. Schedulers can consume ticks, access labels, and may contain scheduler specific parameters. In this thesis, however, the focus is mainly on FPPS and RMS, which have already been outlined in Section 2.1.1. In general, a scheduler requires an executing PU and it can have multiple responsible PUs if the scheduler employs

[35]Reservation-based server scheduler release a-periodic tasks whenever no periodic task is active

a global scheduling mechanism. More details of these entities are defined by the mapping model described in the next Section 3.1.5.

3.1.5 Mapping Model

The mapping model primarily contains the task to PU mappings as the result of the task mapping process (cf. Section 5.2) and the label to memory mappings (cf. Section 5.7). The basic entities to be modeled are shown in Figure 3.8.

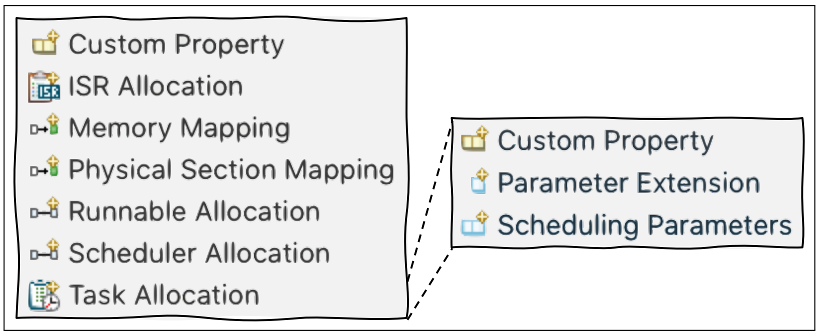


Figure 3.8: App4mc mapping model v0.9.8 entities excerpt [45]

The following Definition 3.7 outlines the task to PU mapping terminology regarding notation and semantic.

Definition 3.7: Task-PU Mapping

A task to PU mapping

$$M^P_{\tau_i} = x \Rightarrow \tau_i \text{ is allocated to } P_x \tag{3.8}$$

is distinct such that a task must be mapped to exactly one PU. $M^P_{\tau_i}$ can be represented as (i) a task ~ or (ii) a scheduler allocation if the latter references a task scheduler. A task mapping is transitive with the runnable mapping, i.e. :

$$M^P_{\tau_i} = x \Leftrightarrow M^P_{r_a} = x \;\forall\; r_a \;:\; M^\tau_{r_a} = i \text{ respectively } M^{\tau_i}_{r_a} = 1 \text{ or } r_a \in \mathcal{R}_i \tag{3.9}$$

Since this thesis assumes a distinct runnable to task partitioning $M^\tau_{r_a}$, an Amalthea `Runnable Allocation`, which provides $M^P_{r_a}$, i.e. a runnable to PU mapping, is implicit and can be derived transitively from the task mapping. Hence, no additional (redundant) model elements are used for $M^P_{r_a}$.

A `Task Allocation` (cf. Definition 3.7 (i)) defines $M^P_{\tau_i}$ for each task and hence references a task τ_i and a PU P_x. It may also refer to the scheduler of the OS model and may contain `Parameter Extensions` and/or `Scheduling Parameter` similar to the `Scheduler Associations` outlined in OS model of Section 3.1.4. The same definition holds for Interrupt Service Routine (ISR) allocations. If no affinity is modeled within the task allocation, the mapping can be derived from the referred scheduler's responsible PU, which must be distinct for partitioned-FPPS.

`Scheduler Allocation` (cf. Definition 3.7 (ii)) entities refer to (a) a scheduler of the OS model and define (b) the PU a scheduler runs on and (c) what PUs it is responsible for. For RMS and partitioned-FPPS, the scheduler allocation properties `Executing-` and `Responsible` PU entities are the same: $\forall\ M^P_{\tau_i} = x\ \exists!$ a scheduler, which executes and schedules tasks on P_x. Multiple responsible PUs are only valid for global and partially-global schedulers.

Definition 3.8: Runnable Partitioning

A runnable partitioning $M^\tau_{r_a} = i$ is distinct such that a runnable must be mapped to exactly one task. It is implicitly given by a task's activity graph, which contains the corresponding runnable call.

$$M^{\tau_i}_{r_a} = 1 \Rightarrow r_a \in \mathcal{R}_i \Rightarrow r_a \text{ is called by } \tau_i \tag{3.10}$$

`Memory Mapping` entities are used for defining the runnable, task, and label (data) to memory allocations: $M^m_{r_a}, M^m_{\tau_i}, M^m_{l_v} = d$. Each memory mapping defines a distinct memory entity of the hardware model and further defines a position address (usually in hexadecimal notation). Memory-division (e.g., Random Access Memory (RAM) / Read Only Memory (ROM) blocks), virtual memory- (controlling allocations in memory by linker of the data specification), and physical-sections (actual allocation after the linker ran) are not in the scope of this thesis, and more information can be found at [45].

3.1.6 Constraints Model

The constraints model plays a major role for this thesis due to its name-implied process affinity to CP used in Section 5. Its entities in APP4MC are partly shown in Figure 3.9.

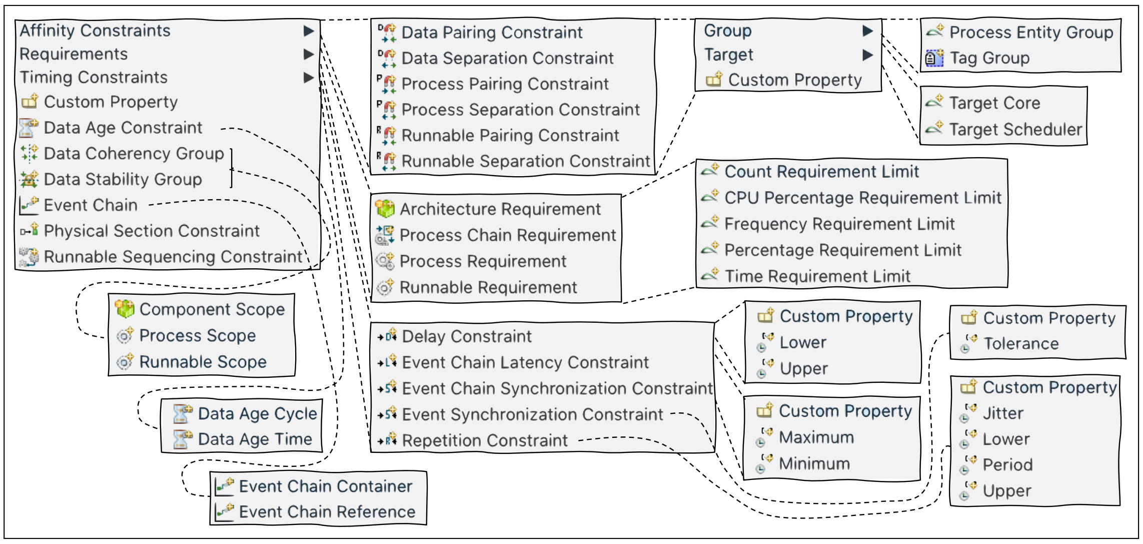


Figure 3.9: APP4MC constraints model v0.9.8 entities excerpt [45]

In the following, various constraint semantics are outlined and their notation and formalism can be found in Appendix H.2.

- **Affinity Constraints**
  - Runnable / Task Pairing with PU(s) / Scheduler: A processing entity (i.e. task or runnable) or set of processing entities must not be executed on PUs or scheduler other than the one(s) specified as the target set. According formalism can be found in Appendix H.2, Eq. H.4–H.5.
  - Runnable / Task Separation from PUs / Scheduler: A processing entity (i.e. task or runnable) or set of processing entities must not be executed on either PUs or scheduler specified as the target set. According formalism can be found at the constraints appendix H.2, Eq. H.6–H.7.
  - Runnable / Task Pairing: If no target is specified in this constraint, all entities across groups must be allocated to the same target. In this case, the number of groups take no effect. In Eq. H.9, x denotes any PU index and indicates that PU mappings must be the same across tasks and i denotes any task index and indicates that runnables within this constraint must be partitioned, i.e. grouped, into the same task. According formalism can be found at the constraints appendix H.2, Eq. H.8–H.9.
  - Runnable / Task Separation: If no target is specified in this constraint, the groups within this constraint must be separated, i.e. each group's entity mapping (for tasks the mapping concerns PUs, runnables are concerned with tasks) must differ from the other groups' entity mappings. According formalism can be found at the constraints appendix H.2, Eq. H.10–H.11.
  - Tag Affinity (e.g. for SWCs, ASILs): Instead of PU entity groups, above outlines affinity constraints can be related to tags, too. Since tags are often used to specify, e.g. SWCs, such tags can be used as groups / sets, so that no new groups need to be added and groups can be consistently used through various processes. Notation wise, replacing τ and r with tag can be used across equations H.5–H.11.
  - Data to Memory Pairing: A label or set of labels must be allocated to the target memory or any of the memories within the set of target memories. An example is later on given along with Eq. 5.77.
  - Data from Memory Separation: A label or set of labels must not be allocated to the target memory or any of the memories within the set of target memories. This constraint is further outlined along with Eq. 5.78 at the software distribution and timing verification Chapter 5.
- **Timing Constraints**
  - Delay: Based on TADL [115] and [189], the occurrences of two different events (source and target) must follow a strong (one-to-one), neutral (reaction), or weak (unique reaction) relation within a specified time window, which consists of a lower and upper bound values. Further formalism and outline can be found at appendix H.2.5 Eq. H.12
  - Event Chain / Task Chain Latency

  * Age: The last event chain response (last event of the chain) must not finish later than the specified deadline based on data produced by the stimulus event (cf. Eq. 5.50)
  * Reaction: At least one event chain response must have finished by the specified relative deadline based on data produced by the stimulus event. The calculations of $\alpha_{\gamma_g}, \rho_{\gamma_g}$ are part of Section 5.5.2.

- **Runnable Sequencing Constraints**: RSCs define a strict partial order of runnables. A runnable must not start executing before all of its predecessors, i.e. runnables contained in prior groups of the RSC, finished execution. More information given in Section 4.2.3.
- **Event Chain Constraints**: Based on TADL [189], this constraint defines a parameter for minimum completed items as well as parallel and sequence sections. This thesis found event chain latency constraints sufficient and hence the TADL-based event chains are omitted here.
- **Data Age Constraints**: The value of a label l_v must not be older than the specified time value. Or in other words: The data of l_v must be updated after at least the specified time value. More information is given in Section 5.5.1.

For any type of requirement constraints, repetition, synchronization, data coherency group, data stability group, and physical section constraints, [45] provides further information that is out of scope here.

3.1.7 Other AMALTHEA Specifics

Figures 3.2–3.9 often show the custom property entity, that can be used for arbitrary needs as outlined in the following Definition 3.9.

Definition 3.9: Custom Property

Custom properties can be used with nearly every AMALTHEA *model entity and allow to specify any values, types, and parameters, which are not already included in the meta-model.*

Finally, runtime [45, Section: Developer Guide>Model Utilities>Runtime Utilities] and deployment [45, Section: Developer Guide>Model Utilities>Deployment Utilities] utility functions are used in this thesis whereas several further functions are implemented. Fixes to the above utilities and new tools and components were committed to the official APP4MC repository during this research.

3.2 Formal Notations

The following notations of Table 3.1 formally outline a subset of AMALTHEA model entities used throughout this thesis. Each notation is further put into context in its respective section, and the following table intends to be used as a consolidated reference table.

Description	**Notation**
PUs set	$\mathcal{P} = \{P_1, ..., P_u\}$
Number of PUs	u
PUs index	$1 \leq x, y \leq u$
Frequency	f_x
Hardware Feature (e.g. IPC)	$\varkappa_x$
Memory block set	$\mathcal{M} = \{m_1, ...m_\mu\}$
Number of memories	μ
Memory index	$1 \leq d \leq \mu$
Memory size	ms_d
Bit width	$bw_{x,d}$ in Bits
Cache line length	cl_{m_d} in Bytes
Data rate	$dr_{x,d}$
Read access latency between P_x and m_d	$\uparrow_{x,d}$
Write access latency between P_x and m_d	$\downarrow_{x,d}$
Label set	$\mathcal{L} = \{l_1, ..., l_q\}$
Number of Labels	q
Label index	$1 \leq v, w \leq q$
Label size	ls_v
Label Mapping	$M_{l_v}^m = [1, \mu]$; or as boolean matrix: $\boldsymbol{M}_l^m$
Local critical section	CS^Θ
Global critical section	CS^ϕ
Critical section length	w_{CS}
Runnable set	$\mathcal{R} = \{r_1, ..., r_p\}$
Number of Runnables	p
Runnable index	$1 \leq a, b \leq p$
Runnable activation	T_{r_a} (in ps for execution time calculation)
Runnable instructions	c_a
Runnable execution time on P_x	$c_{a,x}$
a-th runnable of task τ_i	$r_{i,a}$
Runnable's read labels	$\uparrow_{r_a} \subset \mathcal{L}$
Number of times label l_v is read by r_a	$\uparrow_{r_a,v}^{\#}$
Runnable's written labels	$\downarrow_{r_a} \subset \mathcal{L}$
Runnable r_a's activation	T_{r_a}
Number of times label l_w is written by r_a	$\downarrow_{r_a,w}^{\#}$
Runnable's accessed labels	$\mathcal{L}_{r_a} = (\uparrow_a \cup \downarrow_a)$
Runnable to task index mapping	$M_{r_a}^\tau = [1, n]$
Runnable to task boolean mapping	$\boldsymbol{M}_r^\tau(p \times n); \forall a, i : M_{r_a}^{\tau_i} = \begin{cases} 1 \text{ if } M_{r_a}^\tau = i \\ 0 \text{ otherwise} \end{cases}$
Runnable to PU index mapping	$M_{r_a}^P = [1, u]$
Runnable to PU boolean mapping	$\boldsymbol{M}_r^P(p \times u); \forall a, x : M_{r_a}^{P_x} = \begin{cases} 1 \text{ if } M_{r_a}^P = x \\ 0 \text{ otherwise} \end{cases}$

Description	**Notation**
Runnable's number of CS^{Θ} accesses	$\#CS_a^{\Theta} = \lvert\mathcal{L}_{r_a} \cap \mathcal{L}_{r_b}\rvert : \forall l_v \in \mathcal{L}_{r_b} \exists \tau_j$ with $l_v \in \mathcal{L}_{\tau_j}; M_{r_a}^P = M_{r_b}^P; M_{r_a}^{\tau} \neq M_{r_b}^{\tau}$
Runnable's number of CS^{ϕ} accesses	$\#CS_a^{\phi} = \lvert\mathcal{L}_{r_a} \cap \mathcal{L}_{r_b}\rvert : \forall l_v \in \mathcal{L}_{r_b} \exists \tau_j$ with $l_v \in \mathcal{L}_{\tau_j}; M_{\tau_j}^P \neq M_{r_a}^P; M_{r_a}^{\tau} \neq M_{r_b}^{\tau}$
Task set	$\mathcal{T} = \{\tau_1, ..., \tau_n\}$
Number of tasks	n
Task index	$1 \leq i, j \leq n$
Runnables contained in task τ_i	$\mathcal{R}_i : \forall a$ with $r_a \in \mathcal{R}_i : M_{\tau_a}^{\tau} = i$
b-th runnable within task τ_i	$r_{i,b}$
Task τ_i's accessed labels	$\mathcal{L}_{\tau_i} = (\uparrow_{\tau_i} \cup \downarrow_{\tau_i})$
Task instructions	$C_i = \sum_a c_a \ : \ r_a \in \mathcal{R}_i$
Task execution time on P_x	$C_{i,x} = \sum_a c_{a,x}$
Task execution time on P_x per second	$C_{i,x}^s$
Task (periodic) activation	T_i
Task priority	π_i
Task Deadline	D_i
Tasks on P_x	$\mathcal{T}_x \subset \mathcal{T};\ \forall x, y$ with $x \neq y : \mathcal{T}_x \cap \mathcal{T}_y = \emptyset$
Task to PU index mapping	$M_{\tau_i}^P = [1, u]$
Task to PU boolean mapping	$\boldsymbol{M}_{\tau}^P(n \times u); \forall i, x : M_{\tau_i}^{P_x} = \begin{cases} 1 \text{ if } M_{\tau_i}^P = x \\ 0 \text{ otherwise} \end{cases}$
Task's read labels	$\uparrow_i = \bigcup\limits_b \left(\uparrow_{r_b} : M_{r_b}^{\tau} = i\right)$
Task's written labels of τ_i	$\downarrow_i = \bigcup\limits_b \left(\downarrow_{r_b} : M_{r_b}^{\tau} = i\right)$
Task instance (job)	z
Task instance arrival (absolute)	$\triangledown_{i,z}$
Task instance release (absolute)	$\blacktriangle_{i,z}$
Task instance response (absolute)	$\blacktriangledown_{i,z}$
Task response time (dep. on mapping)	R_i
Task response time for spec. job	$R_{i,z} = \blacktriangledown_{i,z} - \triangledown_{i,z}$
Global blocking of task τ_i	B_i^s
Local blocking of task τ_i	B_i^{pi}
Task / event chain set	$\Gamma = \{\gamma_1, ..., \gamma_k\}$
Number of task chains	k
Task chain index	$1 \leq g, h \leq k$
j-th task within γ_g	$\tau_{g,j}$
Implicit communication	ι
LET communication	λ
Explicit communication	ϵ
Age latency	α[36]
Reaction latency	ρ
Weighted edge set	$\mathcal{E} = \{e_1, ...\}$
Edge index	φ
Edge	$e_{\varphi} = \{e_{\varphi}^s, e_{\varphi}^t, e_{\varphi}^c\}$ with
Edge source	$e_{\varphi}^s = [1, p] : (\downarrow_{e_{\varphi}^s}^{\tau} \cap \uparrow_{e_{\varphi}^t}^{\tau}) \neq \emptyset$

[36]E.g. the worst case task chain age latency for implicit communication is denoted as $\alpha_{g,\iota}^+$; age latency can also be applied to labels denoted as α_{l_v}

Description	**Notation**
Edge target	$e^t_\varphi = [1,p] : (\downarrow^\tau_{e^s_\varphi} \cap \uparrow^\tau_{e^t_\varphi}) \neq \emptyset$
Edge communication cost	e^c_φ
Utilization	U
Jitter	J
Window	w
Network message index	ν
Constraint	Φ
Speedup	S
Parallelism	ξ
Slackness	ζ
Span	ς
Solution	$\mathcal{S}$
Applicable to $c_a, C_i, R_i, \alpha, \rho, B, \uparrow_{x,d}, \downarrow_{x,d}, e_w$:	
Worst-Case	$^+$ e.g. c^+_a
Best-Case	$^-$ e.g. c^-_a

Table 3.1: Formal AMALTHEA-based system model notation

Indexes $x, y, v, w, a, b, i, j, g, h, d$ are $\in \mathbb{N}$, i.e. positive natural numbers. Some indexes, e.g. i, j, k, x, y, are not distinct throughout this entire thesis, due to their additional use along with, e.g. paths, edges, cycles, RSCs, Semaphores, and TDRR conflict intervals, or more. This is due to some notations and definitions just section-wise require indexes, which are not further incorporated at other sections. Hence, no separate and distinct index notation is required and reusing existing indexes at limited extent eases readability due to not introducing a vast amount of indexes.

A runnable, task, or label mapping is either of integer or boolean nature denoted as $M^\tau_{r_a}$ and $M^{\tau_i}_{r_a}$ for runnables and $M^P_{\tau_i}$ and $M^{P_x}_{\tau_i}$ for tasks, respectively. This mapping notation serves readability and comprehension ease and follows the distinct transitivity shown in Eq. 3.11.

$$\begin{aligned} M^\tau_{r_a} &= i \Leftrightarrow \left(M^{\tau_i}_{r_a} = 1 \wedge \forall j = [1,n]; j \neq i \ : \ M^{\tau_j}_{r_a} = 0\right) \\ M^P_{r_a} &= x \Leftrightarrow \left(M^{P_x}_{r_a} = 1 \wedge \forall y = [1,u]; y \neq x \ : \ M^{P_y}_{r_a} = 0\right) \\ M^P_{\tau_i} &= x \Leftrightarrow \left(M^{P_x}_{\tau_i} = 1 \wedge \forall y = [1,u]; y \neq x \ : \ M^{P_y}_{\tau_i} = 0\right) \end{aligned} \tag{3.11}$$

An edge e_φ of a precedence graph denotes a partial order between two runnables, such that $e^s_\varphi \prec e^t_\varphi$ and is represented by an AMALTHEA RSC with two groups, containing e^s_φ in the first group and e^t_φ in the second group. In general, RSCs can be constituted by multiple groups, each containing multiple runnables. The RSCs generated during the partitioning process (cf. Chapter 4) contain only two groups, each of which containing exactly one distinct runnable. Notation wise, $\mathcal{R}^\prec_{i,r_b}$ is a set of runnables preceding r_b within τ_i so that for every runnable within this set, there exists a forward path containing r_b as target, i.e. $\forall r_a \in \mathcal{R}^\prec_{i,r_b} \exists e^s_\varphi = r_a; e^t_{\varphi'} = r_b; e_\varphi, e_{\varphi'} \in path_k$.

The system model notation of Table 3.1 is intended to provide a clear scope but yet flexibility in modeling specific industrial needs. A more comprehensive description of model entities is online available [45] and goes beyond the scope of this thesis.

4 Software Partitioning

This chapter is based on publications [4] and [20], of which concepts of the latter are further provided in Section 4.3.2. As stated in Section 2.5, software partitioning is crucial for optimizing timing properties and investigating different task mapping scenarios on a multi-PU system. However, basic heuristics such as bin-packing algorithms do not consider various constraints regarding timing, reliability, safety, affinity, and others that are mandatory in the automotive industry. Finding an optimal partitioning of runnables to tasks (cf. Definition 3.8) for the purpose of concurrently executing tasks on different PUs is an NP-complete problem [190]. Hence, the following Sections outline custom partitioning heuristics to form tasks from runnables while considering different constraints and targeting goals such as load balancing, reducing inter-task dependencies, or maximizing speedup.

The two graph-based partitioning heuristics Critical Path Partitioning (CPP) and Earliest Start Schedule Partitioning (ESSP) were implemented by the author of this thesis from scratch using the AMALTHEA model, the Eclipse APP4MC platform, and the JgraphT library[33], and the CP-based bin-packing and Constraint-Programming-based Partitioning via Cumulative Constraints (CP-PC) approaches were implemented using AMALTHEA and the choco library[37]. The former two are publicly available as part of the open-source Eclipse APP4MC platform. The latter CP-based partitioning approaches are part of APP4MC in rudimental form[38], but consolidated implementation is planned to be contributed to APP4MC soon, too. No direct contributions to either the AMALTHEA model or to one of the used libraries were necessary.

4.1 Related Work on Software Partitioning

Over the years, several strategies for partitioning and mapping in the context of embedded software have been developed. A recent thesis by Kienberger [90] assesses partitioning approaches along with AMALTHEA and this work, namely CPP and ESSP and shows that they benefit over, e.g., hierarchical task graphs or RunPar [191] by supporting verification, data validation, mapping, tool support, and scalability. In essence, the PCAM approach, denoted as "Split, then analyze" is used along with (a) partitioning that bundles strongly connected runnables based on predefined sizes and (b) bin-packing-based mapping that

[37] http://www.choco-solver.org, visited 11.2020

[38] Rudimentary unit tests for the Democar Model exist in APP4MC

considers timing constraints, dependency conflicts, and constraint violation. Even though no specific algorithmic details or implemented approaches are given in [90], it is shown that corresponding tools form mandatory components within the development process of industrial multi-PU systems in AUTOSAR and the fitness of results can potentially be improved in terms of partitioning and mapping.

Multi-Level Partitioning (MLP), which is based on [192, 193], can be used to avoid tasks that contain a long runnable sequence, i.e., runnables that depend on results produced by direct predecessors such that a long path constituted by a sequence of RSCs exists. MLP has been omitted here in order to decompose (cf. Section 4.2.4) as few edges as possible and due to partitioning results not creating too long runnable sequences in the observed case studies (cf. Section 6). Furthermore, in [90], the model and task splitting is argued to be an appropriate solution for avoiding unnecessary high search efforts. However, this thesis's approaches show that many challenges can still be approached without model or task splitting. Nonetheless, the model and task splitting concepts can be used and applied with concepts presented in this work to reduce DSE resolution time.

Lowinski et al. [92, 93] use HLEFT list scheduling extended by three greedy heuristics to either maximize speedup, called *earliest execution*, reduce cross partition precedence constraints, called *maximal parents*, or increase the gradient, i.e., *slackness* towards a synchronization point, of created partitions. Precedence constraints are ensured by synchronization points, to which runnables may have to wait in case a synchronization point has no gradient. These approaches are very close to CPP and ESSP presented in this work, and follow similar ideas. The ESSP heuristic is a combination of *earliest possible execution* and *min distance* heuristics, and partitioning also uses metrics like speedup, inter-task communications, and slackness as evaluation criteria. However, the significant difference between [93] and this work is the reverse engineering concept, i.e., splitting existing tasks, compared with forming tasks from scratch presented here.

Comparable heuristics used to evaluate partitioning and also task mapping algorithms can be FF, BF, or WF heuristics based on Baruah and Fischer [194]. These have also been used by, e.g., von der Brüggen [29] in a similar context. Given that these heuristics provide simple Greedy distribution mechanisms, their resolution time is significantly low, but on the contrary, data progression in the form of RSCs is not considered, and hence these heuristics are out of focus here.

The work by Saidi et al. in [195] is another approach close to the partitioning ideas presented in this thesis. The work uses an ILP approach to minimize (a) cut-costs (sum of inter-PU communications) of a weighted runnable DAG and (b) deviation from optimal load balancing ($\frac{\sum_a c_a}{u}$, with u being the number of PUs and c_a denoting the ticks of a runnable) for the purpose of forming one partition (sub-graphs) for each PU. However, the approach does not consider heterogeneous platforms or cycles that typically occur in runnable or task graphs. Graph cuts are arbitrary in [195], whereas partitioning in this thesis considers parallel regions via investigating fork and join scenarios.

In [196], partitioning is achieved by creating one or a few dispatcher tasks that contain runnables of different periods, offsets, and execution orders. Relative deadlines guarantee the timeliness, and the approach tends to form a static bare metal runnable scheduling. However, using a dispatcher task for runnables of different periods is not in line with the AUTOSAR standard, which intends OS scheduler to do so. Also, the approach neither considers precedence nor shared resource constraints and indeed imposes several disadvantages when being applied to multi-processor architectures due to the effects of inter-PU communication.

In [95], the used model is close to AMALTHEA-based concepts used here, since it considers (a) execution time for various processors and (b) data access times across different memories. SA, GA, and Tabu-search are used to investigate the design space of distributing runnables across PUs. The optimization goal is comparable to the mapping of Section 5 and minimizes load subject to (i) execution time on a PU, (ii) periodicity, and (iii) data access times as well as (iv) PU-load and (v) memory-load constraints. Though, schedulability, resource conflicts, and advanced timing verification for, e.g., task chains, are not considered.

In [197], runnable partitioning and task mapping are addressed, including the avoidance of cyclic dependencies. Runnable synthesis is based on synchronous models (e.g., derived from Simulink) using modularity (number of runnables), reusability (avoid feedback loops), and schedulability (minimize the number of periods) metrics via MILP. Secondly, a greedy heuristic is used for allocating runnables to tasks and PUs so that a runnable is either added to a task running with the same period on the same PU or to a newly generated task. This phase considers RSCs, memory penalties for order relaxation, as well as offset-based schedulability. Alternatively, SA is also used for the above process in [197]. It is shown that SA has significantly higher resolutions time, generates better solutions in only two of nine cases, and results in even no solution two other cases. The mapping approach is similar to the greedy Data Flow Graph (DFG) approach, which is outperformed by meta-heuristics, especially for larger models in this work (cf. Section 7). The runnable synthesis seems valuable but is out of scope in this thesis due to the assumption that runnables are already given. In general, a comparison of the two-phase approach used here with the one phase methodology, i.e., from runnables to PUs and tasks in a single greedy algorithm presented in [197], could be addressed in future work but is omitted here because greedy approaches show worse results compared with meta-heuristics in observed case study models of Section 6.

Bin-packing, which is used along with the CP approach in this chapter using the choco library[39], is combined with a least-loaded algorithm by Monot et al. in [198]. However, no precedence constraints are considered, and sorting runnables by decreasing CPU utilization should only be applied to independent runnables.

Furthermore, various scheduling publications like [199, 200] or hardware and software co-synthesis articles using GAs like [201] also address challenges close to the partitioning problem of this thesis. However, approaches presented here neither require a unique exit vertex nor entity duplication.

Finally, no research was found that considers all of the following properties, namely (1) considering precedence constraints while optimizing speedup and inter-task dependencies, (2) decomposing cycles into a runnable DAG, (3) using cumulative constraints for solving the partitioning problem, (4) using open-source models compliant to AUTOSAR, and (5) comparing results from greedy partitioning heuristics with those from meta-heuristics. The presented CP-PC approach leaves the greedy heuristic domain that includes ESSP or approaches of [90, 93] so that optimal results can be found given

[39]Thus, bin-packing is dominated by cumulative constraints via ensuing that all RSCs are kept with CP-PC

a corresponding configuration of the CP solver. Along with the presented CP-PC, CPP, and ESSP approaches, all above mentioned properties (1)–(5) can be met.

4.2 DAG-based Runnable to Task Partitioning

Partitioning Amalthea models is divided into four parts, as shown in Figure 4.1.

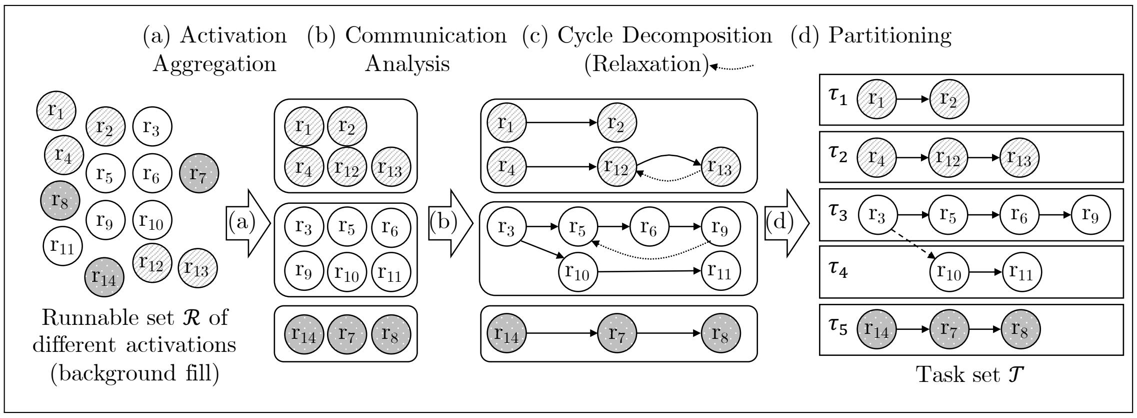

(a) Activation Aggregation
(b) Communication Analysis
(c) Cycle Decomposition (Relaxation)
(d) Partitioning
Runnable set $\mathcal{R}$ of different activations (background fill)
Task set $\mathcal{T}$
τ_1
τ_2
τ_3
τ_4
τ_5

Figure 4.1: Partitioning phases: activation and label access analysis → cycle decomposition → graph partitioning

The first part (a) is the aggregation of runnables that reference the same activation, which is most often defined by a periodic stimulus. Then, communication is analyzed in (b), to construct a directed graph, which is decomposed into a DAG by the third process (c). In general, DAGs used along with Amalthea are weighted, i.e., Weighted Directed Acyclic Graphs (WDAGs), since runnables that represent vertices are weighted according to the ticks they require for execution, just as edges, which are defined by RSCs and the corresponding labels written and read by the edge's source and target entities. Finally, the DAG is analyzed based on greedy heuristics CPP and ESSP to form tasks that can be mapped to PUs in (d). Alternatively to greedy DAG-based partitioning approaches, the CP-PC meta-heuristic (cf. Section 4.3) uses cumulative constraints derived from RSCs together with affinity and separation constraints as well as a task number rather than the complete DAG structure as input data. Each phase (a)–(d) is outlined in more detail in the following.

4.2.1 Activation Aggregation

Before analyzing the runnable communications, the initial phase addresses aggregating runnables referencing the same activation, such that any group only contains runnables of the same activation. Mixing multiple activations within a task or partition has been investigated in, e.g., [196], but is out of scope here due to not being supported by Autosar. Aggregations are modeled within PPs that reference a specific activation and define pre-task states for analysis purposes. PPs are thus temporary and transformed into tasks at the end of the partitioning process.

The definition of Eq. 4.1 is in line with the pairing affinity constraint $\Phi^{r,pair}$ of Eq. H.9 such that runnables referring different activation values are not permitted to be located in the same PP.

$$PP = \quad \{pp_1, ...\} \text{ with } \forall r_a, r_b \in \mathcal{R}_{pp_j}, a \neq b \ : \ T_{r_a} = T_{r_b} \tag{4.1}$$

Programmatically, this phase creates runnable sets for every activation and assigns runnables accordingly.

4.2.2 Pairing and Separation Constraints

Affinity respectively pairing and separation constraints are denoted with Φ and have a single source entity set, e.g. runnables, combined with a set of entities to be paired with or separated from, which can by of type runnable, task, PU, SWC, ASIL, *tag* or similar. In terms of Amalthea, SWCs, ASILs, and tags are represented as tag groups contrarily to PUs, which have a dedicated model entity within affinity constraints. The runnables of an affinity constraint are derived via notation $\mathcal{R}(\Phi)$ and the target set, e.g. PUs via $\mathcal{P}(\Phi)$. The pairing constraint approach shown in Eq. 4.2 forces every runnable within a pairing constraint to be partitioned to the same PP. It uses the runnable to task mapping notation $M^{\tau}_{r_a} = i$, which means that runnable r_a is partitioned to task τ_i, i.e. pp_i.

$$\forall r_a, r_b \in \mathcal{R}(\Phi^{r,pair}); a \neq b \ : \ M^{\tau}_{r_a} = M^{\tau}_{r_b} \tag{4.2}$$

Alternatively, Eq. 4.2 can be replaced with Eq. 4.3 so that all runnables of a PP reference the same ASIL (or SWC, or tag) or none. In other words, runnables contained in a PP are allowed to refer to at most one ASIL (or SWC or tag).

$$\forall r_a \in \mathcal{R}_{pp_j} \ : \ ASIL(r_a) = ASIL(\Phi^{r,pair}_x) \ \vee r_a \notin \bigcup_y \left(ASIL(\Phi^{r,pair}_y)\right) \tag{4.3}$$

The latter allows more flexibility since the runnable set of the pairing constraint is allowed to be allocated to multiple PPs, which is not the case for Eq. 4.2. Eq. 4.3 instead limits the amount of ASIL references per PP to 1 whereas Eq. 4.2 allows multiple ASILs per PP. Since flexibility is of major interest during the parititoning, Eq. 4.3 is chosen for the CP-PC approach. Eq. 4.2 and Eq. 4.3 also apply for constraints, that neither refer a tag group not any target entity set. If a pairing constraint contains a target PU set, the partitioning process applies Eq. 4.3 so that corresponding runnables are simply combined in the same PP and the task mapping process later on has to consider the specific PUs (transitivity of Def. 3.7 imposes that Eq. H.5 includes Eq. H.4).

Separation constraints are covered in Eq. 4.4 exemplary shown for ASILs and they are further extended in the appendix at Eq. H.6 and Eq. H.10.

$$\forall j : \text{ if } \left(\mathcal{R}_{pp_j} \cap \mathcal{R}(\Phi^{r|sep}_x)\right) \neq \emptyset \text{ then } \forall r_a \in \mathcal{R}_{pp_j} \ : \ ASIL(r_a) \neq ASIL(\Phi^{r|sep}_x) \tag{4.4}$$

Eq. 4.4 ensures that for any PP that contains a runnable of the separation constraint, all the corresponding PP's runnables do not reference the entity specified in the separation constraints, exemplary shown for an ASIL in Eq. 4.4. Furthermore, if neither a tag group nor a target set is given in a runnable separation constraint, the runnable set is assumed to be separated from all other separation constraints, which is defined in Eq. 4.5.

$$\forall r_a \in \mathcal{R}_{pp_j} \ : \ r_a \in \left(\mathcal{R} \setminus \bigcup_{y} \left(\mathcal{R}(\Phi_y^{r|sep}) \right) \cup \mathcal{R}(\Phi_x^{r|sep}) \right) \tag{4.5}$$

Figure 4.2 additionally supports Eq. 4.5 by visualizing the runnable sets across all runnables, the union of separation constraint runnables and runnables contained in a single separation constraint.

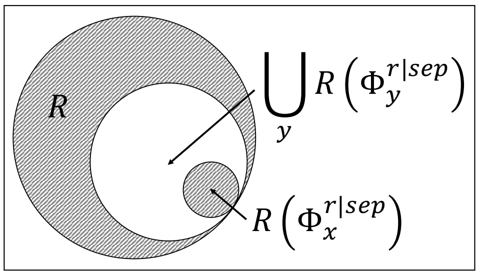

R
$\bigcup_y R\left(\Phi_y^{r|sep}\right)$
$R\left(\Phi_x^{r|sep}\right)$

Figure 4.2: Set illustration for retrieving runnables for a PP according to a separation constraint

4.2.3 Dependency Analysis

After the activation-based aggregation and ensuring that pairing and separation constraints are covered, each PP is analyzed for precedence constraints derived from label accesses. Given a set of runnables $\mathcal{R}$ (cf. Definition 3.1) as the input, a dependency graph (Weighted Directed Graph (WDG)$= \{\mathcal{R}, \mathcal{E}\}$) is formed. Graph vertices are represented by runnables and their required ticks for execution define their weights. If runnable A writes a label and another runnable B reads the same `label`, an AMALTHEA RSC within the constraints model is created. This RSC then contains A as source runnable and B as target runnable. In other words, dependencies are automatically derived from label accesses and stored within the constraints model representing directed edges. The corresponding constraint is defined in Eq. 4.8. A graph's edges $\mathcal{E}$ correspond these RSCs so that every edge has a source and a target entity and hence is directed. Notation wise, an edge's source and target runnables are addressed via e^s and e^t, respectively. As a consequence, an edge's runnable pair e^s, e^t hold the condition $\downarrow_{e^s} \cap \uparrow_{e^t} \neq \emptyset$, i.e. at least one written label of e^s is read by e^t. Additionally, an edge defines a weight e^c that is calculated via Eq. 4.6. The edge weight defines the communication cost between its source and target runnables. It is based on (a) the accessed labels' size ls_v and (b) the read / write latency $\uparrow_{x,d}$ / $\downarrow_{x,d}$ between the PU P_x that executes the runnable and the memory m_d the label is mapped to. The read / write latency (b) computation is based on m_d's cache line length cl_x and either the bit width $bw_{x,d}$ or the data rate $dr_{x,d}$, which may differ among different PUs, memories, and especially ECUs (more info see Example 3.2). If no access latency model entity is modeled for $\uparrow_{x,d}$ or $\downarrow_{x,d}$, 9 cycles are assumed for accessing global memory or local memory of different PUs, and 1 cycle is assumed for local P_x memory as proposed in [187]. By replacing 9 with $9 + n - 1$, maximal FIFO arbitration at the crossbar can be considered as described in [32]. Memory access costs e_φ^c calculation is based on the implicit communication paradigm, i.e., labels are read at the beginning of a task's execution and written at its end (more details are given in Section 5.5.1).

$$\begin{aligned} e^c_\varphi = & \sum_{v:l_v \in \downarrow_{e^s_\varphi}} \left(\frac{10^{12}}{T_{e^s_\varphi}} \cdot \downarrow^{\#}_{e^s_\varphi,v} \cdot \downarrow_{x_v,d} \cdot \left\lceil \frac{ls_v}{cl_{x_v}} \right\rceil \right) + \sum_{w:l_w \in \uparrow_{e^t_\varphi}} \left(\frac{10^{12}}{T_{e^t_\varphi}} \cdot \uparrow^{\#}_{e^t_\varphi,w} \cdot \uparrow_{x_w,d'} \cdot \left\lceil \frac{ls_w}{cl_{x_w}} \right\rceil \right) \\ & \text{with } M^m_{l_v} = d; M^P_{\tau_i} = x_v; M^m_w = d'; M^P_{\tau_j} = x_w \end{aligned} \tag{4.6}$$

As of Table 3.1, $\downarrow^{\#}_{r_a,v}$ denotes the number of times runnable r_a writes label l_v. Eq. 4.6 directly applies to explicit communication, but for implicit and LET communication, the term $\downarrow^{\#}_{e^s_\varphi,v} \cdot \downarrow_{x_v,d}$ must be replaced with $\downarrow^{\#}_{e^s_\varphi,v} \cdot \downarrow_{x_v,dl} + \downarrow_{x_v,d}$, whereas dl denotes the local memory or cache of the entity executing e^s_ϕ, to which the access latency is usually the lowest. The same holds for the second part of Eq. 4.6 that represents delays of written labels. In fact, based on [30], using Local Random Access Memory (LRAM) dl is also mandatory, if source and target entities are mapped to the same PU, i.e. if $M^P_{e^s_\varphi} = M_{e^t_\varphi} = P_x \Rightarrow m_d$ is local memory of PU P_x. More detail about the communication paradigms is given in Section 5.5. If either task to PU mapping or label to memory mapping is not available, $\downarrow_{x_v,d}$ respectively $\uparrow_{x_w,d'}$ are assumed to be 1. To sum up, every edge is represented by $e_\varphi = \{e^s_\varphi, e^t_\varphi, e^c_\varphi\} \in \mathcal{E}$ with $e^s_\varphi, e^t_\varphi \in \mathcal{R}$.

To maximize responsiveness and minimize data propagation delays, an edge's source must have finished its execution before the target starts executing within a PP so that $\forall e_\varphi \in \mathcal{E}_{pp_j} : \blacktriangledown_{e^s_\varphi} \leq \blacktriangle_{e^t_\varphi}$, i.e., the absolute finish time of the source runnable must be smaller than the absolute arrival time of the target runnable. Then, the result of the second partitioning phase are PPs that consider RSCs as shown in Eq. 4.7

$$\begin{aligned} pp_j = & \{\mathcal{R}_{pp_j}, \mathcal{E}_{pp_j}\} \text{ with } \mathcal{R}_{pp_j} \subseteq \mathcal{R}; \mathcal{E}_{pp_j} \subseteq \mathcal{E}; \forall r_a, r_b \in \mathcal{R}_{pp_j} : T_{r_a} = T_{r_b} \\ & \forall e_\varphi \in \mathcal{E}_{pp_j} : e^s_\varphi(pp_j) < e^t_\varphi(pp_j) \end{aligned} \tag{4.7}$$

Here, $e^s(pp_j)$ denotes the position respectively index of runnable r_s in pp_j. However, since RSCs may not inherently be derived from label accesses and to comply with the constraint notation used throughout this document, the notation $\Phi^{\prec}_\varphi = \{r^s, r^t\}$ is used for all RSCs. Consecutively, Eq. 4.7 applies just equally to all RSCs $\Phi^{\prec}_\varphi$ as shown in Eq. 4.8.

$$\forall x \text{ with } \Phi^{\prec}_x = \{r^s_x, r^t_x\} : r^s_x(pp_j) < r^t_x(pp_j) \text{ iff } M^\tau_{r^s_x} = M^\tau_{r^t_x} = j \tag{4.8}$$

Amalthea actually provides several groups within a RSC, which yields in Eq. 4.9, given that $r_{a,x}(\Phi^{\prec})$ denotes the runnable r_a of the x-th RSC group. However, this advanced RSC implementation has been omitted for analyses ease and RSCs are generated with two groups only in this work.

$$\forall a \text{ with } r_{a,x}(\Phi^{\prec}); \forall b \text{ with } r_{b,y}(\Phi^{\prec}); y > x \;:\; r_a(pp_j) < r_b(pp_j) \text{ iff } M^\tau_{r_a} = M^\tau_{r_b} = j \tag{4.9}$$

The following Example 4.1 gives a quick idea of a possible RSC structure and corresponding viable runnable sequences within a PP.

Example 4.1: RSC

Given the following example RSC $\Phi_x^{\prec} = \{\{r_2, r_6, r_5\}, \{r_4, r_9\}, \{r_1\}\}$, valid PP permutations are e.g. $\mathcal{R}_{pp_j} = \{r_6, r_5, r_2, r_9, r_4, r_1\}$ or $\mathcal{R}_{pp_j} = \{r_2, r_6, r_5, r_4, r_9, r_1\}$ since $\forall a \in [2,5,6], b \in [4,9], c = 1 : r_a(\mathcal{R}_{pp_j}) < r_b(\mathcal{R}_{pp_j}) < r_c(\mathcal{R}_{pp_j})$.

As the next step, the following Section 4.2.4 outlines the approach for transforming the recently obtained graphs into runnable DAGs.

4.2.4 Cycle Decomposition

The runnable graph derived from label accesses must be a-cyclic to assess a runnable's topology within the graph and preceding or succeeding paths such that partitions can be formed that potentially run concurrently across PUs without violating RSCs. Therefore, the JgraphT library[33] is used for finding cycles based on cycle algorithms presented in [202, 203] due to being java-based and compliant to Eclipse IP policy. Detected cycles are decomposed, i.e., particular edges are relaxed. Relaxation has been studied in [91] and requires careful timing analysis regarding, e.g., event or task chain latency constraints. However, since manual cycle decomposition, in general, is error-prone and inefficient, a cycle decomposition heuristic is outlined in the following to propose an edge relaxation across a set of cycles for the goal of (i) minimizing the total number of edges being decomposed and (ii) forming a graph that results in a minimal schedule length, respectively topology, given equal runnable weights. The following outlines this relaxation heuristic, which is based on finding the Minimal Feedback Arc Set (MFAS) to relax as few dependencies as possible. Therefore, several assumptions need to be made, and notations and definitions must be outlined. Due to these outlines mostly stem from graph theory and impede the focus here on cycle decomposition, corresponding formalism is shifted to the appendix at H.3, which targets the definition of a path, constituted by edges, a path set, cycles, and a cycle set. The *Cycles* cardinality is the highest number of edges that need to be *relaxed* to obtain a DAG. However, since some edges may be part of more than one cycle, corresponding edges are ordered firstly by the number of cycles they are involved with and secondly by their weight e^c. The goal is to find the minimal amount of edges for forming a DAG, which is known as the MFAS problem [204] and is NP-hard [28][40]. The MFAS set definition is given in Eq. 4.10 and imposes Eq. 4.11, which means that a PP without edges contained in the MFAS_{pp_j} set results in a cycle free graph, i.e. DAG.

$$MFAS_{pp_j} = \{e_\varphi, ...\} \text{ with } \forall\varphi \;:\; e_\varphi \in cycle \tag{4.10}$$
$$\mathcal{E}_{pp_j} \setminus MFAS_{pp_j} \Rightarrow Cycles_{pp_j} = \emptyset; \tag{4.11}$$

The objective can be formalized as minimize $|MFAS_{pp_j}|$ s.t. Eq. 4.10 and Eq. 4.11.

Algorithm 1 presents the Amalthea-based MFAS calculation heuristic that specifically concerns a PP so that cycles and paths involve only parameters of the PP and not the entire system model. Consequently, Algorithm 1 is applied separately to every PP. It is constituted by two parts, which (i) sort edges by the number of cycles they are involved

[40]For the MFAS terminology, a feedback matches a cycle and an edge, respectively RSC, matches an arc.

with as well as increasing edge weight within a potential edge decomposition list and (ii) iteratively take edges from the latter list, put them into a MFAS list and update the former list until no cycles are left, respectively. When the algorithm finishes, all edges contained in the MFAS list are decomposed from RSCs into `access precedence` entities, which indicate follow up processes like timing verification that runnables involved with `access precedences` are allowed to work with data (labels) produced by runnables of prior instances, respectively periods.

The *pem* map of Algorithm 1 is used for keeping the edge prioritization order according to the optimization goals considered in lines 7–9, and 18. It is necessary to update this map for every relaxation iteration since each edge removal results in a new $Cy\#$ value for all map entries that involve the removed edge within one of their cycles. The algorithm stops when all cycles are removed and line 14 returns true.

Algorithm 4.1: AMALTHEA-based MFAS Algorithm

```
Data: $pp_j = \{\mathcal{R}_{pp_j}, \mathcal{E}_{pp_j}\}$, $Cycles$
Result: $MFAS_{pp_j} \; : \; \mathcal{E}_{pp_j} \setminus MFAS_{pp_j} \Rightarrow Cycles_{pp_j} = \emptyset$
1  let $pem$ denote an ordered edge map $map(e_\varphi, \{\#Cy(\varphi), e^c_\varphi, mpl(\varphi)\})$ (initially empty)
2  foreach edge $e_\varphi \in \mathcal{E}_{pp_j}$ do
3      let $\#Cy(\varphi)$ be the number of cycles $e_\varphi$ is involved with
4      if $(\#Cy(\varphi) == 0)$ then continue (next foreach loop iteration)
5      let $e^c_\varphi$ be the edges weight according to Eq. 4.6
6      let $mpl(\varphi)$ initially be the sum of edge and vertex weights $mpl(\varphi) = \sum_{a=1}^{a=|\mathcal{R}_{pp_j}|} c_a + \sum_{b=1}^{b=|\mathcal{E}_{pp_j}|} e^c_b$
7      put $e_\varphi$ to $pem$ (with $\#Cy(\varphi)$, $e^c_\varphi$, and $mpl(\varphi)$ values) at position $k$ so that $\forall i \in [1, k-1]$ :
8          $\#Cy(i) \geq \#Cy(\varphi)$
9          if $(\#Cy(i) == \#Cy(\varphi))$ then $e^c_i \leq e^c_\varphi$
10 end
11 while $|Cycles| \neq \emptyset$ do
12     let $e_\varphi$ be the first entry in the priority edge map $e_\varphi = pem.getkeys.get(0)$
13     let $CC_\varphi$ be the superset of edges across all cycles that involve $e_\varphi$ $CC_\varphi = \bigcup_j cycle_j \; : \; e_\varphi \in cycle_j$
14     if removing $e_\varphi$ results in a path $(e^t_\varphi \wedge e^s_\varphi) \notin Cycles(\mathcal{E}_{pp_j} \setminus e_\varphi)$ then
15         foreach $e_{\varphi_c} \in CC_\varphi$ do
16             set $mpl(\varphi_c)$ to the maximal path length including $e^s_\varphi \wedge e^t_\varphi$ and excluding $e_\varphi$
17             reorder $e_{\varphi_c} \in pem$ to index $k$, so that $\forall i \in [1, k-1]$ :
18             if $(\#Cy(i) == \#Cy(\varphi) \wedge e^c_i == e^c_\varphi)$ then $mpl(i) \leq mpl(\varphi_c)$
19         end
20     end
21     add $e_\varphi$ to $MFAS_{pp_j}$ and add a new $e_\varphi$-correspondent access precedence
22     remove $e_\varphi$ from $\mathcal{E}_{pp_j}$ and $pem$ and all cycles containing $e_\varphi$ from $Cycles$
23     decrease $\#Cy(\varphi)$ in $pem$ for every edge $e(CC_\varphi)$
24 end
```

To retrieve the MFAS, the super set of edges across all cycles (cf. lines 2–4) is ordered by (a) the descending amount of cycles the edges are involved with (cf. Algorithm 1 line 8), (b) ascending weight (cf. Algorithm 1 line 9), and (c) ascending maximal path length (or topology if there are equal runnable weights, its calculation is shown in Eq. 4.12) of the resulting graph(s) (cf. Algorithm 1 line 18). While (a) and (c) correspond to the previously outlined optimization goals, (b) is used to account for edge weights in a way that lower communication cost (edge weight) is preferred over higher costs in case multiple edges exist for the same amount of cycles they are involved with. Line 6 in Algorithm 1 ensures that any edge, which does not result in a DAG when being removed (cf. Algorithm 1 line 14), receives a low priority (high value), i.e. the sum of all vertex and edge weights, to prefer edges for relaxation that receive a lower $mpl(\varphi)$ value according to the resulting path length when being relaxed (cf. Algorithm 1 line 16), which corresponds to goal (c).

Finally, lines 11–22 in Algorithm 1 process the transformation towards effective DAGs such that (a) the transformed edge set is minimal, (b) relaxed (cut) edges involve comparably low communication efforts, and (c) DAGs have a comparably low critical path length. In terms of AMALTHEA, edges, represented by RSCs, are transformed to `AccessPrecedence` entities. `AccessPrecedence`s represent indirect dependencies and allow runnables to work with data of produced by predecessor entities of previous iterations. This process is often utilized in later engineering phases e.g. code generation, which then consider increased data age constraints for data related to the AccessPrecedence. Lines 23 and 22 are necessary to adjust the priority structure *pem* due to an edge removal affecting #*Cy* property and reduces the number of cycles (cf. line 21). The following hypothetical example provides further insights into the algorithm's functionality.

Example 4.2: Cycle Decomposition

Given is directed graph consisting of 5 runnables r_1–r_5, 8 edges, e_1–e_8, and 4 cycles $cycle_1$–$cycle_4$ as shown in Figure 4.3 (a).

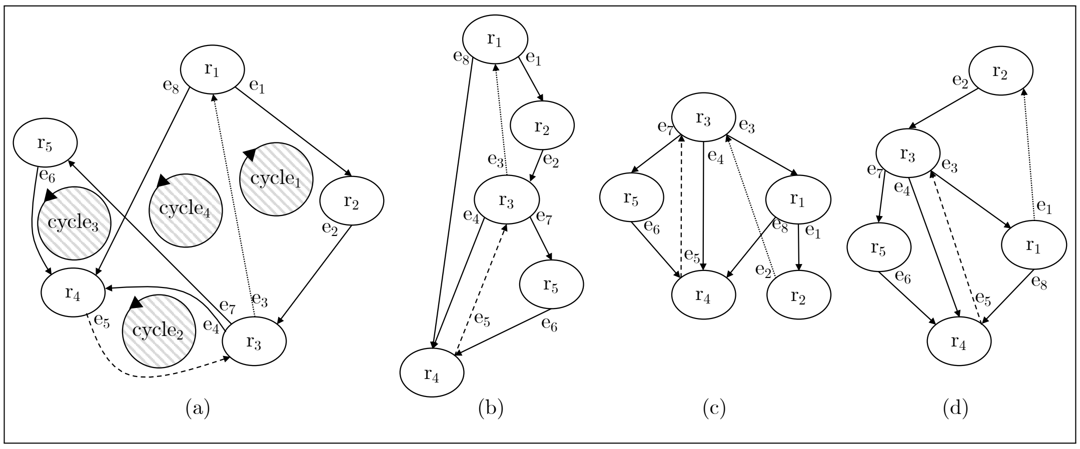


Figure 4.3: Graph cycle relaxation example: (a) runnable input graph with four cycles and (b)-(d) relaxed graphs decomposing (b) e_5, e_3, (c) e_5, e_2, and (d) e_5, e_1

Formally, the following sets are defined: $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}; \mathcal{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}; e_1 = \{r_1, r_2, 1\}; e_2 = \{r_2, r_3, 1\}; e_3 = \{r_3, r_1, 1\}; e_4 = \{r_3, r_4, 1\}; e_5 = \{r_4, r_3, 1\}; e_6 = \{r_5, r_6, 1\}; e_7 = \{r_3, r_5, 1\}; e_8 = \{r_1, r_4, 1\}$ with equal edge and vertex weights shown in Figure 4.3 (a).

Based on Eq. H.15, the following cycles are derived: $cycle_1 = \{e_1, e_2, e_3\}$; $cycle_2 = \{e_4, e_5\}$; $cycle_3 = \{e_5, e_6, e_7\}$; $cycle_4 = \{e_3, e_5, e_8\}$. This set leads to the MFAS $MFAS = \{e_5, e_2\}$, since initially $e_5 \in \{cycle_2, cycle_3, cycle_4\}; e_3 \in \{cycle_1, cycle_4\}$ and $\{e_1, e_2, e_4, e_6, e_7, e_8\}$ are each in exactly one cycle. As a consequence, e_5 is chosen first due to solving three cycles. With this relaxation, only a single cycle remains and e_3 is now involved in as many cycles as e_1 and e_2 such that #*Cy* is not the decisive selection parameter anymore $Cycles = \{cycle_1 = \{e_1, e_2, e_3\}\}$. Since this example assumes equal communication costs, *mpl* forms the primary selection influence. This is important since the final resulting DAG highly depends on the final relaxation decision, as relaxing e_3 results in a single path (sequence, topology 5, cf. Figure 4.3 (b)) whereas relaxing e_2 results in a DAG of topology 3 (cf. Figure 4.3 (c)), i.e. two runnables exist for the second and third topological levels.

The *pem* structure states are as follows with the index denoting the relaxation iteration:

- $pem_1 = \{\ (e_5, \{3,1,13\})\ , (e_3, \{2,1,13\}), ...)\}$
- $pem_2 = \{\ (e_2, \{1,1,3\})\ , (e_1, \{1,1,4\}), (e_3, \{1,1,5\})\}$

And since the first entries in *pem*, highlighted with gray background here, are chosen for relaxation (cf. Algorithm 1 line 12), the MFAS result of the above example is $MFAS = \{e_5, e_2\}$.

In general, relaxation may result in cutting a graph into independent graphs. Therefore, the $mpl_{\mathcal{E}}$ metric (cf. Eq. 4.12) concerns an arbitrary edge set instead of a DAG only. Such set can be derived from a PP for instance, which can contain multiple independent graphs.

$$mpl_{\mathcal{E}} = \max_{i \in [1, |Paths_{\mathcal{E}}|]} \left(\sum_{j=1}^{j=|path_i|} \left(c_{e^s(i,j)} + e^c(i,j) \right) + c_{e^t(i,|path_i|)} \right) \tag{4.12}$$

The notation $c_{e^s(i,j)}$ represents the runnable instructions of j-th edge source within the i-th path. The CPP partitioning also makes use of Eq. 4.12 as outlined in Section 4.2.6. A maximal path length *mpl* defines the longest sequential path from an entry vertex, which has no incoming edge, to an exit vertex, which has no outgoing edge. Minimizing *mpl* is useful for the partitioning to form partitions that can be executed concurrently, i.e. the resulting graph provides a wider structure (more runnables per topological level) rather than fewer and longer sequences of runnables.

Summing up, activation rates (periods T_a, Eq. 4.1), communication costs (edge weights e^c), instructions (runnable weights c_a), affinity constraints (4.2–4.4), and precedence constraints (based on edges $\mathcal{E}$, cf. Eq. 4.8) are now considered. Any further affinity constraints such as runnable separations are aligned with Eq. 4.1 such that separate PPs are created in case runnable separation constraints exist for runnables referring the same activation: $\forall\ r_a \in \Phi_x^{r|sep}, r_b \in \Phi_y^{r|sep}\ : j \neq k$ with $r_a \in \mathcal{R}_{pp_j};\ r_b \in \mathcal{R}_{pp_k}$ whereas $\Phi_x^{r|sep}$ denotes the set of runnables within the x-th group of the separation constraint and $\Phi_y^{r|sep}$ y-th correspondingly. Pairing runnables referring to different activations contradicts the condition of Eq. 4.1, and hence corresponding constraints are ignored.

4.2.5 Independent Graph Aggregation

Finally, to further exploit graph structures, separating independent graphs into different PPs has been added to the partitioning process. Therefore, definitions on a DAG and runnables of a path are given in the appendix at Eq. H.17 and Eq. H.18. The set of runnables involved with a DAG is denoted as $\mathcal{R}(DAG)$ and a DAG's edges as $\mathcal{E}(DAG)$.

Independent graphs do not share any data or runnable, as shown in Eq. 4.13.

$$\forall r_a \in \mathcal{R}(DAG_i); \forall r_b \in \mathcal{R}(DAG_j):$$
$$\left(\bigcup_a \mathcal{L}_{r_a} \cap \bigcup_b \mathcal{L}_{r_b}\right) = \emptyset \; ; \; (\mathcal{R}(DAG_i) \cap \mathcal{R}(DAG_j)) = \emptyset \tag{4.13}$$

Example 4.3: Independent Graph Partitioning Democar

For instance, assuming equal activation references for the Democar model and its runnable graph shown in Figure H.1, three independent graphs are derived namely one for the isolated `DiagnosisArbiter` runnable, another one for the throttle runnables `APedSensor`, `APedVoter`, `ThrottleSensor`, `ThrottleController`, and `ThrottleActuator`, and the third one for the rest of the Democar's runables. Contrarily, taking the actual activation parameters into account, the independent graph aggregation, configured to separate every independent runnable, results in 20 runnables, which is comparably high considering the total number of 43.

Separating independent graphs into different tasks provides less inter-task communication, which can be exploited by the task mapping process since no communication costs emerge for independent tasks when being mapped to different PUs or ECUs. To avoid too many PPs being created based on independent graphs (cf. Example 4.3), which happens for some case study models outlined in Chapter 6, the independent graph aggregation can be configured to only separate graphs beyond a specified threshold, i.e., the number of runnables.

As an intermediate summary, PPs at this point provide WDAGs based on runnables, their instructions as weights, and communication costs for weighted edges. Runnable aggregations consider activation references, independent graphs, ASIL and SWC pairing and separation constraints. Based on this state, DAG-based partitioning heuristics as outlined in the next two Subsections 4.2.6 and 4.2.7 can be applied to generate tasks.

4.2.6 Critical Path Partitioning

Based on DAGs created by the processes described in the previous sections, the partitioning process targets at splitting up PPs to execute runnable sets concurrently. The idea of the CPP is that the Critical Path (CrPa) forms the lower bound on a graph's or set of graphs' schedule length, i.e., the total time to execute a complete DAG [205]. For a graph set, the maximal Critical Path Length (CPL) across all graph's CrPas forms the lower bound on the Schedule Length (SL). The CrPa is defined as the path with the maximal path length across all DAG's paths as shown in Eq. 4.14, which makes use of Eq. 4.12.

$$CrPa(DAG) = path_i \; : \; mpl_{\mathcal{E}(DAG)} = mpl_{\mathcal{E}_{path_i}} \tag{4.14}$$

Relaxing an edge within a CrPa is avoided due to increased execution time caused by communication overheads and data being exchanged across task or possibly even beyond PU or ECU boundaries. In case the graph structure shows an extensive sequence and a significantly low amount of branches (multiple runnables on different topological levels), relaxation at the CrPa could be exceptionally allowed (addressed in, e.g., [91]). However, it is omitted here due to the considered models (cf. Chapter 6) providing good results without splitting the CrPa, primarily due to the generated models not resulting in significant long runnable sequences.

CPP assigns the CrPa to the first PP and the graph's branches to further PPs following the algorithm shown in Algorithm 4.4. The CrPa is defined by runnables and dependencies, forming a path from an entry vertex to an exit vertex, of which the sum of computation and communication costs is the maximum [206]. Branches next to the CrPa never exceed the CrPa's SL execution or cause the CrPa to wait on input data. Yang et al. presented in [207] that CPP, denoted as dominant sequence clustering, provides comparable or even better performance than much-higher-complexity heuristics. The CPP algorithm makes use of earliest initial time eit_a and latest start time lst_a values for runnables. These values are calculated in the Algorithm 2 and Algorithm 3, respectively.

Algorithm 4.2: Calculation of eit_a

```
Data: $DAG, r_a$
Result: $eit_a$
$eit_a \leftarrow 0$
foreach $path_j \in Paths : r_a \in \mathcal{R}(path_j)$ do
    $cpl \leftarrow 0$;                                          /*Current path length*/
    foreach $e_\varphi \in path_j$ do
        $cpl \leftarrow cpl + c_{e^s_\varphi}$
        if $e^t_\varphi == r_a$ then
            $cpl \leftarrow cpl + c_{e^t_\varphi}$
            break
        end
    end
    if $cpl > eit_a$ then $eit_a \leftarrow cpl$;                /*Get maximum here*/
end
```

The shifting potential of a runnable is then defined by the subtraction $lst_a - eit_a$ and defines to what extent a runnable can be shifted with regard to the CrPa to not violate any RSC.

Algorithm 4.3: Calculation of lst_a

```
Data: $DAG, r_a$
Result: $lst_a$
$lst_a \leftarrow mpl_{\mathcal{E}(DAG)}$ ;                             /*CrPa*/
$maxsl \leftarrow 0$;                             /*Maximal succeeding length*/
foreach $path_j \in Paths : r_a \in \mathcal{R}(path_j)$ do
    $sucl_j \leftarrow 0$
    $ind_{j,a} \leftarrow$ index of $r_a$ in $path_j$
    for $int\ i = ind_{j,a}; i \leq |path_j|; i++$ do
        $sucl_j \leftarrow sucl_j + c_{e^s_{j,i}}$
    end
    $sucl_j \leftarrow sucl_j + c_{e^t_{j,|path_j|}}$
    if $sucl_j > maxsl$ then $maxsl \leftarrow sucl_j$;         /*Get maximum here*/
end
$lst_a \leftarrow lst_a - maxsl$
```

Here, $e^t_{i,j}$ denotes the target runnable of the j-th edge of path $path_i$. Based on eit_a and lst_a, a set of runnables assignable ar to some point in time $ppt < mpl(DAG)$ can be derived such that $ar(ppt) \subseteq \mathcal{R}(DAG)$ with $\forall r_a \in at(ppt) \ : \ eit_a \leq ppt \leq lst_a$. This is used in lines 6 of the CPP Algorithm 4.4.

Algorithm 4.4: CPP Algorithm

```
Data: $\mathcal{R}, \mathcal{E}$
Result: PPs, respectively $M^{\tau}_{r_a}\ \forall\ r_a \in \mathcal{R}$
1  find $CrPa = \max_i mpl_{path_i}$ and assign its runnables $\mathcal{R}(CrPa)$ to $pp_1$
2  while not all runnables are assigned to PPs: $\mathcal{R} \neq \bigcup_i \mathcal{R}_{pp_i}$ do
3      let $pp_j$ be a new PP and set the PP time to 0 $ppt = 0$
4      while $ppt < CrPa$ do
5          update $eit_a$ and $lst_a$ values forall $r_a$ in $\mathcal{R}(path(r_a))$
6          let $ar(ppt)$ be the runnable set assignable to $ppt$ so that $(eit_a \geq ppt; lst_a \leq ppt) \forall r_a \in ar(ppt)$
7          if $(|ar| == 0)$ then
8              set $ppt$ to $\min_b (eit_b)\ :\ r_b \in ar(ppt)$
9          else if $(|ar| == 1)$ then
10             add $r_a$ to $\mathcal{R}_{pp_j}$ and set $M^{\tau}_a = j$ with $r_a = ar(ppt).get(0)$
11             $ppt+ = c_a$
12         else
13             add $r_b$ to $\mathcal{R}_{pp_j}$ and set $M^{\tau}_b = j\ :\ (lst_b - eit_b) \leq (lst_a - eit_a)\ \forall\ r_a \in ar$
14             $ppt+ = c_b$
15         end
16     end
17 end
```

Line 5 is necessary, since every assignment of a runnable to a PP may influence *eit* and *lst* values of runnables connected to the assigned runnable through a path. The algorithm creates PPs, respectively tasks, in line 3 and checks time-wise in line 6 what runnables can be assigned according to *eit* and *lst* values, that ensure RSCs. In other words, all preceding runnables must have finished their execution when a runnable starts executing. If no runnables can be assigned to the current time slot *ppt*, line 8 increases its value to a point, at which a runnable or a set of runnables becomes available. If the former situation is the case, i.e. only one runnable is available, line 10 assigns that runnable to the current PP and the *ppt* value at the current PP is increased according to the assigned runnable's instructions c_a. For the latter case, in which multiple runnables are available, line 13 selects a runnable, which has the lowest shifting potential value $(lst - eit)$. This heuristic results in low task numbers while preserving the maximal schedule length in the form of the CrPa. Finally, the algorithm stops when all runnables are assigned (cf. line 4).

Example 4.4: Calculating Runnables' *eit* and *lst* Values & Applying CPP

Given is the cycle decomposition result DAG of Example 4.2's Figure 4.3 (c) represented as $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ and edges $\mathcal{E} = \{e_1, e_3, e_4, e_6, e_7, e_8\}$ with $e_1 = \{r_1, r_2, 1\}, e_3 = \{r_3, r_1, 1\}, e_4 = \{r_3, r_4, 1\}, e_6 = \{r_5, r_4, 1\}, e_7 = \{r_3, r_5, 1\}, e_8 = \{r_1, r_4, 1\}$. The runnables' *eit* and *lst* values are then $eit = lit = \{1, 2, 0, 1, 2\}$. This example results in no shifting potential for any runnable due to the high number of edges and equal runnable weights.

If edge e_3 is removed, runnables r_1 and r_2 would gain a shifting potential of 1 by the following new values $eit^* = \{0, 1, 0, 2, 1\}; lst^* = \{1, 2, 0, 2, 1\}$, e.g. $lst_{r_1} - eit_{r_1} = 1 - 0 = 1$, which is used in Algorithm 4.4 line 13.

When applying Algorithm 4.4 to the example result DAG of Figure 4.3 (c), two PPs are created, i.e. $PP_1 = \{r_3, r_5, r_4\}$ from line 1 and $PP_2 = \{r_1, r_2\}$ via iterating three times through the while loop at line 4 for $ppt \in (0, 2) \cap \mathbb{N}$, whereas line 8 is executed for $ppt = 0$ and line 10 for $ppt = 1$ and $ppt = 2$, since the assignable runnables set contains 0 and 1 entries, respectively.

Since the CPP is not able to limit the number of tasks, the following ESSP algorithm is presented that defines an alternative to CPP.

4.2.7 ESS Partitioning

ESSP is based on earliest start schedule (*eit* cf. Eq. 2) values and features a PP / task number limitation. For this purpose, only runnables' *eit* values (cf. Eq. 2) are needed that define the sum of the longest preceding path's instructions. The value of n, which is the number of tasks being created, is predefined for the following Algorithm 4.5.

Algorithm 4.5: ESSP Algorithm

```
Data: Runnable set $\mathcal{R}$, Edge set $\mathcal{E}$, Partition number $n$
Result: PPs repsectively $M_a^\tau \in [1, n] \forall r_a \in \mathcal{R}$
1  create empty PPs: $\forall i \in [1, n] \; \exists \; pp_i$
2  let $ppst[n]$ denote a sum of assigned instructions array for each $pp_i$, initially 0 for all entries
3  while not all runnables are assigned to PPs: $\mathcal{R} \neq \mathcal{R}(\cup_i pp_i)$ do
4      let $r_b$ be the runnable with the lowest $eit$ value across unassigned runnables
         $eit(r_b) = min_a\,(eit_a)\,\forall r_a \in \mathcal{R} \setminus \cup_i \mathcal{R}(pp_i)$ (choose runnable with $\max c_a$ if there are multiple)
5      let $pr_b$ be the set of direct predecessors of $r_b$: $\forall r_a \in pr_b \;\; : \;\; e^t(r_a) = r_b$
6      let $pp_i$ denote the selected PP
7      if the last entry of any $pp_j$ contains a runnable in $pr_b$ then
8          set $pp_i$ to $pp_j$, which has the highest $ppt$ value and contains $r_b$'s predecssor
             $pp_i = pp_j$ with $ppst[j] = max_k(ppst[k]); pr_b \cap \mathcal{R}(pp_j) \neq \emptyset$
9      else
10         let $lpt$ be the lowest PP time across all PPs :$lpt = \min_j ppst[j]$
11         let $ppt(pr_b)$ be the highest finish time of all predecessor runnables
12         if $ppt(pr_b) > lpt$ then
13             select a PP so that runnable $r_b$ starts after all its predecessors
                 $pp_i = pp_j \;\; : \;\; ppst[j] \geq ppt(pr_b)$
14         else
15             select PP with lowest start time $pp_i = pp_j \;\; : \;\; ppst[j] = lpt$
16     end
17     assign $r_b$ to $pp_i$, set $M_b^\tau = i$ and increase $ppst[i] + = c_b$
18 end
```

In line 11, $ppt(pr_b)$ denotes the maximal finish time across PPs of all predecessor runnables pr_b, i.e. $ppt(pr_b) = \max_a ppt_a$ with $r_a = e_\varphi^s$ and $e_\varphi^t = r_b$. Assuming that intra-task communication can be neglected and only inter-task costs should be considered, line 12 must be extended from $ppt(pr_b) > lpt$ to $ppt(pr_b) + e_\varphi^c$ with $e^s(\varphi) = r_a$ and $e^t(\varphi) = r_b$. However, the shown algorithm assumes communication costs to be even for intra and inter task data exchange for the same PU and additional communication costs between PUs are accounted during the task to PU mapping process.

In case a complete sequential runnable ordering would define the algorithm's input, all runnables are assigned to the same partition (task), since line 7 in Algorithm 4.5 considers dependencies and distributing a sequence of runnables would increase overall execution time due to additional synchronization and communication. Line 10 in Algorithm 4.5 ensures that load is balanced among partitions in case no parent of the runnable is located at the current PPs, since the runnable is assigned to the PP featuring the lowest utilization derived from ppt and no RSC is violated (parent assignment consideration of line 12).

ESSP provides a fixed number of partitions, few inter-task communications, and a balanced load across partitions. Assessments are presented in Section 7.1.

Example 4.5: ESSP Applied to the Cycle Decomposition Example Result

Given is the same runnable DAG as of Example 4.4, i.e. $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ and edges $\mathcal{E} = \{e_1, e_3, e_4, e_6, e_7, e_8\}$ with $e_1 = \{r_1, r_2, 1\}, e_3 = \{r_3, r_1, 1\}, e_4 = \{r_3, r_4, 1\}, e_6 = \{r_5, r_4, 1\}, e_7 = \{r_3, r_5, 1\}, e_8 = \{r_1, r_4, 1\}$.

The first assigned runnable is r_3 with the lowest *eit* value 0 at Algorithm 4.5 line 2. Since weights are equal, line 2 arbitrarily selects either r_1 or r_5 as the next runnable to be assigned. In this example, r_1 is selected and assigned to the same PP as r_3, since line 5 found r_3 as a direct predecessor of r_1 and line 8 gets executed accordingly. The next runnable is r_5 and since the single existing PP does not contain r_5's direct predecessor r_3 as its last entry, line 10 and following lines are executed, which assign r_5 to the second PP at position 1, which means that there is an empty slack between 0 and 1 at the second PP. With runnable r_2 being assigned next, its direct predecessor r_1 is located as the last entry of the first PP and line 8 allocates r_2 accordingly. Finally, r_4 is selected and allocated to the second PP via lines 12 and 8 and the final result is $pp_1 = \{r_3, r_1, r_2\}$, $pp_2 = \{-, r_5, r_4\}$.

Due to arbitration at line 4, the second possible result is $pp_1 = \{r_3, r_5, r_4\}$ and $pp_2 = \{-, r_1, r_2\}$.

4.3 CP-based Partitioning

As a further alternative to DAG-based partitioning, CP-based approaches are outlined in the following that ease programming extensions and new constraints in a programmatically natural way. Equations outlined in the following partially show '`.constraint`' notations in line with the used java API of the used Choco library.

4.3.1 CP-P via Arithmetical Constraints

The CP-based partitioning can make use of various arithmetical constraints, similar to using ILP. The Choco library[37] has been found a reasonable CP-solver since it not only provides easy to use arithmetical as well as pairing or separation constraints that can be combined with RSCs mentioned in Section 3.1.6, but also bin-packing constraints as outlined in Section 4.3.2. More details on the CP paradigm itself are given in Section 5.1.

The following equations restrict the problem of partitioning runnables to tasks via a variety of constraints. The first constraint shown in Eq. 4.15 outlines the activation aggregation and applies consecutively to ASILs, SWCs, tags, or runnable pairings. It ensures that runnables referencing the same activation are assigned to the same partition, i.e., task. In other words, a partition must not contain runnables that commonly reference more than one activation.

$$\forall\ a, i \text{ with } M^{\tau}_{r_a} = i\ :\ \texttt{.allEqual}\,(T_{r_a}) \tag{4.15}$$

However, the amount of tasks per activation is not restricted by this constraint, which means that multiple tasks can exist for the same activation. Since runnables inherit their properties such as instructions and activations to tasks, similar constraints also hold for the mapping process, i.e., the constraint of Eq. 4.15 is also applied to the task mapping (cf. Section 5.2).

After ensuring the activation aggregations and affinity constraints, Eq. 4.8 and 4.9 are used for ensuring RSCs. As mentioned earlier, RSCs can potentially have multiple groups that each reference an arbitrary amount of runnables. Here, CP has benefits over the heuristic implementation of Section 4.2.7 and 4.2.6 since the constraint implementation is of Eq. 4.9 is much less verbose. The CP implementation uses only arithmetical constraints with the smaller expression for every RSC group and corresponding runnables in combination with their position within a PP. Subsequently, the runnable to task partitioning constraint can be defined that implements the optimization goal. This definition can be done in two different ways, either using a boolean matrix or an integer array, outlined in the following.

CP-P Using a Boolean Assignment Matrix

A boolean runnable assignment matrix can be used with two `.sum` constraints as shown in Eq. 4.16 and Eq. 4.17.

$$\forall\, r_a \in \mathcal{R} : \sum_i M_{r_a}^{\tau_i} = 1 \tag{4.16}$$

Eq. 4.16 ensures that a runnable is assigned to exactly one task.

$$\forall\, \tau_i \in \mathcal{T}, r_a \in \mathcal{R} : \sum_{a,i} M_{r_a}^{\tau_i} = |\mathcal{R}| = o \tag{4.17}$$

Eq. 4.17 ensures that all runnables are assigned.

To further balance runnable loads across tasks, Eq. 4.18, a minimal task load can be set to a lower bound value (lb_τ) for each task. This lower bound shrinks the solution space and increases efficiency by reducing the CP solver's resolution time.

$$\forall\, \tau_i \in \mathcal{T} : \sum_a \left(M_{r_a}^{\tau_i} \cdot c_a \right) \geq lb_\tau \tag{4.18}$$

Instead of the lower task load bound and with the assumption that the number of tasks is always lower than the number of runnables ($o < m$), Eq. 4.18 can be replaced by Eq. 4.19.

$$\forall\, \tau_i \in \mathcal{T} : \sum_a M_{r_a}^{\tau_i} \geq 1 \tag{4.19}$$

This ensures that at least one runnable is assigned to each task. The lower bound definition in Eq. 4.18 has though been found useful for larger models to significantly reduce resolution time.

CP-P Using an Index Integer Array

Instead of a boolean assignment matrix, an integer array can be used that directly refers to a task index for an assignment decision. This is shown Eq. 4.20, which replaces Eq. 4.16–4.17.

$$M_{r_a}^{\tau} \in [1, |\mathcal{T}|] \tag{4.20}$$

Accordingly, the lower bound equations of Eq. 4.18–4.19 change to Eq. 4.21–4.22.

$$\forall\ a, j \text{ with } M^{\tau}_{r_a} = j\ :\ \sum_{a} (c_a) \leq lb_{\tau_j} \tag{4.21}$$

$$\forall j : \texttt{.count}(M^{\tau}_{r_a} = j) \geq 1 \tag{4.22}$$

However, summing up a subset of array values based on index assignments (Eq. 4.21), has been found most efficient when altering the index with the help of another boolean matrix combined with a `.scalar` constraint as shown in the following Java-Listing 4.1.

```
1  [...]
2  // n=NB_TASKS, p=NB_RUNNABLES
3  Model choco = new Model();
4  int[] runnableCosts = getRunnableCosts();
5  IntVar[] taskLoad = choco.intVarArray("TaskLoad", n, getMinRunInstr(),
       getInstrSum());
6  IntVar[] rnnblMppngInt = choco.intVarArray("RunMapInt", p, 1, n);
7  BoolVar[][] rnnblMppngBool = choco.boolVarMatrix("RunMatBool", n, p);
8  IntVar minLoad = choco.intVar("MinLoad",getMinRunInstr(), getInstrSum());
9  for (int i=0; i<n;i++) {  //every task
10   for (int a =0;a<p;a++) {  //every runnable
11     rnnblMppngBool[i][a].eq(rnnblMppngInt[a].eq(i)).post();
12   }
13   choco.scalar(rnnblMppngBool[i], runnableCosts, "=", taskLoad[i]).post();
14 }
15 for (int a=0; a<p; a++) { //every runnable
16   choco.sum(ArrayUtils.getColumn(rnnblMppngBool, a),"=",1).post();
17 }
18 for (RSC rsc : getRSCs()){  //every RunnableSequencingConstraint
19   choco.arithm(getRunIndInTask(rsc.getSource()), "<", getRunIndInTask(rsc.
        getTarget())).post();
20 }
21 IntVar minTaskLoad = choco.intVar("minTaskLoad",getMinRunInstr(),
       getInstrSum());
22 choco.min(minTaskLoad, taskLoad).post();
23 choco.setObjective(Model.MAXIMIZE, minTaskLoad);
24 choco.getSolver().solve();
25 [...]
```

Listing 4.1: CP-P for Load Balancing Using Index Variables

In Listing 4.1, line 11 is the main logic for binding an index to a vector of a matrix by setting $M^{\tau_i}_{r_a}$ to 1 (true), iff $M^{\tau}_{r_a} = i$, and to 0 otherwise. Combining the integer array with reification for applying certain constraints only for specific conditions is much less efficient than the approach presented in Listing 4.1. The same holds for a set-constraint-based CP partitioning so that the approaches using reification and set constraints are omitted for measurements of Section 7.1. Using either a boolean matrix directly or combining an integer array and a boolean matrix has not shown significant efficiency differences. Since the boolean assignment matrix approach requires a fewer amount of variables, it is chosen for measurements presented in Chapter 7.

4.3.2 CP-P via Bin-Packing Constraints

The bin-packing constraint requires an integer-based bin variable array *bins*, a binary bin assignment matrix $ba^{items \times \#bins}$, and a *load* array that accumulates weights of the items. In terms of AMALTHEA, PPs represent the bins, and runnables represent the items so that c_a values are used for the item weights. The constraint is then straight forward and shown in Eq. 4.23.

$$\texttt{.binPacking}(ba, weights, load); \tag{4.23}$$

Optimization for bin-packing is in line with previous approaches, respectively Lines 8–23 of Listing 4.1. The bin-packing constraint can be combined with others such as the previously outlined activation, SWC, *tag*, ASIL, pairing aggregation or separation, sequencing (cf. Section 3.1.6) or similar constraints for a valid CP-based partitioning DSE. Minimizing the maximal value of the *load* array via Eq. 4.24 ensures balanced loads across tasks.

$$\text{minimize} \ \max_j(load_j) \tag{4.24}$$

Since the number of *bins* is configurable and not only the maximal bin load but also, e.g., the total cost of inter-task communication can be minimized (among others) while satisfying all constraints, the CP-based partitioning forms a practical approach for addressing the partitioning problem of assigning runnables to tasks based on AMALTHEA.

4.3.3 CP-P via Cumulative Constraints

The used Choco library also provides a constraint namely *cumulative*, which can be exploited for the partitioning process. Instead of integers, the cumulative constraint applies to task variables, which do not correspond AMALTHEA but Choco tasks, and consist of three parameters, i.e. *start*, *weight*, and *end*. For the partitioning purpose, Choco tasks are derived from AMALTHEA runnables. A task's *weight* value is modeled via its WCET and the constraint ensures that $end_\tau = start_\tau + weight_\tau$. In addition to task variables, the *cumulative* constraint requires (a) *heights* and (b) *capacity* arguments. The former defines resources the task requires for being executed and is set statically to 1 for every task. The latter argument, i.e., *capacity*, is the number of available resources and set to the number of partitions that are supposed to be generated. Then, Listing 4.2 can be used for applying the cumulative constraint to a Choco model derived from AMALTHEA model properties.

```
chocoModel.cumulative(tv, heights, chocoModel.intVar(n)).post();
```

Listing 4.2: Choco cumulative constraint on CP-P

Similar to the bin-packing constraint, the objective for cumulative task partitioning is modeled via minimizing the maximal end time across all tasks. RSC are ensured via an arithmetical expression for each runnable and all its predecessors that ensures a runnable start variable being higher than all its predecessors end values, which is shown in the following Listing 4.3

```
chocoModel.arithm(runStart[rindex], ">=", runEnd[predindex]).post();
```

Listing 4.3: Precedence constraints on CP-P using cumulative constraint

Although providing nearly the same functionality, the CP-PC approach has shown much lower resolution times compared to the bin-packing methodology and hence it is used for comparisons in the remainder of this thesis prior to the bin-packing methodology.

4.4 Partitioning Metrics and Summary

The partitioning approaches CPP, ESSP, and CP-PC form tasks with the primary goal of balancing load while considering precedence constraints, cycles, activation periods, as well as separation and pairing, respectively affinity constraints. Reasonable metrics used to evaluate the effectiveness of DAG-based partitioning are derived from [205] and defined as

(i) **span**, which is the length of the CrPa for CPP and the maximal partition length in general,

$$\varsigma^{\infty} = \max_i C_i = \max_i \sum_a c_a : M_{r_a}^{\tau_i} = 1 \tag{4.25}$$

(ii) **parallelism**, which is the sequential runtime, respectively *work* (i.e. the sum of all DAG instructions), divided by the parallel runtime, i.e. span ς^{∞} when assuming that the number of PUs is at least as high as the number of tasks $u \geq n$ respectively all tasks can be executed concurrently,

$$\xi = \frac{\sum_i C_i}{\varsigma^{\infty}} \tag{4.26}$$

(iii) **slackness**, which is the factor by which the parallelism exceeds the number of PUs,

$$\zeta = \frac{\sum_i C_i}{n \cdot \varsigma^{\infty}} \tag{4.27}$$

(iv) and **speedup**, which is defined by the fraction of sequential runtime (on the fastest PU) and the span for a specific task allocation

$$S^M = \frac{\sum_i C_{i,y}^{+,s} : P_y = (\max_x puc_x)}{\left(\sum_{i:M_{\tau_i}^{P_x}=1} C_{i,x}^{+,s}\right) : P_x = \max_y U_y} \tag{4.28}$$

In general, assessing speedup requires a reference, which can be either the sequential execution assuming a single PU (cf. [205], used in Eq. 4.28), or the proportion of sequential program code (cf. [208]), which usually is defined by the span ς, or the number and capacities of PUs assuming optimal concurrent program code such that every single instruction can be distributed. For the latter, which forms the coarse-grained theoretical bound, heterogeneous capacities can be considered via factors concerning the maximal PU capacity, as shown in Eq. 4.29.

$$\hat{S}^P = \sum_x \frac{puc_x}{\hat{puc}} \text{ with } \hat{puc} = \max_x puc_x \tag{4.29}$$

Eq. 4.29 hence gives the maximal theoretical speedup value for a heterogeneous system. However, Amdahl's law rather considers improving program code fractions by specific factors, which is not quite applicable to the partitioning and task allocation approaches. Instead, concurrently executing tasks running on other PUs may mitigate their original execution time by 100%, and hence Amdahl's law reduces to $\frac{1}{p_{ssw}}$, which is the inverse of sequential code fraction. For instance, given that the CrPa constitutes 10% of the program and the parallel code can be evenly distributed across other PUs without exceeding the CrPa length, the maximal speedup of $S^M = 1/(1/10) = 10$ can be achieved. If load balancing results in the situation of a PU running the CrPa and executing additional tasks, the speedup must be derived from the PU with highest utilization, i.e., $S^M = \frac{1}{\max_x U_x}$. The actual speedup calculation of a task allocation set is shown in Eq. 4.28 and defined by the sequential execution time on the fastest PU divided by the parallel run time. It is used as a rough estimation since no information about scheduling, network delays, memory contention, or resource blocking is incorporated. Eq. 4.28 normalizes towards one second just as puc_x and $C^{+,s}_{i,x}$ do. If assuming unlimited resources and perfect resource utilization without any overheads, the speedup corresponds to the parallelism value. The following example shows the calculation of span, parallelism, and slackness along with a hypothetical runnable DAG consisting of ten runnables.

Example 4.6: Partitioning Metrics

The example of Figure 4.4 shows a runnable DAG on the left-hand side, whereas runnables are indicated as an oval forms their height represents the required ticks to execute the runnable. Dependencies are represented as arrows and derived from runnables' label accesses. On the right-hand side of Figure 4.4, results of CPP and ESSP heuristics are shown, of which the latter is configured towards both two and three amounts of tasks.

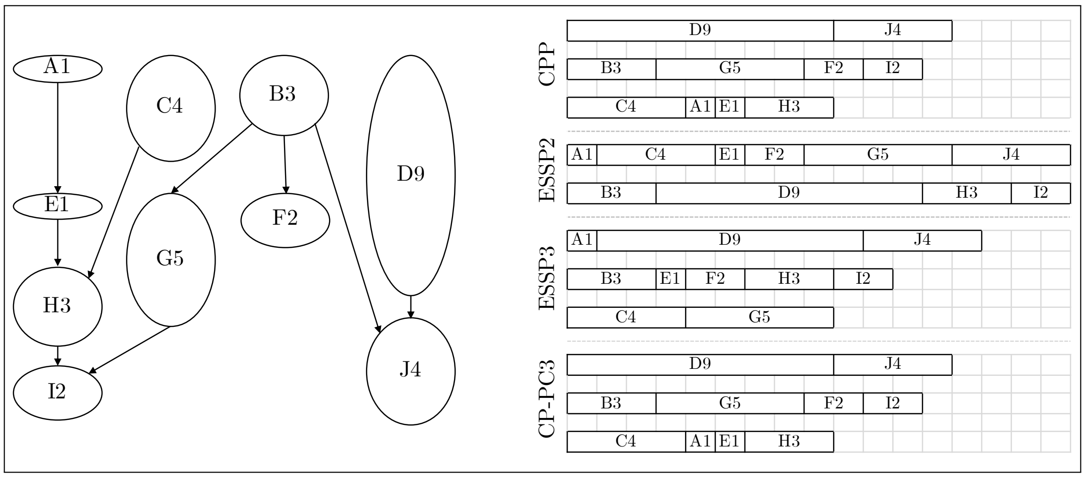


Figure 4.4: Example runnable DAG and partitioning solutions

The previously outlined metrics are presented in the following Table 4.1 given that $work = \sum_a c_a = 34$.

	$\mathcal{T}_{CPP} = \mathcal{T}_{CP-PC}$	$\mathcal{T}_{ESSP2}$	$\mathcal{T}_{ESSP3}$
ς	13	17	14
$\xi = S^M$	$\frac{34}{13} \approx 2.62$	$\frac{34}{17} = 2$	$\frac{34}{14} \approx 2.43$
ζ	$\frac{34}{3 \cdot 13} \approx 0.87$	$\frac{34}{2 \cdot 17} = 1$	$\frac{34}{3 \cdot 17} \approx 0.87$

Table 4.1: Example partitioning metric results for span, parallelism, and slackness

Since CPP targets exactly the sequential code fraction for being assigned to a separate task, CPP achieves the optimal speedup factors according to Amdahl's law when ignoring scheduling and other overheads produced by concurrent software execution. Additionally, the CPP algorithm considers all precedence constraints and assures that no other generated partition (task) exceeds the CrPa's length. Both other algorithms ESSP and CP-PC trade span ς, i.e., the lower bound of execution time, for task number limitation and thereby often do not achieve maximal speedup values when reducing the number of partitions. However, for the same amount of partitions, both ESSP and CP-PC approaches often show similar or even equal speedup results compared to CPP. CP-PC finds optimal speedup values for even a lower number of tasks compared with CPP. Figure 4.5 shows a system's theoretical speedup S as a function of the CrPa length ς for three different amount of partitions.

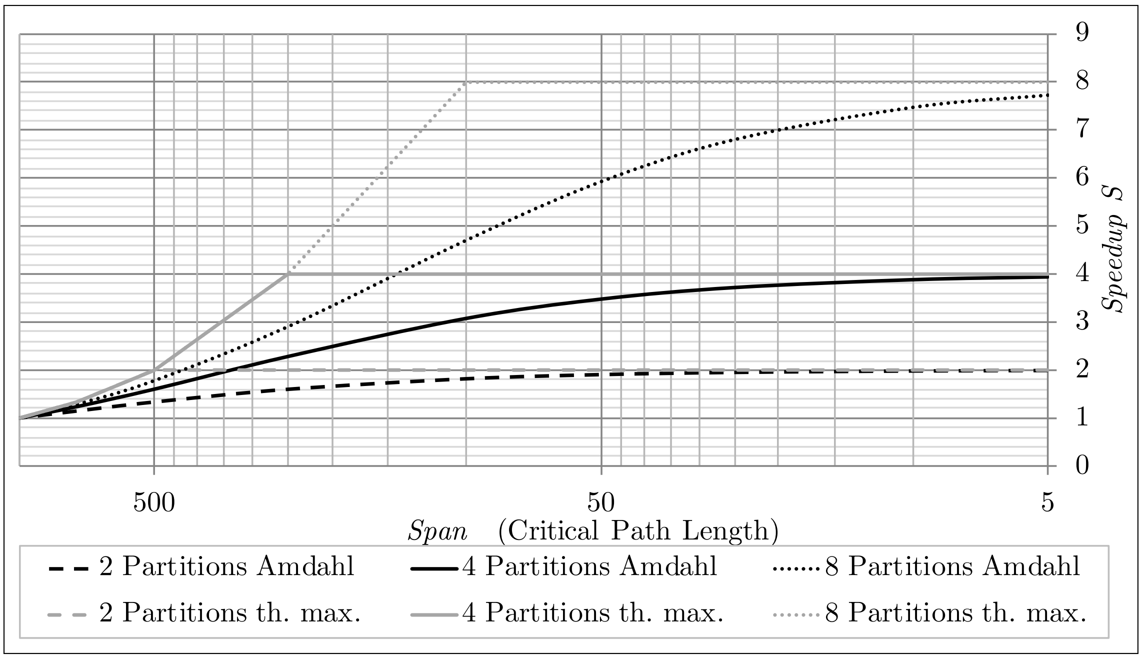


Figure 4.5: Theoretical speedup for 2, 4, 8 partitions based on [208] and [205] for 1000 instructions

Obviously, the shorter the CrPa length ς is, the higher speedup can be achieved. It is an essential characteristic of both ESSP and CP-PC to minimize the partition number, i.e., if the configured partition amount exceeds the number of critical path partitions, both approaches create empty partitions and hence provide results equal or close to CPP. If activation parameters are considered, which is ignored in this section for investigating broader graphs, the partitioning approaches iteratively split existing activation partitions, starting with the one consuming maximal instructions per second. If this runnable set is an entire sequence, the next highest instruction consuming runnable activation set is selected. Hence, this process applies CP-PC and ESSP to runnable activation subsets in a subdivision-by-two-manner, which is greedy on the one hand, but still respects activation parameters, load balancing, and precedence constraints on the other hand.

The partitioning approaches' application to, as well as further evaluation and discussion on the case study models of Chapter 6, is given in the consolidated evaluation Chapter 7. It shows that partitioning using cumulative task constraints, which is denoted as CP-PC, is more effective than the greedy ESSP, since partition lengths are generally lower than ESSP results while considering all RSCs. In contrast, the resolution time of CP-PC is higher compared with ESSP.

Summing up this chapter, three major approaches are shown that automatically form tasks as an ordered list of runnables referring to the same activation and considering several constraints, primarily RSCs. Formally, the partitioning result is given as the uniqueness quantification $\forall\ a\ \exists!\ M^{\tau}_{r_a} \in [1, n]$. As the next step, tasks can be allocated to PUs, and a partitioned scheduler for every PU can activate these tasks so that they potentially execute concurrently. By following the two-phased approach, i.e., runnable partitioning and task mapping, the number of allocations is subject to a mere fraction compared with allocating runnables directly to PUs.

5 Constrained Software Distribution & Timing Verification

This chapter primarily deals with DSE of the mapping problem, i.e. allocating tasks to PUs using different meta-heuristics as well as several timing verification techniques. Contents of this chapter have been partially published in [4] and [20].

In contrast to the partitioning's focus on determining an ideal granularity and runnable distribution for the executable software partitions, the goal of the mapping process is to find a task allocation to the PUs of a multi- or many-PU hardware target system while fulfilling constraints and optimizing goals such as load balancing. Based on the partitioning's outcome, tasks with different priorities, execution times, activation periods, communication demands, and more must not violate constraints such as (PU-) affinities, timing, safety (ASIL) aggregations, communication cost limits, as well as deadlines i.e., timing constraints, and consider various aspects of heterogeneous, networked, hierarchical, embedded, multi-PU architectures when being assigned to PUs. Software distribution is considered as a holistic allocation across heterogeneous PUs but also across ECUs connected through various network types in contrast to software parallelization, which rather concerns partitioning and allocation across homogeneous multi-PUs. Since partitioned scheduling allows using well studied single PU schedulers, the multi-PU challenge significantly manifests in finding a static and optimized mapping. This challenge is closely related to bin-packing, which is known to be NP-hard [28], and corresponding (meta-)heuristics are required to cope with intractability.

A variety of algorithms and technologies exist along with DSE for software distribution across real-time, embedded, multiprocessor, and mixed-critical systems. The automotive domain not only combines those domains (cf. beginning of Chapter 2), but even introduces further constraints and requirements due to several design decisions, standards, or evolved methodologies (cf. Section 2.3). Though, solutions and tools are predominantly proprietary, often lack in perspicuity, and sophisticated approaches towards the comprehensive concern of constraints are relatively rare.

Along with the presentation and application of typical constraints required for distributing automotive software across the PUs of vehicles in this chapter, three example mapping COP approaches are outlined in Chapter 7 to investigate and compare resolution time, scalability, and result quality between each other as well as GA, ILP, and dedicated heuristics. Benchmarks on hypothetical and industrial models (cf. Chapter 7) shows that the constraint-based approaches sometimes outperform other meta-heuristics regarding quality and effectiveness and predominantly provide more natural and advanced programming of intertwined constraints.

5.1 Related Work on Constrained Software Distribution

Before presenting the COP in Section 5.2, SotA software distribution approaches especially in context of CP are highlighted in this section.

Xiao et al. have shown in [209] that satisfying reliability goals and reducing resource consumption is challenging for precedence-constrained, mixed-critical, parallel, and embedded systems. However, the Amalthea model used in this thesis is based on Autosar and highly differs from the presented reliability goal in [209], which is based on the *constant failure rate per time unit* combined with ensuring that tasks are mapped to processors that maximize a particular reliability value. In contrast, approaches presented in this thesis ensure reliability via considering the various constraints such as affinities, pairings, separations, activations, safety levels, and more.

Thiruvady et al. studied the component deployment problem for vehicles in [158], which is similar to the partitioning and mapping problems of this work via CP. However, due to the consideration of only three constraints (memory, colocation, and communication), their approach covers just a subset of this thesis's constraints.

Oliveira et al. provide one of the few publications that compares MILP with CP for the Job Shop Scheduling Problem (JSSP) in [210]. Their results show that CP outperforms MILP in many cases and that CP is assessed as being the prior choice over MILP in generic cases. The GA approach, however, has neither been applied to the JSSP problems in [210] nor compared with CP or MILP. Also, JSSP does not cover the specific constraints presented in this work.

Limtanyakul et al. apply CP to test scheduling approaches for the automotive industry in [211]. Although the problem is different from this work's partitioning and mapping approaches, results have shown that the automotive domain comprises typical requirements and constraints that CP can ideally utilize such that CP-based DSE can potentially be more effective and efficient.

Along with the Fmtv benchmark, which is further outlined in Section 6.1 and used in Chapter 7, several research activities have been published in the recent years regarding solutions towards event chain latency calculation [212], memory contention delay analysis under different communication paradigms [34, 213], WCET and WCRT analyses [84], label mapping [85] and more. More precisely, properties of the initial Fmtv model are published in [187] along with the challenges of calculating effect-chain delays, worst-case response times considering memory accesses, optimizing the task to PU mapping, and evaluating execution platforms. The original challenge is extended in [128] by providing an Amalthea model and the outline of challenges for timing verification considering heterogeneous memory access delays, RTA for mixed preemptive tasks activated periodically, sporadically, and engine synchronously, and optimizing label placement, i.e., data to memory mapping. Solutions are proposed in [62] and [84] via analytical approaches, in [82] using the MAST tool, in [40] via the RTAna tool, and in [83] via the RTSim tool. In 2017, additional challenges were outlined in [31] for addressing memory contention, event chain reaction and age delays, and explicit, implicit, and LET communication paradigms. The AMALTHEA model was updated to provide label update frequencies, a sophisticated memory architecture, and event chains. Solutions were again presented accordingly along with [85] using analytical approaches along with MILP and GA, in [34] using solely analytical methods, in [86] using the Prelude tool, in [32] using the MAST tool, and in [33] using CPA via the pyCPA tool. In 2019, a new AMALTHEA model and new challenges were outlined in [30] for addressing RTA and optimizing task to PU mapping for CPU-GPU task offloading, CE operations, synchronous and asynchronous offloading analyses, and CPU-GPU memory interference. Solutions are proposed accordingly in [19, 87, 88, 214]. The latter is a fundamental part of and extended in this Chapter. Along with the FMTV publications, none is found covering the broad constraints and properties described in this thesis and, e.g., Table 2.2.

Hilbrich et al. present a CP approach towards safety, time, and mixed critical systems in [215]. However, there are specific differences to the AMALTHEA model. For instance, label (memory) accesses, label sizes, access rates, runnable sequencing, stimuli diversity, or event chains of AMALTHEA require new solutions for approaches of [215].

Krawczyk et al. present task mapping algorithms based on AMALTHEA via ILP and GA in [41]. Cuadra et al. in [11] extended these towards the incorporation of SA. These approaches are extended and taken as references to compare them to CP results obtained in Chapter 7. GA-based applications to automotive systems have also been investigated in [216]. Those results show, similar to generic multi-objective GAs in [217], that evolutionary algorithms scale well, especially for large-scale problems.

Along with the investigation of related work, no publication could be found that considers (i) a broader set of automotive constraints, (ii) timing verification, and (iii) an analysis of different DSE methodologies for the partitioning and mapping problems, based on AUTOSAR or AUTOSAR compliant models. The broader constraint set (i), which is part of the AMALTHEA meta-model, provides typical combinatorial patterns, which can be exploited by CP in a programmatically natural as well as straightforward way. In contrast, programming many-fold intertwined linear inequalities via ILP or expansive fitness functions requiring sophisticated gene and chromosome altering as well as crossover operation configuration for GA can significantly aggravate the DSE so that the CP approach is presented and investigated in the following. For comparison and evaluation purposes, ILP and GA are used and evaluated, too, and presented in Chapter 7.

Before starting with presenting the various CP constraints, some notations and model coherences must be outlined. The partitioning result denoted as $M^{\tau}_{r_a} \in [1, n]$ is a distinctive runnable to task assignment based on Chapter 4 and the task to PU mapping result is denoted accordingly as $M^{P}_{\tau_i} \in [1, u]$. The latter mapping is distinctive too, i.e., each runnable and task must be assigned statically to exactly one target, i.e., task and PU, respectively. With the partitioning result, a task τ_i contains an ordered sequence $path_i$ of r_a runnables: $\mathcal{R}_{\tau_i} = \{r_{i,1}, ...r_{i,r_x}\}$. PUs $\mathcal{P}$ can be obtained from higher abstraction levels such as microcontrollers, PUs, or ECUs, whereas communication costs are considered according to the modeled architecture properties such as labels, label sizes, label accesses, and access rates (cf. Eq. 4.6, respectively Definition 3.3). Each PU P_x is associated with a frequency f_x and an instructions per cycle value $\varkappa_x$ such that execution times $C^{+}_{i,x}$ can be calculated for every task as of Eq. 3.6. Runnables inherit their activation to the tasks they are assigned to: $\forall\ a$ with $M^{\tau_i}_{r_a} = 1 : T(r_a) = T(\tau_i)$. Just as runnable instructions c_i, task instructions are derived based on worst-cases, i.e. upper bounds of, e.g. Weibull or other distributions (cf. Definition 3.2) in AMALTHEA. Average values, lower bounds, and especially the distribution type can be utilized by simulation approaches such as [218] or others, but are out of scope here. The instruction cost of a task is defined by the sum of its contained runnable instructions as of Eq. 3.3 and can be transformed into a time-based value for a specific PU based on Eq. 3.6. It is assumed that tasks' deadlines are implicit to their activation. For sporadic activation, the lower bound value is used to consider worst-case arrival rates for the corresponding tasks.

Data propagation between tasks is assumed to be covered by the asynchronous use of shared labels. To determine communication costs, a communication model like explicit, implicit, or LET is preferred to be used in terms of AUTOSAR. The WCRTs of runnables and tasks depend on scheduling and corresponding preemptions. The response time analysis from Baruah et al. in [219] that has been extended by Balsini et al. in [83], can be used to calculate WCRT as well as event chain latency properties via recurrence relations. Besides, recent response time analysis solutions for adaptive variable-rate tasks presented by Biondi et al. in [220] can be further incorporated into advanced RTA for EMS-typical software with a precise estimation of worst-case interference. The configuration presented in [220] is omitted here, to achieve comparable results regarding the existing evolutionary algorithms, ILP solutions, and heuristics in Chapter 7. The mentioned model entities and constraints excellently apply to CP paradigm due to the highly combinatorial parameter and property coherencies. The vast amount of sets, relations, and properties constitute all data a constraint model requires to potentially solve a CSP and further optimize parameters of a COP. For example, aggregations such as activations, tags, or ASIL properties can be directly converted to `.allEqual`$(x[])$ constraints that guarantee that a valid solution must have equal values for all variables in x. As this is a noteworthy advantage over, e.g. ILP, which is used in Chapter 7 for comparison, the main ILP implementation on a utilization constraint is given in the following Excursus 1.

Excursus 1: ILP Mapping

This excursus is mostly part of publication [4] and forms the ILP-based mapping referred to in Chapter 7 for comparing and evaluating the different DSE paradigms. ILP-based strategies have been studied to support the creation of Pareto-optimal mapping solutions, optimized towards specific goals, e.g., minimizing the total execution time [221] or energy consumption [222]. The execution time minimization strategy minimizes the accumulated instructions of all tasks assigned to a PUs across all PUs. Since determining such an optimal allocation is well known to be an NP-complete problem [28], finding solutions for significantly larger models usually requires a substantial amount of time. The functionality for solving the ILP models is provided by the open-source *oj! Algorithms* library[a]. The resolution time of the execution time minimization approach is exponential with the number of tasks and hence not appropriate for larger models. The load balancing ILP formulation is given in Eq. 5.1.

$$\text{minimize Load}^{\max} \text{ s.t.}$$
$$\begin{aligned} \forall x &: \sum_i M_{\tau_i}^{P_x} \cdot C_{i,x} \leq \text{Load}^{\max} \text{ with } M_{\tau_i}^{P_x} = \begin{cases} 1 : M_{\tau_i}^{P} = x \\ 0 \text{ otherwise} \end{cases} \\ \forall i &: \sum_x M_{\tau_i}^{P_x} = 1 \end{aligned} \tag{5.1}$$

Eq. 5.1 uses $C_{i,x}$ of Eq. 3.6 and hence considers heterogeneous PU systems. However, Eq. 5.1 may quickly result in infeasible solutions since no periods are considered (no normalization). As a consequence, the ILP formulation is adjusted to allocate tasks to PUs under the goal of keeping all PU utilization values as low as possible. Therefore a PU's utilization is based on the sum of task utilization values shown in Eq. 5.2.

$$\text{minimize } \hat{U}^P \text{ s.t. } \forall x : \hat{U}^P \geq U_x^P \tag{5.2}$$

This is implemented as an addition to the original publication in [4] to produce comparable results for Chapter 7.

[a] http://ojalgo.org/, visited 11.2020

In general, the simplified derivation of a PU's utilization and the maximal PU utilization are shown in Eq. 5.3 and 5.4, respectively. For getting more realistic utilization values, contention, blocking, OS, and more delays must be added, which is shown in the later course of this section in Eq. 5.9.

$$U_x^P = \sum_{i:M_{\tau_i}^P = x} U_{i,x} \text{ with } U_{i,x} = \frac{C_{i,x}}{T_i} \tag{5.3}$$

$$\hat{U}^P = \max_x U_x^P \tag{5.4}$$

Yet, Excursus 1 shows a rather simple ILP-based utilization minimization and does not cover the following properties:

1. Blocking times imposed by locally and globally shared resources (cf. OPCP, solution in Section 5.4.1)
2. Memory contention (see Section 5.4.1)
3. Network message transmission times (Eq. 5.10)
4. Memory access delays (Eq. 4.6)
5. Task chain latency constraints for age and reaction delays as well as implicit and LET communication patadigms (cf. Section 5.5)
6. Any type of timing verification such as RTA, synchronous and asynchronous GPU offloading (cf. Section 7.3.3) etc.

As a consequence, the following sections present decoding AMALTHEA models into constraints used by a COP solver for the mapping of tasks to PUs while respecting the above mentioned properties.

5.2 Task Mapping Constraints

This Section describes a subset of constraints applied to the AMALTHEA meta-model outlined in Chapter 3, for calculating the task to PU mapping. Constraints are modeled using the open-source choco-library and its solver from Prud'homme et al. [44]. The task mapping problem is transposed into a COP so that CP techniques can be used to solve it. Various constraints are outlined, and their application to the mapping problem is formally presented. Because using task indexes as identifiers imposes additional constraints with increased domains compared to using a boolean matrix when using CP, which is already mentioned in the partitioning Chapter 4, the boolean matrix $\boldsymbol{M}_{\mathcal{T}}^{P}$ shown in Eq. 5.5 is used in favor to the index mapping $M_{\tau_i}^{P}$. The latter can, in fact, still be used if being arithmetically put into relation with the boolean matrix, as shown earlier at the notation section in Eq. 3.11. This coherency can be beneficial for certain constraints that require using indexes and for readability. A column of $\boldsymbol{M}_{\mathcal{T}}^{P}$, denoted as $\vec{M}_{\mathcal{T}}^{P_x}$, gives a mapping vector to PU P_x for all tasks, and a row, denoted as $\vec{M}_{\tau_i}^{P}$ gives the mapping vector for task τ_i across all PUs.

$$\boldsymbol{M}_{\mathcal{T}}^{P}(n \times u) = \begin{pmatrix} M_{\tau_1}^{P_1} & \dots & M_{\tau_1}^{P_u} \\ \vdots & \ddots & \vdots \\ M_{\tau_n}^{P_1} & \dots & M_{\tau_n}^{P_u} \end{pmatrix} \text{ with } M_{\tau_i}^{P_x} = \begin{cases} 1 \text{ if } M_{\tau_i}^{P} = x \\ 0 \text{ otherwise} \end{cases} \tag{5.5}$$

Just as for the partitioning problem, the mapping must implement <u>distinction</u> constraints similar to equations 4.16 and 4.17. This means that tasks must be mapped to exactly one PU, i.e., when having the boolean matrix $\boldsymbol{M}_{\mathcal{T}}^{P}(n \times m)$, the sum of boolean values (`true` $= 1$) across all PUs must be 1 for each task as stated in the following Eq. 5.6.

$$\forall\ \tau_i \in \mathcal{T}\ :\ \sum_{x} M_{\tau_i}^{P_x} = 1 \tag{5.6}$$

And as a consequence, every task is mapped to exactly one PU and hence the sum over all boolean assignment values equals the number of tasks n as shown in Eq. 5.7.

$$\forall\ P_x \in \mathcal{P}, \tau_i \in \mathcal{T} : \sum_{i,x} M_{\tau_i}^{P_x} = |\mathcal{T}| = n \tag{5.7}$$

Additional constraints follow that consider the various model properties such as safety levels and avoid arbitrary results. Eq. 5.9 begins with defining the PU utilization constraint and ensures that no PU is assigned with a set of tasks that exceeds the PU's execution capacity puc_x (cf. Eq. 3.4). Therefore, the task assignment vector $\vec{M}_{\mathcal{T}}^{P_x}$ for all tasks at PU P_x (a column of the matrix shown in Eq. 5.5) is multiplied with the execution time vector $\vec{C}_{\mathcal{T}}^{P_x}$ for all tasks at the PU P_x. Hence, given the scalar product $\langle \vec{M}_{\mathcal{T}}^{P_x}, \vec{C}_{\mathcal{T}}^{P_x} \rangle = \sum_i \left(C_{i,x}^{+,s} \cdot M_{\tau_i}^{P_x} \right)$, the PU <u>capacity</u> constraint is defined by Eq. 5.8 using the normalized execution time of Eq. 3.5.

$$\forall\ P_x \in \mathcal{P} : \left\langle \vec{M}_{\mathcal{T}}^{P_x}, \vec{C}_{\mathcal{T}}^{P_x} \right\rangle \le puc_x \tag{5.8}$$

However, this capacity constraint does not consider OS overheads, blocking, contention, or memory access delays. Thus, instead of the capacity constraint of Eq. 5.8, a <u>utilization</u> constraint is implemented that considers these additional overheads with the task's periodic activation, as shown in Eq. 5.9.

$$\forall\ P_x \in \mathcal{P} : \left\langle \vec{M}_{\tau}^{P_x}, \vec{U}_{\tau}^{P_x} \right\rangle \leq 1 \text{ with } U_{\tau_i}^{P_x} = \left(\frac{C_{i,x}^{+} + B_i^{+} + L_i^{+}(com)}{T_i} \right) + O_x \tag{5.9}$$

For a correct utilization, it is important that all values have the same time unit, e.g., picoseconds. The accumulated worst-case blocking B_i^+ and label access delays $L_i^+(com)$ are outlined in Eq. 5.28 and 5.30, respectively. They consider both locally shared resources, i.e. pi-blocking, and globally shared resources , i.e. s-blocking. Under AUTOSAR, the latter are usually protected by spinlocks and have a much higher influence on task execution and response times due to busy waiting. The calculation of s-blocking B_i^s for a task τ_i is outlined in Eq. 5.24, data access costs are defined in Eq. 5.30, and the calculation of memory contention is presented in Eq. 5.25. These delays are accounted for along with execution times $C_{i,x}^+$ (cf. Eq. 5.33) and OS overheads on PU P_x denoted O_x in Eq. 5.9. The additional overheads O_x on a PU P_x are assumed to be produced by OS tasks, scheduling overheads, context switches, etc. and are accounted for via static constants. An example for such overheads is a task scheduler that consist of `computation items` that are required for each task invocation. Just as for task execution times, the overheads are summed up towards one second based on the task periods of every task the scheduler is responsible for. Additionally, OS overheads have a dedicated model entity in the AMALTHEA OS model, provide `tick` parameters, and refer to APIs or ISRs. The former can be of various types, such as task activation, event call, Inter-OS-Application Communicator (IOC) read or write access, request or release resource or spinlock, among others. For each occurrence of any of those model entities within a runnable's or task's activity graph, the corresponding overhead ticks are normalized to one second via the invoked periodicity and added to O_x accordingly.

Another constraint to be considered is the costs of data being exchanged between PUs either on the same ECU or even across ECUs. Since intra-ECU communication is accounted for within access delays to labels, under the assumption that corresponding data is mapped to memory accessible from every PU within the ECU, inter-ECU communication is of primary interest here, which is derived from the amount and delays of bus messages. Data shared between tasks running on different ECUs requires transmission over a bus and induces delays denoted as R_ν^+, which are outlined in the dedicated Section 5.7.2. Accumulated inter-ECU communication costs are denoted as cc_x for a PU P_x and calculated exemplary according to CAN messages for tasks accessing the same data and running on different ECUs, i.e. $\mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j} \neq \emptyset : i \neq j; M_{\tau_i}^{ECU} \neq M_{\tau_j}^{ECU}$. Inter-ECU communication costs depend on the task to PU mapping $\boldsymbol{M}_\tau^P$ and label to memory allocation $\boldsymbol{M}_l^m$. Therefore, the label mapping approach of Section 5.7 is assumed to be used before retrieving CAN message delays. For a given label to memory and task to PU mapping, communication costs are derived from (a) the labels shared across tasks and (b) the normalized label access delays $L_i^s(com)$, which include the access rate T_i, the label size in Byte ls_v, and either the hardware cache line in Byte cl_d, bit with $bw_{x,d}$, or data rate $dr_{x,d}$ between a PU P_x and a memory m_d.

Eq. 5.10 outlines the communication cost calculation for a PU P_x, which accumulates the CAN message transmission delays outlined in Section 5.7.2. Here, mappings of tasks across hardware hierarchies are transitive similar to transitivity of Definition 3.7 so that

$M_{r_a}^{ECU} = j \Rightarrow M_{r_a}^{\tau} = i; M_{\tau_i}^{P} = x; M_{P_x}^{ECU} = j$.

$$cc_x^+(M_\tau^P) = \sum_{\varphi} R_\nu^+(\varphi) \text{ with } \forall\varphi \ : M_{e_\varphi^s}^{ECU} \neq M_{e_\varphi^t}^{ECU}; M_{e_\varphi^s}^{P} = x \tag{5.10}$$

The total communication cost across all PUs is then $cc^+ = \sum_x cc_x^+$. The condition of Eq. 5.10 regarding φ is the requirement on message delays $R_\nu(\varphi_\phi)$ between PUs across ECUs connected through a bus (inter-ECU communication). The network message delays depend on the bus type, and calculating response times of CAN messages is presented in Section 2.6.2, respectively Eq. 5.69. A message ν is derived from the written label set of a task for labels read by any other task on a different ECU.

To ensure timeliness for all network messages, the constraint of Eq. 5.11 ensures that all network message response times are lower or equal to their deadline. Once again, deadlines are assumed to be implicit and hence derived from the task's period sending the message. It must be noted here that the notation *response* does not relate to a response message rather than the time difference between the message's ready time, i.e. the message is queued at the message controller, and the message's successful transmission time.

$$\forall\varphi : R_\nu^+(\varphi) \leq D_\nu(\varphi) \text{ with } M_{e_\varphi^s}^{ECU} \neq M_{e_\varphi^t}^{ECU}; M_{e_\varphi^s}^{P} = x \tag{5.11}$$

For calculating Eq. 5.10 programmatically using CP, either reification (e.g., using `.ifThenElse` Choco notation) or boolean variables are required that are used in combination with the task to PU mapping variables to estimate the communication costs for a given mapping. For instance, adding a CAN message to the communication cost calculation of Eq. 5.10, requires that the corresponding tasks are mapped to different ECUs. Along with using the Choco library, boolean variables were found more efficient.

Finally, to reduce the solution space and speed up the resolution process, a lower bound on PU utilization can be added by using the following equations 5.12 and 5.13 (notably similar to partitioning equations 4.18 and 4.19).

$$\forall\ P_x \in \mathcal{P} : \sum_i M_{\tau_i}^{P_x} \cdot C_{i,x} \geq lb_P \tag{5.12}$$

$$\forall\ P_x \in \mathcal{P} : \sum_i M_{\tau_i}^{P_x} \geq 1 \tag{5.13}$$

In addition to the feasibility check using utilization constraints, timing verification and RTA are required to verify that response times are always lower than the tasks' deadlines, which guarantees schedulability. This verification is part of Section 5.4.

5.3 Affinity Constraints

Affinity constraints in terms of AMALTHEA are outlined in Section 3.1.6 and formalized in more detail here. Pairing and separation constraints are the main affinity types and stretch over different software (left stack in Figure 5.1) as well as hardware abstraction levels (right stack in Figure 5.1).

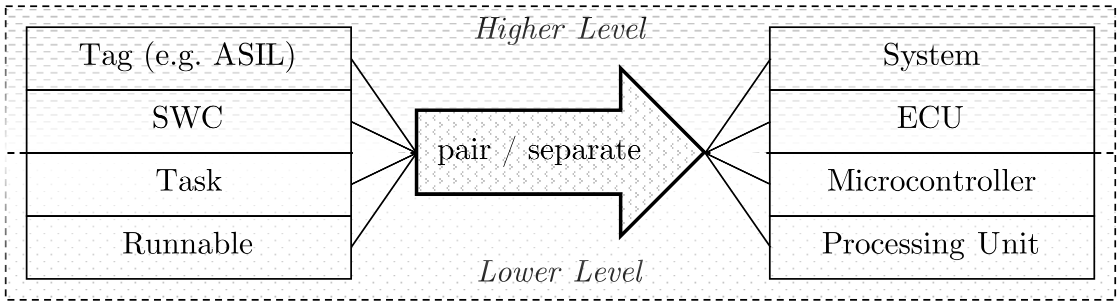


Figure 5.1: Affinity constraint levels

Any software entity or set of entities from the left can have an arbitrary pairing to or separation from an entity or set of entities of the hardware abstractions on the right. These constraints include the lowest functional level (runnables) over task and up to application levels for the software as well as the lowest hardware level for PUs over microcontroller to ECU- and system levels. For generality purposes, any abstraction levels are denoted as *swa* and *hwa* in the following for software abstraction and hardware abstraction, respectively.

A pairing $\Phi_{swa}^{hwa} = \{swa, hwa\}; swa = \{swa_i, ...\}; hwa = \{hwa_j, ...\}$ is implemented such that for all entities in $swa_i \in swa$, the relative complement indexes of hwa_j i.e. hwa'_j must be 0 as shown in Eq. 5.14.

$$\forall\ i \text{ with } swa_i \in \Phi_{swa}^{hwa} \ : \ M_{swa_i}^{hwa'_j} = 0 \tag{5.14}$$

For separation constraints, the constraint implementation is even easier, since no target set relative complement has to be found as shown in Eq. 5.15.

$$\forall\ i \text{ with } swa_i \in \Phi_{swa|\emptyset} \ : \ M_{swa_i}^{hwa_j} = 0 \tag{5.15}$$

The following example gives a quick look into applying the constraints to a SWC.

Example 5.1: SWC Pairing

Assuming a SWC pairing constraint for $SWC = \{\tau_1, \tau_3\}$ with a micro controller $\mu C = \{P_2, P_4\}$ within a system that provides four PU in total ($\Phi_{SWC}^{\mu C} = \{SWC, \mu C\}$), the pairing constraint imposes $M_{\tau_1}^{P_1} = 0; M_{\tau_1}^{P_3} = 0; M_{\tau_3}^{P_1} = 0; M_{\tau_3}^{P_3} = 0$, since $hwa'_j = \{P_1, P_3\}$, i.e. all indexes not in hwa_j, respectively $hwa'_j = \mathcal{P} \setminus hwa_j$ with $hwa_j \subset \mathcal{P}$.

The abstraction levels of Figure 5.1 manifest in hierarchies, i.e. a tag is a set of SWCs or other groups, a group / SWC is a set of tasks, which is a set of runnables. The pairings are then applied to the lowest levels, i.e., runnable and PU set-wise.

Taking the same example as separation constraint $\Phi_{SWC|\mu C} = \{\{\tau_1, \tau_3\}, \{P_2, P_4\}\}$, the constraint imposes $M_{\tau_1}^{P_2} = 0; M_{\tau_1}^{P_4} = 0; M_{\tau_3}^{P_2} = 0; M_{\tau_3}^{P_4} = 0$.

A typical use case for a separation constraint is the separation of a highly critical task from an interfering task using the same hardware, e.g., a FPU, an I/O interface, or similar.

In addition to software entity pairing with, respectively, separating from hardware entities, the target hardware group can remain empty within such constraint to separate or pair software entities disregarding any hardware. Such pairing has been found useful in industrial projects for, e.g., event chains, functionally strong connected software, or FFI requirements. However, since no strict target is defined, statically removing solutions from the solution space as done in Eq. 5.14 and Eq. 5.15 is not possible. Dynamic approaches using CP can be implemented either by applying reification (`.reifyWith()`), `If-Then` constraints, which impose reification, element constraints, or expressions to derive, e.g., task sets for every PU index x, exemplarily shown in Listing 4.1, line 11 for the runnable partitioning. The latter uses equal (`.eq`) expressions for setting $M^{\tau_x}_{r_a} = M^{\tau}_{r_a} = x?1:0$, i.e. coupling the boolean matrix with integer mapping values. Once again, index i is used for software abstraction entities, and PU-independent pairing is shown in Eq. 5.16.

$$\forall\ swa_i \in \Phi^{\emptyset}_{swa}\ :\ M^P_{swa_i} = x \text{ with } x \in [1, u] \tag{5.16}$$

Accordingly, the separation constraint without hardware entities is shown in Eq. 5.17.

$$\forall\ swa_i \in \Phi_{swa|\emptyset}\ :\ M^P_{swa_i} \neq x \text{ with } x \in [1, u] \tag{5.17}$$

The constraints can be applied recursively through the hierarchies down to runnables, but as of task to PU mapping, the lowest software abstraction level is defined by tasks. The main difference to the constraint referring a hardware entity is that some software entity values are just constrained to have the same value, but not a specific value. The constraints above mostly show inequality equations but their implementation does not necessarily require arithmetical constraints using $(=, \neq, \leq, \geq, <, >)$ operators and instead use, e.g., `.allDifferent(`$M^P_{swa_i}$`)` for separation constraints or `.Member(`$M^P_{\tau_i}$`,`hwa_j`)` for PU pairing constraints.

Finally, another affinity constraint is implemented that ensures the existence of a hyper period for PU's task set. This constraint is outlined in Excursus 2 and allowed to be violated if the activation parameters are too diverse.

Excursus 2: Hyper-periods

Scheduling tasks that are part of a hyper period was, and still is, a common practice in industrial, automotive application development. Using hyper periods is not only involved with legacy software, which, by the time it was developed, hardly could be timely verified otherwise, but also beneficial for more humanly reasoning of cause and effect chains. The corresponding analysis includes verification, which can be covered solely by considering a hyper period, i.e., the least common multiple of all activation values, which is by a considerable magnitude lower than its pendant, i.e., a time frame by the length of all tasks periods' multiplication.

As a consequence, another optional constraint is implemented for considering a hyper period existence as shown in Eq. 5.18:

$$\forall\ \tau_i \text{ with } M_{\tau_i}^{P_x} = 1 : \exists\ v \cdot T_i = Th_x;\ v \in \mathbb{N};\ Th_x < \prod_i T_i \tag{5.18}$$

By replacing $M_{\tau_i}^{P_x} = 1$ with $M_{\tau_i}^{P} = x$, the constraint can be applied to one PU or a subset of PUs. Eq. 5.18 states that each task τ_i of a task set mapped to a PU P_x must reference an activation that is part of a hyper period set, i.e. a least common multiple exists for all activation periods on a PU, that is lower than the multiplication of all periods. In other words, a hyper period is the least common multiple of a set of periods, so that a single integer (v) exists for every period that results in the hyper period when being multiplied with every period ($v \cdot T_i = Th_x$). Such validation can be used to lessen the pessimism of RTA approaches such as [219] and examine WCRT values for scheduling approaches for a given task to PU mapping set.

Given the above and the previous sections' constraints, the CP-based task to PU mapping can already be executed. However, results may only be feasible but not schedulable. Consequently, the following Section 5.4 presents the timing verification mechanisms and CP application for typical automotive software.

5.4 Timing Constraints and Verification

Since an environment determines the pace of software execution, timing verification according to various timing constraints is crucial to guarantee a safe, correct, efficient, and requirements satisfying system. In the following, such timing constraints are outlined, and corresponding analysis techniques are defined and applied to Amalthea models and its notation of Section 3.2. More precisely, analysis techniques include the investigation of blocking and contention delays (cf. Section 5.4.1) as well as schedulability along with CPU WCRTs for RMS as an FPPS example (cf. Section 5.4.2), which are further extended by reaction, propagation, and age latency constraints across implicit and LET communication for task chains (cf. Section 5.5) and RTA for mixed CPU-GPU environments in Section 5.6.

5.4.1 Blocking and Memory Contention

Data operations can be delayed in multi-PU and shared memory environments due to mutual-exclusive access to a specific memory. Various hardware interfaces may be mutually exclusive too, but here, the focus is on simultaneous access from different PUs to the same memory, which results in delays due to contention for at least one PU [223]. Specifically, memory blocking is often also referred to as the average miss penalty [175]. Such CSs require appropriate protection or access techniques to ensure correct and deterministic CS usage and hence avoid race conditions. With the locking concept of, e.g., semaphores, such mutual exclusion can be addressed, and as a consequence, blocking delays can be accounted for during the timing verification process. If no explicitly modeled Amalthea semaphore is used to protect a CS, the access time between the PU executing the corresponding task and the memory the labels are mapped to, as well as the label sizes, are used to derive the blocking delay.

Before the blocking delays are formally presented, the actual critical section windows must be defined. A CS window is denoted as w_{CS} and either depends on a specific label (cf.

Eq. 5.19) or a semaphore access (cf. Eq. 5.21) as part of a task's activity graph ag_i.

$$w_{CS}(\tau_i, l_v) = \downarrow_{x,d} \cdot \left\lceil \frac{ls_v}{cl_x} \right\rceil \text{ with } M^P_{\tau_i} = x; d = \begin{cases} M^m_{l_v} \text{ for} & com = \epsilon \\ \text{local memory of } P_x \text{ for} & com = \iota \vee \lambda \end{cases} \tag{5.19}$$

$$w^+_{CS}(\tau_i) = \max_{l_v \in \downarrow_{\tau_i}} w_{CS}(\tau_i, l_v) \text{ with}$$
$$l_v \in \begin{cases} CS^\Phi_{\tau_i} = \left(\mathcal{L}_{\tau_i} \cap CS^\Phi\right) & \text{for s-Blocking} \\ CS^\Theta_{\tau_j} \;:\; \tau_j \in lp(\tau_i) \wedge M^P_{\tau_i} = M^P_{\tau_j} & \text{for pi-Blocking} \end{cases} \tag{5.20}$$

Here, $CS^\Phi_{\tau_i}$ is the intersection of labels accessed by task τ_i and labels contained in the set of global critical sections CS^Φ, i.e. $CS^\Phi_{\tau_i} = \left(\mathcal{L}_{\tau_i} \cap CS^\Phi\right)$ and $\forall l_v \in CS^\Phi : l{\in}\mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j}$ with $M^P_{\tau_i} \neq M^P_{\tau_j}$. As a reminder, ϵ stands for the explicit communication model and ι, λ for implicit and LET communication paradigms. In the worst-case scenario, every label access is fraught with contention due to other PUs being granted access to the same shared label first under the assumption that cross PU memory accesses are granted in FIFO fashion.

With semaphores, AMALTHEA provides protecting entire parts of a runnable's or task's execution graph. Therefore, Eq. 5.21 accumulates all activity graph item costs $c_{agi_j,i,x}$ that are located in the activity graph sequence between locking a semaphore Sem_k and unlocking it.

$$w_{CS}(Sem_k, \tau_i) = \sum_{agi_n \in ag_i : (lock(Sem_k) \prec agi_n \prec unlock(Sem_k))} c_{agi_n,x} \text{ with } M^P_{\tau_i} = x \tag{5.21}$$

$$w^+_{CS}(Sem, \tau_i) = \max_k w_{CS}(Sem_k, \tau_i) \tag{5.22}$$

Activity graph items are shown in Figure 3.2, second column, and can be transformed to time values based on Definition 3.2 and Eq. 3.6. Given that the execution of code is a often a multitude higher than memory accesses, the time value of Eq. 5.21 is usually much higher than the time value of Eq. 5.19.

In addition to resource blocking, non-preemptive blocking, which results in priority-inversion, must be analyzed, too, due to AUTOSAR potentially allowing cooperative and preemptive task types. Cooperative tasks can be preempted immediately by higher priority preemptive tasks or at runnable bounds by higher priority cooperative tasks such that any cooperative task $\tau_i : pt_i = c$ can be blocked by at most the longest runnable of any lower priority cooperative tasks on the same PU.

Furthermore, CS accesses are assumed to only happen at a task's response time or period under the assumption that labels are copied into local (cache) memory and written back to shared memory at response time or period for implicit and LET communication, respectively. Reading data from memory is henceforth assumed to be potentially concurrent without memory protection. Concurrent read accesses make blocking investigation much easier since data blocking may only occur once for every task execution instance.

Based on the assumptions mentioned above, blocking in AUTOSAR is categorized into the following types, which require a distinct label mapping $d = M^m_{l_v}$ as well as distinct runnable and task mappings $M^\tau_{r_a}$ and $M^P_{\tau_i}$.

1. **Local direct blocking** B^{pi}: A high priority task τ_1 wants to acquire a lock, which is already hold by a low priority task τ_2. Both tasks run on the same PU. Task τ_1 is locally blocked by task τ_2. The locks are called *semaphores* and the arbitration protocol *suspension-based* [77, p. 55]. This blocking type is also known as *pi-blocking* [29, 224].

2. **Global direct blocking** (non-preemptive) B^{s}: A task τ_1 holds a lock, which task τ_2 wants to lock from a different PU. Task τ_2 is blocked remotely by task τ_1 and actively spins to acquire the lock. The lock is called *spinlock* and the protocol *spinning-based*. In [224], this blocking type is referred to as *s-blocking*.

3. **Non-preemptive blocking caused by cooperative tasks** B^{npc}: A cooperative task is blocked due to a lower priority cooperative task is executing a runnable.

4. **Non-preemptive blocking caused by ignored preemption** B^{np}: A task is blocked due to a lower priority task being executed and preemption is ignored.

5. **Memory Contention** B^{mc}: A task is blocked by the label it is trying to access due to the label being already accessed by a task running on another PU. The memory access policy is assumed to be ordered in a FIFO queue for PUs.

The following blocking types are often addressed in related work, but do not apply to work of this thesis as described accordingly.

(a) **Local push-through (indirect) blocking based on PCP**: A task is blocked due to a higher priority task being directly blocked by a low priority task via priority inheritance. Since OPCP/HLP does not inherit priorities, this local indirect blocking is not necessary for PCP.

(b) **Local ceiling blocking**: A task τ_1 is blocked by a lower priority task τ_2 that inherited τ_1's priority due to chained blocking risk. This situation is also only necessary for PCP, not OPCP / HLP.

(c) **Local indirect blocking caused by globally blocked tasks**: A task τ_1 is preempted by a higher priority task τ_2 locally, which is globally blocked on a spinlock hold by a task τ_3 on another PU. Indirect blocking is already accounted for during RTA, since direct local and global blocking is added to WCETs (cf. Eq. 5.33), and preemption time frames (higher priority tasks' interference on lower priority tasks) are considered in RTA equations.

(d) **Global indirect blocking**: A task is blocked on a globally shared resource hold by another task, which is preempted or blocked on a different (nested) resource. Since nesting is not allowed and interrupts are disabled for running tasks running a global CS, this blocking type does not apply to AUTOSAR and work of this thesis.

Blocking situations (a), (b), and (d) do not have to be considered in AUTOSAR, since according to the AUTOSAR OS specification [47]:

(i) OPCP (HLP) does not inherit priorities,

(ii) global CSs are always non-preemptive[41],

[41]While running in a spinlock protected CS, interrupts are assumed to be suspended, i.e., a task working with a spinlock protected resource can not be preempted

(iii) nesting spinlocks should be avoided in general.

If necessary, the AUTOSAR specification requires tasks to access spinlocks in a specific order, free of cycles and deadlocks.

With the CS windows and blocking types being sorted out, the local and global blocking delays can be defined and calculated for tasks. Assuming OPCP and that multiple low priority tasks (index j) sequentially lock different semaphores (index k), during which multiple activity graph items are called (index n, Eq. 5.21), the higher priority task is locally blocked at most a single maximal critical section length B_i^{pi} as shown in Eq. 5.23.

$$\begin{aligned} B_i^{pi,+} &= \max_{j \in lp(i)} w_{CS}^{\Theta}(\tau_j) \\ &\text{with } M_{\tau_j}^P = M_{\tau_i}^P \wedge w_{CS}^{\Theta}(\tau_j) = \max\left(\max_k w_{CS}(Sem_k, \tau_j), w_{CS}^{+}(\tau_j)\right) \\ &\text{and } Sem_k \in ag_i \end{aligned} \tag{5.23}$$

Here, the maximal CS window length is based on either the semaphore locking from Eq. 5.21 or the label contention from Eq. 5.19.

Having the LET and implicit communication paradigms in mind, the CS window length should be increased to the execution time of a runnable, because any shared label update in between runnables' executions would be overwritten without protecting the shared labels and eventually cause data inconsistencies. For instance, preemption may cause a high priority task's result to be overwritten by a lower priority task after being resumed, which copied data much earlier than the high priority task. However, this would drastically increase the spurious effect of blocking times and hence implicit and LET-based communications allow using shared resources concurrently given that data progression is much easier to trace (strict points in time to write data into shared memory) and control. If specific shared resources must still be locked for safety purposes, semaphores should be added to ensure corresponding mutual exclusion.

For deriving the global s-blocking delays, spinlocks, used in AUTOSAR to protect globally shared resources, are assumed to be granted in a FIFO fashion [31] and implemented via semaphores. The worst-case global blocking time is then the sum of maximal semaphore locking times across other PUs for every semaphore lock window, as shown in Eq. 5.24.

$$B_i^{s,+} = \sum_{k: Sem_k \in ag_i} \left(\sum_{y \setminus M_{\tau_i}^P} \max_{j: Sem_k \in ag_j \wedge M_{\tau_j}^P = y} \left(w_{CS}(Sem_k, \tau_j)\right) \right) \tag{5.24}$$

Other spinlock types than FIFO-ordered spinlocks have been studied in [35] but are out of scope here. Because Eq. 5.24 distinguishes between both CS lengths and corresponding maximal interference for each of the CSs, the s-Blocking computation forms a precise blocking analysis for AMALTHEA models, which is finer-grained than, e.g., an integer multiple of the worst-case CS accesses time of lower priority tasks based on the amount of CS accesses used in [60].

In addition to s-blocking and pi-blocking, which rely on semaphore implementation, additional delays must be accounted for when shared data is accessed from shared memory and not protected by semaphores. These delays stem from memory contention across PUs, and they are dynamically calculated using different (i) source PUs, (ii) target memories, (iii) accesses delays, and (iv) data from different tasks shown in Eq. 5.25.

$$B_i^{mc,+} = \sum_{l_v \in CS^{\Phi}_{\tau_i} : M^m_{l_v} \neq LRAM_x} \left(\sum_{y \setminus (M^P_{\tau_i} = x)} \max_{j : l_v \in CS^{\Phi}_{\tau_j} ; M^P_{\tau_j} = y} (w_{CS}(\tau_j, l_v)) \right) \text{ with } |P_{l_v} \setminus P_x| \geq 1 \tag{5.25}$$

Here, P_{l_v} is the set of PUs that execute at least one task accessing label l_v, LRAM_x is the local RAM of P_x, and task τ_i is mapped to PU P_x, i.e., $M^P_{\tau_i} = P_x$. This contention equation significantly improves the contention calculation of [30], since

(a) varying label accesses of different tasks,

(b) various access latency values between different PUs and memories,

(c) only shared labels (CSs),

(d) varying label sizes,

(e) dual-ported LRAM[42],

(f) arbitrary baseline parameters, and

(g) FIFO memory arbitration [31]

are considered.

Calculating blocking delays caused by cooperative tasks only applies to those of the corresponding preemption type, i.e., $\tau_i : pt_i = c$, and it is defined by the longest runnable of other lower priority cooperative tasks mapped to the same PU as shown in Eq. 5.26.

$$B_i^{npc,+} = \max_a (c_{a,x}) : M^{\tau}_{r_a} = j ; pt_j = c \text{ (is cooperative) } ; i \neq j ; M^P_{\tau_i} = M^P_{\tau_j} = M^P_{r_a} = x \tag{5.26}$$

For ignoring preemption entirely, the worst-case non-preemptive blocking is defined by the most prolonged task running on the same PU as shown in Eq. 5.27.

$$B_i^{np,+} = \max_j C_{i,x} : M^P_{\tau_i} = M^P_{\tau_j} = x \wedge pt_j = np \text{ (is non-preemptive)} \tag{5.27}$$

Eq. 5.27 usually applies to OS tasks following AUTOSAR and the blocking time is outlined here for reasons of integrity, but the investigated models (cf. Chapter 6) do not include such model entities. In a mixed-preemptive environment, both above outlined blocking times are combined to $B_i^{mp} = \max_{j \in lp(i)}(\max(B_j^{np}, B_j^{npc}))$. Both cooperative and non-preemptive blocking delays must be accounted during RTA via an increased level-i window and adjusted equations for task start and end times as provided in [62], which is analyzed in the following Section 5.4.2. Generally, since only either local resource, cooperative, or non-preemptive blocking can interfere locally with a task, Eq. 5.26 and Eq. 5.27 are included in Eq. 5.23 as additional parameters of the outer max term $B_i^{pi,+} = \max\left(\max_{j \in lp(i)} w^{\Theta}_{CS}(\tau_j), B_i^{npc,+}, B_i^{np,+}\right)$.

The following Example 5.2 gives an overview of the typical pi- and s-blocking delays.

[42] Labels mapped to LRAM_x can be accessed concurrently from PU P_x and the crossbar

Example 5.2: Local and Global CS Blocking

Given are three tasks running on the same PU with descending priority $\mathcal{T} = \{\tau_H, \tau_M, \tau_L\}$, with arrival times $\nabla_H = 3, \nabla_M = 2, \nabla_L = 0$ and activity graphs $ag_H = \{tick, tick, CS2, tick\}, ag_M = \{tick, CS1, tick\}, ag_L = \{tick, CS1, CS1, CS1, tick, tick, CS2\}$. Figure 5.2 shows the Gantt chart for the execution of these three tasks. Arrows pointing up denote a task arrival.

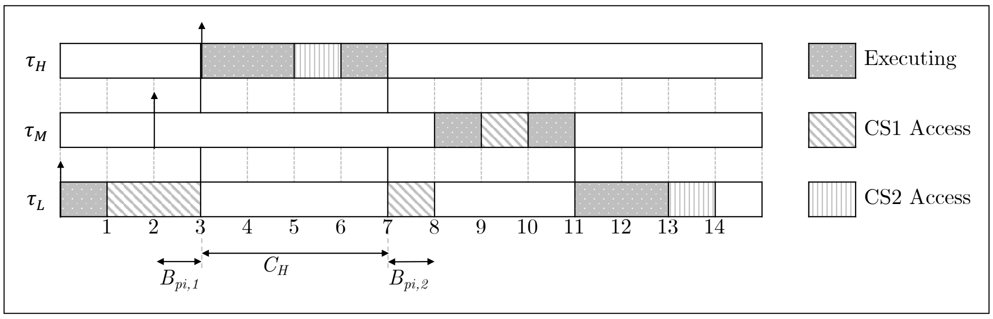


Figure 5.2: Local blocking example

Here, the C_H length is the direct preemption of τ_L by τ_H, which is part of the RTA calculation and also affects the response time of τ_M. B_1^{pi} is the local blocking of τ_M by τ_L, which continues after the preemption C_H as B_2^{pi}. The example shows a total pi-blocking to τ_M of two ticks caused by τ_L. The worst-case situation, which results in $B_{\tau_M}^{pi} = 3$, arises with τ_M arriving one tick earlier. Since CS_2 is accessed by at most a single tick by τ_L, the total local blocking imposed to τ_H is 1 correspondingly.

The same example is shown in the following Figure 5.3, with the minor derivation that all tasks are running on different homogeneous PUs ($\tau_H \rightarrow P_1, \tau_M \rightarrow P_2, \tau_L \rightarrow P_3$). Here, task τ_H is preempted on P_1 for one time instant after one tick due to an additional hypothetical task on P_1. This adaption has been made to show the worst-case blocking situation to τ_H, which is imposed by τ_L at time instant 6 for a single tick. Furthermore, τ_M is blocked by τ_L for one cycle at time instant 3. Since τ_L actually occupies CS_1 for three cycles, the worst-case global blocking to task τ_M is $B_{\tau_M}^{s,+} = 3$.

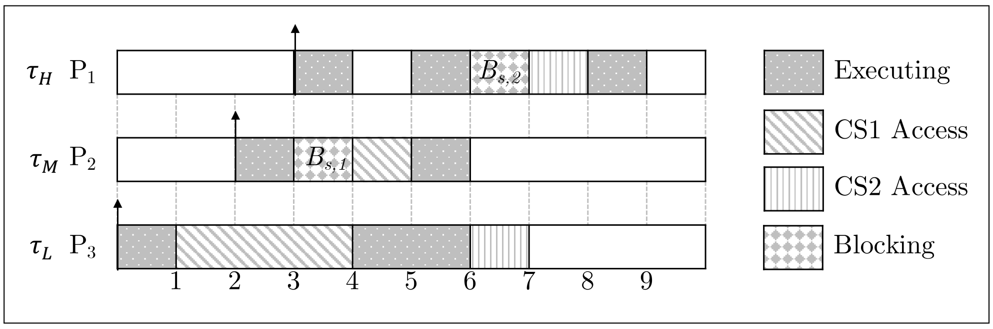


Figure 5.3: Global blocking example

Finally, pi- and s-blocking as well as memory contention delays are accumulated for retrieving a single blocking delay value imposed to a task via Eq. 5.28.

$$B_i^{+} = B_i^{pi,+} + B_i^{s,+} + B_i^{mc,+} \tag{5.28}$$

Equations 5.23–5.27, i.e. memory contention latency and blocking values, are considered as B_i during the task mapping DSE process as part of the RTA for CPUs, which is further outlined in the next Section 5.4.2 (Eq. 5.34). These techniques are used in Autosar for ensuring mutual exclusive memory access across PUs globally and tasks locally. According to GPUs, a dedicated model is presented in Section 5.6 Eq. 5.56.

5.4.2 CPU Response Times and Schedulability

To formally verify that all tasks meet their deadlines, formal methods can be used to derive WCRTs that must be smaller than the tasks' deadlines, as shown in Eq. 5.29. These formal methods include appropriate RTA that can be applied to Amalthea models as outlined in the following.

$$\begin{aligned} \forall i &: R_i^{+}(\mathcal{S}) &\leq D_i \\ \forall z &: R_{i,z} &\leq D_i \end{aligned} \tag{5.29}$$

Here, $\mathcal{S}$ denotes a solution, i.e. a runnable to task partitioning, task to PU mapping, and label to memory mapping (cf. Definition 5.11). An important assumption for calculating response times as well as event chain latency values across a networked environment is the incorporation of varying instruction values (cf. Definition 3.2). This becomes of major importance not only when considering heterogeneous PUs, but especially for executing complex image processing applications, which significantly benefit from GPUs regarding execution time.

As stated in Section 2.6.2, the RTA used in this thesis is based on the critical instant introduced by Liu and Layland in [53], extended first by Lehoczky for arbitrary deadlines in [38], later on by Tindell and Clark in [136] for holistic RTA, and by Keskin et al. in [225] for preemption threshold scheduling, which is incorporated via its revision of Buttazzo et al. in [59]. New adjustments of this thesis include the incorporation of precise label access delays Eq. 5.30, copy operations caused by implicit and LET communication paradigms Eq. 5.32, and various blocking types of Eq. 5.28 into Eq. 5.33.

Before the RTA methodology is presented, some adaptations to a task's execution time must be outlined first. Based on several aforementioned assumptions, a task's actual execution time not only depends on (a) its instructions on a PU (cf. Eq. 3.6), but also on (b) the latency of accessing data, (c) the contention that may occur for accessing unprotected shared resources, and the blocking of either (d) cooperative (or non-preemptive) lower priority tasks or (e) critical section accesses. The execution time for instructions is covered by $C_{i,x}$ whereas the factor (c) is accounted for in B_i. Hence, only the label access latency must be defined since neither communication costs cc_x nor edge costs $e_c(\varphi)$ presented in Eq. 5.10 and 4.6 account access delays for a single tasks based on a communication paradigm. Accordingly, Eq. 5.30 is presented, whereas the communication paradigm $com \in (\epsilon, \iota, \lambda)$ can be either explicit, implicit, or LET-based. Implicit and LET paradigms both copy data into local memory at the beginning of their execution and write back labels either at the end of the task execution for implicit communication or at the end of the period for LET. Thus, the label access costs for implicit

(ι) and LET (λ) communication shown in Eq. 5.30 must be advanced with copy in/out latency parameters $L_{cio,i}$, which is further shown in Eq. 5.32. Equations 5.30–5.32 assume $M^P_{\tau_i} = x$; m_{dl} being the local memory of P_x ; $M^m_{l_v} = d$; and $M^m_{l_w} = d'$.

$$L_i(com) = \begin{cases} \sum_{l_v \in \uparrow_{\tau_i}} \left(\uparrow^{\#}_{i,v} \cdot \uparrow_{x,d} \cdot \left\lceil \frac{ls_v}{cl_x} \right\rceil \right) \\ \quad + \sum_{l_w \in \downarrow_{\tau_i}} \left(\downarrow^{\#}_{i,w} \cdot \downarrow_{x,d'} \cdot \left\lceil \frac{ls_w}{cl_x} \right\rceil \right), & \text{if } com \text{ is } \epsilon \\ L_{cio,i} + \sum_{l_v \in \uparrow_{\tau_i}} \left(\uparrow^{\#}_{i,v} \cdot \uparrow_{x,dl} \cdot \left\lceil \frac{ls_v}{cl_x} \right\rceil \right) \\ \quad + \sum_{l_w \in \downarrow_{\tau_i}} \left(\downarrow^{\#}_{i,w} \cdot \downarrow_{x,dl} \cdot \left\lceil \frac{ls_w}{cl_x} \right\rceil \right), & \text{if } com \text{ is } \iota \vee \lambda \end{cases} \tag{5.30}$$

If an access delay (e.g. read $\uparrow_{x,d}$) is given in instructions, its time value is calculated just as for task instructions as given in Eq. 3.6, e.g., 50 instructions on a 200MHz PU with 1 instructions per cycle: $\uparrow_{x,d} = \frac{50}{200*10^6 Hz} = 250ns$. $L_i(com)$ can further be normalized to one second, denoted $L^s_i(com)$ by considering a task's period as shown in Eq. 5.31.

$$L^s_i(com) = L_i(com) \cdot \frac{10^{12}}{T_i} \text{ with } T_i \text{ in } ps \tag{5.31}$$

Since LET and implicit communication induce additional copy operations, Eq. 5.32 provides the delay calculation for (i) reading read labels from any memory location they are mapped to, (ii) writing the same into local memory, (iii) reading all written labels from local memory, and (iv) writing the same to their original location.

$$L_{cio,i} = \sum_{l_v \in \uparrow_{\tau_i}} \left((\uparrow_{x,d} + \downarrow_{x,dl}) \cdot \left\lceil \frac{ls_v}{cl_x} \right\rceil \right) + \sum_{l_w \in \downarrow_{\tau_i}} \left((\uparrow_{x,dl} + \downarrow_{x,d'}) \cdot \left\lceil \frac{ls_w}{cl_x} \right\rceil \right) \tag{5.32}$$

Given all possible blocking delays, data access costs based on different communication paradigms, and execution times derived from an instruction set of a task, the total worst-case execution for a task on a CPU $C^{+,CPU}_{i,x}$ can be calculated, which is shown in the following Eq. 5.33. Finally, the total WCET influenced by (a)–(e) is shown in Eq. 5.33.

$$C^{+,CPU}_{i,x} = C^+_{i,x} + B^+_i + L_i(com) \tag{5.33}$$

Based on Eq. 5.33, the first constraint covers the feasibility test shown in Eq. 5.9 based on a utilization constraint, implemented to quickly sort out invalid solutions during the DSE process.

Classical RTA for mixed-preemptive fixed-priority tasks using recurrence relation [62] incorporates mapping decisions and varying instruction sets correspondingly as shown in Eq. 5.34 with $R^{0,+}_{i,x} = C^{+,CPU}_{i,x}$.

$$R^{z,+}_{i,x} = C^{+,CPU}_{i,x} + \sum_{h \in hp(i)} \left\lceil \frac{R^{z-1,+}_i}{T_h} \right\rceil \cdot C^{+,CPU}_{h,x} \tag{5.34}$$

As defined in Section 3.2, $R^+_{i,x}$ denotes the WCRT of task τ_i on PU P_x. However, Eq. 5.34 requires that the worst-case situation happens at the critical instant, which denotes the simultaneous release of all higher priority tasks at the same time the task under investigation is released. The critical instant can be used with RMS, but when considering arbitrary deadlines, the level-i busy period method is required, expressed in the following section.

WCRT for Arbitrary Deadline Fixed Priority Mixed-Preemptive Scheduling

Based on Lehoczky's work [38], which is applied to AMALTHEA in [62], and revised for mixed-preemptive task sets in [59], the WCRT for Fixed-Priority Mixed Preemptive Scheduling (FPMPS) using arbitrary deadlines, i.e. a task's deadline and response time is allowed to be greater than its period $D_i \geq T_i, R_i \geq T_i$, can be calculated using the level-i busy period window, which assumes that the worst-case response time may not necessarily occur at the critical instant but within the busy period. Based on Eq. 5.33, the level-i busy period length is derived by Eq. 5.35 using recurrence relation and the initialization $w_{i,x}(0) = B_i^{mp} + C_{i,x}^{+,CPU}$. This includes the local and global blocking as well as memory contention accounted for with $C_{i,x}^{+,CPU}$ as of Eq. 5.33 and the blocking delays of cooperative and non-preemptive tasks via B_i^{mp}.

$$\begin{aligned} w_{i,x}(q) =& B_i^{mp} + \sum_{h \in hep(i)} \left\lceil \frac{w_i(q-1)}{T_h} \right\rceil \cdot C_{h,x}^{+,CPU} \\ &\text{with } q \in [1,n] \cap \mathbb{N} \ : \ w_{i,x}(n) = w_{i,x}(n-1) \end{aligned} \tag{5.35}$$

Here, $hep(i)$ includes all tasks (task indexes) of higher priority than τ_i and τ_i itself too (hep = higher and equal priority). Consequently, the number of task jobs that need to be checked for the worst-case situation is $Z_i = \left\lceil \frac{w_{i,x}(q)}{T_i} \right\rceil$. During the busy period $w_{i,x}(q)$, the worst-case finish time of a task instance (job) is calculated using Eq. 5.36.

$$f_{i,x}^{+}(z) = \sum_{h \in hp(i)} \left\lceil \frac{f_i^{+}(z-1)}{T_h} \right\rceil \cdot C_{h,x}^{+,CPU} + z \cdot C_{i,x}^{+,CPU} \tag{5.36}$$

And finally, the worst-case response time for a fixed priority mixed preemptive task with an arbitrary deadline is defined in Eq. 5.37 as the maximal time interval between the job's release and finish time over all jobs within the busy period.

$$R_{i,x}^{+,CPU} = \max_{z \in [1,Z_i]} \left(f_{i,x}^{+}(z) - (z-1) \cdot T_i \right) \tag{5.37}$$

Equations 5.35–5.37 are derived from [62], respectively [59], and depend on a partitioning as well as task and label mappings (cf. Def. 5.11). The initialization of $f_{i,x}^{+}(0); s_{i,x}^{+}(0)$ are in line with the references, i.e., $f_{i,x}^{+}(0) = s_{i,x}^{+} C_{i,x}$ and $s_{i,x}^{+}(0) = B_i^{mp} + \sum_{h \in hp(i)} C_h$, respectively. The advancement over [62] and [59] presented here is the fine grained incorporation of various resource blocking delays ($B^p i, B^s, B^{mc}$) and the application to a multi-PU ECU network (B^{mp} considers PU mapping and hence applies to an entire AUTOSAR system rather than a single PU) based on data available through AMALTHEA.

For mixed-preemptive tasks, the level-i window length and the finish time calculation are added with blocking delays by the maximal lower priority non-preemptive runnable B^{npc} or task B^{np}, which has also been used in [62]. Non-preemptive sections may result in the *self-pushing-phenomenon*, i.e., high priority tasks being released during a non-preemptive lower priority task pushing everything ahead such that higher delays of subsequent tasks arise. Hence, not only the level-i busy window is required, but also a slight adjustment of response time analyses based on preemption thresholds [62], denoted ϖ so that Eq. 5.36 is advanced to Eq. 5.38.

$$\begin{gathered} f^{+}_{i,x}(z) = s^{+}_{i,x}(z) + C_{i,x} + \sum_{h \in \varpi(i)} \left(\left\lceil \frac{f^{+}_{i,x}(z-1)}{T_h} \right\rceil - \left(\left\lfloor \frac{s^{+}_{i,x}(z-1)}{T_h} \right\rfloor + 1 \right) \right) C_{h,x} \\ \text{with } s^{+}_{i,x}(z) = B^{mp}_i + \sum_{h \in hp(i)} \left(\left\lfloor \frac{s^{+}_{i,x}(z-1)}{T_h} \right\rfloor + 1 \right) C_{h,x} + z \cdot C_{i,x} \end{gathered} \tag{5.38}$$

Assigning equal thresholds to tasks based on the highest priority across all cooperative tasks results in the original Eq. 5.36. However, as shown in [59], a careful threshold assignment is necessary to avoid deadline misses, e.g. the approach by Saksena and Wang in [226].

5.5 Task Chain Latency Analyses

Task Chain Latency Analysis (TCLA) and corresponding bounds on worst-case data propagation and cause-effect chains is just as important as meeting task deadlines for automotive systems. Therefore, related work, assumptions, and definitions are given in the following Section 5.5.1 and AMALTHEA-based approaches to calculate the various latency values are given in Section 5.5.2. Assumptions and definitions include implicit and LET communication paradigms, which are part of the AUTOSAR-Specification [51], as well as data sampling, reaction, propagation, and age delay interpretations.

5.5.1 Related Work, Assumptions, and Definitions on TCLA

The determination of cause and effect chain delays, which is an AUTOSAR timing extensions term [94], can stretch across abstractions like events, tasks, and even runnables, as well as call graph items. While end-to-end latency analysis is in general reasonable and possible across all these abstractions, task chains are in focus in the following, due to typical automotive requirements mostly refer to tasks such that, e.g., a task chain between a sensor task to an actuator task with all processing in between must take no longer than x time units. The motivation of TCLA stems from vehicle body electronics, at which reaction delays have to be strictly bounded and verified, as well as typical control engineering applications that necessitate the verification of data age delays [227]. However, related work often uses varying semantics for the same or similar entities. For instance, Feiertag et al. define in [227] a *reaction delay* as an interval between the first initiator and the first responder of a task chain and an *age delay* as the time interval between the last initiator and the last responder of a task chain. A valuable contribution has been presented by Becker et al. in [228] that identifies the data age constraints as "the most complex timing requirement in these systems". Becker also considers explicit, implicit, and LET communication for task chains.

In this thesis and in line with the semantic of the word *chain*, a task chain γ_g represents an ordered sequence of tasks on the same PU (cf. Eq. 5.39) such that no task appears twice in the chain and for every two subsequent tasks $\tau_{g,j}$, $\tau_{g,j+1}$, a dependency exists such that at least one label is written by $\tau_{g,j}$ and read by $\tau_{g,j+1}$, i.e. $\forall j \in [1, |\gamma_g| - 1] \ : \downarrow_{\tau(\gamma_{g,j})}$

$\cap \uparrow_{\tau(\gamma_{g,j+1})} \neq \emptyset$ (cf. *path*, Eq. H.13).

$$\begin{aligned}\Gamma =& \{\gamma_1, ...\}\\ \gamma_g =& \{\tau_{g,1}, ...\}\\ \tau_{g,j} =& j\text{-th task in task chain } \gamma_g\end{aligned} \tag{5.39}$$

Consequently, in contrast to a task DAG, a task chain is a serial sequence of tasks with no fork or join dependencies. Accordingly, a task chain also represents data propagation and a cause-effect chain. This definition is in line with [62, 87, 115, 214, 228–230]. Davare et al. differ slightly in [231] by allowing different paths, but the proposed calculation stays the same by adding up all periods and response times. Schlatow et al. mention in [230] that task chains do not necessarily have to propagate data through labels and can be arbitrary. In general, Schlatow et al. are correct with the assumption, and concepts presented in this thesis generally also apply to task chains without data propagation. However, the main purpose of timing constraints for task chains is to bound data propagation delays and hence cause-effect chains, which form important requirements for developing automotive systems and hence are of concern in this thesis. In contrast to [227], the work presented in this thesis assumes a single (distinct) initiator for a task chain instance.

Definition 5.1: Task Chain Initiator

A task chain initiator is the first task of a task chain (entry task).

$$\tau_{g,1} = \text{ initiator of } \gamma_g = \{\tau_{g,1}, \tau_{g,2}, ...\} \tag{5.40}$$

A task chain instance is initiated by its initiator, and as a consequence, the number of initiated task chain instances within an arbitrary time interval t equals the number of its initiator's periods, i.e. $z(\gamma_g, t) = \left\lfloor \frac{t}{T_{\gamma_{g,1}}} \right\rfloor$. Potentially, every task chain initialization can propagate data to the task chain's responder (cf. Definition 5.2) but over- (cf. Definition 5.4) and under-sampling (cf. Definition 5.3) may result in *newer or more recent data* that dominates the former task chain instance.

Definition 5.2: Task Chain Responder

A task chain responder is the last task of a task chain (exit task).

$$\tau_{g,n} = \text{ responder of } \gamma_g = \{\tau_{g,1}, \tau_{g,2}, ..., \tau_{g,n}\} \textit{ with } n = |\gamma_g| \tag{5.41}$$

Each of the task chain instances can have multiple responding tasks, such that early and late reaction delays can be derived. Related work also often refers to event [115, 232], cause-effect [62, 228] or data chains [229] rather than task chains [214, 230]. However, the concepts mostly apply to the TCLA of this thesis, since just the abstraction level or notation differs. Data sampling denotes the propagation of data between two entities, e.g., tasks. Due to the fact that tasks can have varying periods across the task chain, propagation between task chain entities can be over- or under sampled such that a task's result (a) serves as an input for several subsequent task chain entity instances or (b) does not serve as an input at all due to the fact that the subsequent tasks already use newer results produced by new task instances. Corresponding definitions are given in Definition 5.3 and 5.4.

Definition 5.3: Under Sampling

The situation when a source task is executed more than once before its successor executes is denoted under-sampling.

Under-sampling results in some data produced by a source task not being used throughout the rest of the task chain at all, since newer data produced by the same task is already available.

Definition 5.4: Over Sampling

When a target task executes more than once before its predecessor provides new results, data produced by the predecessor task is oversampled, and the situation is denoted oversampling.

Oversampling results in the same data being used multiple times by a target task and is the only reason that reaction and age latency values of a task chain can differ.

As a next step, latency types must be defined, i.e. reaction (cf. Definition 5.5) and age latency (cf. Definition 5.6).

Definition 5.5: Task Chain Reaction Latency ρ_g

The time between a task chain's initiator arrival to the $\underline{\textit{first}}$ task response of the task chain's responder is denoted as task chain reaction latency. This delay can differ for each task chain instance, such that the worst and best-case latency values can be analytically derived.

In other words, a task chain's reaction time latency is the earliest time data traverses a task chain from an entry task to an exit task. By analyzing a task chain's worst reaction latency based on Definition 5.5, time-critical delays can be verified, such as the delay between a brake pedal sensor activation and a brake actuator task.

Definition 5.6: Task Chain Age Latency α_g

The time between a task chain's initiator arrival to the last task response of the task chain's responder is denoted as task chain age latency. The next execution instance (job) of the task chain's responder uses data produced by a newer, i.e., later, task chain initiator.

When analytically bounding the worst-case task chain age latency, a responder task can be guaranteed to use data produced at most the specified time value ago. Definitions 5.5 and 5.6 are in line with [115, 228] but not with [30, 227, 232] since the latter use multiple initiators.

Definition 5.7: Data Age Latency α_{l_v}

The duration a label version persists in memory is denoted as data age latency.

A data age constraint is typically applied for, e.g., control engineering applications, to guarantee that data is regularly updated within at least the specified time bounds. Usually, data (a label) age latency is defined by the lowest period of tasks writing to the label shown in Eq. 5.42.

$$\alpha_{l_v} = \min_i T_i \ : \ l_v \in \downarrow_{\tau_i} \tag{5.42}$$

If the label under consideration is written by more than one task, the interleaving periods of tasks writing to the label usually result in a significantly lower average data age delay. For all data age constraints, the analyzed data age must stay below a specified value as shown in Eq. 5.43.

$$\Phi_{l_v,\downarrow} = \{l_v, t_{spec}\} : \alpha_{l_v} \leq t_{spec} \tag{5.43}$$

The three latency types data aging ($\alpha_{l_v,\iota}$ assuming $l_v \in \downarrow_{\tau_i}$), task chain reaction ($\rho_{g,\iota}$), and task chain aging ($\alpha_{g,\iota}$) are shown exemplary for implicit communication in Figure 5.4 based on an example $\mathcal{T} = \{\tau_i, \tau_j, \tau_k\}; \gamma_g = \{\tau_k, \tau_j, \tau_i\}; T_i = 15; T_j = 20; T_k = 30$. Further Gantt charts based on the same example are given in the appendix at Section H.7 for outlining different communication, worst- and best-case, and varying period situations.

For deriving a task chain end-to-end latency, assumptions towards communication paradigms must be made, which define at what point in time data is read from and written to memory. Although AUTOSAR provides three major paradigms in general, namely explicit, implicit, and LET communication, only the latter two are considered in this thesis due to being more used in industrial applications and their benefit of eased timing analysis. While explicit communication allows random access to labels at any point in time, implicit communication strictly reads all labels at the beginning of the accessing entity's execution and writes labels at the end of its execution. LET spreads these memory accesses even further as labels are not written at the end of an entity's execution, but at the end of its period. These paradigms are proposed as challenges in [31] and have been studied in, e.g., [34, 212].

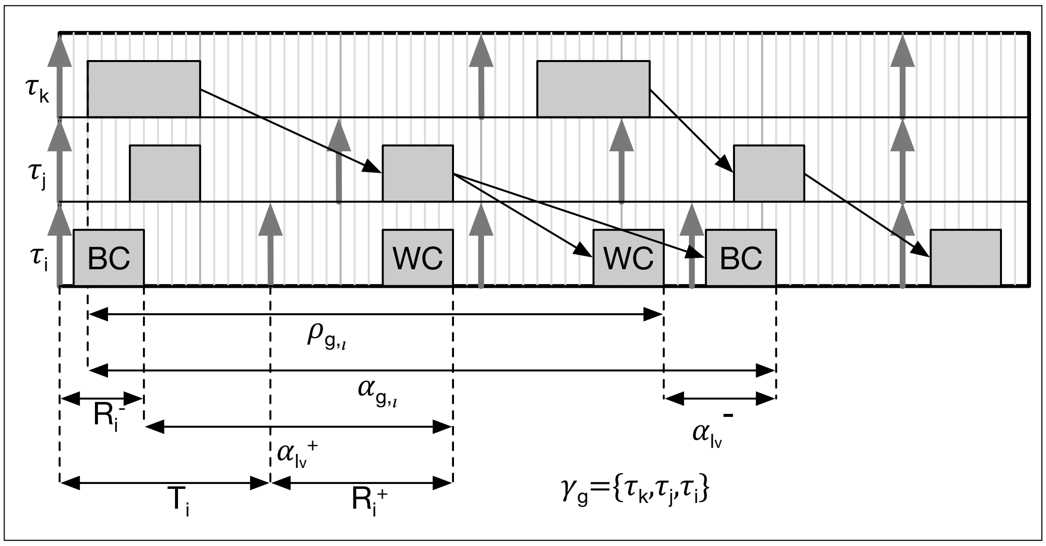


Figure 5.4: Example Gantt chart on task chain reaction and aging as well as data aging latency

Definition 5.8: Implicit Communication ι

Implicitly communicating entities, e.g., tasks, copy all read labels into local memory at the beginning of their execution (release time) and write written labels into shared memory by the time their instructions finished execution.

Implicit communication follows read-execute-write semantic [31].

Definition 5.9: LET Communication λ

To ease data propagation analysis, tasks under LET communication always read labels at a task's arrival time and write written labels after the task's period, i.e., at the arrival of the task's next job.

Explicit data accesses result in high uncertainty about data propagation across tasks, runnables, or task chains. No assumptions are made such that data accesses can occur at arbitrary points in time. Due to having no memory copy overheads, explicit communication can result in rapid data propagation across tasks sharing data. However, the crucial disadvantage of explicit communication is the aggravated analysis of data propagation along cause and effect chains over arbitrary time intervals. Hence, the AUTOSAR consortium decided to use implicit and LET communication, such that explicit data access analyses are omitted in this thesis. TCL assessment for LET, which decouples computation and communication, is targeted in [31] as part of the WATERS FMTV challenges, and addressed in [32–34, 212] with the result that the overhead imposed by LET is neglectable. LET significantly reduces latency deviation and jitter, and has been used in the industry [212]. Isolating computation from communication is a practical approach to reduce shared cache conflicts, bus contention, shared cache contention, buffer conflicts, and request reordering processes in the memory controller [233].

5.5.2 Identifying Task Chain Latency Bounds

This section outlines calculations required to bound age and reaction delays of task chains for implicit and LET communication. Examples on various worst and best-case situations regarding LET and implicit communication for age and reaction delays are presented in the appendix Section H.7 along with three tasks in Figures H.2–H.11. More precisely, Figures H.2–H.11 show Gantt charts for task chains, respectively **A**: worst-case age latency on increasing periods and implicit communication; **B**: early worst-case reaction latency on increasing periods and implicit communication; **C**: best-case reaction and worst-case age latency on decreasing periods and implicit communication: **D**: worst-case age and reaction latency on decreasing periods implicit communication; **E**: worst-case age latency on alternation periods and implicit communication; **F**: worst-case reaction latency on decreasing periods and LET communication; **G**: best-case reaction latency on decreasing periods and LET communication; **H**: worst-case reaction latency on increasing periods and LET communication; **I** worst-case age and reaction latency on alternating periods and LET communication; and **J**: best-case reaction latency on increasing periods and LET communication.

The association between Figures H.2–H.11' examples, latency types, and communication paradigms is given in Table 5.1.

	ρ_ι^+	ρ_ι^-	α_ι^+	ρ_λ^+	ρ_λ^-	α_λ^+
γ_1	B / H.3	C[43] / H.4	A / H.2	H / H.9, I[44]	J / H.11	H / H.9
γ_2	D / H.5	C / H.4	E[45] / H.6, C	F / H.7	G / H.8	I[44] / H.10

Table 5.1: Association between Figures H.2–H.11's task chain examples and corresponding latency types as well as communication paradigms

Figures H.2–H.11 assume implicit deadlines derived from task periods, and response times being arbitrary, but lower than the period. Black arrows denote worst-case data propagation, light gray arrows represent data propagation of a different task chain instance, dotted arrows denote best-case data propagation, and dashed arrows represent the propagation of data produced by a succeeding task instance that dominates data of the task chain in focus. The latter situation is shown in Figure H.9 and increasing periods can be observed that result in over-sampling and not every task chain instance provides a distinct output. Furthermore, diagonal filled boxes indicate the execution of a task at a specific point in time, necessary for the corresponding task chain to be valid, i.e. a worst-case situation to not dominate data used in the task chain under analysis (e.g. (C): the second job of task τ_k must execute at the end of its period such that τ_j uses data from τ_k produced $2 \cdot T_k$ ago).

The best-case reaction delays are presented first. Since best-case age delays are not

[43]The best-case reaction delay is the same across varying task periods.
[44]The period of τ_j *H.9*/ *H.10* has been reduced to 10 to show a difference in α and ρ.
[45]The period of τ_k is used twice (replaces τ_i) to show the situation of alternating periods.

of interest and do not provide meaningful information to be used during developing automotive systems, corresponding analysis is omitted here. The **best-case task chain reaction latency for implicit communication** can be calculated by considering the sum of all task's best-case response times within task chain $\gamma = \{\tau_0, ...\}$, as given in Eq. 5.44.

$$\rho^-_{g,\iota} = \sum_j R^-_j + 1 \ : \ \tau_j \in \gamma_g \tag{5.44}$$

The best-case task chain reaction latency is shown in Figure H.4.

For LET communication, the calculation (**best-case task chain reaction latency based on LET**) is similar to Eq. 5.44, except replacing the response time with the period, since communication via shared data only takes place at each task's period. The equation is shown in Eq. 5.45 for Figures H.8 and H.11 accordingly.

$$\rho^-_{g,\lambda} = \sum_j T_j + 1 \ : \ \tau_j \in \gamma_g \tag{5.45}$$

To guarantee that written data is read by a task chain's successor, the best-case equations add a single time value (+1), which is also shown in a single time value between tasks in Figures H.4, H.8, and H.11.

The **worst-case task chain reaction latency** value calculations are nearly the same for **implicit** and LET communication as shown in Eq. 5.46 and 5.47, whereas the latter replaces the response time with the period due to the same reason mentioned for Eq. 5.44 and 5.45.

$$\rho^+_{g,\iota} = \sum_j R^+_{g,j} + \sum_{j=2}^{j=|\gamma_g|} (\min(T_j, T_{j-1})) \tag{5.46}$$

Eq. 5.46 assumes that data is read at the beginning of a task's arrival since the jitter between arrival and release can vary for every task job and hence a more tight task chain reaction latency bound can not be provided. Compared with the calculation of task chain reaction delays presented in [234], Eq. 5.46 uses $min(T_j, T_{j-1})$ instead of T_j only, since data propagation is not only constrained by a single task, but also its predecessor, whereas the worst-case situation depends on the lower period, e.g., the first task τ_i shown in Figure H.3 on the one hand, but also the second task τ_j in H.5, on the other hand. Additionally, Eq. 5.46 does not consider best-case response times as done in [83] (from all found reaction calculations, [83] is the closest to equations used here), as the worst-case reaction slightly differs from (last-to-first) L2F semantics in [83]. In fact, [227] and [83] use different semantics, especially for L2F, L2L since the former refers to specific input and output intervals and the latter to data propagation. This thesis uses the reaction notation defined in [115], which defines the time between a task chain's initiator arrival and the first responder's finish time. This definition is also denoted as L2F in [83]. An age latency is the last finish time of a task chain's responder before the next responder job uses data produced in a newer task chain instance. This definition is denoted as L2L in [83].

The **worst-case task chain reaction latency for LET** communication assumes that for every two consecutive tasks in the task chain, the target task just started execution when the source task wrote its results. As a consequence, every result (task write process) is delayed by two periods, except for the initiator task, which is accounted for with a single period as shown in Eq. 5.47.

$$\rho^+_{g,\lambda} = T_{g,1} + \sum_{j=2}^{j=|\gamma_g|} (2 \cdot T_j) \text{ with } \tau_j \in \gamma_g \tag{5.47}$$

Equations 5.46 and 5.47 hold for Figures H.3 and H.5 for implicit communication as well as H.7, H.9, and H.10 for LET.

Next, the most straightforward approach of getting the **worst-case task chain age latency** is to consider two consecutive task chain instances and assume the worst-case reaction for the second one. Then, by adding the period of the initiator to the worst-case reaction delay and subtracting the response time of the responder, the worst-case age delay can be calculated, as shown in Eq. 5.48.

$$\alpha^+_{g,\iota} = T_{g,1} + \rho^+_{g,\iota} - T_{g,|\gamma_g|} \tag{5.48}$$

In Eq. 5.48, '$-T_{g,|\gamma_g|}$' comes from '$-R^+_{g,|\gamma_g|} - (T_{g,|\gamma_g|} - R^+_{g,|\gamma_g|})$'. As shown in H.2 and H.4, increasing or decreasing periods have no effect on the age latency derivation.

When calculating the **worst-case age delay for LET** communication, the response time must be replaced with the period for the responder task as well as the communication type for the reaction delay to LET, i.e. λ, from Eq. 5.48, as shown in Eq. 5.49.

$$\alpha^+_{g,\lambda} = T_{g,1} + \rho^+_{g,\lambda} - T_{g,|\gamma_g|} \tag{5.49}$$

Finally, instead of task chain propagation, data propagation delays are also of interest during the development of automotive systems. Therefore, entities reading a particular label and all succeeding entities that depend on those reading entities must be considered. For instance, in Figure H.2, any read label by task τ_i (e.g., label A) influences the entire task chain. Even if other entities in the task chain do not use the label, the results (other labels) can be influenced by the computation based on label A. Consequently, data propagation can be calculated for all entities reading the label set X, and all entities with any dependency to set X. This approach results in a vast amount of data propagation paths and requires DAGs, i.e., no cyclic dependencies. In general, every possible path of a task DAG could be transformed into a task chain and Eq. 5.44–5.49 can be applied to all of those paths to investigate data propagation on a holistic level.

Furthermore, data propagation can also be related to both, a specific task chain and dedicated data. If the label is read at the task chain's source entity, then the data propagation equals the task chain reaction latency. If the label is read in between tasks of the task chain, by e.g. Task τ_j from Figure H.2, the data propagation is lower than the task chain reaction, since it excludes T_i, respectively R^-_i, across equations Eq. 5.44, Eq. 5.45, and Eq. 5.46. Dedicated data propagation calculation is a matter of finding the shortest (best-case) and longest (worst-case) path over a DAG that consists of response time or period-weighted vertices. In this thesis, propagation delays are considered for task chains only and data-dedicated analysis is omitted here.

To add task chain age and reaction delay constraints to the solution space investigation of the mapping process (cf. Section 5.2), the following Eq. 5.50 is applied, which uses Φ_{γ_g} across age α and reaction ρ delays as well as implicit ι and LET λ communication in the form of a specified deadline for the task chain γ_g in, e.g., picoseconds.

$$\forall g \ : \ \begin{cases} \rho^{+}_{\gamma_g,\iota} \leq \Phi^{\rho}_{\gamma_g,\iota} \\ \rho^{+}_{\gamma_g,\lambda} \leq \Phi^{\rho}_{\gamma_g,\lambda} \\ \alpha^{+}_{\gamma_g,\iota} \leq \Phi^{\alpha}_{\gamma_g,\iota} \\ \alpha^{+}_{\gamma_g,\lambda} \leq \Phi^{\alpha}_{\gamma_g,\lambda} \end{cases} \tag{5.50}$$

In addition to task chain delays, runnable event chains may also be of interest and provide a fine-grained analysis of data propagation [34]. A runnable event chain reaction latency defines the time it takes for data (i.e. a label or set of labels) to propagate to the end of the chain, respectively from the initiator runnable to the responder runnable (notation in line with Definition 5.1 and 5.2). However, [34] is based on a different assumption regarding propagation, reaction, and age latency values. In line with [115], a runnable chain latency only differs from TCL results when assuming explicit communication. For implicit and LET communication, data only propagates at task response time or period bounds, which includes all data accessed by runnables, which are part of the task's activity graph. Hence, runnable event chain delays are out of scope here. For explicit runnable communication and corresponding delay analyses, the approach of Martinez et al. [34] can be used.

5.6 GPU Timing Verification

Recent development activities in the automotive domain have risen challenges when applying RTA, memory contention analysis, and data access latency estimation to AUTOSAR compliant models, including GPUs. The WATERS community published a corresponding challenge in 2019 [30]. This community established along with the ECRTS conference and has been working on solving various challenges since 2010, whereas specifically automotive challenges are outlined since 2015 [187], such as worst-case end-to-end latency derivation along complex cause-effect chains [128], communication paradigms [31], WCET / WCRT for advanced shared memory architectures [62], optimized application mapping, and sophisticated models for multi-core execution platforms. Mapping tasks to PUs to optimize various metrics increase complexity and require appropriate advancements to conventional formal RTA methods. Metrics have to address a) task chain reaction and aging delays, b) response times under the consideration of costs induced by offloading instructions to GPUs and copying data to GPU dedicated memory regions across highly heterogeneous hardware architectures with different memory types, processing speeds, peripherals, accelerators, and more, and c) the use of sophisticated memory contention models. The proposed challenge [30] can be tackled by advanced RTA for mixed-preemptive tasks under FPMPS running on CPUs using the windowing technique of [38] and [225] combined with WRR RTA for GPU tasks [36]. Additionally, delays produced by GPU CE operations and differences of asynchronous and synchronous GPU offloading mechanisms must be accounted for. The challenge has been addressed in [87, 88, 214], and [19], whereas the latter is presented in the following.

Instead of using WRR scheduling RTA for the GPU, Capodieci et al. exemplary show in [235] how EDF policies can be applied to the Nvidia TX2 platform, which is in the focus of the WATERS2019 challenge. They also show that the use of GPUs in automotive systems is reasonable and provides significant benefits over using CPUs only. They compare the schedulability ratio with the original Nvidia scheduler as a function of task set utilization and different time slice lengths. Therefore, the *host*, i.e., GPU dispatcher that schedules copy, computation, and graphic engine operations, is triggered by a new software scheduler to implement EDF policies. Results show that trading performance for real-time requirements significantly increases schedulability. Unfortunately, contributions are not publicly available, so that approaches to derive response times presented here remain in line with [30], i.e., WRR as well as NVidia TX2 Rule-based Scheduling (TX2RS) based on [236]. The latter forms an alternative approach to WRR and is based on information revealed in [236]. This alternative is presented in Section 5.6.2 using the Nvidia-specific scheduling rules to avoid invasive software dispatching in addition to the host scheduler as of [235].

Recent research also identifies various pitfalls for executing time-sensitive tasks on a GPU [237]. Especially explicit (user-defined) and implicit (API defined) synchronization causes unclear blocking delays due to kernels waiting for others to start, operations preventing concurrent kernel execution, or CPU tasks being blocked even though asynchronous offloading is issued. Although Multi-Process Service (MPS) overcomes many of those issues as stated in [237], MPS is not available for the Nvidia TX2 platform such that work of this thesis assumes running tasks as user-defined streams in a shared address space. Memory operations such as *allocation*, *set*, or *copy* cannot run concurrently across streams on the Nvidia TX2 such that the WATERS2019 challenge assumes a single custom stream per task as well as memory operations being accounted within the tasks' GPU ticks.

The WATERS2019 challenge model is given as an AMALTHEA model[2] and can be accessed by the APP4MC [5] platform. By using above mentioned assumptions, the WCRTs for CPU tasks are calculated using the approach presented in Section 5.4.2 and GPU response times are calculated by using the methodology of the following Section 5.6.

Since (a) Eq. 5.25 already accounts all PUs accessing a label written by a CPU task denoted with index y and (b) the copy operation is handled by the dedicated CE, no additional costs must be added to CPU tasks in addition to ticks, preemption, blocking, and contention interference presented in Section 5.4.2 for CPU task response times.

5.6.1 Copy Engine

Before a GPU task starts executing, the CE needs to copy all accessed labels into the dedicated GPU region. The CE reads all labels accessed by a task from different memories, writes them into a dedicated GPU memory location of the global memory, and after the GPU execution finished, all labels are written back to their original location. Since this mechanism conceptually corresponds the copy operations for implicit and LET communication, equations 5.30 5.32 can be reused here by simply replacing m_{dl} with the dedicated GPU memory region.

The resulting data copy flow is then $CE_{\uparrow_{x(GPU),d(l_v)}} \Rightarrow CE_{\downarrow_{x(GPU),d(GPU)}} \Rightarrow GPU_{execution} \Rightarrow CE_{\uparrow_{x(GPU),d(GPU)}} \Rightarrow CE_{\downarrow_{x(GPU),d(l_v)}}$, i.e. accessed data is read and written for both their original location and the dedicated GPU region. Eq. 5.51 presents the delay imposed by reading labels to be copied from their original location m_d and writing them into GPU memory $m_{d'}$ as well as reading them from the same location $m_{d'}$ and writing them back to where they were read at the beginning m_d.

$$CE^{+}_{a,i} = \sum_{l_v \in \mathcal{L}_{\tau_i}} \left\lceil \frac{ls_v}{cl_{GPU}} \right\rceil \cdot \left(\uparrow_{GPU,d} + \downarrow_{GPU,d} + \downarrow_{GPU,d'} + \uparrow_{GPU,d'} \right) \text{ with } M^{m}_{l_v} = d \tag{5.51}$$

In addition to the actual copy operation time $C^{+}_{a,i}$, contention $CE^{+}_{c,i}$ and queuing $CE^{+}_{q,i}$ delays need to be considered. The former is shown in Eq. 5.52 as a function of a label set to be copied and shared globally as well as varying access delays from other PUs to the labels' original location (cf. $w_{CS}(\tau_j, l_v)$ Eq. 5.19).

$$CE^{+}_{c,i} = \sum_{l_v \in (\downarrow_{\tau_i} \cap CS^{\Phi})} \left(\sum_{x \setminus GPU} \max_{j: M^{P}_{\tau_j} = x} \left(w_{CS}(\tau_j, l_v) \right) \right) \tag{5.52}$$

Due to queuing copy operations at a single CE, no pi-blocking and only s-blocking (caused by a PU P_x in Eq. 5.52) affects the CE.

The CE queuing delay is derived from the maximal copy operations among tasks mapped to the GPU and triggered from a CPU, as shown in Eq. 5.53, which assumes FIFO-ordered CE queuing.

$$CE^{+}_{q,i} = \sum_{x \setminus M^{P}_{\tau_i}} \max_{j: M^{P}_{\tau_j} = x} CE^{+}_{j} \tag{5.53}$$

Given predefined label and task mappings and latency values from each PU to each memory, the total CE time that sums up label access, queuing, and contention delays is provided in Eq. 5.54.

$$CE^{+}_{i} = CE^{+}_{q,i} + CE^{+}_{c,i} + CE^{+}_{a,i} \tag{5.54}$$

Equations 5.51–5.54 assume that multiple read accesses can be executed concurrently, and write operations must always be mutually exclusive. During the implementation of the CE latency calculation, situations were identified, to which labels were written back to their original place, but not changed during the GPU execution. Consequently, a small adjustment of the CE access delay in Eq. 5.51 is implemented, which only accounts written, i.e., changed labels to be chosen for being written back to their original location.

Example 5.3: CE Time Calculation

Given is a task τ_1 that accesses a single label l_1 of size 128 Bytes located at memory m_1. The is offloaded to a GPU running at 2GHz. The GPU access delays are five read cycles and six write cycles for memory m_1, as well as seven read cycles and eight write cycles to access GPU memory. The CE access delay is then $CE_1 = \frac{\left(\frac{128}{64}\right)\cdot(5+6)}{2\cdot 10^9} + \frac{2\cdot(7+8)}{2\cdot 10^9} = 26ns$. Label access times during a task execution are accounted for within $C^{+,GPU}_{i}$. For instance, assuming that label l_1 is read three times and written twice during execution, label access latency $L_{\tau_1} = \frac{\left(\frac{128}{64}\right)\cdot(3\cdot 7+2\cdot 8)}{2\cdot 10^9} = 37ns$ is already account for within $C_{i,x}$ if P_x is of type GPU.

5.6.2 GPU RTA based on WRR Scheduling

WRR scheduling applies to the Nvidia TX2 GPU, since offloaded instructions from CPU tasks enter a run-queue and are then scheduled as channels in round-robin fashion [236]. Therefore, channels have different or equal time slice lengths, three different priorities, and run non-preemptively. During a WRR schedule, preemption can only take place at time slice boundaries.

Since notations for executable entities deviate compared to CPUs, GPU semantics are as follows. An application is subdivided into a single (cf. [30]) custom (user-defined) stream, which is a queue (sequence) of copy and computation operations, i.e. kernels[46]. As a consequence of using custom streams that do not run as a NULL stream [237], kernels are free from synchronizing with previous ones [30]. A stream is further subdivided into channels based on a predefined time slice property, such that a hardware WRR scheduler schedules channels. Due to separate copy and execution engines, copy operations of one channel may execute concurrently to the kernel execution of another channel since the usual execution flow follows *copy in, execute, copy out.* In line with [30], the following assumptions are made:

1. CPU tasks are scheduled in fixed-priority mixed-preemptive fashion (RTA presented in Section 5.4.2)

2. Memory access delays for GPUs are already accounted for within a kernel's ticks, such that L_i is set to 0 for GPU kernels when calculating the execution time (cf. Eq. 5.33).

3. CPU contention is given in [30] as:

$$B^{mc}_{i,x} = bl_{i,x} + (K_x \cdot \#C_i) + sGPU \cdot bGPU \tag{5.55}$$

   with $\#C_i$ denoting the number of processors that run at least one task, which accesses at least one label accessed by τ_i. The baseline $bl_{i,x}$ is derived from a PU's access latency to memory, all labels accessed by τ_i, and the cache line length cl_x so that $bl_{i,x} = L_i(com)$ cf. Eq. 5.30. K_x and $sGPU$ are constants derived from the memory contention model [238] in conjunction with information given in the forum[47]. This Eq. 5.55 has been implemented in favor of Eq. 5.25 for producing comparable results along with the WATERS community. However, Eq. 5.25 in combination with concrete labels, label sizes, semaphores, access latency between different PUs and memories, as well as the PU's cache line, gives more realistic bounds than Eq. 5.55. Eq. 5.55 only accounts a single cache line access delay multiplied with the number of PUs accessing the same data as the task under consideration, and critical sections of semaphores are not covered at all.

4. GPU contention is given as:

$$B^{mc,GPU}_{i} = bl^{GPU} + 0.5 \cdot \#C \tag{5.56}$$

   with bl_{GPU} being a $3ns$ constant for a 64B cache line and $\#C$ the number of CPUs

[46] A kernel represents a GPU task and inherits its notation, i.e. τ_i is a kernel iff $M^P_{\tau_i} = x$ and P_x is of type GPU

[47] WATERS forum thread `https://bit.ly/2I1LXTe`, visited 11.2020

offloading tasks to the GPU. Again, this Eq. 5.56 can be advanced by considering model entity properties as shown in Eq. 5.52 and Eq. 5.53.

5. Copy operations of the CE are handled by the GPU.
6. Data is always transferred in form of an integer multiple of a complete cache line (i.e. 64 Bytes, cf. Eq. 5.25).
7. In line with [30], any memory access-, blocking-, or contention- delays are already accounted for within the ticks for the GPU kernels[48].
8. A GPU kernel's execution strictly follows the kernels' time slice lengths, which may only be shortened by finishing a kernel's execution at some round-robin turn before its time slice terminates.
9. As a consequence to the previous item 8, no cooperative or non-preemptive blocking must be imposed to $C_{i,x}^{+,GPU}$. Due to the fact that a GPU utilizes highly concurrent code, adding non-preemptive policies could significantly reduce concurrency, e.g. 9/10 GPU resources waiting for a slow 1/10 GPU occupying non-preemptive kernel.
10. In line with item 9, cooperative and non-preemptive tasks must not be allocated to a GPU.

Excursus 3: Round Robin Turn

A Round Robin Turn (RRT) under WRR is the sum of each GPU kernel's time slice lengths θ_i shown in Eq. 5.57.

$$RRT = \sum_i \theta_i \ : \ M_i^P = x; x \text{ is GPU} \tag{5.57}$$

After the CE operation time is calculated in Section 5.6.1, WCRTs of GPU kernels must be evaluated based on WRR scheduling (cf. [30]) using the GPU kernel execution time shown in Eq. 3.6. Note here that as stated at the previous seventh assumption item 7, memory access times and blocking delays are accounted for within the GPU ticks already so that $C_{i,x}^{+}$ is used instead of $C_{i,x}^{CPU,+}$, respectively no L_i and B_i instructions are included. The execution time can also be normalized towards one second via Eq. 3.5. The work of this thesis includes the implementation of RTA described in [36] for WRR scheduling apart from the specific burst stimulus consideration, which is not part of the model in scope. The implementation makes use of the windowing-technique proposed by Lehoczky in [38] to check for the WCRT of kernels with arbitrary deadlines within the busy period derived from the critical instant, i.e., the situation when all kernels arrive at the same time. The algorithm considers interference of other kernels within a round-robin turn, kernel interference of the previous round-robin turns, requested execution times until each time slice window, periodic kernels with different execution times and time slices to derive

[48]Local and global resource blocking (pi- and s-blocking) would impose high real-time uncertainty and it is nearly impossible to derive predictable scheduling that includes blocking delays in a highly concurrent GPU system [237].

accurate round-robin timing behavior without much pessimism. The following calculation of a GPU kernel response time $R_i^{+,GPU}(q)$ is further explained in [36], whereas $\mathcal{I}(q)$ denotes the interference of the analyzed task with other tasks, during the q-th window.

$$\begin{aligned} R_i^{\text{GPU}}(q) &= q \cdot C_i^{+,GPU} + \mathcal{I}(q) - (q-1) \cdot T_i \\ R_i^{\text{GPU},+} &= \max_q R_i^{\text{GPU}}(q) \end{aligned} \tag{5.58}$$

The essential benefit of this approach and its analytical calculation is its work-conserving manner that considers the cases of kernels not utilizing their entire time slice, such that round-robin turns vary in length over time. An example Gantt chart in line with the example of [36], except considering burst stimuli, is given in the Appendix H.9.

Since copy operations are performed by a dedicated CE, no label copy in/out costs induced by the communication paradigm must be accounted for GPU kernels, i.e. $L_{cio,i} = 0$ if $M_{\tau_i}^P = x$ and P_x is GPU (cf. Eq. 5.32 and Eq. 5.30).

To optimize the task to CPU-GPU mapping, a GA is implemented that includes potential task allocations to the GPU. The GA is configured to either optimize (a) the response time sum across all tasks and kernels, (b) task chain reaction or age delays, or (c) load balancing. For every solution, various metrics can be measured, such as individual and accumulated response times, the PU utilization values and their consolidated standard deviation, memory accesses delays, task chain reaction and age delays, the CE operation time, and contention delays, each of which can be considered for worst- and best-case execution times, different communication paradigms, and synchronous or asynchronous offloading delays. In terms of schedulability, the simple utilization test of Eq. 5.3 is added to the GA chromosome validation to traverse the solution space more quickly. This validation especially sorts out kernel sets for the GPU that exceed the GPU's computational resources.

The consideration of synchronous and asynchronous GPU offloading operations is presented in Section 5.6.4 right after the following Section 5.6.3 that presents a WRR scheduling alternative that covers more properties of GPU kernels for the Nvidia TX2 platform, i.e., blocks, grids, and threads. The alternative, denoted as TX2RS, primarily reflects existing knowledge and insights about real-time properties discovered for the Nvidia Jetson TX2 hardware, which is in focus of the WATERS2019 challenge.

5.6.3 GPU RTA based on Nvidia Jetson TX2 rules

Apart from WRR scheduling, commercial and proprietary details of the Nvidia TX2 scheduler, which is the target platform of the WATERS2019 challenge, are investigated in, e.g., [236, 237, 239]. Assuming run-to-completion policy on the GPU, its execution engine is never preempted and no interference has to be considered under TX2RS. Instead, advanced properties, semantics, and notations have to be outlined for GPU instructions when considering properties provided by Nvidia. A **kernel** τ_i, consist of a single **grid** of blocks $g_i = (x \times y)$. A single **block** $b_{a,b}$ with $a \in [1, x]; b \in [1, y]; a, b \in \mathbb{N}$ consists of another grid of $b_i = (i \times j)$ threads. As a consequence, a kernel consists of $g_i \cdot b_i = x \cdot y \cdot i \cdot j$ **thread**s and its sequential execution time based on instructions is calculated via Eq. 5.59.

$$C_{i,x}^{TX2R} = \frac{C_i \cdot g_i \cdot b_i}{f_x \cdot \varkappa_x} \tag{5.59}$$

$$C_{i,x}^{TC2R,s} = C_{i,x}^{TX2R} \cdot \frac{10^{12}}{T_i} \text{ with } T_i \text{ in picoseconds} \tag{5.60}$$

A kernel's utilization is given by $U_i^{TX2R} = \frac{C_i \cdot g_i \cdot b_i}{T_i}$, such that the GPU utilization can be derived as $U_x^P = \sum_{\tau_i : M_{\tau_i}^P = x} U_i^{TX2R}$. Due to finding an optimal number of threads per kernel being a challenging process as various publications show, e.g., [240–243], the following assumes equal thread numbers per kernel and a single symmetric multi-processor potentially serving 4096 threads. Threads are scheduled in **warps**, which represent a group of 32 threads, by the warp scheduler. Nvidia developers recommend using block sizes 128, 256, 512, or 1024 since these values are more likely to take full advantage of the GPU resources as stated in [244, 245]. Although providing a wide range of memory types, e.g., global-, constant-, texture-, block-shared-, thread-local-, and thread-register memories, as shown in Table 5.2, the WATERS2019 model does not contain this granularity. Due to this lack of information, memory access latency values used in the following are based on the simplified version from [30], i.e., using constant read and write access latency values for Global Random Access Memory (GRAM) and cache memories.

Memory	**Properties**	**Scope**	**Lifetime**
Global	R/W, slow, big	Grid	Application
Texture	ROM, fast, optimized for 2D/3D processing	Grid	Application
Constant	ROM, fast, constants	Grid	Application
Shared	R/W, fast, on-chip	Block	Block
Local	R/W, slow, used if registers are full	Thread	Thread
Registers	R/W, fast	Thread	Thread

Table 5.2: Overview of Nvidia TX2 GPU memory types, properties, scope, and lifetime

The following additional assumptions are used to ease RTA for TX2RS:

1. The TX2RS scheduler's FIFO run-queue is filled with entire kernels[49].
2. A single symmetric multiprocessor SM is assumed for the Nvidia TX2 platform such that the maximal number of threads that can be allocated in the SM at some point in time is $b_{max} = 4096$.
3. All blocks have the same amount of threads: $\forall \tau_i : b_i = b$ with $M_{\tau_i}^P = x$ and P_x being a GPU (b_i is a kernel's block height in Figure 5.5).
4. Kernels can have different numbers of blocks g_i ($g_i \cdot b$ is the kernel height in Figure 5.5) as well as different block execution times (kernel's block width in Figure 5.5).
5. Blocks within the same kernel have equal execution time.
6. The maximal number of blocks executing on the GPU is $g_{\max} = \frac{b_{\max}}{b}$.
7. The number of free blocks at time t is denoted as $g_f(t)$.

[49]Theoretically, a combination of WRR and TX2RS could be implemented that does not assume entire kernels but kernel time slices as entities in the run-queue. This would manifest in (a) replacing $C_{i,x}$ with θ_i and (b) counting the scheduled time slices ($\#\theta$) and instances (activations) for every kernel scheduled on the GPU such that Algorithm 5.1 stops when $\#\theta_j \cdot \theta_j \geq T_j$.

8. Kernels are de-queued in FIFO order either immediately (the first kernel(s) at the queue's head), delayed, or partially delayed (for a block subset), which depends on already de-queued running kernels.

9. *Later* arriving kernel jobs entering the run-queue for being released due to, e.g., their periodic activation, do not affect the response time calculation due to being behind the kernel under consideration in the run-queue.

10. To retrieve the WCRT for a GPU kernel, the critical instant is assumed so that all other kernels potentially running on the GPU are in front of the kernel under consideration τ_j in the run-queue, i.e. τ_j is located at the run-queue's tail and all other kernels are located before τ_j. An alternative approach for FIFO scheduling is given in [246] and could decrease pessimism of the previous assumption. However, significant adaptation would be required to consider grids, blocks, and threads in addition to the used $\tau_i = \{O_i, C_i, T_i, D_i\}$ task model in [246].

11. The previous assumption item 10 imposes $\sum_{i:M^P_{\tau_i}=x;x \text{ is GPU}} (C_i \cdot g_i \cdot b_i) \leq \min_i T_i \cdot g_{\max}$ and $\forall i : C_i \leq \min_i T_i$.

12. No kernel exists more than once in a run-queue. Otherwise, implicit deadlines would be violated.

Alternatively to the worst-case run-queue assumption 10, a hyperperiod existence and distinct priority-based ordering of kernels for entering the run-queue can be assumed, which reduces pessimism. The alternative approach results in a distinct run-queue for the critical instant and lower response times for higher priority kernels compared with assuming a worst-case run-queue being filled with all other kernels already. However, the entire hyperperiod must be investigated for finding worst-case response times because frequent high priority kernels being executed first along with their initial instance may experience a non-empty run-queue at later instances, which results in increased response times compared with the initial instance correspondingly. This approach is promising for being investigated in future work mentioned in Section 8.2.

In contrast to partitioned PU scheduling that assumes a single computational resource (CPU) for each scheduler shared across tasks over time, GPU TX2RS adds a dimension by the availability of multiple resources (in form of threads). Hence, TX2RS requires the $g_f(t)$ metric to define the available amount of blocks at some point in time t based on dispatched kernels (scheduled kernels until t). Figure 5.5 shows exemplary the semantics of $g_f(t), g_{\tau_i}, b$, and $g_{\max}$. In contrast to CPU tasks, GPU kernels are a tuple of period,

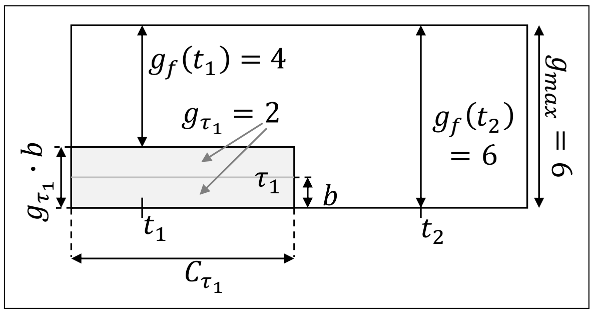


Figure 5.5: Semantics of available blocks at different points in time on a GPU

execution time, grid size, and block size, respectively $\tau_i = \{T_i, C_i^{GPU}, g_i, b_i\}$. The following Algorithm 5.1 constructs a complete worst-case schedule to find a GPU task's WCRT.

Algorithm 5.1: Nvidia TX2 Response Time Algorithm

```
Data: $\mathcal{T}_x$ with $\forall \tau_i \in \mathcal{T}_x : M^P_{\tau_i} = x, P_x$ is GPU
Result: $R_{i,x} \forall \tau_i \in \mathcal{T}_x$ and $R^+_{j,x}$
initialize $t_a = 0; g_f = g_{\max}; rq = \{\mathcal{T} : \tau_j = rq[|\mathcal{T}|]$ i.e. queue tail$\}; h = \emptyset$;
let $e$ denote the index for entries in $h$ such that $h_e[1]$ is the first value of the $e$-th entry in $h$;
while $rq$ contains $\tau_j$ do
    let $\tau_i$ denote the head of $rq$
    if $g_f(t_a) \geq g_i$ then
        $R_{i,x} = t_a + C_i$;                          /*set response time for $\tau_i$*/
        add $(f_i, g_i)$ to $h$;                        /*update the current schedule*/
        $g_f = g_f - g_i$;                              /*update the available blocks*/
        dequeue $\tau_i$ from $rq$;
    else
        $g_i = g_i - g_f$;                              /*update required blocks of $\tau_i$*/
        add $(t_a + C_i, g_f)$ to $h$;                  /*update the current schedule*/
        $index = e : h_e[1] = \min_x h_x[1]$;           /*find next available blocks*/
        $t_a = h_{index}[1]$;                           /*update current time*/
        $g_f = h_{index}[2]$;                           /*update the available blocks*/
        remove $h_{index}$ from $h$;
    end
end
$R^+_{j,x} = R_{|rq|,x}$;
```

A good way to traverse the stages of this Algorithm 5.1 is using an example, which is shown in Example 5.4.

Example 5.4: TX2R Scheduling

The following example of Figure 5.6 is used to illustrate the progression of Algorithm 5.1. The input is defined by $\mathcal{T} = \{\tau_1 = \{15, 4, 2, 512\}, \tau_2 = \{15, 8, 2, 512\}, \tau_3 = \{15, 2, 4, 512\}, \tau_4 = \{15, 5, 7, 512\}\}$, the run-queue is filled chronologically, i.e. $rq = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ and the response time to be found is $R_4^{+,GPU}$. The result schedule is shown in Figure 5.6, which is constructed iteratively by Algorithm 5.1.

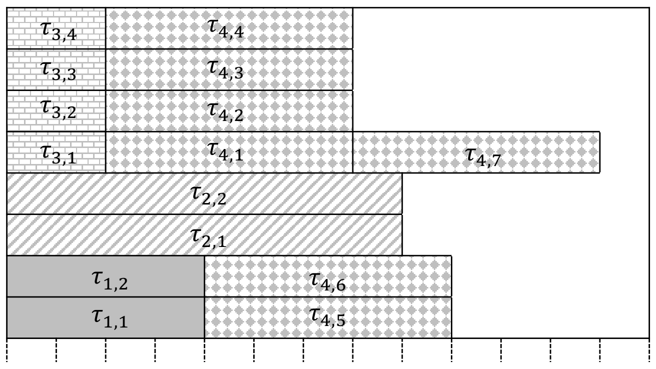


Figure 5.6: Nvidia TX2 schedule example & trace of Algorithm 5.1

The first three while loop iterations (cf. Algorithm 5.1 line 3, rep. rows # 1–3 in Table 5.3), are straight forward since $g_f(t_a) \geq g_i$ is true such that Algorithm 5.1 lines 6–9 are executed, which results in $h = \{(4,2),(8,2),(2,4)\}$. In the fourth iteration, no free threads are available, such that Algorithm 5.1 lines 11–16 are executed. In fact, τ_4 must be split into three regions for being completed, which is done by four times executing Algorithm 5.1 lines 11–16 and once executing Algorithm 5.1 lines 6–9 as shown in the following trace Table 5.3.

#	t_a	g_f	$g_f \geq g_i$?	g_i	h	R_i
1	0	8	y	2	$h = \{(4,2)\}$	$R_1 = 4$
2	0	6	y	2	$h = \{(4,2),(8,2)\}$	$R_2 = 8$
3	0	4	y	4	$h = \{(4,2),(8,2),\mathbf{(2,4)}\}$	$R_3 = 2$
4	0	0	n	7	$h = \{\mathbf{(4,2)},(8,2),(5,0)\}$	–
5	2	4	n	3	$h = \{(8,2),\mathbf{(5,0)},(7,4)\}$	–
6	4	2	n	1	$h = \{(8,2),\mathbf{(7,4)},(9,2)\}$	–
7	5	0	n	1	$h = \{(8,2),(9,2),(10,0)\}$	–
8	7	4	y	–	$h = \{(8,2),(9,2),(10,0),(12,1)\}$	$R_4^+ = 12$

Table 5.3: Example TX2RS WCRT algorithm trace for Figure 5.6

Bold highlighted values in h of Table 5.3 indicate the value pair used in Algorithm 5.1 line 13 to find the next closest possible start time for the current kernel to be scheduled or a block subset of the same.

Hence, Algorithm 5.1 calculates response times for a given run-queue and the WCRT for the last entry in the run-queue. For calculating WCRTs across all kernels, Algorithm 5.1 is applied to $n = |\mathcal{T}_x|$ run-queues, so that every kernel is located at a run-queue's end once, i.e. $\forall i \in [1,n]\ \exists!\ rq_i\ :\ \tau_i = rq_i[n]$ with $i \in \mathbb{N}; n = |\mathcal{T}_x|; M^P_{\tau_i} = x; P_x$ is GPU . Furthermore, the order of kernels in the run-queue can influence a kernel's response time, such that meta-heuristics can potentially be used to identify a worst-case run-queue order. However, due to lack of required information such as blocks, threads, or warps in the available model, the investigation of worst-case run-queue orders is omitted here, just as the comparison of TX2RS RTA for the evaluation Chapter 7.

5.6.4 CPU-GPU Response Times

In 2011, Lakshaman already mentioned in [247] that the use of GPUs in real-times systems is likely to increase in the future. Corresponding research has still open challenges, which are outlined, e.g., in [30] to analyze not only mapping entire tasks to GPUs as kernels, but also to investigate offloading instructions to a GPU either synchronously or asynchronously. Therefore, a CPU task (I) starts execution on a CPU, (II) triggers the GPU by starting (II.a) data copy operations for the CE and (II.b) instructions for the execution engine, (III.a) waits for the GPU to finish the offloaded computation or (III.b) suspends to yield PU resources to other tasks on the same CPU, and (IV) continues execution on the CPU either (IV.a) using results of the offloaded and finished GPU instruction set or (IV.b) not using job-related GPU results. Distinguishing (III.a)/(III.b) and (IV.a)/(IV.b) is necessary since in the synchronous offloading case, the triggering CPU task actively waits until the GPU finishes its instructions, which is not the case for asynchronous offloading. On the one hand, the waiting yields an immediate propagation of data due to no synchronization being required. On the other hand, CPU cycles are wasted during the active waiting period. To avoid wasting CPU cycles, the CPU computing resources can be exploited by other tasks when asynchronously offloading instruction sets to the GPU. However, this asynchronous offloading requires additional synchronization efforts to ensure the offloading task's progression using results of the offloaded (GPU) instruction set. The two different concepts are shown in Figures 5.7 and 5.8 for synchronous and asynchronous offloading, respectively.

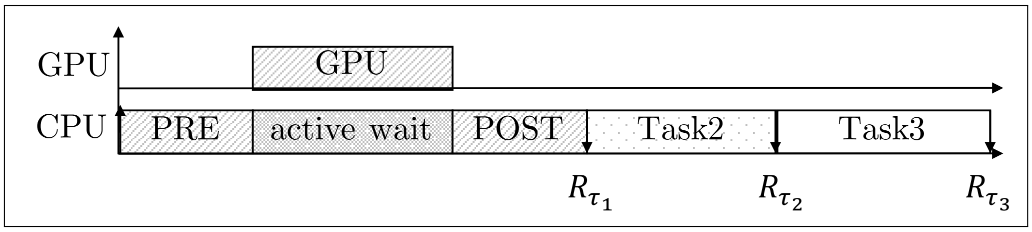


Figure 5.7: Synchronous GPU kernel offloading without copy operations

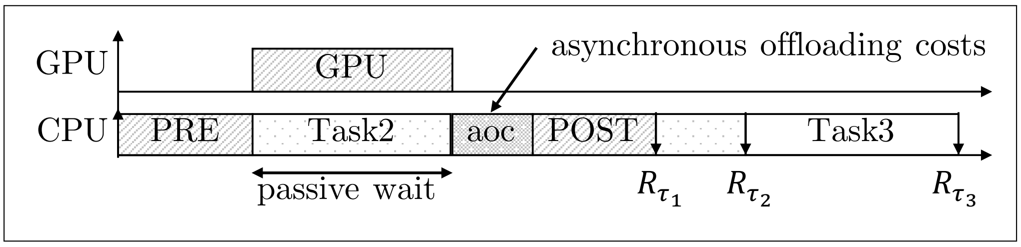


Figure 5.8: Asynchronous GPU kernel offloading without copy operations

To offload an instruction set to a GPU as a kernel in terms of AMALTHEA, an offloading CPU task must exist[50] that follows the structure (a) pre processing (PRE) → (b) trigger GPU instruction set via `inter-process-trigger` event → (c) wait & clear events → (d) post processing (POST). The wait and clear events are only required for synchronous offloading. If those events are not contained in an offloading task, it is assumed to be asynchronous. The offloading task AMALTHEA structure is exemplary shown in Figure 5.9.

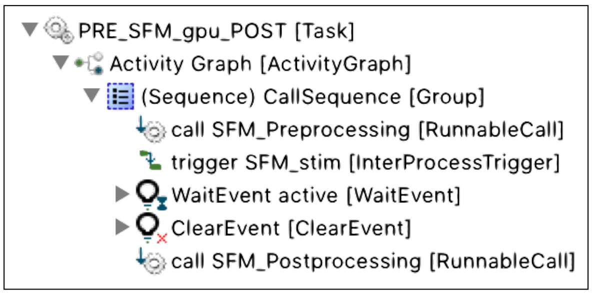

```
▼ PRE_SFM_gpu_POST [Task]
  ▼ Activity Graph [ActivityGraph]
    ▼ (Sequence) CallSequence [Group]
        call SFM_Preprocessing [RunnableCall]
        trigger SFM_stim [InterProcessTrigger]
      ▶ WaitEvent active [WaitEvent]
      ▶ ClearEvent [ClearEvent]
        call SFM_Postprocessing [RunnableCall]
```

Figure 5.9: AMALTHEA example structure for a CPU task offloading a GPU kernel

[50]For the WATERS2019 challenge, the naming of offloading tasks follows a "PRE_TASKNAME_gpu_POST" naming convention.

Since asynchronous offloading, i.e., passive waiting, allows other tasks to execute (cf. Task2 in Figure 5.8), the throughput is higher for asynchronous offloading compared to the synchronous case, if the penalty, denoted as Asynchronous Offloading Costs (AOC), is shorter than the active waiting section. An AOC represents the latency between the end of GPU kernel and the start of the post-processing phase. Those additional costs are required to reconstruct the offloading task's state to the time it was released initially. AOCs usually occupy less processing resources compared with the relatively longer active waiting period during synchronous offloading. The structure shown in Figure 5.9 is only required if the triggered task is mapped to a GPU. Otherwise, the triggering task is obsolete and can be ignored for RTA. For the task mapping process, this means that only tasks that have a corresponding offloading respectively triggering task can potentially be mapped to the GPU. This limitation is another example of pairing or separation constraints, such that either (a) CPU only tasks must be separated from GPUs or (b) paired with the GPU set.

Synchronous Offloading

The synchronous offloading RTA is implemented using the conventional RTA from Section 5.4.2 based on Lehoczky's work [38]. The only necessary adaption for synchronous offloading is the extension of the triggering task's τ_i execution time via using Eq. 5.61.

$$C_{i,x}^{+,sync} = C_{i,x}^{+,CPU} + CE_i^+ + R_{j,y}^+ \text{ with } \tau_j \text{ being triggered by } \tau_i \tag{5.61}$$

In Eq. 5.61, CE_i^+ is used from Eq. 5.54 for considering CE operations. Here, the CE always runs sequentially to the execution engine of the same kernel and might be further delayed due to queuing caused by multiple tasks issuing the CE at the same time. A CE queuing delay is included in CE_i^+ and described in Eq. 5.53. Hence, the original execution time for a task is extended towards the inclusion of the CE time for the triggered task and its response time at the GPU. The term '*being triggered by* τ_i' requires an `inter-process-trigger` event from τ_i to τ_j as part of τ_i's activity graph, which is shown as the fifth element from the top in Figure 5.9.

Asynchronous Offloading

To calculate response times that consider passive waiting for asynchronously offloading instructions, the triggering task is split into two parts, i.e., PRE (denoted i') and POST (denoted i'') processing tasks. While the former receives every activity graph item's execution time until the trigger event, the latter receives all execution time after the trigger event, including PRE processing costs and the AOC penalty. Additionally, the latter task obtains an offset value O_i, which equals the PRE task's length plus the triggered GPU task's response time shown in Eq. 5.62.

$$\begin{aligned} &\tau_i \text{ is split into } \tau_{i(\text{PRE})} \text{ and } \tau_{i(\text{POST})}, \text{ i.e., } \tau_{i'} \text{ and } \tau_{i''} \\ C_{i',x}^{+,async} &= \sum_{k=1}^{k<n} c_{agi_{i,k}}^+ \text{ with } n = \text{ index of trigger event of } \tau_i\text{'s activity graph} \\ C_{i'',x}^{+,async} &= \sum_{k=n}^{k=|ag_i|} c_{agi_{i,k}}^+ + \text{AOC}_i \\ O_{i''} &= C_{i',x}^+ + R_{j,y}^+ \text{ with } \tau_i \text{ triggering } \tau_j \end{aligned} \tag{5.62}$$

Due to the additional offset, the existing RTA is extended based on [39] for the asynchronous offloading approach, in which passive waiting can be utilized by other tasks. The offset consideration makes use of the *imposed interference* method since the critical instant derivation used for the synchronous offloading is not viable when having offsets for the asynchronous case. Therefore, task sets with the same periodic activation but different offsets are combined in transactions denoted as $\Gamma = \{\Gamma_1, ...\} \ : \ \Gamma_d = \{\tau_{d,1}, ...\}$, whereas the index d is used as a transaction index. Based on these transaction sets, a transaction's imposed interference during an iteratively increasing time interval is computed (denoted $W_{d,h}(R_i, t)$). The iterations on increasing the time interval end via fix-point lookup for the response time calculation of the task under consideration, i.e. $R_{i,x}^{+,\text{offs}} = R_{i,x}^{+,\text{offs}(n)}$ with $R_{i,x}^{+,\text{offs}(n)} = R_{i,x}^{+,\text{offs}(n-1)}$. The offset-based RTA using transactions is used from [39] and available in the appendix H.4. Notations of Equations H.19–H.24 are derived from [39] and just slightly adapted to fit notations of this thesis.

Asynchronous instruction set offloading to the GPU allows tasks to use the PU resources of the PU that offloads a task until results of the offloaded instruction sets are available from the GPU. With the implemented combination of FPPS using the windowing technique and the offset-based CPU RTA for asynchronous offloading as well as WRR scheduling for offloaded GPU tasks, timing verification can be investigated for a hybrid CPU-GPU environment. This timing verification is used for the mapping DSE in a mixed CPU-GPU environment to guarantee schedulability and both valid and optimized mapping solutions as well as both synchronous and asynchronous offloading configurations. The latter is evaluated for the WATERS model in Section 7.3.3 as being the only available model providing mixed CPU-GPU properties.

5.7 Data to Memory Mapping

Data to memory mapping that includes labels, i.e. shared variables and constants, code, and OS entities is often assumed to be static and either defined by the system designer or by default set to affinity locations of software components mainly working with the corresponding memory. However, optimized solutions can significantly mitigate temporal costs caused by accessing data distributed across Non Uniform Memory Access (NUMA) architectures. Therefore, event chains, response times, activation patterns, contention, and a variety of hardware properties such as memory type, memory size, memory access type, and memory affinities must be considered. Furthermore, label mapping costs must be analyzed regarding ECU networks consisting of buses, hardware hierarchies, and arbitrary connections of ports and hardware entities. Various timing properties for automotive applications are effected by data to memory mapping, especially in a typical environment of distributed, heterogeneous, and mixed-critical systems. System designers may overlook optimal data mapping solutions due to various constraints emerging from safety, affinity, timing, reliability, fault-tolerance, and similar demands. Since the modeling of comprehensive system environments is common practice in the automotive industry, new technologies can automatically cope with such challenges and overcome manual and error-prone processes by addressing problems on a much broader level using DSE along with timing verification techniques. For example, centralization effects can be investigated in early design phases without the need for hardware validation, actual software implementation, or sophisticated simulation tools.

Figure 5.10 shows a typical constellation of an ECU network subset in the automotive context. Real-world automotive networks usually consist of significantly more ECUs,

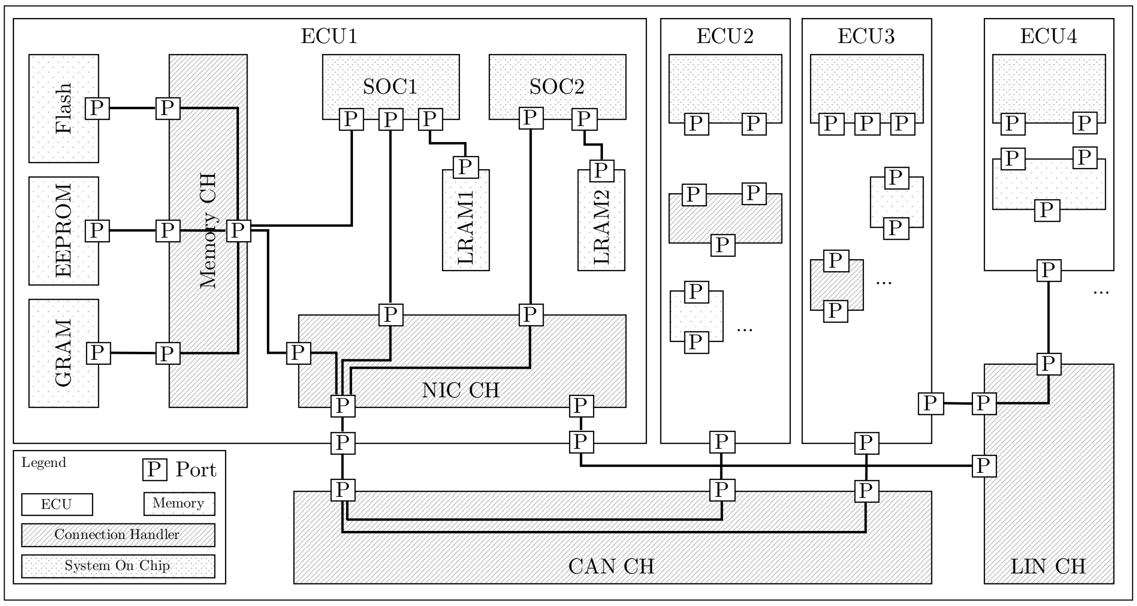


Figure 5.10: ECU network example with ports, connections, memories, connection handler, and hardware structures

which is omitted here for comprehension purposes. The same holds for detailed views of ECUs 2–4. The example is constructed using the AMALTHEA hardware model notations consisting of ports, internal and external connections, connection handlers, memories, and computational elements on various abstraction levels, e.g., DSP, FPGAs, CPUs, GPUs, micro-controllers, or similar (cf. Section 3.1.2). Connections within a connection handler (denoted 'CH') are shown as internal connections. Ports can be defined as initiators or responders and can implement interfaces such as CAN, Flexray, LIN, MOST, Ethernet, Serial Peripheral Interface (SPI), Inter Integrated Circuit (I2C), Advanced eXtensible Interface Bus (AXI), Advanced High-performance Bus (AHB), or Advanced Peripheral Bus (APB). ECU1 features five memory instances of four different types as well as two PUs in Figure 5.10. Due to contention and blocking interference and varying access delays, execution and response times are significantly influenced by data being located in different memories, such as global or local RAM, Non Volatile Random Access Memory (NVRAM), ROM, Electrically Erasable Programmable Read Only Memory (EEPROM), Flash or similar, and the data sizes, access rates, or the hardware structure. For instance, the PU on SoC1 can take between 2 and 3 cycles to access LRAM1, whereas accesses to the Flash memory cache can be 20 cycles or more. While most of the existing research assumes fixed worst-case memory access costs driven by contention, blocking, and scheduling, a sophisticated network, as shown in Figure 5.10 comprises a broad heterogeneous NUMA structure that needs to consider the dynamics of network protocols, connection handler properties, access patterns, and port interfaces to make memory mapping as effective and the execution of tasks as efficient, as possible.

Data distribution across different memories of various types while considering access types, access costs, access rates, contention, blocking, and various hardware properties to minimize total interference delays is an NP-complete problem [248]. To cope with the intractability, a GA, a CP approach and a dedicated heuristic are outlined in the following that minimize costs influenced by the data to memory mapping, after revising related work in the next Section 5.7.1.

5.7.1 Related Work on Data to Memory Mapping

The RTC [249] (cf. Section 2.3.4) toolbox for Matlab provides calculating end-to-end delays, buffer requirements, response times, or throughput of networked systems, among others. While RTC can assess real-time properties, deployment feasibility, and more, it does not provide tools to adjust or optimize a given memory mapping.

Schneider [250] outlines the challenges and requirements of memory management units for automotive ECUs and investigates approaches of the general-purpose computing domain regarding protection granularity, memory efficiency, and real-time behavior. The publication clearly shows the importance and relevance of sophisticated memory management for the automotive domain.

Kumar et al. [248] present formulations and solutions to the data layout problem, which is closely related to the label mapping problem of this thesis. In [248], the data layout problem is addressed via ILP, SA, GA, and a heuristic, whereas work presented here uses a GA and CP. Another difference is that Kumar et al. take memory stall cycles as a performance metric, derived from the number of data accesses and the stall cycles, which do not depend on the size of accessed data. Furthermore, affinity constraints are not covered in [248], which are crucial for automotive systems. Yet, there are valuable extensions in [248] such as memory architecture exploration and the comparison of logical and physical data mapping approaches.

In [251], Broquedis, et al. advance the OpenMP runtime to dynamically perform thread and memory placement to provide dynamic load distribution under application requirements and hardware constraints such as affinities. In contrast, this thesis's approaches account for memory utilization offline to provide a static mapping, which complies with the AUTOSAR standard and considers explicitly various timing and network constraints.

Antony et al. [252] account various memory placements along with access delays and memory bandwidth assessments on different NUMA platforms running Solaris and Linux. Although the benchmarks do not cover specific real-time properties, results show that local placement is not always the best strategy, and sophisticated mapping significantly improves application performance.

Avissar et al. present a compiler-based approach to automatically partition data for memory units using binary ILP in [253]. The used system model is close to this thesis' model and corresponds to the minimization of label mapping costs MC_l. Nevertheless, this thesis's contribution goes beyond [253] via considering AUTOSAR related constraints, analyzing response times, and incorporating sophisticated network structures containing, e.g., the CAN bus.

Along with the WATERS community, label mapping optimization is the third part of the WATERS2016 [128] and the fourth part of the WATERS2017 [31] challenges. However, presented solutions are more rudimental than this thesis' approach due to distinguishing between local, remote, and global memory only and having homogeneous access delays across PUs i.e., nine cycles to global and remote as well as one cycle to local memory. In contrast, by considering arbitrary access delays and hierarchies of memories, this thesis's approach provides an AMALTHEA-based data mapping solution with increased generality. Solutions of the WATERS community are presented as follows with short solution descriptions, whereas all approaches lack network delay consideration, heterogeneous memory access delays, and GPU interference.

- In [82] and [32], shared labels are mapped to global memory and labels only accessed by one task are mapped to the local memory of the PU the accessing task is mapped to. This straightforward procedure misses minimized temporal interference and results are especially ineffective if memory limits are reached and frequently accessed or huge shared data chunks are allocated to local memory.
- In [84], a greedy label mapping algorithm is proposed that maps all labels to global memory and secondly maps labels to local memory in descending order of label access number divided by the period (access rate) based on a label index, a memory index, and the memory size. Albeit avoiding arbitrary allocations for reaching memory size limits, this greedy algorithm maps labels to global memory if the memory with the highest access rate is already fully utilized, although there could be another local memory that may reduce access delays even further, and hence the algorithm misses optimal label mappings, too. In general, due to the greedy structure, the approach may also miss global optima.
- In [85], a GA and an ILP solution to the label mapping problem are presented. The approach is close to this thesis's solution since response times are taken as an optimization goal, and memory contention is considered as well. However, the model in [85] only considers local and remote access delays, and no memory size constraint is considered.
- In [86], a greedy label mapping is presented that considers memory size, access rates, hyper periods, and minimizes contention. Even though this seems quite reasonable, response time optimization, including network delays and heterogeneous memory access delays, gives more realistic and usable solutions in terms of timing efficiency.

Before starting to outline the advanced label mapping approaches, some assumptions must be noted as follows.

- A PU's local memory is dual-ported so that both the PU and the crossbar can access the local memory at the same time. Consequently, contention effects only remote PUs.
- Data progression between runnables of the same task is considered through local (private) memory.
- LET and implicit communication based Copy-in and copy-out operations are implemented as CSs.

These assumptions are in line with [33], memory contention Eq. 5.25, access latency Eq. 5.30, and edge costs Eq. 4.6 calculations.

For optimization purposes along with the presented meta-heuristics, i.e., GA and CP approaches for data to memory mapping, CAN message transmission delays, denoted as CAN message response times R_ν between ECUs, must also be considered that are subject to (a) lower priority bus blocking, (b) priority-based queuing, and (c) several message constants and bus properties as outlined in the next Section 5.7.2. Based on this holistic interference analysis, data propagation through an ECU network can be bounded according to worst-case situations to minimize access, blocking, contention, and network transmission delays. The latter are outlined in the following Section 5.7.2.

5.7.2 CAN Message Transmission Delays

Typically, the start of a CAN message's transmission occurs at the end of a task's execution by e.g., following the implicit communication paradigm (cf. Definition 5.8), if task results are required by one or more tasks running on another ECU, as shown in Eq. 5.63 with ν denoting a CAN message.

$$\forall \nu : \exists l_v \text{ with } |\mathrm{ECU}_{l_v}| \geq 2 \text{ and } \mathrm{ECU}_{l_v} = \bigcup_{\mathrm{ECU}_y : l_v \in \mathcal{L}_{\tau_i} ; M_{\tau_i}^{\mathrm{ECU}} = y} \tag{5.63}$$

Additionally, due to varying task response times and the corresponding variety in CAN release times as well as mutually exclusive CAN bus occupation, the WCRT for CAN messages must be analyzed in an appropriate context. The CAN bus can be seen as a globally shared resource granted access to based on a priority ordered queue. Sent messages are non-preemptive and have a static priority similar to Fixed Priority Non-Preemptive (FPNP) scheduling. The approach by Tindell et al. [254] published in 1994 was refuted by Davis et al. [145] in 2007 about 13 years later, which shows that CAN messages must be analyzed over a busy-window that begins with the critical instant to provide realistic CAN transmission delays. Here, the window is denoted as *level-ν busy-period*, which is the maximum consecutive amount of time the CAN bus is occupied by messages that have an equal or greater priority than message ν. The analytic approach for determining worst-case response times in CAN networks under consideration of busy-periods is further presented in [143]. After ensuring that the bus utilization is less than one ($\sum_\nu \frac{C_\nu}{T_\nu} \leq 1$, ν is used as the index for a CAN message), the duration w_ν of a *level-ν busy period* is calculated via Eq. 5.64.

$$w_\nu^z = B_\nu + \sum_{\nu_i \in hep(\nu)} \left\lceil \frac{w_\nu^{z-1} + J_{\nu_i}}{T_{\nu_i}} \right\rceil C_{\nu_i} \tag{5.64}$$

The minimal interval between two occurrences of the ν_i-th higher priority message is represented by T_{ν_i} and equals the sending task's period for periodically sent messages. In case a task is activated sporadically, its minimal inter-arrival time is used for T_{ν_i}.

Eq. 5.65 defines the payload of a CAN message $c_\nu(\tau_j)$ based on summed up label sizes of labels that are part of the intersection between (1) τ_j's accessed labels $\mathcal{L}_{\tau_j}$, and (2) globally shared labels CS^Φ, which are accessed by at least one task τ_i mapped to a different ECU.

$$\begin{aligned} c_\nu(\tau_j) = & \sum_{l_v \in \left(\mathcal{L}_{\tau_j} \cap CS^\Phi\right)} ls_v + COM_{\tau_j} \\ & \text{with } \forall v \; : \; l_v \in CS^\Phi \wedge l_i \in \left(\mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j}\right) \text{ with } M_{\tau_i}^{ECU} \neq M_{\tau_j}^{ECU} ; i \neq j \end{aligned} \tag{5.65}$$

Here, AUTOSAR COMmunication Module (COM)$_{\tau_j}$ is given as a custom property (cf. Def. 3.9) of τ_j and defines the AUTOSAR COM BSW costs for inter-ECU communication, similar to IOC for inter-PU communication on the same ECU. After the CAN message payload is known, some constants must be derived from the AMALTHEA model to estimate the maximum transmission time C_ν of a CAN message. One of these properties is the transmission time for a single bit τ_{bit}, which can be derived from the static CAN network's baud rate. Another property is the CAN message identifier bit length, which can be either 11 or 29 according to the CAN protocol. Based on the payload and the identified length, the transmission time can be calculated as shown in Eq. 5.66. The constants $55, 80$, and 10 are obtained from CAN protocol properties such as bit stuffing that includes Cyclic Redundancy Check (CRC) bits, error frames, as well as control, arbitration and data fields [143].

$$C_\nu = \begin{cases} (55 + 10 s_\nu)\, \tau_{bit} & \text{for 11-bit identifiers} \\ (80 + 10 s_\nu)\, \tau_{bit} & \text{for 29-bit identifiers} \end{cases} \tag{5.66}$$

As the next step, the non-preemptive priority-based CAN bus arbitration scheme must be taken into account to estimate the WCRT $R^+_\nu(q)$ for the q-th instance of a CAN message ν. It is obtained as shown in Eq. 5.67 by summing up the queuing jitter J_ν, the queuing delay $W_\nu(q)$, and the transmission time C_ν. By subtracting $q \cdot T_\nu$ in Eq. 5.67, the relative WCRT for the q-th instance of message ν is calculated. The queuing jitter J_ν is usually derived from hardware profiling, and an analytical jitter derivation is omitted here.

$$R_\nu(q) = J_\nu + W_\nu(q) - q \cdot T_\nu + C_\nu \tag{5.67}$$

The queuing delay $W_\nu(q)$ for the q-th instance of ν (shown in Eq. 5.67) is determined using the recurrence relation Eq. 5.68 and $W^z_\nu(0) = B_\nu + C_\nu$.

$$W^z_\nu(q) = B_\nu + q \cdot C_\nu + \sum_{\forall h \in hp(\nu)} \left\lceil \frac{W^{z-1}_\nu + J_h + \tau_{bit}}{T_h} \right\rceil C_h \tag{5.68}$$

The last parameter to be determined is the blocking delay B_ν. As CAN messages are naturally non-preemptive, the worst-case blocking time B^+_ν imposed to ν is defined by the maximal transmission time $C_{\nu'}$ of lower priority (lp) messages on the bus. At most one lower priority message can block the CAN bus for a message ν, i.e. $B_\nu = \max_{\nu' \in lp(\nu)} C_{\nu'}$.

Finally, the WCRT of a CAN message ν is determined by finding the maximal CAN message response time across all instances within the level-ν-busy-period.

$$R^+_\nu = \max_z R_\nu(z) \text{ with } z \in \left(\left[1, \left\lceil \frac{w^z_\nu}{T_\nu} \right\rceil \right] \wedge \mathbb{N} \right) \tag{5.69}$$

Based on Eq. 5.64–5.69 [143], data to memory mapping can be approached holistically, i.e. across ECUs. Therefore, a greedy heuristic, an EA, and a CP approach are outlined in the following. The heuristic is a quick and straightforward method to map labels, ordered by size decreasingly, to memory featuring the lowest $MC^{m_d}_{l_v}$ value, i.e. $M^m_{l_v} = d \;:\; MC^{m_d}_{l_v} = \min_{d'} MC^{m_{d'}}_{l_v}$. If a memory instance is full, the memory with the next lowest $MC^{m_d}_{l_v}$ value is chosen. Since this Greedy approach may miss optimal mappings, Sections 5.7.3 and 5.7.5 present the CP and EA approaches to solving the data to memory mapping optimization problem.

5.7.3 CP-based Data to Memory Mapping

In terms of CP, the initial constraint that has to be applied to the label mapping matrix $\boldsymbol{M}_l^m$ is a sum constraint that ensures that all labels are mapped to exactly one memory instance as shown in Eq. 5.70. This methodology has already been used for the partitioning and task mapping (cf. Chapter 4 $\boldsymbol{M}_r^\tau$ and 5 $\boldsymbol{M}_\tau^P$, respectively), and the label mapping is represented by a $(q \times \mu)$, i.e., number of labels $\times$ number of memories, boolean matrix $\boldsymbol{M}_l^m = M_{l_v}^{m_d}(q \times \mu)$.

$$\forall\, l_v \in \mathcal{L} \;:\; \sum_d M_{l_v}^{m_d} = 1 \tag{5.70}$$

Additionally, the memory size constraint must be respected such that no memory is assigned with more bits than it actually can save, as shown in Eq. 5.71.

$$\forall m_d \;:\; \left(\sum_{v:M_{l_v}^{m_d}=1} ls_v + \sum_{i:M_{\tau_i}^{m_d}=1} cs_i + \sum_{s:M_{os_s}^{m_d}=1} os_s \right) \leq ms_d \tag{5.71}$$

Eq. 5.71 shows $M_{\tau_i}^{m_d}$, which denotes the mapping of a task to memory, cs_i, which denotes the code size of τ_i, $M_s^{m_d}$ denoting the mapping of an OS service to memory, and os_s representing the OS service memory size required to operate the OS service.

After ensuring that labels are mapped once, and memory sizes are not exceeded, optimization goals can be defined. For optimizing label to memory mapping only (a broader optimization methodology is presented in Section 5.9), the goal is twofold: minimizing (I) the overall access latency and (II) the accumulated network message delays. The first optimization goal considers label access numbers and access rates, memory access latency delays, the runnable to PU mapping, label sizes, and the bit width of the communication channel, respectively cache line length. Consequently, the total label mapping cost is derived from the binary label mapping $\boldsymbol{M}_l^m$, the number of accesses to the label per second derived from activation rate T_a, read and written labels $\uparrow_a, \downarrow_a$, runnable mapping $\boldsymbol{M}_r^P$ (to PUs), and read as well as write latency values between PU and memories $\uparrow_{x,d}, \downarrow_{x,d}$. The overall label mapping cost calculation is given in Eq. 5.72 and constituted by the sum of all labels' mapping costs, each of which is defined by the dot product of cost and mapping vectors.

$$\begin{aligned} MC_l =& \sum_v \vec{M}_{l_v}^m \cdot \vec{MC}_{l_v}^m \qquad \text{(scalar constraint)} \\ &\text{with } \vec{MC}_{l_v}^m = \left(MC_{l_v}^{m_1}, ..., MC_{l_v}^{m_q}\right) \\ &\text{and } MC_{l_v}^{m_d} = \sum_{a:l_v \in \uparrow_{r_a}} \left(\uparrow_{r_a,v}^{\#} \cdot \uparrow_{x,d} \cdot \left\lceil \frac{ls_v}{cl_x} \right\rceil \right) + \sum_{b:l_v \in \downarrow_{r_b}} \left(\downarrow_{r_b,v}^{\#} \cdot \downarrow_{y,d} \cdot \left\lceil \frac{ls_v}{cl_y} \right\rceil \right) \\ &\qquad\qquad \text{with } M_{r_a}^P = x; M_{r_b}^P = y \end{aligned} \tag{5.72}$$

Here, $\uparrow_{r_a,v}^{\#}$ denotes the number of read accesses per second of r_a and $\downarrow_{r_b,v}^{\#}$ the number of write accesses per second of r_b to label l_v. In terms of implicit and LET communication, these values are derived from the periodicity only due to assuming data being copied into local (cache) memory for a task instance. Copy operation of Eq. 5.32 are therefore accounted within $C_{i,x}$. Although being close to the dependency Eq. 4.6, Eq. 5.72 does not consider a source and target runnable from a dependency, but rather accumulated and normalized access costs over all runnables for a specific label to memory mapping in $MC_{l_v}^{m_d}$, all possible memory mappings for a specific label $\vec{MC}_{l_v}^{m}$, and the accumulated label mapping costs for a specific label mapping MC_l^m.

As an intermediate summary for the label mapping optimization approach, solutions can be assessed by access delays across all tasks and inter-ECU communication costs defined by the amount and response times of CAN network messages.

To get a better understanding of variables and constraints, the following hypothetical and small Example 5.5 provides an analysis of mapping costs for three memories, two PUs, four runnables, and five labels. The optimization goals are to minimize (1) the total label mapping costs in Eq. 5.73 via Eq. 5.72 and (2) the maximal PU load in Eq. 5.74 via Eq. 5.4. If Eq. 5.9 is used instead of Eq. 5.4, label access costs are also accounted in the utilization optimization, which is omitted here to keep Example 5.5 simple. Equations 5.73 and 5.74 are preferred over, e.g., load balancing in Eq. 5.1 or the sum of label access costs via Eq. 5.30, since the former use normalization via taking periodicity into account. WCRT optimization of tasks and CAN messages (cf. Eq. 5.98) is outlined in Section 5.9.

$$\text{minimize } MC_l \tag{5.73}$$

$$\text{minimize } \hat{U}^P \tag{5.74}$$

Example 5.5: CP-based Data Mapping Cost Optimization

Assuming three memories $M = \{m_1, m_2, m_3\}$, two PUs $P = \{P_1, P_2\}$, four Runnables $\mathcal{R} = \{r_1, r_2, r_3, r_4\}$ that access five labels $\mathcal{L} = \{l_1, l_2, l_3, l_4, l_5\}$ such that $\uparrow_{r_1} = \emptyset, \downarrow_{r_1} = \{l_1, l_3\}, \uparrow_{r_2} = \{l_2\}, \downarrow_{r_2} = \{l_3\}, \uparrow_{r_3} = \{l_3\}, \downarrow_{r_3} = \{l_5\}, \uparrow_{r_4} = \{l_2, l_5\}, \downarrow_{r_4} = \{l_4\}$, and access latency values (same for read and write) $\uparrow_{P_1,m_1} = 2, \uparrow_{P_2,m_1} = 1, \uparrow_{P_1,m_2} = 3, \uparrow_{P_2,m_2} = 2, \uparrow_{P_1,m_3} = 1, \uparrow_{P_2,m_3} = 4$. The two homogeneous PUs can execute up to four instructions per second, i.e., $puc_1 = puc_2 = 4$, the four runnables each execute a single instruction and they are executed once per second, i.e., $\forall r_a \in \mathcal{R} : T_{r_a} = 1; c_{r_a,x} = 0.25s; u_{r_a} = \frac{1}{4}$. There are $5^3 = 125$ solutions for mapping the labels to memories and $4^2 = 16$ solutions for mapping runnables to PUs and $16 \cdot 125 = 2000$ solutions for all combinations of runnable and label mappings. The dependency graph derived from label accesses is shown in the right part of Figure 5.11.

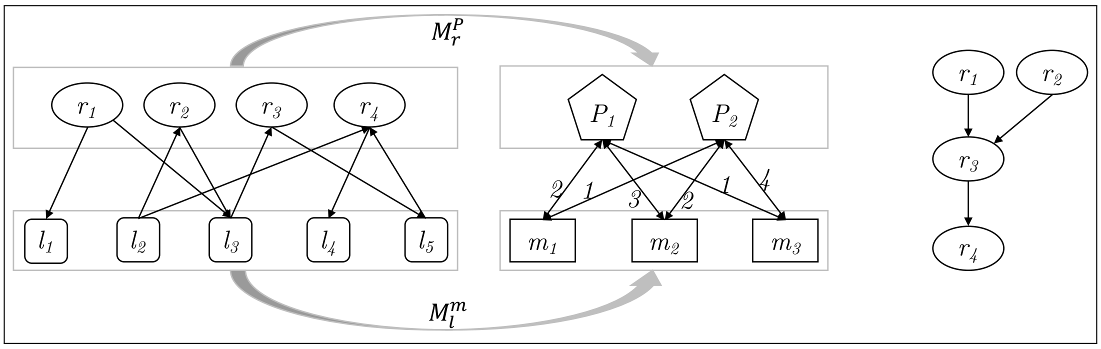


Figure 5.11: Memory mapping example

Without load balancing constraints, the result is shown in Eq. 5.75. Adding load balancing as the primary goal, the result changes as shown in Eq. 5.76.

$$\boldsymbol{M}_l^m = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}; \boldsymbol{M}_r^P = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{pmatrix}; \boldsymbol{MC}_l^m = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{pmatrix}; MC_l = 9; \hat{U}^P = 1 \tag{5.75}$$

$$\boldsymbol{M}_l^m = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}; \boldsymbol{M}_r^P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}; \boldsymbol{MC}_l^m = \begin{pmatrix} 0 & 0 & 1 \\ 2 & 0 & 0 \\ 5 & 0 & 0 \\ 1 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix}; MC_l = 12; \hat{U}^P = 0.5 \tag{5.76}$$

The Pareto-optimal solution set is shown in Figure 5.12 along with the two optimization goal values PU utilization (y-axis) and label mapping cost (x-axis). Solutions of the Pareto front are indicated with circles, of which the most left one represents Eq. 5.75 and Eq. 5.76 gives the details of the solution (12; 0.5) in Figure 5.12.

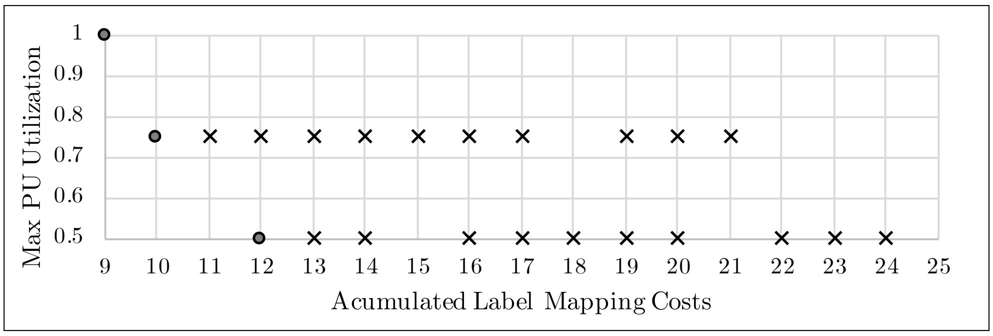


Figure 5.12: Dominating solutions and pareto front of the label mapping example

Access latency calculation outlined in Eq. 5.30 covers write and read dependent delays based on instructions $c_{x,d}$, contention B^{mc}, the cache line length cl_x, bit width $bw_{x,d}$, or data rate $dr_{x,d}$, labels $\mathcal{L}_i$ and their size ls_v, as well as PU properties such as frequency f_x and IPC $\varkappa_x$. With considering R_ν^+, data propagation delays across ECUs, i.e., network transmission delays are also considered via protocol properties such as the type and bit width of ports and, e.g. message priority queuing, bus data rate, and data overhead caused by identifiers, additional checksum data etc. as shown in Eq. 5.64–5.69, respectively Section 5.7.2. As the next step, necessary data separation, pairing, and communication paradigm dependent data aging constraints must be incorporated, which is part of the next section.

5.7.4 Data Separation, Pairing, and Aging Constraints

FFI is not only of relevance for tasks across PUs, but also for data across memories. Hence, the label to memory mapping must consider separation and pairing constraints just as tasks in Eq. 5.14 and 5.15. Therefore, data pairing and separation constraints are defined in Eq. 5.77 and 5.78, respectively.

$$\Phi_l^m = \{\mathcal{L}(\Phi_l^m), \mathcal{M}(\Phi_l^m)\} \text{ with } \mathcal{L}(\Phi_l^m) \subseteq \mathcal{L}; \mathcal{M}(\Phi_l^m) \subseteq \mathcal{M} \tag{5.77}$$

For instance, $\Phi_l^m = \{\{l_1, l_2\}; \{m_2\}\}, \{\{l_3\}; \{m_1, m_3\}\}\} \Rightarrow M_{l_1}^m = M_{l_2}^m = m_2; M_{l_3}^m = m_1 \vee m_3$ means that labels l_1 and l_2 must be allocated to memory m_1 and label l_3 can be allocated to either memory m_1 or m_3.

$$\Phi_{l|m} = \{\mathcal{L}(\Phi_{l|m}), \mathcal{M}(\Phi_{l|m})\} \text{ with } \mathcal{L}(\Phi_{l|m}) \subseteq \mathcal{L}; \mathcal{M}(\Phi_{l|m}) \subseteq \mathcal{M} \tag{5.78}$$

These constraints can now be applied to the label mapping variables as shown in Eq. 5.79 and 5.80.

$$\forall\, l_v \in \mathcal{L}(\Phi_{l|m}); \forall m_d \in \mathcal{M}(\Phi_{l|m}) \;\; : M_{l_v}^m \in (([1, q] \setminus d) \wedge \mathbb{N}) \tag{5.79}$$

$$\forall\, l_v \in \mathcal{L}(\Phi_l^m) : M_{l_v}^m \in \mathcal{M}(\Phi_l^m) \tag{5.80}$$

Finally, the data aging constraints can be applied to either implicit or LET communication. Data aging constraints mostly have their origin in requirements for sensors refreshing data in memory so that precise analysis for event propagation and event reaction is possible and corresponding timing guarantees can be made.

Definition 5.10: Data Aging

A data aging constraint demands a label to be updated at least after n time units.

A data age constraint's CP application for AMALTHEA models based on notations of Section 3.2 and Definition 5.10 is given in the following Eq. 5.81. The communication paradigm *com* is either implicit (cf. Definition 5.8) or LET (cf. Definition 5.9), i.e. $com \in \{\iota, \lambda\}$, due to being used in favor of explicit communication under AUTOSAR, which may result in inconsistent data propagation [212].

$$\Phi_{l_v,\downarrow} = \{L_\alpha\} \;\; : \;\; w_{l_v,\downarrow}^+ \leq L_\alpha(\Phi_{l_v,\downarrow}) \tag{5.81}$$

Eq. 5.81 makes use of the worst-case write window $w_{l_v,\downarrow}^+$ for a label l_v, which is derived from all tasks writing to l_v in terms of implicit communication. This derivation is required because the worst-case window of a label not being updated does not necessarily depend on the task with the lowest period for implicit communication. Instead, it is defined by a consecutive occurrence of a task's BCRT followed by its WCRT. The window the label is not updated in such a situation is then the sum of a tasks' periods and its WCRT subtracted by its BCRT, as shown in Eq. 5.82.

$$w_{l_v,\downarrow}^+(\iota) = \min_{i: l_v \in \mathcal{L}_i} \left(T_i + R_i^+ - R_i^-\right) \tag{5.82}$$

Under LET, the writing process of data to memory is executed at the end of a period across tasks. This approach makes data age derivation significantly easier since distinguishing between worst and best cases is not necessary, and the refresh rate is simply a task's period, as shown in Eq. 5.83.

$$w^{+}_{l_v,\downarrow}(\lambda) = \min_{i:l_v \in \mathcal{L}_i} T_i \tag{5.83}$$

The CP approach to the data mapping problem uses the choco library[37] to find a data mapping to memory represented as the boolean variable matrix $\boldsymbol{M}^m_l$. Results are assessed by the accumulated mapping cost MC_l. Solutions are required to satisfy all of the following constraints.

1. A sum constraint on each array of the $M^m_{l_v}$ vector ensuring that each label is exactly mapped once.
2. A scalar constraint across $M^m_{l_v}$ and $MC^{m_d}_{l_v}$ used for optimization, i.e. reducing timely memory access interference.
3. A sum constraint over $MC^{m_d}_l$ to derive the total mapping costs MC_l for the above optimization.
4. A scalar constraint for calculating the sum of label sizes assigned to a memory instance and an arithmetical constraint to ensure that this sum does not exceed the memory size. The latter is applied to each memory.
5. Several arithmetical constraints for various affinities.

In the next Section 5.7.5, the evolutionary data mapping algorithm is presented that uses Eq. 5.64–5.83 and forms an alternative to the CP-based approach. Due to the high amount of labels beyond ten-thousand, e.g., for the FMTV model, the GA provides a reasonable alternative to CP in terms of scalability.

5.7.5 Evolutionary Data to Memory Mapping

The implemented EA uses the jenetics java library [42]. The fitness function is presented in Eq. 5.85, whereas $\mathcal{S}$ denotes a chromosome that encodes M^m_l based on a static $M^P_{\tau_i}$. The notation $\overline{R}^N$ represents the average CAN message response time of Eq. 5.84 and N is the set of CAN messages, each represented with index ν.

$$\overline{R}^N = \sum_{\nu} \left(\frac{R^{+}_{\nu}}{|N|} \right) \tag{5.84}$$

Additionally, each fitness calculation is accompanied by various constraint checks to ensure validity, listed below.

1. Distinct label, code, and OS service to memory mapping as of Eq. 5.70
2. Memory size limitation in Eq. 5.71
3. Data separation in Eq. 5.79
4. Data pairing in Eq. 5.80
5. Data Aging in Eq. 5.42

If a constraint is violated, the solution, i.e., a chromosome encoding the label allocation matrix, is marked invalid and omitted for the GA's crossover operation, respectively it is not chosen for reproduction to form a new population. To speed up the solution process and weighting the optimization goals (a) accumulated normalized label access delay and (b) average CAN message response time, a reference is required to align the fitness function to. Then each optimization goal can be normalized with regard to an initial solution, which is defined by the first valid solution found during the GA resolution shown as the denominators of Eq. 5.85.

$$\text{minimize } f(\mathcal{S}) = \frac{MC_l}{MC_l^1} + \frac{\overline{R}^N}{\overline{R}^{N,1}} \tag{5.85}$$

More information on the GA configuration and results of mapping data to memory of the hardware primarily assessed by the total data mapping cost MC_l is given in Section 7.5. A concrete label mapping affects results of the task to PU mapping namely (accumulated) task response times $\sum_i R_i^+$, (accumulated) task chain reaction and age delays $\sum_\gamma \rho_\gamma$ and $\sum_\gamma \alpha_\gamma$, PU utilization, but also blocking and contention times.

5.7.6 Summary on Memory Mapping

The presented description of typical memory mapping constraints under a variety of automotive specific hardware properties as well as the consideration of CAN network properties as an example, provides valuable insights into modern memory mapping for automotive systems. The formal outline and description of constraints, timely costs, and optimization criteria present important characteristics and foundations for data mapping analysis. The application of CP and EA being appropriate meta heuristics to meet the problem's intractability yield optimized solutions incorporating multiple optimization goals and the consideration of various constraints. Therefore, CP provides easy to use variables, domains, and constraint types in terms of implementation, which, in contrast, need to be manually programmed for the EA at its fitness function. However, the latter benefits from shorter resolution time in most cases.

The next Section 5.8 presents a novel approach to reduce global blocking interference for Autosar based on Amalthea. The outlined concept improves execution and response times by reducing busy waiting for globally shared mutually exclusive resources across tasks.

5.8 Improved Global Critical Section Management

CSs are one of the major influences on jitter and execution time deviation in a distributed multi-PU context. Since the Autosar standard intends to use spinlocks for global CSs, varying busy waiting time frames occur due to tasks spin-locking on CSs, which is inefficient for inherently long CSs. Foundations and backgrounds to spin-locking is presented in Section 2.8.2 and a further analysis of blocking times B^s, which correspond busy waiting, is presented in Section 5.4.1 (cf. $w^+_{CS}(\tau_i)$ for s-Blocking and $w^+_{CS}(Sem_k, \tau_i)$). In terms of Amalthea, a CS is defined by a not empty accessed label union of two tasks either mapped to the same PU for local CSs or mapped to different PUs for global CSs, i.e. $CS^\Theta : \emptyset \neq \mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j}$ with $i \neq j, M^P_{\tau_i} = M^P_{\tau_j}$ and $CS^\phi : \emptyset \neq \mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j}$ with $i \neq j, M^P_{\tau_i} \neq M^P_{\tau_j}$, respectively.

The automotive industry uses spinlocks due to (a) implementation simplicity, (b) the resulting ease of determining timely coherencies, (c) spinlocks maintaining cache affinity, (d) the avoidance of scheduler invocations and context switches, and (e) claiming that the CS accesses are inherently short. Nonetheless, spinlocks permit priority inversion and deadlocks, of which the former are usually only bounded in a FIFO-queuing sense. Some of these challenges are avoided by preventing spinlock nesting entirely. This nesting avoidance is implemented either by returning an error to the task requesting an already hold lock or via suspending all interrupts such that the *locking* task can not be preempted. Both approaches are ineffective regarding implementation overhead and starvation [255]. Additionally, development efforts have to be explicitly invested to ensure such inherently short CS accesses. Also, CS locking times may increase in modern automotive systems due to the growth of shared data in, e.g., image processing or the use of CEs. As a consequence, spinlocks may impose significant blocking delays to the system, as shown in [35][51]. Despite that, it is desired not to deviate from the Autosar defined standard by, e.g., introducing new mechanisms to cope with global CSs. Instead, the spin locking periods can potentially be evaded due to flexibility in executing instructions of a task based on runnables while preserving the spinlock mechanism. This flexibility stems from runnable dependency graphs presented in the partitioning Chapter 4.

Whenever two or more runnables exist for the same topological level of a runnable DAG, i.e., runnable fork dependencies exist within the same task, the corresponding runnables' execution order has no influence on the task outputs and also no effect on the task's results. In other words, two runnables r_a and r_b on the same topological runnable DAG level within the same task are allowed to switch positions such that either $r_a \prec r_b$ or $r_b \prec r_a$, since no order constraint, respectively RSC, is violated and hence no deviation in execution time or data propagation occurs. This fundamental reordering concept also prevents race conditions since the task result does not change even when using a reordered runnable set. This ordering flexibility not only holds for the same topological level but also for the entire graph paths, as shown in the following Example 5.6.

[51]Wieder et al. investigate Autosar spinlock blocking in [35] especially regarding queuing such as FIFO, unordered, priority-ordered, and combined priority- and FIFO-ordered spinlock queues under preemptable and non-preemptable task scheduling.

Example 5.6: Runnable Ordering Flexibility

Given five runnables $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ with dependencies $\mathcal{E} = \{\{r_1, r_2\}, \{r_2, r_3\}, \{r_1, r_4\}, \{r_4, r_5\}\}$, there exist six possibilities to order runnables within the task, i.e.,

$$\begin{aligned}
s_1 &= \{r_1, r_2, r_3, r_4, r_5\};\\
s_2 &= \{r_1, r_2, r_4, r_3, r_5\};\\
s_3 &= \{r_1, r_2, r_4, r_5, r_3\};\\
s_4 &= \{r_1, r_4, r_2, r_3, r_5\};\\
s_5 &= \{r_1, r_4, r_2, r_5, r_3\};\\
s_6 &= \{r_1, r_4, r_5, r_2, r_3\};
\end{aligned}$$

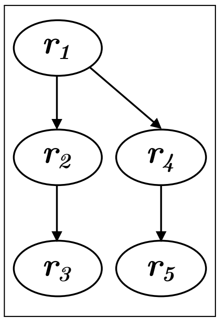


Figure 5.13: Simple example runnable DAG

Using any of these six task permutations effects the same task result due to no RSC, respectively cause-effect, violation.

Assuming possible preemption at any time or at runnable bounds for preemptive and cooperative tasks respectively, a change of the runnable order does not affect the system, since task results are published at the end of the execution or at the end of the period for implicit and LET communication respectively. Based on such flexibility in ordering runnables within a task, tasks can avoid busy waiting by reordering runnables for estimating concurrent situations. This approach is denoted TDRR in the following. TDRR assumes that CSs accesses are not part of accessed labels being copied into local memory by a task's release time any copied back by the task's response time (at next arrival for LET). Hence, CSs are explicitly part of a runnable in form of spinlock *locking* and *unlocking* activity graph items. For measurements presented in Chapter 7, these spinlock activity graph items are generated for all accesses to globally shared labels. In other words, copied labels for implicit LET communication are those locally shared, or not shared at all, and globally accessed labels are handled in explicit fashion and extended by spinlock accesses.

Some research activities can be found that address the challenge of reducing CS accesses in AUTOSAR such as [256]. Via using the concept of pseudo resources, resource locking sequence lists are calculated offline, and resource accesses are then called at appropriate points in time depending on those offline calculated lists. Pseudo resources are used to increase a task's priority by specific values required for fixed-priority scheduling with deferred preemptions. To avoid deadlocks and priority inversion, CSs are exited in reverse order as they entered the CSs. However, in contrast to [256], TDRR applies to runnables within a task based on the runnable DAG and it is not limited to limited-preemptive fixed priority scheduling.

In contrast to RunPar [191] that schedules tasks sequentially, TDRR assumes that runnables of different tasks can be executed concurrently, resulting in less idle intervals. Moreover, the supertask approach presented in [257] improves response time based on the speedup metric for a single instance of a task but has the disadvantage of requiring a least common multiple period of tasks contained in the supertask. Consequently, the runnables of tasks with higher periods are scheduled more often, and the total system load is increased.

Due to the construction of tasks, the task to PUs mapping, and scheduling being statically defined once during application configuration [191], runtime calculation of new runnable orders may involve excessive overheads. However, Kluge et al. have shown in [166] that online task filtering can be used in AUTOSAR to efficiently schedule tasks while considering resource conflicts and active tasks. This approach can be extended to schedule task instances with off-line calculated runnable orders for a given point in time to reduce busy waiting periods while not violating precedence constraints.

In general, managing mutual exclusion of shared resources has been addressed by various locking protocols and evolved to meet different challenges, e.g., dynamic priorities or nested global CSs. Instead of busy waiting for a resource for spinlock protocols, semaphore protocols use the suspension-based scheme to yield lower priority tasks to execute if an occupied resource blocks the higher priority task. Such protocols have been investigated for priority inversion, schedulability analysis, deadlocks, mutual exclusion, nested resource accesses and more properties and further protocols were developed, e.g., FMLP [172], Real-time Nested Locking Protocol (RNLP) [258], Preemptable Waiting Locking Protocol (PWLP) [259], O(m) Locking Protocol (OMLP) [260] and others. A recent study [261] shows for global fixed-priority scheduling that the early FMLP and PIP protocols still outperform newer semaphore protocols in most scenarios. In [262], Brandenburg shows how MPCP, FIFO Multiprocessor Locking Protocol (FMLP+), DPCP, and Distributed FIFO Locking Protocol (DFLP) perform for partitioned fixed-priority scheduling. Alternatively, Gai et al. further provide in [171] blocking analyses for local blocking, non-preemptive blocking, and remote blocking upon the MSRP protocol, i.e., one of the few spinlock-based protocols applicable to AUTOSAR.

The significant difference of TDRR compared to above mentioned suspension-based semaphore protocols and MSRP is the crucial concept of preserving the spinlock global CS protection mechanism while minimizing busy waiting within the same task, i.e., exploiting computing resources by runnables of the same task instead of grating the resources to other tasks. This approach significantly eases the analysis of cause-effect chains and timely determination across blocking, contention, and response times. Hence, TDRR aims at filling the gap between semaphore-based protocols and spinlock-based protocols, such that the latter can also be applied to applications with longer CSs without creating significant spinning delays. Therefore, TDRR assumes given initial runnable orders from the partitioning process. In [92, 93] and [4], respectively Chapter 4, precedence constraint based partitioning is presented for the purpose of parallelizing program code in form of a runnable set based on the AMALTHEA model. CPP, ESSP, and CP-based partitioning strategies of Chapter 4 preserve precedence constraints, which are required for TDRR and ensure a causally correct task behavior. Other approaches apply to TDRR too, but require re-validation if precedence constraints are violated.

In the following, the TDRR approach is illustrated to achieve a reduction of remote blocking under the assumption of spinlocks being used to protect globally shared resources.

5.8.1 Task-Release-Delta-based Runnable Reordering

TDRR serves the purpose of *sequentializing* parallel accesses to shared resources, resulting in reduced task response times, improved timing predictability, and increased parallel efficiency. To achieve sequential resource accesses, runnables are reordered within a task based on precedence constraints and the system state in form of release-delta times. Instead of introducing another protocol, an estimation of possible resource conflicts, i.e., concurrent accesses to a shared resource according to task release times, is used to mark situations in which busy waiting occurs. Based on the release time of a predecessor task $\blacktriangle_{\tau_{\prec}}(\tau_i)$ and the release time of the current task (to be scheduled) $\blacktriangle_{\tau_i}$, a runnable order is chosen from a list of possible task release delta situations $\delta_{\tau_j \to \tau_i} = \{\sigma_{i,j,1}, ...\}$ calculated at design time to reduce conflicts as much as possible. Hence, TDRR is subdivided into the following six processes for all possible task pairs.

(I) Identification of all tasks' access intervals to CSs.

(II) Calculation of all possible task release delta conflicts and corresponding intervals based on release delta situations $\sigma_{i,j,x} \in \delta_{\tau_i \to \tau_j} \in \Delta^{\phi}$ that result in busy waiting due to spin locks being accessed concurrently.

(III) Combination of release delta conflict intervals for considering multiple CSs for interleaving release-delta conflicts.

(IV) Retrieving all conflicting runnables' conflict interval for a specific release delta conflict.

(V) Calculating runnable orders $\mathcal{RO}_{i,j,x}$ for every release delta conflict interval $\sigma_{i,j,x} \in \delta_{\tau_i \to \tau_j} \in \Delta^{\phi}$.

(VI) Keeping track of release times at runtime to derive task release delta values upon a task's release and choose a design-time calculated runnable order at run time correspondingly.

A runnable order $\mathcal{RO}_{i,j,x}$ defines a runnable sequence for the x-th conflict interval $\sigma_{i,j,x}$ for $\delta_{\tau_i \to \tau_j} \in \Delta^{\phi}$. The approach's benefit is the analysis of concurrent accesses to shared memory at early design steps, its application to AUTOSAR models, and improved system performance due to reduced task execution and response times. The TDRR concept is shown in Figure 5.14.
Figure 5.14 shows the runnable dependency graph on the left, which is used to (1) create tasks in form of initial runnable orders during the partitioning process shown at top centered rectangle in Figure 5.14 and (2) calculate additional runnable orders based on specific release time delta conflicts via TDRR shown at the bottom centered rectangle of Figure 5.14. If a specific release delta value is detected at runtime of the program for a task pair, which means that the release delta is in between the release delta conflict interval shown in the right part of Figure 5.14, the corresponding new runnable order replaces the initial order.

A task constructed by the partitioning process defines the initial runnable order based on precedence constraints, which are preserved for new runnable orders calculated by the TDRR process (V). Precedence constraints are derived from the edge set $\mathcal{E}$. A task τ_i of the task set $\mathcal{T}$ contains a runnable order in form of its activity graph, which contains runnable call activity graph items based on the outcome of the partitioning. An alternative

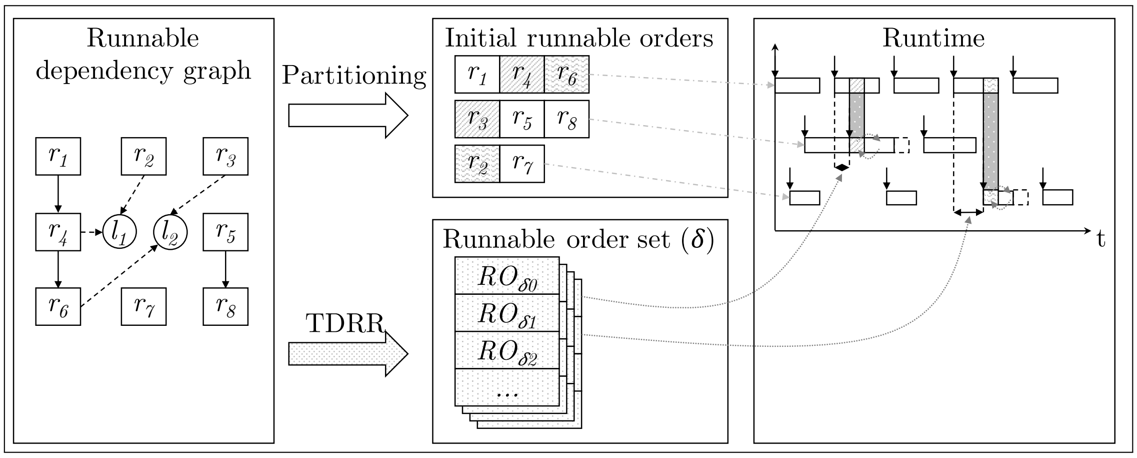


Figure 5.14: TDRR concept to reduce busy waiting

runnable order $\mathcal{RO}_{i,j,x}$ is a task's runnable order permutation respecting all precedence constraints. TDRR introduces some new notations outlined in the following Table 5.4. In general, a conflict interval ci consists of a start and an end time, i.e., $ci = [t_s, t_e)$. The superscript notation ^{p} denotes derivation from partitioning, i.e. initial runnable orders. Here, $\mathcal{R}^{\dashv}(\sigma^{\delta}_{i,j,x})$ denotes the set of runnables in task τ_i, of which each shares

Description	Symbol & Details
Runnable access conflict interval	$ci^p_{r_a}$ cf. Eq. 5.88
Task's access conflicts interval set	$ci^p_i = \bigcup_{r_a \in \mathcal{R}(\sigma^p_x)} ci^p_{r_a}$
Task's access conflicts	$\sigma^p_i = \{\sigma^p_{i,1}, ...\} : \sigma^p_{i,x} = \{\mathcal{R}_{i,x}, \{CS_{i,x,1}, ...\}\}$
Task release delta conflict matrix	$\Delta^{\phi}(n \times n)$
Task release delta conflict set of a task pair	$\delta_{\tau_i \to \tau_j} = \{\sigma^{\delta}_{i,j,1}, ...\}$ i-th row and j-th column of Δ^{ϕ}
Delta conflict	$\sigma^{\delta}_{i,j,x} = \{ci^{\delta}_{i,j,x}, \mathcal{R}^{\dashv}, \mathcal{R}^{\vdash}\}$
Runnable conflict interval for a release delta and runnable	$ci^{r_b}_{i,j,x} : r_b \in \left(\tau_j \cup \mathcal{R}^{\vdash}(\sigma^{\delta}_{i,j,x})\right)$
Runnable order	$\mathcal{RO}_{i,j,x}$ deals with $(\sigma^{\delta}_{i,j,x})$

Table 5.4: TDRR notations

one or more labels with at least one runnable in $\mathcal{R}^{\vdash}(\sigma^{\delta}_{i,j,x})$ of task τ_j. Conversely, every runnable in $\mathcal{R}^{\vdash}(\sigma^{\delta}_{i,j,x})$ shares one or more labels with at least one runnable in $\mathcal{R}^{\dashv}(\sigma^{\delta}_{i,j,x})$. Notations of Table 5.4 are exemplary applied to the example runnable DAG of Figure 5.14 in Example 5.7.

Example 5.7: TDRR Notations of Figure 5.14's Runnable DAG

For the example runnable DAG of Figure 5.14, the following details are calculated under the assumption that every runnable executes a single instruction / tick.

$$\begin{aligned}
\sigma^p_{\tau_1} &= \{\{\{r_4\},\{l_1\}\},\{\{r_6\},\{l_2\}\}\} \\
ci^p_{r_4} &= [1,2); ci^p_{r_6} = [2,3) \\
ci^p_{\tau_1} &= \{ci^p_{r_4}, ci^p_{r_6}\} \\
\sigma^p_2 &= \{\{r_3\},\{l_2\}\} \\
ci^p_{\tau_2} &= \{[0,1)\} \\
\sigma^p_3 &= \{\{r_2\},\{l_1\}\} \\
ci^p_{\tau_3} &= \{[0,1)\} \\
\Delta^\phi &= \begin{pmatrix} 0 & [0,2) & [1,3) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
\sigma^\delta_{\tau_1,\tau_2,1} &= \{[0,2),\{r_4\},\{r_3\}\} \\
\sigma^\delta_{\tau_1,\tau_3,1} &= \{[1,3),\{r_6\},\{r_2\}\}
\end{aligned}$$

Based on the above data, runnable orders can be calculated, e.g., for $\sigma^\delta_{\tau_1,\tau_2,1} = \{[0,2),\{r_4\},\{r_3\}\}$, so that r_3 is not executed within $[0,2)$ via, e.g., runnable order $\mathcal{RO}_{\tau_1,\tau_2,1} = \{r_5, r_8, r_3\}$, which avoids busy waiting at runnable r_3 for τ_1 being released in between $[0,2)$ before τ_2.

The purpose and methodology of corresponding calculations is presented in the next Section 5.8.2.

5.8.2 Calculating Potential Conflicts

This section includes processes (I)–(III) mentioned in the previous Section 5.8.1.

Firstly, a conflict of a task pair requires that the intersection of written labels is not empty as shown in Eq. 5.86.

$$\left(\downarrow_{\tau_i} \cap \downarrow_{\tau_j}\right) \neq \emptyset \text{ with } i \neq j \wedge M^P_{\tau_i} \neq M^P_{\tau_j} \wedge \left(\downarrow_{\tau_i} \cap \downarrow_{\tau_j}\right) \subseteq CS^\phi \tag{5.86}$$

Due to the runnables potentially accessing different global CSs, and multiple tasks being released before the task under analysis accessing those CSs, the idea is first to investigate all possible access conflicts across task pairs, which can, later on, be considered in common sense to identify the most valuable runnable order for a runtime situation (cf. Section 5.8.4). Hence, a $(n \times n)$ matrix for all tasks is used that defines for every task pair a set of release delta conflicts that define the cause and situation of busy waiting. This matrix is denoted as Δ^ϕ with entries $\delta_{\tau_i \to \tau_j}$, which correspond the i-th row and j-th column of Δ^ϕ as shown in Eq. 5.87.

$$\Delta^{\phi} = \begin{pmatrix} \delta_{\tau_0 \to \tau_0} & \dots & \delta_{\tau_0 \to \tau_n} \\ \vdots & \ddots & \vdots \\ \delta_{\tau_n \to \tau_0} & \dots & \delta_{\tau_n \to \tau_n} \end{pmatrix} : n = |\mathcal{T}| \tag{5.87}$$

Within Δ^{ϕ}, a single element defines a release delta conflict set $\delta_{\tau_i \to \tau_j} = \{\sigma^{\delta}_{i,j,1}, ...\}$, each of which consists of a delta conflict interval $ci^{\delta}_{i,j,x}$, a former runnable set $\mathcal{R}^{\dashv}_{i,j,x} \subseteq \mathcal{R}_i$, and a latter runnable set $\mathcal{R}^{\vdash}_{i,j,x} \subseteq \mathcal{R}_j$ with $\forall r_a \in \mathcal{R}^{\dashv}_{i,j,x}, \forall r_b \in \mathcal{R}^{\vdash}_{i,j,x} : (\downarrow_{r_a} \cap \downarrow_{r_b} \cap CS^{\phi}) \neq \emptyset$. Collecting both conflicting runnable sets is essential to, later on, derive the time frame the latter runnables must not be executed, which depends on the time frame of the former runnables. An interval $[t_s, t_e)$ includes the lower but excludes upper limit. The release delta conflict defines that if τ_i is released within the intervals $ci_{i,j,x}$ before τ_j, a busy waiting period occurs. In other words, the relative release time of τ_i, denoted as $\blacktriangle_{\tau_i}(t)$, must be within any release delta conflict interval $\left(t_s(ci^{\delta}_{i,j,x}) \leq \blacktriangle_{\tau_i}(t) < t_e(ci^{\delta}_{i,j,x})\right) \forall x \in \delta_{\tau_i \to \tau_j}$ prior to τ_j for busy waiting to occur. The relative release time $\blacktriangle_{\tau_i}(t)$ is the delta, i.e. difference, of a task's release time to time t, e.g. assuming τ_i was released at $\blacktriangle_{\tau_i} = 4$ and the current time is $t = 14$, then $\blacktriangle_{\tau_i}(t) = t - \blacktriangle_{\tau_i} = 14 - 4 = 10$.

(I) The calculation of access intervals is based on the start and end times of runnables accessing CSs at their initial order (partitioning), which are given as $ci^{p}_{r_a} : (\downarrow_{r_a} \cap CS^{\phi}) \neq \emptyset$. In contrast to release delta conflicts $\sigma^{\delta}_{i,j,x}$, an access conflict σ^{p}_{i} contains a single runnable set and a CS set instead of a conflict interval and two runnable sets. Since runnables of a task can access different CSs multiple times, all access-intervals for a task pair to CSs accessed by these two tasks must be investigated. These intervals are collected together with the corresponding CSs so that the CS access interval sets for two tasks are given as $\sigma^{p}_{i} = \{\sigma^{p}_{i,x}, ...\}, \sigma^{p}_{j} = \{\sigma^{p}_{j,y}, ...\} : \sigma^{p}_{i,x} = \{\mathcal{R}_{i,x}, \{CS_{i,x}\}\}$. A runnable access conflict interval is defined in Eq. 5.88.

$$\begin{aligned} ci^{p}_{r_a} &= [t^{p}_{s}, t^{p}_{e}) : t^{p}_{s} < t^{p}_{e} \\ t^{p}_{s}(ci^{p}_{r_a}) &= t^{p}_{s}(r_a) = \sum_{b=1}^{b=pos_{r_a}-1} c_{r_b} \\ t^{p}_{e}(ci^{p}_{r_a}) &= t^{p}_{e}(r_a) = t^{p}_{s}(r_a) + c_{r_a} \end{aligned} \tag{5.88}$$

Here, $t^{p}_{s}(r_a) = t_s(ci^{p}_{r_a})$ denotes the start time of runnable r_a within its task based on the initial order created by the partitioning and is defined by the sum of runnable execution times of runnables positioned prior to r_a, i.e. $t^{p}_{s}(r_a) = \sum_{b=1}^{b=pos_{r_a}-1} c_{r_b}$ and pos_{r_a} representing the position of a runnable within the task τ_j[52].

(II) Afterwards, the release delta conflict intervals are calculated for every task pair's access intervals that (i) access the same CS and (ii) hold the condition that $t^{p}_{s}(ci^{p}_{r_a}) < t^{p}_{s}(ci^{p}_{r_b}) : r_a \in \mathcal{R}_{i,x}; r_b \in \mathcal{R}_{j,y}; (\downarrow_{r_a} \cap \downarrow_{r_b} \cap CS^{\phi}) \neq \emptyset$, i.e. the access interval of the former released task starts *later* than the access interval of the latter task. This is calculated in Algorithm 5.2 line 5 i.e. via Eq. 5.89.

$$\begin{aligned} ci^{\delta}_{i,j,x} &= [t_s, t_e) : t_s < t_e \\ t_s(ci^{\delta}_{i,j,y}) &= (t^{p}_{s}(ci^{p}_{r_a}) - t^{p}_{e}(ci^{p}_{r_b})) \cap \mathbb{Q}^{+}_{0} \text{ with } r_a \in \mathcal{R}(\sigma^{p}_{i,x}); r_b \in \mathcal{R}(\sigma^{p}_{j,y}) \\ t_e(ci^{\delta}_{i,j,y}) &= (t^{p}_{e}(ci^{p}_{r_a}) - t^{p}_{s}(ci^{p}_{r_b})) \cap \mathbb{Q}^{+}_{0} \text{ with } r_a \in \mathcal{R}(\sigma^{p}_{i,x}); r_b \in \mathcal{R}(\sigma^{p}_{j,y}) \end{aligned} \tag{5.89}$$

[52]For instance, given $\tau_i = \{r_a, r_b, r_c\} \Rightarrow pos_{r_c} = 3$

Here, the '$\cap\mathbb{Q}_0^+$' notation is included to extract negative values in case the end time of r_b is higher than r_a's start time. Positive rational numbers are chosen here, to cover situations handling, e.g., fractions of a second. If time values are all given based on a smallest scale, e.g., pico seconds, $\mathbb{Q}_0^+$ can be replaced with $\mathbb{N}$. The following Example 5.8 shows this process along with a simple hypothetical model.

Example 5.8: Calculation of Access Intervals / Conflicts

Given is a task τ_j calling three runnables in order $\mathcal{R}_j = \{r_1, r_2, r_3\}$, each of which executing one time unit whereas r_2 executes a CS CS_k. First, the access interval for τ_j is $ci_{r_2}^p = \{[1,2)\}$ and the access conflict is $\sigma_{\tau_j}^p = \{\{r_2\}, \{CS_k\}\}$. A second task τ_i calls another three runnables $\mathcal{R}_i = \{r_4, r_5, r_6\}$, each of which executing two time units and the second runnable r_5 executes the same CS CS_k. Consequently, the access interval is $ci_{r_5}^p$ is $\{[2,4)\}$ and the access conflict is $\sigma_{\tau_i}^p = \{\{r_5\}, \{CS_k\}\}$. Based on Eq. 5.89, the resulting delta conflict interval is $ci_{i,j,1}^\delta = [2-2, 4-1) = [0,3)$ and the conflict $\sigma_{i,j,1} = \{\{[0,3)\}, \{r_2\}, \{r_5\}\}$.

Note that there is no conflict for calling τ_j before τ_i in this example, since no access conflict's start time of task τ_j is higher than an access conflict start time at τ_i.

(III) To consider all possible busy waiting situations across CS accesses on a holistic level, release delta conflict interval intersections must be found and split into (1) the lower relative complement, (2) the intersection, and (3) the higher complement. The new intersection conflict (2) combines both conflicts' CS accesses and a corrsponding runnable set is derived that should be avoided for being executed at the latter task. This interval splitting process is outlined in the following Example 5.9.

Example 5.9: Splitting Conflict Intervals

Assuming two delta conflicts $\sigma_{i,j,1}^\delta = \{[1,4), \{r_a\}, \{r_c\}\}, \sigma_{i,j,2}^\delta = \{[3,5), \{r_b\}, \{r_c\}\}$, a release delta conflict intersection exists at $[3,4)$ for runnables r_a and r_b. Hence, the intersecting release delta conflicts are divided into $\sigma_{i,j,1} = \{[1,3), \{r_a\}, \{r_c\}\}, \sigma_{i,j,2} = \{\{[3,4)\}, \{r_a, r_b\}, \{r_c\}\}, \sigma_{i,j,3} = \{[4,5), \{r_b\}, \{r_c\}\}$, which correspond the lower complement, intersection, and higher complement.

The formalism of merging release delta conflicts via interval intersections is shown in lines 13–22 of Algorithm 5.2. As a consequence, all release delta conflict intervals for a $\delta_{\tau_i \to \tau_j}$ set are distinct as shown in Eq. 5.90.

$$\begin{aligned}\forall x, y \text{ with } \sigma_{i,j,x}, \sigma_{i,j,y} \in \delta_{\tau_i \to \tau_j}; (x,y) \in \mathbb{N}; x < y :\\ t_s(ci_{i,j,y}^\delta) \geq t_e(ci_{i,j,x}^\delta);\\ ci_{i,j,x}^\delta \cap ci_{i,j,y}^\delta = \emptyset\end{aligned} \tag{5.90}$$

The following Algorithm 5.2 provides the process of calculating release delta conflicts $\delta_{\tau_i \to \tau_j}$

based on access intervals so that the former contain (a) the release delta intervals resulting in busy waiting for τ_i being released prior to τ_j and (b) the corresponding runnable sets $\mathcal{R}^{\dashv}_{i,j,x}, \mathcal{R}^{\vdash}_{i,j,x}$ for every release delta conflict interval that provides necessary data to reorder runnables at τ_j by shifting $\mathcal{R}^{\vdash}_{i,j,x}$ backwards based on $\mathcal{R}^{\dashv}_{i,j,x}$ as shown in the $\mathcal{RO}$ Algorithm 5.3, to reduce busy waiting.

Algorithm 5.2: Release-δ Interval Conflict Calculation based on Access Conflicts

```
Data: CS access intervals for two tasks $\sigma^p_i, \sigma^p_j$
Result: Release delta conflict set $\delta_{\tau_i \to \tau_j}$
1  n←1
2  $\delta_{\tau_i \to \tau_j} \leftarrow \emptyset$
3  forall access conflicts $\sigma^p_{j,y} \in \sigma^p_j$ do
4      forall access conflicts $\sigma^p_{i,x} \in \sigma^p_i : \emptyset \neq (CS_{i,x} \cap CS_{j,y}{}^{a}); t_e(ci^p_{r_a \in \sigma^p_{i,x}}) \geq t_s(ci^p_{r_b \in \sigma^p_{j,y}})$ do
5          $ci^\delta_{i,j,n} \leftarrow [t_s(ci^p_{i,x}) - t_e(ci^p_{j,y}), t_e(ci^p_{i,x}) - t_s(ci^p_{j,y}))$; /*set release delta conflict interval*/
6          $\mathcal{R}^{\dashv}_{i,j,n}$ = runnables accessing $CS_{j,y}$ from
             $\tau_i : (CS_{j,y} \in \downarrow r_a) \wedge t_s(ci^p_{r_a}) - t_s(ci^\delta_{i,j,n}) = t_e(ci^p_{r_b}) \forall r_a \in \mathcal{R}^{\dashv}_{i,j,n}$
7          $\mathcal{R}^{\vdash}_{i,j,n}$ = runnables accessing $CS_{i,x}$ from
             $\tau_j : (CS_{i,x} \in \downarrow r_b) \wedge t_e(ci^p_{r_b}) + t_s(ci^\delta_{i,j,n}) = t_s(ci^p_{r_a}) \forall r_b \in \mathcal{R}^{\vdash}_{i,j,n}$
8          $\sigma^\delta_{i,j,n} \leftarrow \{ci^\delta_{i,j,n}, \mathcal{R}^{\dashv}_{i,j,n}, \mathcal{R}^{\vdash}_{i,j,n}\}$;                /*set release delta conflict*/
9          $\delta_{\tau_i \to \tau_j} \leftarrow (\delta_{\tau_i \to \tau_j} \cup \sigma^\delta_{i,j,n})$;      /*Changing result → add new release delta conflict*/
10         n←n+1
11     end
12 end
13 forall $y, x$ with $(\sigma^\delta_{i,j,x}, \sigma^\delta_{i,j,y}) \in \delta_{\tau_i \to \tau_j} : x \neq y$ do
14     if $(ci^\delta_{i,j,x} \cap ci^\delta_{i,j,y}) \neq \emptyset$ then
15         $ci^\delta_{i,j,n} \leftarrow (ci^\delta_{i,j,x} \cap ci^\delta_{i,j,y})$; /*set new release delta conflict interval via intersection*/
16         $\mathcal{R}^{\dashv}_{i,j,n} \leftarrow (\mathcal{R}^{\dashv}_{i,j,x} \cup \mathcal{R}^{\dashv}_{i,j,y})$;                  /*combine conflicting prior runnables*/
17         $\mathcal{R}^{\vdash}_{i,j,n} \leftarrow (\mathcal{R}^{\vdash}_{i,j,x} \cup \mathcal{R}^{\vdash}_{i,j,y})$;                 /*combine conflicting latter runnables*/
18         $\sigma_{i,j,n} \leftarrow \{ci^\delta_{i,j,n}, \mathcal{R}^{\dashv}_{i,j,n}, \mathcal{R}^{\vdash}_{i,j,n}\}$;                   /*set new release delta conflict*/
19         $\delta_{\tau_i \to \tau_j} \leftarrow (\delta_{\tau_i \to \tau_j} \cup \sigma_{i,j,n})$;      /*Changing result → add new release delta conflict*/
20         $ci^\delta_{i,j,x} \leftarrow (ci^\delta_{i,j,x} \setminus ci^\delta_{i,j,y})$;             /*Changing result → reduce to lower complement*/
21         $ci^\delta_{i,j,y} \leftarrow (ci^\delta_{i,j,y} \setminus ci^\delta_{i,j,x})$;             /*Changing result → reduce to upper complement*/
22         n←n+1
23 end
24 sort release delta conflicts chronologically and update indexes
```

[a] $CS_{i,x}$ denotes the CS set for access interval $\sigma^p_{i,x}$

In the following, a 'conflict' relates to a release delta conflict if 'access conflict' is not explicitly mentioned. Lines 13–22 of Algorithm 5.2 create new conflicts based on corresponding interval intersections to significantly reduce the total number of conflicts. This may sound contradicting, but due to lines 20–21 often result in empty conflict intervals if one is a proper subset of the other, i.e. $t_s(ci^\delta_{i,j,x}) \geq t_s(ci^\delta_{i,j,y}) \wedge t_e(ci^\delta_{i,j,x}) \leq t_e(ci^\delta_{i,j,y})$, the conflict combination actually results in a reduced number of conflicts. The implementation of the algorithm further ensures that release delta interval complements (set in lines 20–21) are distinct such that if a complement results in two or three intervals, a new conflict is created accordingly (e.g. $[2,8) \setminus [3,4) \Rightarrow [2,3),[4,8)$). Furthermore, lines 13–22 ensure that all runnables potentially conflicting for a release time interval are considered on a holistic basis and that all conflicts are distinct for $\delta_{\tau_i \to \tau_j}$, i.e. $\forall (x,y) \in \mathbb{N}, x < y$ with $\sigma_{i,j,x}, \sigma_{i,j,y} \in \delta_{\tau_i \to \tau_j} : (t_e(ci^\delta_{i,j,x}) \leq t_s(ci^\delta_{i,j,y}))$. This approach is realized to avoid inserting a runnable at a conflict situation that solves one conflict but not another one due to different CS accesses. For example, having $\sigma_{i,j,x} = \{[1,2),\{r_a\},\{r_c\}\}$ and $\sigma_{i,j,y} = \{[0,5),\{r_b\},\{r_c\}\}$ after executing lines 3–12, lines 13–22 result in $\sigma_{i,j,x} =$

$\{\emptyset, \{r_a\}, \{r_c\}\}, \sigma_{i,j,y} = \{\{[0,1),[2,5)\}, \{r_b\}, \{r_c\}\}$, and $\sigma_{i,j,n} = \{\{[1,2)\}, \{r_a, r_b\}, \{r_c\}\}$. Finally, line 24 cleans up results since lines 20–22 probably result in inconsistent indexes n and the chronological sorting of conflicts is required for Algorithm 5.3 of the next section.

Instead of lines 13–22 of Algorithm 5.2, which combine and split conflicts, merging them is possible, too. The merge process could reduce the number of runnable orders even further and avoid likely redundant $\mathcal{RO}$, however, it results in more runnables being prevented to execute at the access interval so that finding runnable orders to avoid busy waiting gets more difficult. Not only the number of runnables increases, but also the runnable conflict interval $ci^{r_a}_{i,j,x}$ length, these runnables are not *supposed* to be executed within, increases. Ultimately, no available runnables may fit into the runnable conflict interval and less busy waiting can be reduced. Thus, instead of merging multiple conflict intervals, the chosen algorithm, used in analyses of Section 7.4, splits and combines them resulting in a less reduction runnable order number on the one hand but potentially increased busy waiting reduction.

5.8.3 Calculating Runnable Orders for Conflicts

This section includes processes (IV) and (V) outlined in Section 5.8.1. The calculated runnable order is distinct for three parameters, i.e. a predecessor task τ_i, a task to which the runnable order applies τ_j, and a delta conflict interval $ci^{\delta}_{i,j,x} \in \sigma_{i,j,x}$.

(IV) Due to a runnable order considering not a precise delta release time but an entire release delta conflict interval, the time frame for a conflicting runnable must be calculated correspondingly. This approach is realized using the runnable's access interval and extending it by the conflict interval length, as shown in Eq. 5.91.

$$\begin{aligned} ci^{r_b}_{i,j,x} &= [t_s, t_e) \\ t_s(ci^{r_b}_{i,j,x}) &= (t^p_s(r_a) - t_e(ci^{\delta}_{i,j,x})) \cap \mathbb{Q}^+_0 \\ t_e(ci^{r_b}_{i,j,x}) &= t^p_e(r_a) - t_s(ci^{\delta}_{i,j,x}) \text{ with } r_a \in \mathcal{R}^{\dashv}(\sigma^{\delta}_{i,j,x}); r_b \in \mathcal{R}^{\vdash}(\sigma^{\delta}_{i,j,x}) \end{aligned} \tag{5.91}$$

This process is visualized along with an example in Figure 5.15.

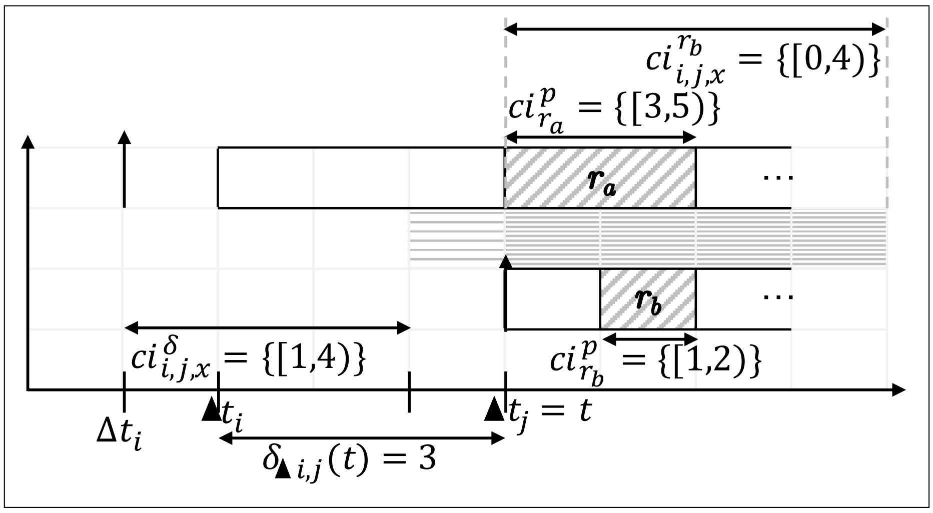

$ci^{r_b}_{i,j,x} = \{[0,4)\}$
$ci^{p}_{r_a} = \{[3,5)\}$
r_a
...
r_b
...
$ci^{\delta}_{i,j,x} = \{[1,4)\}$
$ci^{p}_{r_b} = \{[1,2)\}$
Δt_i
t_i
$t_j = t$
$\delta_{\blacktriangle i,j}(t) = 3$

Figure 5.15: Intervals of label accesses, release delta conflicts, and runnable conflicts

(V) The second last and most crucial TDRR process is the runnable order calculation. Therefore, precedence constraints on task level are used, i.e. all edge's source and target runnables are contained in the task under consideration. Precedence constraints (causality) of tasks across PUs may span over two task periods due to uncertainty of release jitter, such that considering causality within runnables of a single task is reasonable for TDRR. Precedence constraints are represented as edges $\mathcal{E}(DAG_{\tau_j})$ derived from AMALTHEA RSCs, which are part of the runnable DAG of a task τ_j. In addition to edges, a runnable DAG also contains runnables $\mathcal{R}(DAG_{\tau_j})$ so that $DAG_{\tau_i} = \{\mathcal{E}, \mathcal{R}\}$. A runnable's predecessor set is derived from all preceding paths $path_x^{\prec}(r_a) = \mathcal{E}_{r_a}^{\prec} \subseteq \mathcal{E}(DAG_{\tau_j}) : e^t_{|\mathcal{E}_{r_a}^{\prec}|} = r_a$ and its runnables are represented as shown in Eq. 5.92[53].

$$\mathcal{R}_{r_a}^{\prec} = \bigcup_{\varphi} e_{\varphi}^s : e_{\varphi} \in \bigcup_{x} path_x^{\prec}(r_a) \tag{5.92}$$

It is important to note that the runnable set positioned prior to runnable r_a, i.e. $\cup_{r_b} : pos_{r_b} < pos_{r_a}$ does not necessarily match its predecessors $\mathcal{R}_{r_a}^{\prec}$ but in turn, all predecessors must have a lower position, i.e. $\forall r_b \in \mathcal{R}_{r_a}^{\prec} : pos_{r_b,\tau_i} < pos_{r_a,\tau_i}$.

Using the runnable DAG and calculations described above, available runnables for replacing a conflicting runnable while not violating intra-task precedence constraints can be derived. Runnables available to be shifted towards a conflict interval have to fulfill the following three conditions. Firstly, since the initial order is retained before the conflict's start time, only *later* runnables can be moved to the conflict interval (first condition in Eq. 5.93). This approach ensures that no precedence constraint is violated due to reordering a predecessor runnable. In addition, runnables assignable to $t_s(ci_{i,j,x}^{r_a})$ must all have their predecessors assigned prior to the conflict interval start time (second condition in Eq. 5.93). Finally, reordering runnables of a task may result in new undetected conflicts that were not present using initial orders. Hence, calculating $\mathcal{R}^{avail}(t)$ only considers runnables that do not access CSs (third condition in Eq. 5.93).

$$\forall r_a \in \mathcal{R}^{avail}(t) \;:\; \begin{cases} t_s^p(r_a) & > t \\ \forall r_b \in \mathcal{R}_{r_a}^{\prec} : t_e(r_b) & \leq t \\ (\mathcal{L}_{r_a} \cap CS^{\phi}) & = \emptyset \end{cases} \tag{5.93}$$

The corresponding process of conflict identification and retrieving runnables for such conflicts is outlined along with the following Example 5.10.

[53]More DAG-based notations are given in the Appendix H.3.

Example 5.10: Identifying Conflicts & Potentially Resolving Runnables

Given are two tasks with runnable orders $\mathcal{R}_i = \{r_1, r_2, r_3\}$ and $\mathcal{R}_j = \{r_4, r_5, r_6\}$, whereas all runnables execute *one time unit*, except r_6, which consumes 2 *time units*. Runnables r_2 and r_5 access a global CS. No further dependencies exist between the runnables for simplicity reasons, and no further CSs are accessed to have the highest flexibility in reordering runnables. The start time of the conflict runnable r_5 is $t_s^p(r_5) = 1$ since only r_4 is called prior to r_5. Its end time is $t_e^p(r_5) = 1 + c_{r_5} = 2$.

For this example, the conflict set $\delta_{\tau_i \to \tau_j}$ contains only a single conflict due to having only one runnable pair accessing the same CS. The corresponding conflict interval is based on access conflicts, i.e. $\sigma_{\tau_i}^p = \{\{(1,2)\}, \{CS\}\}, \sigma_{\tau_j}^p = \{\{(1,2)\}, \{CS\}\}$ (cf. line 5 in Algorithm 5.2) such that the following conflict is calculated: $\sigma_{i,j,1}^\delta = \{\{(0,1)\}, \{r_2\}, \{r_5\}\}$. According to Eq. 5.91, the runnable conflict interval is then $ci_{i,j,1}^{r_5} = (1-1, 2+1) = (0,3)$. Available for jumping in for r_5 is only $\mathcal{R}^{avail}(t = 1 = t_s^p(r_b)) = \{r_6\}$ (cf. Eq. 5.93). Hence, calling the runnable order $\mathcal{RO}(\sigma_{i,j,1}) = \{r_4, r_6, r_5\}$ solves the conflict and busy waiting can be prevented.

To retrieve new runnable sequences, an algorithm is required to investigate valid and appropriate task permutations, i.e. runnable orders for conflicts $\mathcal{RO}(\sigma_{i,j,x}^\delta)$. This algorithm depends on (a) a task pair (τ_i, τ_j), (b) all conflicts $\sigma_{i,j,x} \in \delta_{\tau_i \to \tau_j}$, (c) the runnable dependency graph DAG(τ_j), and (d) the initial runnable orders $\mathcal{RO}_{\tau_j}^p$, provided by the partitioning process (cf. Chapter 4; (c) and (d) are indirectly given with τ_i, τ_j). Algorithm 5.3 shows the implementation of the described concept and calculates a set of runnable orders $\mathcal{RO}(\delta\tau_i \to \tau_j)$ for conflict intervals given in $\sigma_{i,j,x}$ via moving conflict-unaffected runnables to conflicting intervals based on their precedence constraints, i.e. $\forall r_a, r_b : pos_{r_a,\tau_i} < pos_{r_b,\tau_i}$ with $r_a \prec r_b$, execution time, and global CS accesses.

In Algorithm 5.3 line 4, it should be noted that t_s^p is based on the partitioning and calculated via Eq. 5.88, which, as noted already, includes predecessors $\mathcal{R}^{\prec}(r_a)$ but also runnables without a dependency relation to the considered runnable. In other words, runnables positioned prior to the runnable under consideration are addressed based on the initial task runnable order, which is denoted with $pos_{r_b} < pos_{r_a}$, i.e. r_b has a lower task position index than r_a. Lines 10 and 11 are used to fill the conflict interval as *soon* as possible, i.e., with a runnable that fits effectively into the conflict interval. Hence, via sorting the available runnables in line 10 and choosing one, filling the conflict interval most appropriately, ensures that the amount of runnables being *moved* is kept low. This greedy decision could potentially be improved, since, e.g., using a combination of *short* runnables may result in filling the conflict interval more appropriately, i.e. without exceeding the interval significantly, than the single *longer* runnable. However, since the greedy approach has shown valuable improvements already, this optimization is omitted here.

Algorithm 5.3: Calculation of $\mathcal{RO}(\delta_{\tau_i \to \tau_j})$ for a Conflict Interval Set

```
Data: $\tau_i, \tau_j, \delta_{\tau_i \to \tau_j}$, TRES_BWA
Result: $\mathcal{RO}(\delta_{\tau_i \to \tau_j}) \forall x$ with $\sigma_{i,j,x}$
1  forall $ci^{\delta}_{i,j,x}$ with $\sigma^{\delta}_{i,j,x} \in \delta_{\tau_i \to \tau_j}$ do
2      persistBW $\leftarrow 0$;                    /*Optimistically, all busy waiting can be removed*/
3      forall $r_a \in \mathcal{R}^{\dashv}(\sigma^{\delta}_{i,j,x})$ do
4          $t_s \leftarrow (t^p_s(r_a) - t_e(ci^{\delta}_{i,j,x})) \cap \mathbb{Q}^+_0$
5          $t_e \leftarrow (t^p_e(r_a) - t_s(ci^{\delta}_{i,j,x})) \cap \mathbb{Q}^+_0$
6          $\mathcal{RO}(\sigma_{i,j,x}) \leftarrow \mathcal{RO}^p_{\tau_j}$
7          while $t_s < t_e$ do
8              let $\mathcal{R}^{avail}(t_s)$ denote assignable runnables at $t_s$ without the conflicting runnables
                 $r_b \in \mathcal{R}^{\vdash}(\sigma_{i,j,x}) : \mathcal{L}_{r_a} \cap \mathcal{L}_{r_b} \neq \emptyset$ according to Eq. 5.93
9              if $|\mathcal{R}^{avail}(t_s)| > 1$ then
10                 sort $\mathcal{R}^{avail}(t_s)$ by increasing $c_{r_n}$ with $r_n \in \mathcal{R}^{avail}(t_s)$
11                 let $r_s$ denote the first entry with $c_{r_s} \geq (t_e - t_s)$ or the last entry of $\mathcal{R}^{avail}(t_s)$ if no
                     runnable exceeds $(t_e - t_s)$
12                 move $r_s$ to $t_s$ within $\mathcal{RO}(\sigma^{\delta}_{i,j,x})$;                  /*Changing result*/
13                 $t_s \leftarrow t_s + c_{r_s}$
14             else if $|\mathcal{R}^{avail}(t_s)| == 1$ then
15                 move $r_s = \mathcal{R}^{avail}(t_s)[0]$ to $t_s$ within $\mathcal{RO}(\sigma^{\delta}_{i,j,x})$ ;      /*Changing result*/
16                 $t_s \leftarrow t_s + c_{r_s}$
17             else
18                 persistBW $\leftarrow$ persistBW + 1;          /*Not all busy waiting can be removed*/
19                 $t_s \leftarrow t_s + 1$
20         end
21     end
22     if persistBW>TRES_BWA then split $ci^{\delta}_{i,j,x}$ and repeat from line 4;            /*ci splitting*/
23     add $\mathcal{RO}(\sigma_{i,j,x})$ to $\mathcal{RO}(\delta_{\tau_i \to \tau_j})$
24 end
```

Algorithm 5.3 is applied to all system's task pairs vise versa, i.e. $n^2 - n$ times. Consequently, calculating a runnable order for each $ci^{\delta}_{i,j,x}$ interval ensures that all conflicts, all release δ values, and all tasks are considered. An interval may combine multiple overlapping CSs such that no runnables are available at line 11 in Algorithm 5.3 to avoid all busy waiting periods for a specific situation. Nevertheless, lines 7 and 8 try to reduce this as much as precedence constraints allow it. The first loop at line 1 iterates among all conflict intervals and corresponding runnables for each of these intervals. Line 7 then considers all points in time within these intervals. Line 8 identifies assignable runnables to a specific position and line 12 shifts the most effective runnable (identified by line 10) one after another into the conflicting interval and correspondingly moves the conflicting runnable(s) backwards until the problematic runnable left the conflict interval ($t_s \geq t_e$ cf. line 7). If no runnable is available in line 8, busy waiting periods can not be prevented but are minimized due to checking every time instant via line 19. During the application of Algorithm 5.3 to real-world scenarios (cf. Chapter 6), cases can be observed that show relatively long conflict intervals that reduce the flexibility of reordering runnables because many of a task's runnables are writing to CSs. To overcome this issue, lines 2, 18, and 22 are added. These lines keep track of whether the time frame of unavoidable busy waiting (line 18) goes beyond some threshold (`TRES_BWA`), and if yes, splits the conflict interval into disjoint parts (e.g. $\sigma_{1,2,1} = \{[2,8), \{r_2\}, \{r_7\} \Rightarrow \sigma_{1,2,1} = \{[2,5), \{r_2\}, \{r_7\}; \sigma_{1,2,1'} = \{[5,8), \{r_2\}, \{r_7\}\}$. As later on shown along with Example 5.11, this conflict interval splitting increases the effectiveness of TDRR for inherently long CSs.

Finally, runnable orders are calculated for all conflicts $\delta_{\tau_i \to \tau_j} \in \Delta^{\phi}$ and saved as $\mathcal{RO}_{i,j,x}$ in $\mathcal{RO}^{\phi}$, whereas the number of conflicting delta situations x is not constant across task pairs. $\mathcal{RO}^{\phi}$ is intended to be saved into a read-only memory for the PU that schedules the task set. Thus, TDRR applies to every PU in a partitioned scheduling based system for a task set scheduled at the corresponding PU, i.e., $\Delta^{\phi}_{P_x}(n \times n) : M^{P}_{\tau_i} = x; n = |\mathcal{T}_{P_x}|; \tau_i \in \mathcal{T}_{P_x}$.

5.8.4 Applying new Runnable Orders at Runtime

This section includes the last process (VI) outlined in the TDRR overview of Section 5.8.1 and describes the approach of applying TDRR at runtime.

At each release time of a task τ_j, the system is observed whether there are tasks executing that fulfill all of the following three conditions:

(1) They share a conflicting label with τ_j,

(2) τ_i was released earlier than τ_j and did not yet finish its execution (reach the response point), and

(3) the release time difference between τ_i and τ_j is within at least one $ci_{i,j,x} \in \delta_{\tau_i \to \tau_j}$ conflict interval.

Each of these conditions is formally outlined in Table 5.5, respectively.

(1)	$((\mathcal{L}_{\tau_i} \cap \mathcal{L}_{\tau_j}) \neq \emptyset) \subseteq CS^{\phi};\ i \neq j; i, j \in \mathbb{N}$ (cf. Eq. 5.86)
(2)	$(\blacktriangle_{\tau_i} \leq \blacktriangle_{\tau_j}) \wedge (T_i + \blacktriangle_{\tau_i} > \blacktriangle_{\tau_j})$
(3)	$(t_s(\sigma_{i,j,x}) \leq (\blacktriangle_{\tau_j} - \blacktriangle_{\tau_i}) \leq t_e(\sigma_{i,j,x}))$ for at least one Interval $\sigma_{i,j,x} \in \delta_{\tau_i \to \tau_j}$

Table 5.5: Conditions to identify conflicting tasks at runtime

If all three conditions are given, the latter task τ_j is released with a runnable order that minimizes busy waiting, calculated via Algorithm 5.3. An empty set element in Δ^{ϕ} denotes that the corresponding tasks do not conflict in any way, e.g. $\delta_{\tau_0 \to \tau_1} = \emptyset$ means that if τ_0 is released earlier than τ_1, no conflict occurs between them. The following Example 5.11 shows all six processes (I)–(VI) of TDRR.

Example 5.11: TDRR with three Tasks

Figure 5.16 illustrates three tasks $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$ calling the following runnables $\mathcal{R}_1 = \{r_1, r_2, r_3, r_4, r_5\}, \mathcal{R}_2 = \{r_6, r_7, r_8\}, \mathcal{R}_3 = \{r_9, r_{10}, r_{11}, r_{12}\}$, each of which executes a single time unit. Runnables accessing the same CS are r_2, r_5, r_6, r_9, and r_{11} indicated with 'L' in Figure 5.16. As soon as two runnables access the same label at the same time, shown as vertical gray filled rectangles with white dots in Figure 5.16, a conflict occurs such that one task has to busy-wait for the resource due to the spinlock protecting the CS, i.e., 'L'. Task release delta values[a], to which such conflicts occur, are indicated below the horizontal time axis of the first upper Gantt chart in Figure 5.16. The same chart shows the initial $\mathcal{RO}$ execution resulting in busy waiting, and the lower Gantt chart presents task executions for the same release delta values without any busy waiting due to reordered runnables. In this example, it is assumed that there are no precedence constraints between the runnables and that task release times are recorded during runtime upon a common time basis.

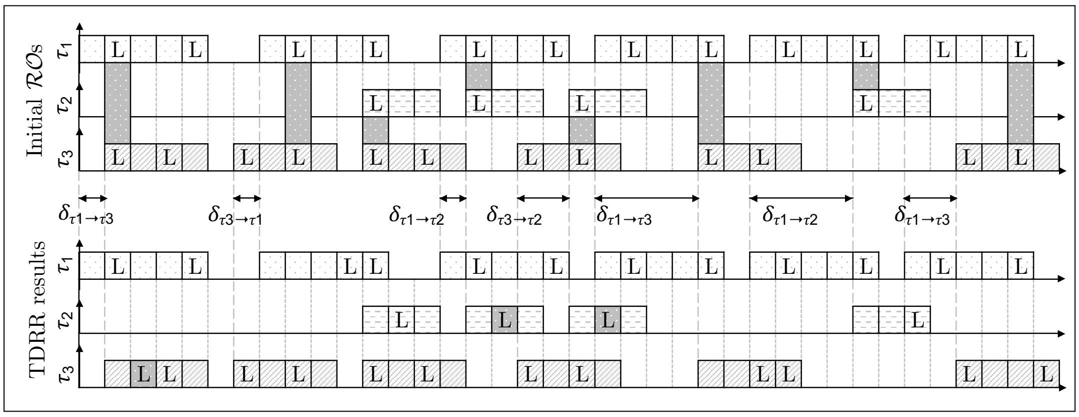

Initial $\mathcal{RO}$s
TDRR results
τ_1
τ_2
τ_3
L
$\delta_{\tau1\rightarrow\tau3}$
$\delta_{\tau3\rightarrow\tau1}$
$\delta_{\tau1\rightarrow\tau2}$
$\delta_{\tau3\rightarrow\tau2}$
$\delta_{\tau1\rightarrow\tau3}$
$\delta_{\tau1\rightarrow\tau2}$
$\delta_{\tau1\rightarrow\tau3}$

Figure 5.16: Example on three conflicting tasks, five label accesses, and eight task release delta values resulting in busy waiting

The corresponding delta matrix is shown i Eq. 5.94.

$$\Delta^{\phi} = \begin{pmatrix} 0 & \{\sigma^{\delta}_{1,2,1}, \sigma^{\delta}_{1,2,2}\} & \{\sigma^{\delta}_{1,3,1}, ..., \sigma^{\delta}_{1,3,4}\} \\ \{\sigma^{\delta}_{2,1,1} = \emptyset\} & 0 & \{\sigma^{\delta}_{2,3,1}\} \\ \{\sigma^{\delta}_{3,1,1}\} & \{\sigma^{\delta}_{3,2,1}\} & 0 \end{pmatrix} \tag{5.94}$$

The contents of Eq. 5.94's delta entries are given in Eq. 5.95.

$$\begin{aligned} \sigma^{\delta}_{1,2,1} &= \{[0,2), \{r_2, r_5\}, & \{r_6\}\} \\ \sigma^{\delta}_{1,2,2} &= \{[3,5), \{r_2, r_5\}, & \{r_6\}\} \\ \sigma^{\delta}_{1,3,1} &= \{[0,1), \{r_2\}, & \{r_9\}\} \\ \sigma^{\delta}_{1,3,2} &= \{[1,2), \{r_2, r_5\}, & \{r_9, r_{11}\}\} \\ \sigma^{\delta}_{1,3,3} &= \{[2,3), \{r_5\}, & \{r_{11}\}\} \\ \sigma^{\delta}_{1,3,4} &= \{[3,5), \{r_5\}, & \{r_9\}\} \\ \sigma^{\delta}_{2,3,1} &= \{[0,2), \{r_6\}, & \{r_9\}\} \\ \sigma^{\delta}_{3,1,1} &= \{[0,2), \{r_{11}\}, & \{r_2\}\} \\ \sigma^{\delta}_{3,2,1} &= \{[0,2), \{r_{11}\}, & \{r_6\}\} \end{aligned} \tag{5.95}$$

The next consecutive step is to calculate runnable orders, which is exemplary shown for $\sigma^{\delta}_{1,3,3}$. Initially, the $ci^{r_{11}}_{1,3,3}$ interval, i.e. the time frame r_{11} must not be executed due to r_5 is derived via $r^p_5 = [4,5)$ and is $ci^{\delta}_{1,3,3} = \{[2,3), \{r_5\}, \{r_{11}\}\} = [4-3, 5-2) = [1,3)$. Then, at $t_s = 1$ runnables r_{10} and r_{12} are available and r_{10} is chosen since it corresponds to the initial runnable order. Here, the sorting and runnable section across lines 10 and 11 in Algorithm 5.3 do not take effect due to equal execution times. At $t_s = 2$, runnable r_{12} is available and assigned correspondingly such that the result is $\mathcal{RO}ci^{\delta}_{1,3,3} = \{r_9, r_{10}, r_{12}, r_{11}\}$, which is the last result shown at τ_3 at

Figure 5.16's bottom right corner.

It must be noted here that three solutions within the lower Gantt chart of Figure 5.16 represent exceptional cases indicated by dark gray filled and white dotted boxes with "L" inside the box. These runnable orders are valid for the specific release delta value rather than a release delta interval. For the latter, which is used by TDRR, the provided solutions of Figure 5.16 do not directly represent results based on conflict intervals of Eq. 5.95, since they do not avoid busy waiting for the values provided in Eq. 5.95. The solutions are instead created by reducing the conflict interval to shorter intervals, which increases the amount of total runnable orders for a task set on the one hand, but also increases the chances to reduce further busy waiting via a more fine-grained analysis of task release delta values on the other hand. The latter is achieved via conflict interval splitting such that shorter periods potentially allow more flexibility in reordering runnables. For instance, the situation $\sigma^{\delta}_{1,2,1}$ including $ci_{1,2,1} = \{[0,2), \{r_2, r_5\}, \{r_6\}\}$ should be split into $[0,1)$ and $[1,2)$ since $[0,2)$ results in $ci^{r_2}_{1,2,1} = [1-2, 2-0) = [0,2)$ and $ci^{r_5}_{1,2,1} = [4-2, 5-0) = [2,5)$ such that r_2 and r_5 occupy the shared resource for the entire execution time of τ_2 i.e. $[0,5)$ and no runnable order can be found to avoid busy waiting. However, splitting the conflict interval into $[0,1)$ and $[1,2)$ results in $ci^{r_2}_{1,2,1} = [1-1, 2-0) = [0,2)$ and $ci^{r_5}_{1,2,2} = [4-1, 5-0) = [3,5)$ such that the time slot $[2,3)$ gets available and $\mathcal{RO}_{1,2,1} = \{r_7, r_8, r_6\}$ can be used to avoid busy waiting for $\blacktriangle_{\tau_1 \to \tau_2} \in [0,1)$, respectively $\mathcal{RO}_{1,2,2} = \{r_7, r_6, r_8\}$ for $\blacktriangle_{\tau_1 \to \tau_2} \in [1,2)$.

[a]Specific values are shown, but the delta release conflict interval is always a time range.

Having the above metrics, $\mathcal{RO}$s can be calculated that define for each interval $ci^{\delta}_{i,j,x} \in \delta_{\tau_i \to \tau_j} \in \Delta^{\phi}$ and corresponding conflicting tasks a runnable order for a latter released task. The following last TDRR example gives a short outline of how to derive the available set of runnables at a certain point in time based on a runnable DAG.

Example 5.12: Retrieving $\mathcal{R}^{avail}(t)$ from a Runnable DAG

Figure 5.17 illustrates (a) an example runnable DAG for a task whereas two runnables r_1 and r_4 are assumed to access a global CS, (b) a corresponding initial runnable order, and (c) a runnable order representing the result of Algorithm 5.3. Dependencies are shown as arrows and runnables as rectangles with a specific width that represents their corresponding execution time.

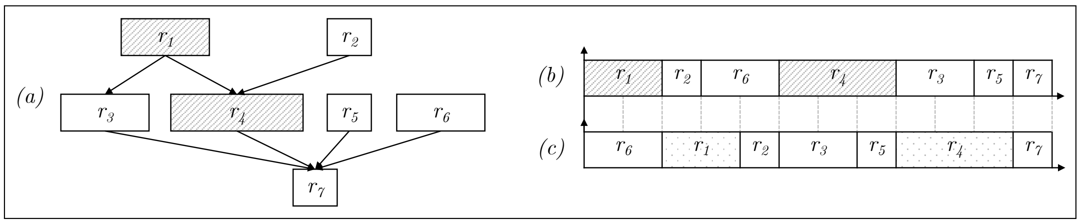


Figure 5.17: (a) Example DAG, (b) initial $\mathcal{RO}$, (c) adapted $\mathcal{RO}$ no conflicts

The conflict produced by a prior released task is assumed to be given as $\sigma_{i,j,1} = \{[3,5), \{r_a, r_b\}, \{r_1, r_4\}\}$ and hence affects the two runnables highlighted with diagonal gray lines in Figure 5.17. From reverse engineering, the following access intervals of τ_i can be derived: $r_a^p = [3,5)$ and $r_b^p = [10,11)$. Algorithm 5.3 initially starts with $t_s = 3 - 5 \cap \mathbb{N}_0 = 0$ and ends with $t_s = 5 - 3 = 2$ (cf. Eq. 5.91, $ci_{i,j,1}^{r_1} = [0,2)$). For the start point, available runnables are $\mathcal{R}^{avail}(0) = \{r_2, r_5, r_6\}$, since these runnables neither have predecessors nor access a global CS. From this set, line 11 of Algorithm 5.3 identifies r_6 as the most suitable runnable being inserted at $t = 0$ since it fills the gap entirely and does not further exceed the interval. Afterwards, the interval $ci_{i,j,1}^{r_4} = [5,8)$ is investigated since $t_s = t_s^p(r_b) - t_e(ci_{i,j,1}^{\delta}) = 10 - 5 = 5; t_e = t_e^p(r_b) - t_s(ci_{i,j,1}^{\delta}) = 11 - 3 = 8$. Thus, at $t_s = 8$, available runnables are $\mathcal{R}^{avail}(8) = \{r_3, r_5\}$, since r_4 accesses a global CS and $\{r_5, r_4\} \in \mathcal{R}_{r_7}^{\prec}$ are not assigned yet, i.e., r_7 is not assignable. Then, r_3 is preferred over r_5 due to filling the conflict interval more than r_5. Afterwards, r_5 is assigned subsequently for $t_s = 7$ so that the final result is defined by $\mathcal{RO}(\sigma_{i,j,1}^{\delta}) = \{r_6, r_1, r_2, r_3, r_5, r_4, r_7\}$.

The quality of TDRR results can be assessed by the longest busy waiting period avoided with the help of new runnable orders calculated by TDRR for all tasks, as shown in Eq. 5.96. This time frame depends on a task pair τ_i and τ_j as well as delta conflicts $\sigma_{i,j,x} \in \delta_{\tau_i \to \tau_j}$ and is normalized with regard to a task's execution time C_j.

$$C^{-,improved} = \max_j C_j^{-,improved}$$

$$C_j^{-,improved} = \frac{\max\limits_{i \in [1,n] \setminus j, x \in [1, |\delta_{\tau_i \to \tau_j}|]} \left(\sum\limits_{r_b \in (\mathcal{R}_j \cap \mathcal{R}_{i,j,x}^{\vdash}) \wedge (\downarrow_{r_b} \cap \downarrow_{\tau_i}) \neq \emptyset} (c_{r_b}) - \text{persistBW}_{i,j,x} \right)}{C_j} \tag{5.96}$$

The metric of Eq. 5.96 is used in Section 7.4 to derive the maximal task execution time reduction in % of the case study model outlined in Chapter 6.

5.9 Summary of Constrained Software Distribution & Timing Verification

This chapter includes various solutions for distributing software across a heterogeneous network of PUs under strict timing verification and the consideration of many-fold constraints. Therefore, various challenges are solved via different (meta-)heuristics roughly summarized in Table 5.6.

Challenge	**(Meta-) Heuristics**	**Reference and Notes**
Partitioning	CP, CPP, ESSP, CP-PC	Chapter 4
Task to PU Mapping	CP, ILP, GA, DFG	Section 5.2; RTA for CPUs, GPUs, RMS, FPPS, FPMPS, WRR, and Offsets considering blocking and contention as well as TCLA for implicit and LET communication.

Challenge	**(Meta-) Heuristics**	**Reference and Notes**
Data (Label) to Memory Mapping	CP, GA	Section 5.7
WCET minimization	TDRR	Section 5.8.1

Table 5.6: Tackled challenges and used (meta-) heuristics

Formally, a mapping solution to these challenges is defined by the following Definition 5.11.

Definition 5.11: Mapping Solution

A mapping solution $\mathcal{S}$ consists of the runnable partitioning matrix $\boldsymbol{M}_{r_a}^{\tau}$, the task to PU mapping matrix $\boldsymbol{M}_{\tau}^{P}$, and the label to memory mapping $\boldsymbol{M}_{l}^{m}$. A solution is valid iff all constraints[a] are satisfied.

[a]The summary of constraint types and their implementation is given in Tables 5.8 and 5.9.

A mapping solution can be assessed by various metrics, which are configured along with the DSE process' optimization criteria. Typical optimization goals are shown in Table 5.7. Many metrics are assessed as mean values via over-line notation over a set of metrics, whereas bounds, i.e., maximal or minimal values of the corresponding set can be of interest, too. Although optimizing mean values increases resolution time compared to bounded optimization, the mean values represent an entire metric set instead of min/max bounds only and hence potentially yield better overall representation of the optimization goal.

Criteria	**Description**	**Notation**	**Eq.**
Responsiveness	The earlier a task responds, the better is its responsiveness. To assess relative and not absolute values, a task's response time can be put into relation with its deadline. The lower the corresponding value is, the better is the responsiveness. The metric is known as normalized response time and can be minimized towards a set's maximal value or its average. Additionally, the total response time sum RTS can be minimized.	$\mathrm{mNRT} = \widehat{\mathrm{NRT}}$ $\overline{\mathrm{NRT}}$ RTS	5.97 (via 5.37, 5.58, and H.19)
Network Load	A lower average normalized CAN message response time results in a lower network load. The partitioning approach of Chapter 4 further constitutes an essential role for the number of CAN messages.	$\overline{R}^{N,+}$	5.69, 5.98

Criteria	**Description**	**Notation**	**Eq.**
Load Balancing (PU Utilization)	By minimizing the maximal PU load, average load balancing is achieved, which potentially yields in lowering the voltage and energy consumption[54]. A PU's utilization is derived from the division of its load by its capacity and provides the proportion of used computing resources.	$\hat{U}^P = \max_x(U_x^P)$	5.3, 5.74; 4.23, 4.24; 5.9
Task Chain Reaction	When minimizing average or maximal task chain reaction delay, the system responsiveness is improved regarding specific task chains instead of all tasks (cf. responsiveness).	$\overline{\rho_\gamma}; \hat{\rho_\gamma}$	5.46, 5.47
Task Chain Aging	Task chain aging minimization ensures that task chain data is updated as frequent as possible.	$\overline{\alpha_\gamma}; \hat{\alpha_\gamma}$	5.48, 5.49
Memory Access Costs	Minimizing label mapping costs requires a task to PU mapping and can be either calculated in isolation or together with the task to PU mapping. Results serve a similar goal such as responsiveness.	MC_l	5.73, 5.72, 5.30
Multi-Criteria	The criteria above can be combined such that Pareto fronts serve choosing solutions from a solution set serving multiple goals (exemplary shown in Example 5.5).	-	E.g. Ex. 5.5

Table 5.7: Optimization criteria

Table 5.7 provides four average notations that sum up entity related properties and divide the result by the number of entities, which is shown exemplary for the average normalized response time $\overline{\text{NRT}}$ in Eq. 5.97 ($n = |\mathcal{T}|$ and $i \in ([1, n] \cap \mathbb{N})\ \forall\ \tau_i \in \mathcal{T}$). The maximal Normalized Response Time (mNRT) metric is used in, e.g., [90] or [263] and also shown in Eq. 5.97, along with the response time sum RTS, which is used for this thesis.

$$\text{minimize mNRT } = \max_i \left(\frac{R_i^+}{D_i}\right); \text{minimize } \overline{\text{NRT}} = \frac{\sum_i \left(\frac{R_i^+}{D_i}\right)}{n}; \text{minimize RTS} = \sum_i R_i^+ \tag{5.97}$$

The difference between mNRT and $\overline{\text{NRT}}$ metrics is the optimization focus, which lies on all tasks for the latter case and on a single task for mNRT. Assuming that the system's highest priority task has a relatively high utilization, the corresponding task defines the lower bound on mNRT. All other tasks are then not optimized due to this upper bound, which is not the case for minimizing the average $\overline{\text{NRT}}$. In contrast, the latter approach may result in some task's response time being very close to its deadline, because the average slack influenced by other tasks may be lower due to other tasks achieving high slack values. Chapter 7 uses the RTS metric (denoted RTSO) to focus solely on response times rather than responsiveness. The latter can be configured easily but is omitted for measurements

[54]Assuming the simplistic energy voltage approximation of appendix H.1.

in Chapter 7, due to already having a generic PU utilization metric (denoted LBO).

The load balancing optimization is usually implemented via minimizing the maximal PU utilization in percent as shown in Eq. 5.74. Instead of minimizing the maximal PU utilization, maximizing the minimal PU utilization is also possible. However, the latter approach has shown a longer resolution time, which is disadvantageous for large industrial models. This is due to the fact that if the upper bound on PU utilization is found, the minimize($\max_x U_x^P$) approach would stop the resolution process whereas the maximize($\min_x U_x^P$) process still continues aligning (maximizing) the remaining utilization values. Since minimizing the maximal PU utilization provides sufficient results across cases studied in Section 6, the approach has been chosen over maximize($\min_x U_x^P$).

Since no explicit deadlines are given for CAN messages, average CAN message response times are used as a metric, which is shown in Eq. 5.98. Therefore, N is the set of CAN messages and hence $|N|$ is its number.

$$\text{minimize } \overline{R}^{N,+} = \frac{\sum\limits_{\nu \in [1,|N|] \cap \mathbb{N}} R_\nu^+}{|N|} \tag{5.98}$$

Due to the many-fold requirements crucial to the automotive industry, single-objective optimization is often not sufficient. Various multiple criteria such as FFI, average normalized response time, load balancing, inter-PU communication or similar can be in focus of optimization at the same time such that multi-objective approaches are reasonable. As a result of a multi-objective optimization, several Pareto-optimal solutions may exist that feature different optimized criteria values located at the Pareto-front, i.e., all those optimization value pairs have at least one better criteria value than other value pairs of the Pareto-front. Formally, given a Pareto-Front consisting of a set of value pairs x_i, y_i, then $\forall i, j \in [1, i] \cap \mathbb{N}; i \neq j : (x_i > x_j \Rightarrow y_i < y_j) \vee (x_i < x_j \Rightarrow y_i > y_j)$. Optimization parameters can also be weighted and combined into a single fitness value, which requires aligning the criteria values. For example, having three solutions and two optimization criteria (oc_1, oc_2) to be minimized such as $oc_1 = [2, 3, 8]; oc_2 = [4, 3, 1]$, the third solution would be omitted when equally weighting the optimization criteria $(\cdot 1)$, since the third solution's accumulated value is higher than the results from solutions 1 and 2 $(2 + 4 = 6; 3 + 3 = 6; 8 + 1 = 9)$. However, when weighting oc_1 with 3, the results are $2+3\cdot4 = 14, 3+3\cdot3 = 12, 8+3\cdot1 = 11$ and consequently solution three would be the *best* one. For the measurements presented along with the software distribution approaches in Section 7.2, namely CPMO, a regular Pareto-front is calculated, which is also assessed in Figures 7.11 and 7.12. Therefore, inter-PU communication costs, cf. Eq. 5.10, are combined with load balancing for the task to PU mapping, so that both criteria are optimized towards their minimum value for the CPMO approach of Section 7.

Table 5.8 briefly summarizes the AMALTHEA constraints and their CP constraint pendant based on [44], respectively their implementation context.

AMALTHEA constraint	**Choco [44] constraint**
Runnable-, Task-, ASIL-, or Tag- Pairing	`.allEqual`; `.arithm(=)`; `.element`
Runnable-, Task-, ASIL-, or Tag- Separation	`.allDifferent`; `.arithm(≠)`; `.notMember`
Runnable Sequencing	`.arithm(<,=)`;`.allDifferentEx0`; `.max`

Amalthea constraint	**Choco [44] constraint**
PU Utilization	`.scalar; .count; .min / .max`
Partitioning	`.binpacking; .min; .sum; .scalar; .cumulative`
Activations	`.allEqual; .arithm(≤); .and; .or`
Inter-task Communication	`.addClausesXorEqVar; .ifThen;.arithm(=); .count; .and`

Table 5.8: Used constraint types for partitioning and task to PU mapping

Table 5.8 implicitly shows the CP benefits in contrast to, e.g. MILP, since the number of constraints remains relatively small and their usage is more natural compared with a combination of multiple inequality definitions.

Finally, Table 5.9 gives an overview of the various constraints, corresponding notations, and applications in the form of equation references outlined in this chapter.

Name	**Equation(s) / Listing**
Runnable Activation	4.1, 4.15
Runnable/ Task/ SWC/ Tag pairing with PU/ micro controller / ECU	4.2, 4.3 / 5.14, 5.16
Runnable/ Task/ SWC/ Tag separation from PU/ micro controller / ECU	4.4, 4.5/ 5.15, 5.17
Runnable sequencing	4.8, 4.9, Lst. 4.3
Cyclic Dependencies	4.11
Runnable Partitioning	4.16, 4.17, Lst. 4.2
Runnable Balancing	4.23, 4.24, Lst. 4.1
Task Load Balancing	5.1
Task PU Mapping	5.6, 5.7
PU Capacity	5.8 using 3.4
PU Utilization	5.9
Deadline	5.29
Data Age	5.42
Label Mapping	5.70
Memory Size Limit	5.71
Label Pairing with Memory	5.77, 5.80
Label Separation from Memory	5.78, 5.79

Table 5.9: Constraints summary and equation references

In general, CP-based approaches benefit from a vast amount of constraints available to be applied to different variables and various intertwined relationships. A wide flexibility in combining optimization goals across minimizing average or maximal normalized response times, average slackness, end-to-end task chain latency values, memory access times, CAN message response times and more can be achieved. Both bad load balancing and high communication costs increase response times caused by additional interference of CAN messages or increased blocking or contention such that timing verification and schedulability tests may reach their limits. Especially highly utilized models cause constraint validations, which accompany the DSE approaches (cf. Table 5.9), to fail for a significant part of the problem space, such that the solution space is often quite narrow. For any meta heuristic approaching the DSEs for large models and highly utilized PUs, its configuration must be carefully set, which is crucial for solving the challenges of Table 5.6 along with case study models presented in the next Chapter 6.

In terms of novelty, this chapter presents

I. precise outlines, definitions, and applications of various automotive specific constraints for the DSE of software distribution across PUs and ECUs,

II. advanced RTA considering

   (a) data access delays between PUs and memories,

   (b) CE operations,

   (c) synchronous and asynchronous task offloading,

   (d) contention and blocking interference,

   (e) WRR scheduling and TX2RS for GPUs, and FPPS or FPMPS with considering offsets for CPUs,

   (f) inter-PU and inter-ECU communication costs,

III. task chain reaction and age delay calculations across both implicit and LET communication paradigms,

IV. a multi-objective data to memory mapping GA , and

V. TDRR as a method to reduce busy waiting for spinlocks.

No related work is available that either covers such holistic analyses or applies to an AUTOSAR compliant modeling environment. When including the various approaches I–V to the DSE of task mapping for AMALTHEA models, it is expected that

(i) performance bottlenecks, error-prone manual task distribution processes, and constraint violations can be avoided and detected in early system design phases via I,

(ii) realistic WCRTs can be calculated in a mixed CPU-GPU environment such as the Nvidia TX2 hardware via II,

(iii) system design phases are improved due to coherent and extensive timing analyses via II/III,

(iv) task execution efficiency, respectively timeliness, is improved via incorporating label mapping IV to RTA and using V,

(v) data access and blocking delays can be reduced via IV, and

(vi) automotive software execution efficiency increases as a consequence of realistic WCRT bounds and a holistic task distribution DSE.

6 Case Study Models

This chapter briefly introduces the origin, scope, intention, and contents of various Amalthea models, which are used as inputs for the previous chapters' contributions to obtain and evaluate results. The focus is on seven models presented in the following sections 6.1–6.5. Beyond these seven models, several hypothetical examples were created in correspondence with both examples used in this thesis (an overview is available in Appendix D) and examples from literature, e.g., [36, 38, 62]. Though, these example models are used primarily for approach validation and consequently omitted for evaluation here. A comparison of model entities and properties such as the number of runnables, tasks, activations, labels, dependencies, the sum of all instructions, average label accesses per task and runnable is part of Section 6.6 and shown in Table 6.2 and Figure 6.3.

6.1 Fmtv

The initial Fmtv-challenges were published in 2015 by researchers of the Robert Bosch GmbH company in [187]. It was the initial step to provide real-world automotive benchmarks without violating IP restrictions by providing abstract information. Many model details are part of the original publication and challenge outlines, which have become a fundamental part of the Waters workshop since then, because such challenges attracted many researchers over the years. The main topic is name implied, Fmtv, which form crucial requirements to cope with, e.g., safety, reliability, real-time, or fault-tolerance demands. The Waters19 model (cf. Section 6.2), for instance, is the latest Amalthea model published along with the Waters workshop by the time this thesis is written.

The initial challenges address verifying timing properties and coherences in the automotive domain alongside the following four topics.

1. TCLA
2. RTA and WCET analysis taking into account shared memory blocking and contention for multi-PU platforms
3. Runnable to task partitioning (cf. Chapter 4), task to PU mapping (cf. Chapter 5) and label to memory allocation (cf. Section 5.7) optimization problems demanding for DSE

4. Application of the FMTV AMALTHEA model to the above outlined challenges

Major foundations for addressing these topics are inherent parts of the background Chapter 2 such as the system's heterogeneities and domains along with many physical processes, varying dynamics, the co-existence of sampled and reactive data, different timing domains, and sophisticated communication channels and dependencies.

In 2016, abstractions published in [187] turned into a publicly available AMALTHEA model along with [128], which did not violate IP policies since AMALTHEA does not contain executable code or concrete software behavior implementation. The modeled system contains abstract entities of a modern EMS with hard real-time demands, criticality levels, task chains, and more. Furthermore, new challenges were presented, i.e., investigating end-to-end latency bounds across different scheduling approaches, varying activation types, cause-effect chains, and finding optimized label mapping methods.

In 2017, the challenges were further extended by adding the consideration of the three different communication paradigms (a) Explicit Communication (EC), (b) Implicit Communication (IC), and (c) LET along with [31]. The WATERS19 model, outlined in the next Section 6.2 [30], is the latest addition to the set of published FMTV (WATERS) models.

The real-time community took great acceptance of the WATERS challenges and tackled these in several publications, which are predominantly available at the WATERS community forum[55] but also used in further research context such as [264]. A throughout analysis of research addressing FMTV challenges along with the WATERS community is analyzed in Section 5.1.

6.2 WATERS19

For the WATERS workshop in 2019, an entirely new model was published in line with new challenges that arise with the interest in autonomous driving and increasing computing power. Image processing is a mandatory part of the former evolution, which makes the use of GPUs inevitable. The WATERS19 AMALTHEA model[56] contains much fewer entities compared to the previously described FMTV model (cf. Figure 6.3) and features abstractions of an end-to-end autonomous driving application prototype for the Nvidia TX2 platform, which is used for GPU RTA in Section 5.6 of this thesis. The application contains throttle, steering, and brake signals required to navigate a vehicle through a map created by image processing tasks. The challenge also states that the application also addresses emergency maneuvers by high priority tasks that detect potential risks such as obstacles amid the driving path. Out of the nine major tasks, five tasks can only be executed on the CPU, three are can potentially be executed on any PU (namely `SFM`, `Localization` and `Lane_Detection`), and one task must run on Nvidia's Pascal^{TM} GPU architecture (namely `Detection`). The basic application structure in form of a task DAG is shown in Figure 6.1 and based on the original publication [30].

[55] WATERS community forum online at: `https://bit.ly/2NEkwFI`, visited 11.2020
[56] WATERS19 model available online at: `https://bit.ly/37IarPZ`, visited 11.2019

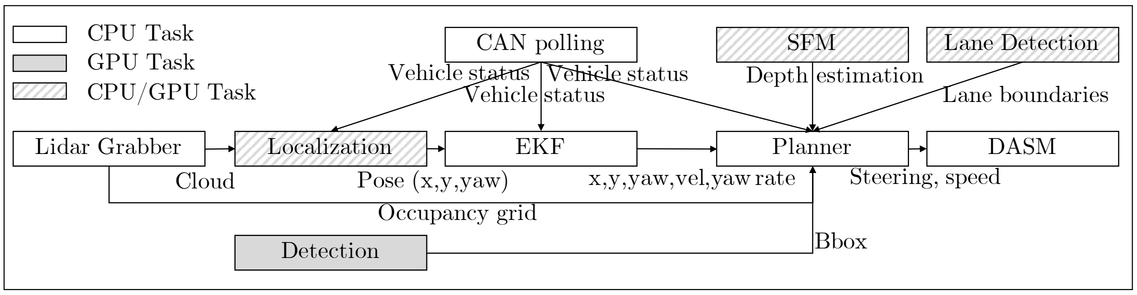


Figure 6.1: WATERS19 application structure based on [30]

Rectangles represent tasks that communicate with others via data being exchanged and represented by directed labeled arrows, of which the label indicates the label name written by the source task, where the arrow begins, and read by the target task. In addition to the above mentioned nine major tasks, which form the application itself, five other tasks are contained in the model that address generic OS overheads and *PRE/POST* tasks, required for all tasks that may execute on the GPU. The latter tasks explicitly perform the offloading operations, i.e., calling a PRE-processing runnable, an inter-process event trigger for the GPU task, optionally the wait event (cf. synchronous offloading), and a POST-processing runnable according to [30]. Not only *PRE/POST* tasks have to be executed for starting a GPU task, but also dedicated copy in/out runnables within the call sequence of the offloaded task, which performs the data copy operations. In turn, if a task, which could potentially run on a GPU, is scheduled on a CPU, neither *PRE/POST* tasks nor copy in/out runnables have to be executed, and hence, these overheads can correspondingly be ignored during the RTA process.

The missing content of the WATERS19 AMALTHEA model for TCLA are task chains. Hence, for analyses in this thesis, task chains are derived based on the task graph of Figure 6.1 to tackle TCLA. Therefore, all paths from entry to exit tasks are collected and shown in Table 6.1.

Task chain	**Tasks**
γ_1	Lane_detection, Planner, DASM
γ_2	SFM, Planner, DASM
γ_3	CANbus_polling, Planner DASM
γ_4	CANbus_polling, EKF, Planner, DASM
γ_5	CANbus_polling, Localization, EKF, Planner, DASM
γ_6	Lidar_Grabber, Localization, EKF, Planner, DASM
γ_7	Lidar_Grabber, Planner, DASM
γ_8	Detection, Planner, DASM

Table 6.1: Task chains of the WATERS model

Task chains of Table 6.1 are manually created, which requires (I) an additional event model that contains events for every task, (II) event chains γ_g within the constraints model, and (III) sub event chains within event chains (II) for every source target task pair within the task chain. For instance, γ_8 requires two sub event chain entities that contain `Detection` and `Planner` as *stimulus* entries and `Planner` and `DASM` as *response* entries, respectively.

The proposed challenges are similar to the previous FMTV ones with the major difference of considering the GPU architecture. More precisely, the focus is on RTA that includes CE operations, memory contention between different CPUs and the GPU, as well as synchronous or asynchronous offloading types, which are all covered in Section 5.6. The second challenge is to minimize task chain latency delays based on the limited mapping of the nine tasks.

6.3 AIM

The anonymized industrial model (AIM) has been granted access to by a project partner during the AMALTHEA4PUBLIC project, and it represents a real automotive ECU similar to the FMTV model. It is anonymized such that mock-up descriptions replace original names. The amount of labels is ~ 4.7 times higher than for the FMTV model, but other model entities are relatively close to the FMTV model contents.

6.4 Democar

The Democar model has been the initial example in APP4MC. It was published by P. Frey along with his dissertation [265] in 2011 and also used in, e.g., [67, 266]. The model entity names are derived from an EMS, but in terms of instructions, most values are evenly modeled as value 80000, except for two runnables, which require 160000 instructions each, such that its hypothetical basis is apparent. However, label accesses and the task periods are close to real-world combustion EMS scenarios by using, e.g., accelerator pedal sensor values combined with properties such as mass airflow, throttle angle, and engine speed to derive the desired throttle position for a target engine speed via a control loop implementation. The runnable dependency graph is included in the appendix as Figure H.1. Initial tasks contain runnables ordered by periodic activations, namely 5ms, 10ms, and 20ms. As soon as partitioning is used to subdivide the original three tasks, inter-task communication and the number of conflicts regarding TDRR increases, since DAGs are cut, and more labels are shared between tasks.

6.5 Generated Models

Three additional generated models are included in measurements of the next Chapter 7 in order to investigate a broader model diversity. The generation process applies to software and hardware models. The software model generation defines min and max values for labels, runnables, ticks, and stimuli, used with random and probabilistic functions to generate AMALTHEA model entities. Generated values are uniformly distributed between min and max, except for ticks, which are probabilistically derived from an $\alpha = 2, \beta = 2$ BetaDistribution density function shown in Figure 6.2 with the dotted line, and read and write accesses from runnables to labels, which are based on the $\alpha = 1, \beta = 3$ BetaDistribution density function shown in Figure 6.2 with the dashed line. Values are calculated by the use of the `apache.commons.math3`[57] library. The BetaDistribution has been chosen due to the fact that it is capable of generating appropriate density functions by using only the two α, β values, which eases the probabilistic generation process. In general,

[57] https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html, visited 11.2020

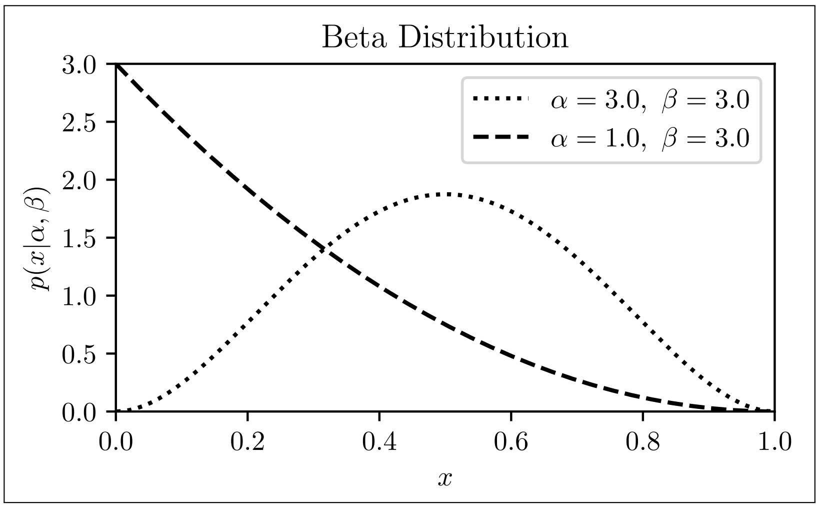


Figure 6.2: Beta distributions used for probabilistic software entity generation

other similar functions can be used, too. Tasks are not generated due to the partitioning process of Chapter 4 already providing this functionality. Therefore, the partitioning can be configured to consider activation values, precedence constraints, cycles, and a specific number of tasks. The hardware generation uses uniform distribution and generates random numbers in between given lower and upper bounds for ECUs, CPUs, memory modules, and frequencies, which are then randomly referenced by the generated PUs.

6.6 Comparison of the Case Study Models' Properties

This section briefly compares the models introduced in Sections 6.4–6.5 based on fundamental properties influencing efficiency and effectiveness of partitioning, task to PU mapping, data allocation, and timing verification processes.

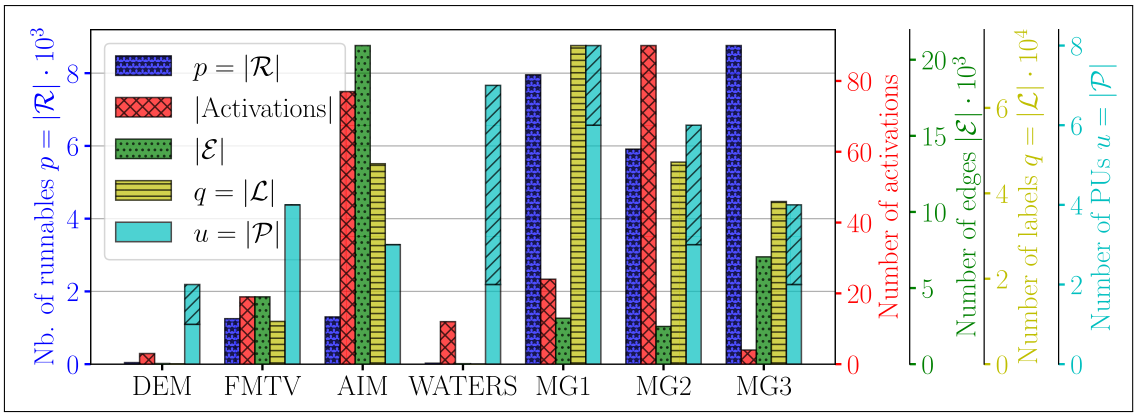


Figure 6.3: Bar chart on various properties of case study models

The precise data of Figure 6.3 is the available along with Table 6.2, whereas the former visually presents the model properties of the seven models using different y-axis.

	DEM.	AIM	FMTV	WAT.	MG1	MG2	MG3
Nb. Tasks $n = \|\mathcal{T}\|$	3	77	21	14	30	120	100
Nb. TaskDeps $\|\mathcal{E}\|$	2	525	51	23	435	3545	4633
Nb. Run. $p = \|\mathcal{R}\|$	43	1297	1250	27	7954	5913	8759
Nb. Run. depend.	53	218984	4612	33	5931	5067	14114
Nb. Labels $q = \|\mathcal{L}\|$	71	46929	10000	30	74596	47308	38072
Nb. Stimuli	3	77	19	13	24	90	4
TickSum $\sum_a c_a \cdot 10^6$	3.96	5.68	8.59	1132	43.9	3.2	48.5
Avg. LA per Task	29	698	584	4	986	182	325
Avg. LA per Run.	3	52	11	2	3	3	3
Avg. Nb. Bytes accessed by Task	43	1397	1604	2009038	1968	366	648

Table 6.2: Various properties of all case study models used as benchmarks for the evaluation

In Table 6.2, the average number of Bytes accessed by a task is calculated via Eq. 6.1.

$$\frac{\sum_i \left(\sum_{v:l_v \in \mathcal{L}_i} ls_v \right)}{n} \tag{6.1}$$

The average label size accessed by a task is comparably high for the WATERS19 model due to image-processing and stream data occupying a high amount of memory. For the cyan-colored PU bars of Figure 6.3, the upper parts, filled with a diagonal line pattern, represent the amount of PUs deviating from the lower PUs bar part without diagonal lines, and hence give coarse insights about the amount of heterogeneous PUs that differ in frequency, instructions per cycle, or clock ratio. However, in addition to Figure 6.3, Table 6.3 gives more details about the hardware heterogeneities with frequencies given in MHz.

Model	f_1	$\|P_x\| : f_x = f_1$	f_2	$\|P_x\| : f_x = f_2$	f_3	$\|P_x\| : f_x = f_3$
DEM	200	3	-			
AIM	50	3	-			
FMTV	200	4	-			
WATERS	2000	4	2000	2	1500	1
MG1	100	2	50	6	-	
MG2	600	2	200	2	-	
MG3	600	2	200	2	-	
FMTV*	200	16	-			
MG3*	60	32	20	32	-	

Table 6.3: Case study model's PU frequencies in MHz and numbers

In general, the number of PUs shown in Figure 6.3 has been used to calculate quality values for the evaluation shown in, e.g., Figure 7.7, but more PUs can be modeled to calculate wider software mapping as exemplarily shown in Figure 7.8. Compared with other models, the Democar model contains the fewest number of entities because it represents a single hypothetical ECU only, and it has been manually modeled with academic origin in [265]. The amount of RSCs and labels is also comparably low for the WATERS19 model, which shows nearly the most amount of PUs and heterogeneity. The generated models all have heterogeneous hardware and higher numbers of runnables compared to Democar, FMTV, AIM, and WATERS19 models, whereas the FMTV and AIM contain homogeneous PUs only.

In addition to the above-outlined model properties, the appendix provides additional chord charts for some models in H.6 showing dependencies between runnable and task entities. Due to the high amount of entities, not every information can be perceived, even though the figure is vector graphic based. During software application development processes, data dependencies may quickly get out of the developer's scope such that an extensive sharing of variables between tasks can be the result. To avoid these overheads, chord plots can be useful to visualize data dependencies and identify data stressing tasks or parameters. Figure H.1 gives an impression of such data dependencies as well as their communication costs, indicated by the width of connections, respectively arcs, between the entities positioned on the circumference.

With the seven outlined case study models, partitioning, task mapping, TDRR, and the various timing verification approaches can be applied to a sufficiently large, diverse, partially industrial relevant, and realistic set of models that is expected to give valuable insights into the approaches' efficiency, scalability, and result quality to be investigated and compared in the following Chapter 7.

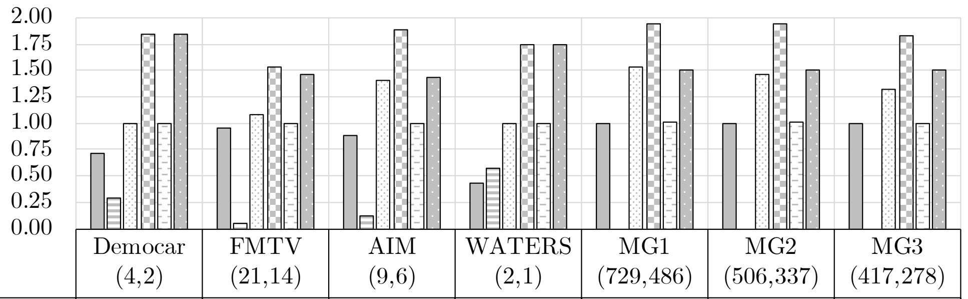
7 Metrics and Evaluation

This chapter presents measurements for the contributions (1) runnable to task partitioning, (2) task to PU mapping, (3) label to memory allocation, (4) RTA for tasks along with results from (2) and (3), (5) TCLA, and specific metrics such as accumulated contention, blocking times, and others obtained from DSE results. The metrics are outlined for assessing result quality and efficiency of solutions to the challenges (1)–(5) outlined in Chapter 4 and 5. The metric measurement results are obtained by applying corresponding calculations to the models described in the previous Chapter 6.

7.1 Runnable to Task Partitioning

To assess and compare the different partitioning approaches, they are applied to each of the case study models of Chapter 6 and assessed according to the parallelism and slackness metrics as well as the proportion of the partitioning result's span compared to the (optimal) CPP's span outlined in Section 4.4. For these assessments, activation groups and constraints that result in a subdivision of runnables are ignored. Consequently, the models are partitioned based on a much broader DAG level and more dependencies to consider for each partition. Partitioning assessment measurements are shown in Figure 7.1, which shows the percentages of parallel ■ and sequential ⊟ code as the main factors influencing the various metrics. Moreover, Figure 7.1 provides insights about the relationship of the maximal partition length to the CrPa length (span ς) across (a) the same amount of partitions for ESSP ▦ and CP-PC ⊟, and (b) $n_{ESSP_l} = n_{CP-PC_l} = \left\lfloor \frac{2}{3} \cdot n_{CPP} \right\rfloor$ i.e two-thirds of the CPP partition number n_{CPP} for ESSP ▩, indicated as ESSP_l, and CP-PC ▩, indicated as CP-PC_l, across all case study models. The two-thirds reduction of the CPP partition number is chosen for getting an impression of the increase in partition length when lowering the number of partitions compared to the CrPa, which forms the lower bound on execution time to execute the complete runnable DAG.

Firstly, the generated models show a much higher percentage of parallel code due to the randomly generated accesses to labels, which results in rather low critical path lengths compared to models with industrial focus, i.e., AIM, FMTV, WATERS, and Democar. However, higher parallel code percentages provide more flexibility for forming the tasks due to less precedence constraints necessary to be considered such that the performance of CPP, ESSP, and CP-PC gain a near bin-packing characteristic for the generated models.



	Democar (4,2)	FMTV (21,14)	AIM (9,6)	WATERS (2,1)	MG1 (729,486)	MG2 (506,337)	MG3 (417,278)
■ % parallel code	0.71	0.95	0.88	0.43	1.00	1.00	1.00
⊟ % sequential code	0.29	0.05	0.12	0.57	0.00	0.00	0.00
□ $\varsigma_{\text{ESSP}} \,/\, \varsigma_{\text{CPP}}$	1.00	1.08	1.41	1.00	1.54	1.47	1.32
▩ $\varsigma_{\text{ESSP}_{l}} \,/\, \varsigma_{\text{CPP}}$	1.85	1.53	1.89	1.75	1.94	1.95	1.83
□ $\varsigma_{\text{CP-PC}} \,/\, \varsigma_{\text{CPP}}$	1.00	1.00	1.00	1.00	1.01	1.01	1.00
■ $\varsigma_{\text{CP-PC}_{l}} \,/\, \varsigma_{\text{CPP}}$	1.85	1.46	1.44	1.75	1.51	1.50	1.50

Figure 7.1: Various partitioning metric results across case study models. The span ς results of ESSP and CP-PC approaches are put in relation to the CrPa span.

This fact manifests in a larger solution space causing higher resolution times, increased number of partitions, and more optimization potential ultimately resulting in a more generic assessment of the partitioning approaches. The solution space is not entirely investigated for metrics presented in Figure 7.1, and hence CP-PC does not equal CPP results since the solution space exploration time was limited to two hours. In Figure 7.1, the first number in brackets behind the model abbreviation name indicates the CPP partition number and the second number represents the lowered partitions amount for ESSP_l and CP-PC_l. Another interesting fact the measurements of Figure 7.1 reveals is the general domination of CP-PC over ESSP. The results of ESSP are worse (higher value) for every case study model except for Democar and WATERS models that have equal span values compared with CP-PC. CP-PC matches the optimal CPP solutions for all models except MG1 and MG2. Even for the latter, the deviation is just one % from the optimum, much lower than up to 54% for ESSP (cf. MG1). As mentioned already, optimal values were just slightly missed due to the resolution time limitation. Only the Democar and WATERS models result in equal partitions for ESSP and CP-PC for both partition numbers due to the relatively small amount of total runnables but many dependencies such that less flexibility is given during the runnable partitioning exploration.

For the AIM and FMTV models, deviations for the number of CPP partitions were discovered during the measurements, due to the cycle decomposition method (cf. Section 5.6.1) not scaling well with the relatively high number of dependencies, especially for the AIM model (218984). No simple cycle algorithm of either Szwarcfiter and Lauer [267] $\mathcal{O}(p+|\mathcal{E}|\cdot|Cy|)$, Tarjan [203] $\mathcal{O}(p\cdot|\mathcal{E}|\cdot|Cy|)$, Johnson [268] $\mathcal{O}(((p+|\mathcal{E}|)\cdot|Cy|)$, or Tiernan [269] $\mathcal{O}(p.const^{p})$ found all cycles within a one hour runtime. Consequently, an alternative cycle decomposition method is used to remove cycles by decomposing a random dependency of the first found strongly connected components (i.e., runnables that can reach each other through a dependency path). Due to this method being highly efficient but not very effective, results can be produced in a short time period and used for the partitioning assessment by ignoring the goal of decomposing as few dependencies as possible. The latter can still be achieved through a more fine-grained analysis of the cycle identification approaches, which is beyond the partitioning assessment scope. The arbitration during the cycle decomposition causes both varying partition numbers and span lengths. This variance is shown in box plots of Figure 7.2 taken from 100 consecutive partitioning measurements, whereas star symbols indicate the solutions arbitrarily chosen for metric comparison measurements of Figure 7.1. In addition to Figure 7.1, Figure 7.3

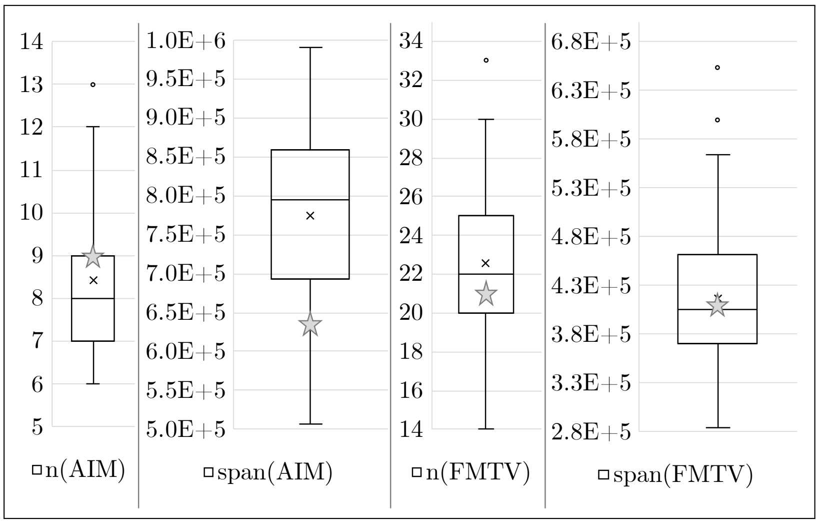
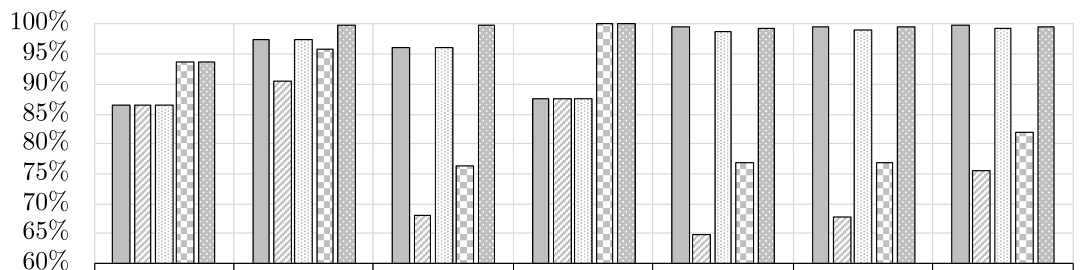


Figure 7.2: Variance in partition number and span for AIM and FMTV models due to arbitration within the cycle decomposition phase

provides slackness metrics and hence the proportion of parallelism and the number of partitions for all partitioning approaches and all case study models. The closer the slackness is to 100%, the more balanced the partitions are. ESSP falls behind and creates much less



	Democar (4,2)	FMTV (21,14)	AIM (9,6)	WATERS (2,1)	MG1 (729,486)	MG2 (506,337)	MG3 (417,278)
■ CPP	86.50%	97.48%	96.00%	87.50%	99.58%	99.51%	99.71%
▨ ESSP	86.54%	90.60%	67.91%	87.45%	64.71%	67.68%	75.57%
□ CP-PC	86.54%	97.47%	95.98%	87.45%	98.81%	98.95%	99.37%
▦ ESSP$_1$	93.75%	95.79%	76.26%	100.00%	76.97%	76.81%	81.86%
▦ CP-PC$_1$	93.75%	99.90%	99.96%	100.00%	99.19%	99.68%	99.66%

Figure 7.3: Slackness ζ results across partitioning approaches and case study models

effective partitions as shown by lower slackness values. Information derived from Figure 7.3 is further twofold. On the one hand, partitioning results show a good load balancing, derived from slackness values close to 100% in general. The smaller the slackness value is, the higher is the variance in the sum of instructions across the created partitions. Actual parallelism (slackness' nominator) are shown in Table 7.1. On the other hand, it is observed that lowering the number of partitions yields in higher slackness values, i.e., maximal parallelism is traded for load balancing across tasks. Apart from the generated models,

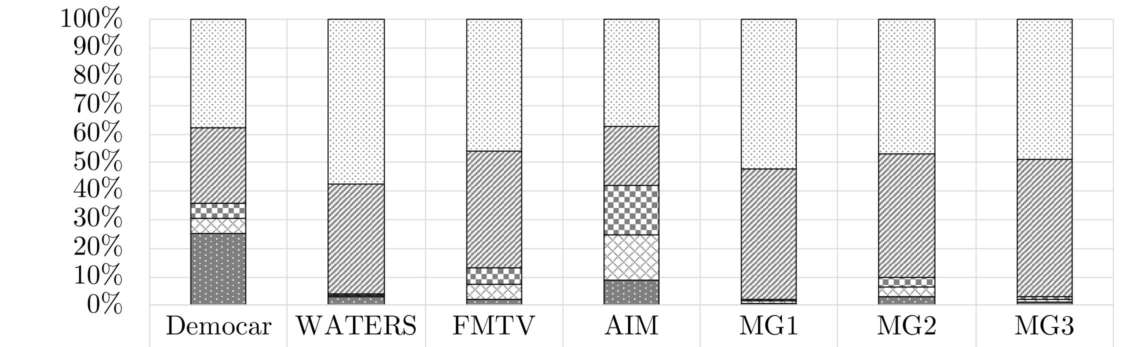
	Democar	**FMTV**	**AIM**	**WATERS**	**MG1**	**MG2**	**MG3**
ξ_{CPP}	3.46	20.47	8.64	1.75	725.94	503.52	415.77
ξ_{ESSP}	3.46	19.03	6.11	1.75	471.73	342.47	315.11
ξ_{ESSP_l}	1.88	13.41	4.58	1.00	374.08	258.85	227.58
ξ_{CP-PC}	3.46	20.47	8.64	1.75	720.29	500.71	414.38
ξ_{CP-PC_l}	1.88	13.99	6.00	1.00	482.07	335.92	277.06

Table 7.1: Parallelism ξ results across partitioning approaches and case study models

CPP results in a partition number that equals the model's parallelism value rounded up to the next integer value. Specific parallelism values are given in Table 7.1. The increase in partition number for the generated models compared to the parallelism value is though very small and just exceeds the value by only three, two, and one partitions for MG1, MG2, and MG3, respectively (cf. the number of partitions indicated by the first number in brackets of Figure 7.3, and parallelism values of Table 7.1). This observation can be referred to the fact that (i) the amount of generated models' partitions is rather high and the span relatively low, and as a consequence (ii) the deviation in instructions required for some runnables necessitates creating additional partitions when respecting precedence constraints.

Finally, efficiency must be compared across the partitioning approaches. Of course, this is a significant disadvantage of CP-PC compared with CPP and ESSP due to using the Choco solver engine instead of a greedy-based heuristic, which requires much more computational resources. The following resolution time measurements are taken from a computer using an Intel i9 9th-gen 8-core processor with 32GB RAM. Measurements are repeated ten times, and the average resolution time values across these ten measurements are shown in Figure 7.4. Due to the high flexibility in allocating runnables across tasks and the high amount of runnables for the generated models, the resolution time limitation was set to one hour compared to one minute for the Democar, WATERS, FMTV, and AIM models. Additionally, the solver was configured to stop as soon as the result quality of ESSP was reached. The reduced task number limitations for ESSP and CP-PC are shown here, too, since they show meaningful resolution time efficiency when the CrPa does no constitute the most prolonged partition. The latter may significantly reduce the solution space, resulting in no meaningful resolution time measures, since having larger solution spaces, which is achieved by reducing the number of partitions, increases resolution time.

On average across all case study models, ESSP takes 1.36 times longer than CPP. CP-PC$_l$, i.e. CP-PC with a task number configured to two thirds of CPP's resulting task number, already struggles at the Democar model, which has only 43 runnables. However, even though CP-PC$_l$ does not investigate the entire solution space due to resolution time limitation, results are still better than ESSP, which is shown by the previous measurements via, e.g., an increased parallelism value. The model property differences cause a high deviation in the resolution time factor for CP-PC compared to ESSP, which reaches from 1.29 for the AIM model up to 150 for the WATERS model. On average across all case study models excluding the small WATERS and Democar models, the resolution time of



	Democar	WATERS	FMTV	AIM	MG1	MG2	MG3
⬚ CPPC$_l$	43	450	30999	58520	1883986	198337	1757597
▨ CPPC	30	300	27486	32365	1630316	183223	1727342
▦ ESSP$_l$	6	3	3843	27253	23074	13738	40219
⊠ ESSP	6	3	3676	25146	23208	13811	40080
■ CPP	29	25	1366	13799	32517	14124	40615

Figure 7.4: Resolution time in milliseconds across all case study models

CP-PC and CP-PC$_l$ is approximately 27 and 30 times the ESSP resolution time. It is important to note that in the general case, CP-PC finds valuable solutions better than ESSP early, and the rest of the CP-PC resolution time is spent on investigating the entire solution space, i.e., all possible results to prove the optimum. Since proving the optimum is not always required and improving ESSP results is of primary interest, it is reasonable to say that CP-PC outperforms ESSP with an acceptable increase in resolution time. Results obtained in Figure 7.4 are therefore configured for at least providing the span of the ESSP result. When comparing the generated models with FMTV and AIM, which provide comparable runnable numbers, Figure 7.4 shows that with an increase of the proportion of parallel code towards one and respectively a proportion of sequential code towards zero, the resolution time increase for CP-PC over ESSP grows. Especially when considering the AIM model, resolution times do not significantly deviate between the partitioning approaches. Here, CP-PC takes approximately 1.29 times the resolution of ESSP for partitioning 1297 runnables into nine tasks.

To conclude, CP-PC is the clear winner for partitioning runnable sets into potentially concurrently executing tasks when strictly defining the number of partitions. Even given a low amount of precedence constraints and hence high flexibility in distributing runnables, the resolution time is still reasonable, and resulting partitions are close to the optimal parallelism of runnable DAGs. If the partition amount is allowed to be arbitrary, CPP should be preferred because it provides partitions respecting the CrPa, i.e., the lower bound on the execution time for the complete runnable set while keeping the resolution time much lower compared to CP-PC. The concurrent execution of a CP-PC, ESSP, or CPP partitioned applications potentially benefits from reduced application execution times and, e.g., less energy consumption than sequential execution due to the potential of running more PUs with lower frequencies [270]. All approaches are based on Amalthea such that a variety of e.g. Autosar models can leverage the benefits of partitioning by using Amalthea importers and exporters to and from established tools used in the automotive industry. CP-PC, ESSP, and CPP form the first Amalthea-based open-source contributions for valuable software partitioning in the automotive industry considering

Abbr.	Description
DFG	Data flow graph heuristic [41]
ILP	Integer linear programming using oj!algo[a] [41]
GA	Genetic algorithm using jenetics [42] [41]
CP	Constraint programming without any optimization using the library from [44]
CPLB	CP + optimization for load balancing, i.e., minimizing the maximal PU utilization using the library from [44]
CPMO	CPLB + optimization for inter PU communication costs (*ipuc*) → multi objective optimization using the library from [44]

Table 7.2: DSE approaches applied to task-PU mapping DSE

the broad constraint set of Table 5.9. The consideration of a broader constraint set provides an automated and optimized generation of tasks that results in efficient data progression as well as tasks potentially being executed concurrently without much inter-task communication.

7.2 Software Distribution

Various (meta-)heuristics are used and implemented along with this thesis to explore the solution space of software distribution for automotive systems. More precisely, each of the use case models' task set (cf. Table 6.2) is explored according to being mapped to the models' PUs via six DSE approaches outlined in Table 7.2. The CP-based approaches, highlighted with gray background in Table 7.2, were implemented from scratch using the Choco library [44]. ILP, DFG, and GA are accessible from the reference [41] as indicated and are used for comparison purposes. However, significant changes had to be made to the latter approaches to be compliant to AMALTHEA 0.9.8, which address heterogeneous hardware features via utilization metrics, and to consider instruction types such as ticks and execution needs, among others. Available implementations only cover minimizing instructions across PUs, which is inappropriate for load balancing according to heterogeneous systems. These necessities were identified when applying the new WATERS19 model to a measurement cluster for solving the mapping problem using ILP, GA, DFG, CP, CPLB, and CPMO, at which the former two approaches failed. As a consequence, implementation is refactored to target PU utilization, which results in more reasonable load balancing.

For the DSE comparison, the maximal PU utilization $\hat{U}^P$ in %, inter PU communication *ipuc*, speedup S, and resolution time $t_{reso}(ms)$ metrics are used. Inter-PU communication considers the access rate, label size (and cache line), and the task to PU mapping shown in Eq. 7.1.

$$ipuc(M_\tau^P) = \sum_{i,j\in[1,n]} \left(ipuc_{i,j} = \begin{cases} 0 & \text{if } i = j \vee M_i^P = M_j^P \\ \sum\limits_{v:l_v \in (\downarrow_{\tau_i} \cap \downarrow_{\tau_j})} \left\lceil \frac{ls_v}{cl} \right\rceil \cdot \frac{10^{12}}{T_i \text{ in ps}} & \text{otherwise} \end{cases} \right) \tag{7.1}$$

The CPMO approach uses the *ipuc* metric as a coarse indicator for inter-PU communication, but CAN message delays can potentially be used here via incorporating equations of Section 5.7.2, too. Access latency values between PUs and memories, denoted as $\downarrow_{x,d}$, require a label to memory mapping and are ignored for the presented metrics due to (1) the models mostly not providing the required information (neither access latency values nor label mappings), and (2) the increasing complexity due to the circular dependency between access cost derivation, which depends on the task to PU mapping that in turn depends on the access cost derivation. The latter is not intractable to solve, but has been omitted along with provided measurement results shown in Table 7.3, since the simplified version already gives a good representation of inter PU communication. However, label to memory mapping evaluation is presented for a static task to PU mapping in Section 7.5. For an advanced RTA-based mapping, a new GA approach is implemented, outlined in Section 7.3, and includes the various constraints and analyses described in Chapter 5.

To conduct the measurement clusters across DSEs and use case models, the partitioning process forms a crucial requirement for configuring the different task numbers. Therefore, the extended ESSP approach is used, which considers the activation groups and splits partitions in order of highest IPC first. The comparison of DSEs approaches in Table 7.2 does not infer generality since there are many-fold parameters to configure, which significantly affect efficiency (resolution time) and effectiveness (e.g. maximal PU utilization). This configuration includes (a) the modeling of genes, chromosomes, population, and others according to parameters to be set during DSE, (b) the implementation of mutation and crossover operations, and (c) the limitation implementation by time, fitness, number generations, or similar for the GA or (i) the modeling of variables (graph, integer, real, set, and more), (ii) constraints (arithmetical, logical, set, graph, and more), and (iii) search strategies (bounded, activity-based, smallest domain / lower bound, greedy branching, random, and more) for CP.

Metric measurement results obtained by applying the DSE heuristics of Table 7.2 to the models of Table 6.2 are shown in Table 7.3 and also as bar chart resolution time and utilization plots across Figures 7.5 and 7.6.

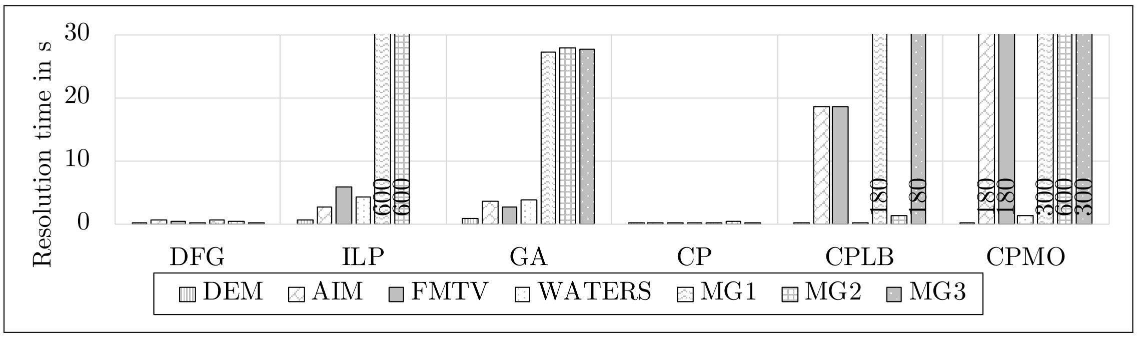


Figure 7.5: Different DSE's resolution times in seconds for case study models

Bars of Figure 7.5 with an open-top (no closed top bar line) required the resolution time indicated in the bar's base. The maximal time is limited to ten minutes, whereas every 10% of the resolution time (1 minute in this case) solutions of CP and CPMO are checked to be existent and whether they increased in number over the last 20% of resolution time. If both conditions are met, the solver is stopped to lower the resolution time since many cases were observed that did not improve the solution *fitness* after *early* solution findings. Consequently, the maximal resolution time is only met twice for the ILP approach and once

Model	**$\|\mathcal{T}\|$**	**|P|**	**$\hat{S}^P$**	**DSE**	**$\hat{U}^P$**	**ipuc**	**t_{reso}**	**S^M**	**DSE**	**$\hat{U}^P$**	**ipuc**	**t_{reso}**	**S^M**
DEM	19	1*3	3	DFG	96.8	5300	82	2.52	CP	96.8	5400	13	2.52
				ILP	96.8	5500	610	2.52	CPLB	96.8	5400	14	2.52
				GA	96.8	4900	916	2.52	CPMO	96.8	4400	117	2.52
AIM	120	3	3	DFG	37.61	13863767	639	2.98	CP	44.08	14017858	34	2.54
				ILP	39.47	14570572	2642	2.84	CPLB	37.40	13859388	18601	3.00
				GA	37.47	14404530	3646	2.99	CPMO	44.19	13822987	60000	2.54
FMTV	100	4	4	DFG	12.41	368754	322	3.63	CP	15.08	357611	38	2.99
				ILP	11.32	398908	5885	3.98	CPLB	11.28	356001	18606	4.00
				GA	11.27	364041	2589	4.00	CPMO	15.70	345107	60000	2.87
WATERS	14	2+4+1	6.75	DFG	89.00	94275247	83	6.16	CP	85.25	60484726	17	6.43
				ILP	88.28	55029358	4190	6.21	CPLB	82.91	84760510	22	6.61
				GA	82.91	82870206	3821	6.61	CPMO	89.00	16426707	1406	6.16
MG1	120	2+6	5	DFG	73.19	25907	585	3.96	CP	90.34	25203	83	3.21
				ILP	64.56	26007	600000	4.49	CPLB	58.50	25148	186000	4.96
				GA	59.03	26078	27288	4.91	CPMO	99.25	23803	306000	2.92
MG2	512	3+3	5	DFG	22.39	73408	485	4.26	CP	30.25	70284	307	3.15
				ILP		-			CPLB	21.40	75939	1373	4.45
				GA	22.39	72023	27955	4.27	CPMO	36.39	64934	600000	2.62
MG3	512	2+2	2.67	DFG	26.05	32927	294	1.33	CP	24.20	32715	162	1.44
				ILP		-			CPLB	13.10	31112	186000	2.65
				GA	13.10	31099	27672	2.65	CPMO	32.20	31544	306000	1.08
FMTV*	128	16	16	DFG	38.27	455684	239	11.78	CP	43.12	457749	250	10.46
				ILP		-			CPLB	29.97	457402	558000	15.05
				GA	33.38	455062	4623	13.51	CPMO	57.74	424543	1800000	7.81
MG3*	512	32+32	42.67	DFG	19.23	42682	1190	18.06	CP	29.53	42718	24862	11.76
				ILP		-			CPLB	08.49	42622	186000	40.88
				GA	15.22	42581	27296	22.81	CPMO		-		

Table 7.3: DSE results for case study models: utilization $\hat{U}^P$ in % and resolution time t_{reso} in milliseconds

for CPMO. In general, the GA DSE appears to be the best trade-off between resolution time and optimization and hence, it is used for the task-chain delay and label mapping DSE in Sections 7.3.2 and 7.5, respectively. Figure 7.6 extends Table 7.3 by comparing the resulting maximal PU utilization.

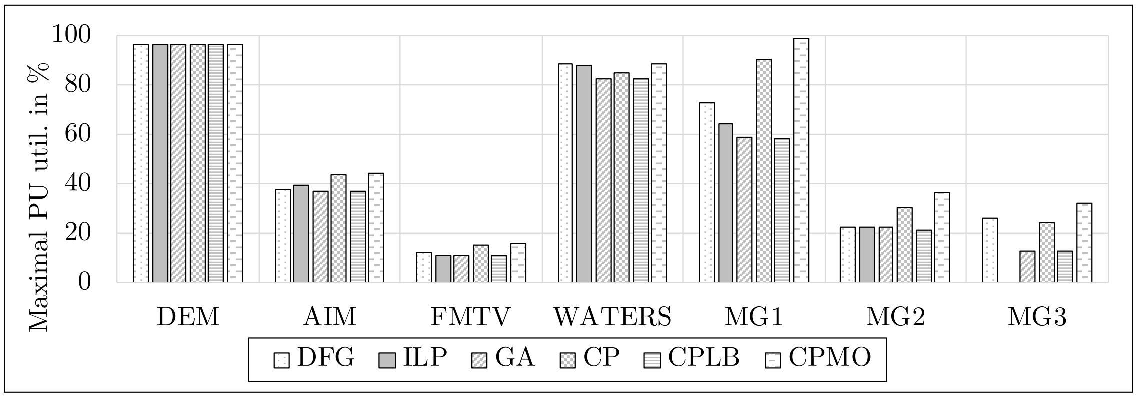


Figure 7.6: Maximal PU utilization across case study models and DSE approaches

Figure 7.6 supports the statement of GA being a general good trade-off due consistently meeting the lowest PU values in line with CPLB. No result for the ILP approach at Amalthea model MG3 is shown due to the solver not finding any feasible solution within the limit of 10 minutes.

The DSE heuristics can further be analyzed to the broader task and PU sets by configuring the partitioning accordingly or adding varying and hypothetical PUs to the Amalthea models. Corresponding results are presented in Figure 7.7, which compares the maximal PU utilization $\hat{U}^P$ and inter PU communication costs *ipuc* according to three different task numbers (light gray markers) across the DSE heuristics for all models (each in a separate scatter plot (a)–(f)) except the WATERS model. The latter is excluded from these measurements because it only provides a single runnable per task (that requires instructions for being executed) with almost individual activation parameters, such that partitioning cannot subdivide the existing task set, and no varying task numbers are feasible. For other case study models, task and PU configurations are indicated along with each subplot's title. The colored (darker filled) markers represent the average over the three (gray) task number configurations. In all cases, the CPLB and GA heuristics result in the two best load balancing results by consistently providing the lowest maximal PU utilization values (cf. Table 7.3). Essentially, the lower-left a marker in Figure 7.7 is, the better the solution is due to the lower maximal PU load and inter-PU communication, respectively. CP and CPMO approaches show worse results compared with GA and CPLB due to no optimizing at all and trading load balancing for reducing inter PU communication costs, respectively. With an increasing number of PUs and tasks, CPMO and ILP results get worse due to scalability issues and resolution time restrictions, which have been set to allow comparing the results based on similar resolution time. The CP approach does not result in very effective outcomes, but as shown in Figure 7.9, its resolution time is the lowest for nearly all measurements. The inter PU communication cost *ipuc* is only optimized for CPMO and hence arbitrary for other DSEs. In general, the best $\hat{U}^P$ values are achieved for CPLB and GA. CP results in the quickest resolution times. As a greedy heuristic, the DFG approach quickly finds good solutions for the Democar and AIM models, but in turn, suffers from

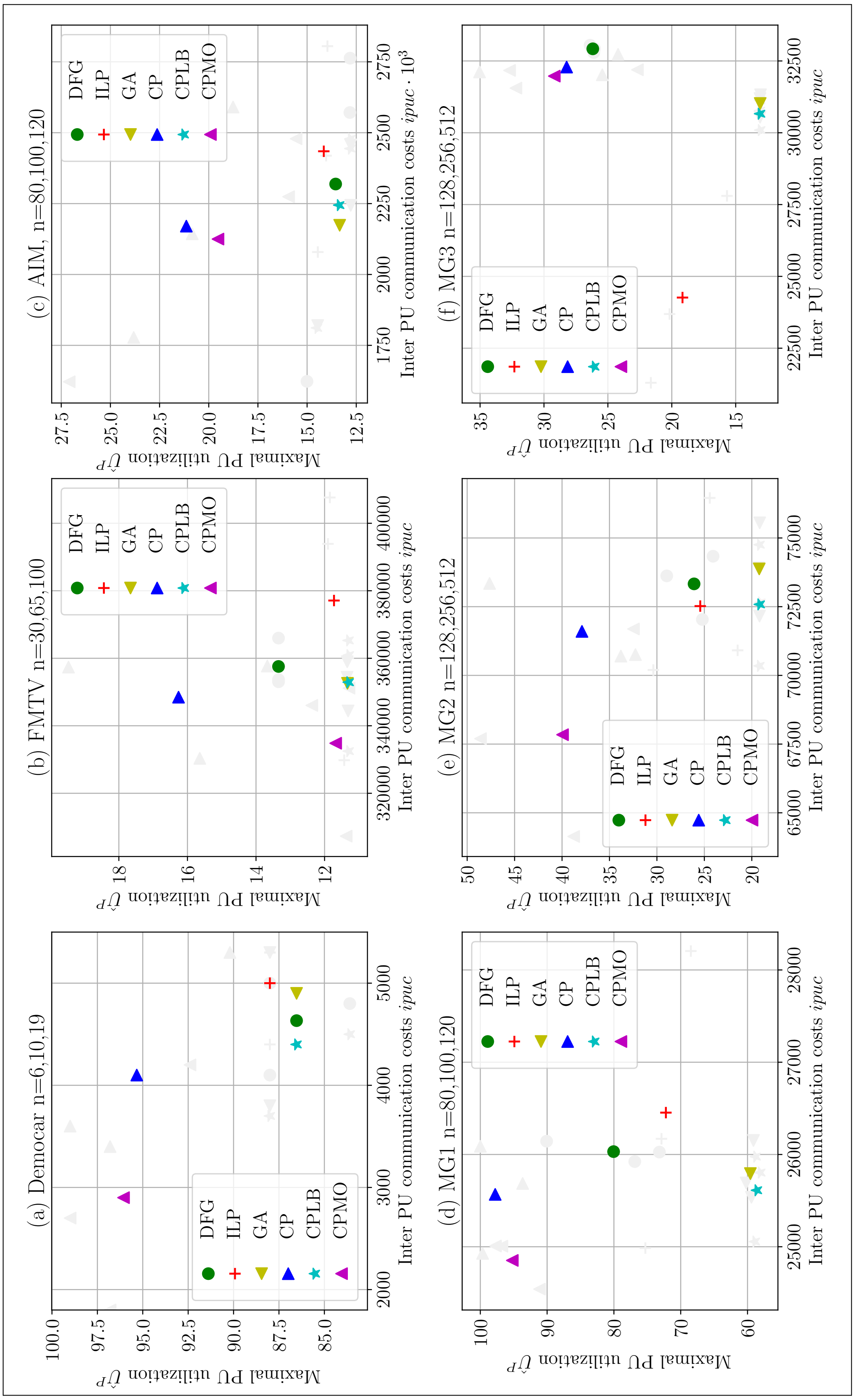


Figure 7.7: $\hat{U}^P(ipuc)$ results for various DSEs across DEM, FMTV, AIM, MG1, MG2, and MG3 case study models

local optima and hence only average results for all other case study models. In general, the DFG approach has no point in favor, albeit not relying on any third-party library.

Given that many-fold constraints can be easily incorporated with CP in contrast to DFG, ILP, or GA, CP is an appropriate choice for quickly identifying valid solutions while considering a variety of constraints that all need to be fulfilled in a given solution. As soon as larger models are addressed, i.e., subplots (b)-(f) in Figure 7.7, CPLB and GA create the best (lower $\hat{U}^P$) results for most measurements. Even setting the resolution time to a single minute creates better results than DFG or ILP approaches in many cases. For instance, at the generated model (d) MG1, CPLB results in the lowest $\hat{U}^P$ across all DSEs. Reducing the resolution time often causes the ILP solver not to provide any feasible result. Measurements provided in Figure 7.7 feature a resolution time of ten minutes, at which ILP does provide results for MG2 at 512 partitions. Since solutions are found after 20 Minutes, resolution times are raised for MG2 and MG3 accordingly. A significant influence on resolution time is whether there exists a relatively large task that contains a sequence of runnables, i.e., a task that can not be further subdivided. If this task occupies more utilization than the sum of all other task instructions divided by the sum of all PU capacities, it forms the lower bound on execution time (assuming that it is mapped to the fastest PU) and investigating a considerable part of the solution space is obsolete since the optimization goal does not change due to the comparably *longer* task. As the DFG heuristic sorts tasks by their instruction costs and assigns those beginning with the largest chronologically to an ordered list of PUs beginning with the fastest (most instructions per second), addressing situations with *long* CrPas (tasks) works well, which is the case for the Democar and AIM cases, but not with other case study models, which provide more flexibility and somewhat balanced tasks.

Figure 7.8 presents the **speedup** of (a), the MG1 model, and (b), the FMTV model, along with an increasing number of homogeneous PUs (speedup(number of PUs u)). The used speedup calculation is based on [28] and provided in Eq. 7.2.

$$S = \frac{\sum_j \left(\min_k (C_{j,k}^{+,s}) \right)}{\max_k \left(\sum_j C_{j,k}^{+,s} \mid ta_{j,k} = 1 \right)} \tag{7.2}$$

Here, the nominator defines the minimal **sequential runtime** of all tasks being mapped to the fastest PU. The denominator depends on the task mapping and identifies the maximal runtime across all PUs, i.e., **parallel runtime**. This speedup calculation applies to a heterogeneous PU structure and can be seen as the fraction of the time before the parallelization and the time after the parallelization, as introduced in [28].

Due to limited dependencies between the tasks and runnables and a relatively homogeneous instruction distribution, almost optimal speedup factors can be reached, whereas CPLB found the best values according to Figure 7.7 (f) and Figure 7.8. Surprisingly, the DFG approach creates better results for the MG3 model compared to GA. However, this is not the case for the FMTV model, as shown in Figure 7.8 (b) due to its more heterogeneous nature. While ILP does not scale beyond eight PUs for the given resolution time at all, CPLB requires significantly more runtime after the amount of 16 PUs. The optimal, model-independent, speedup value equals the number of PUs, but is barely achievable due

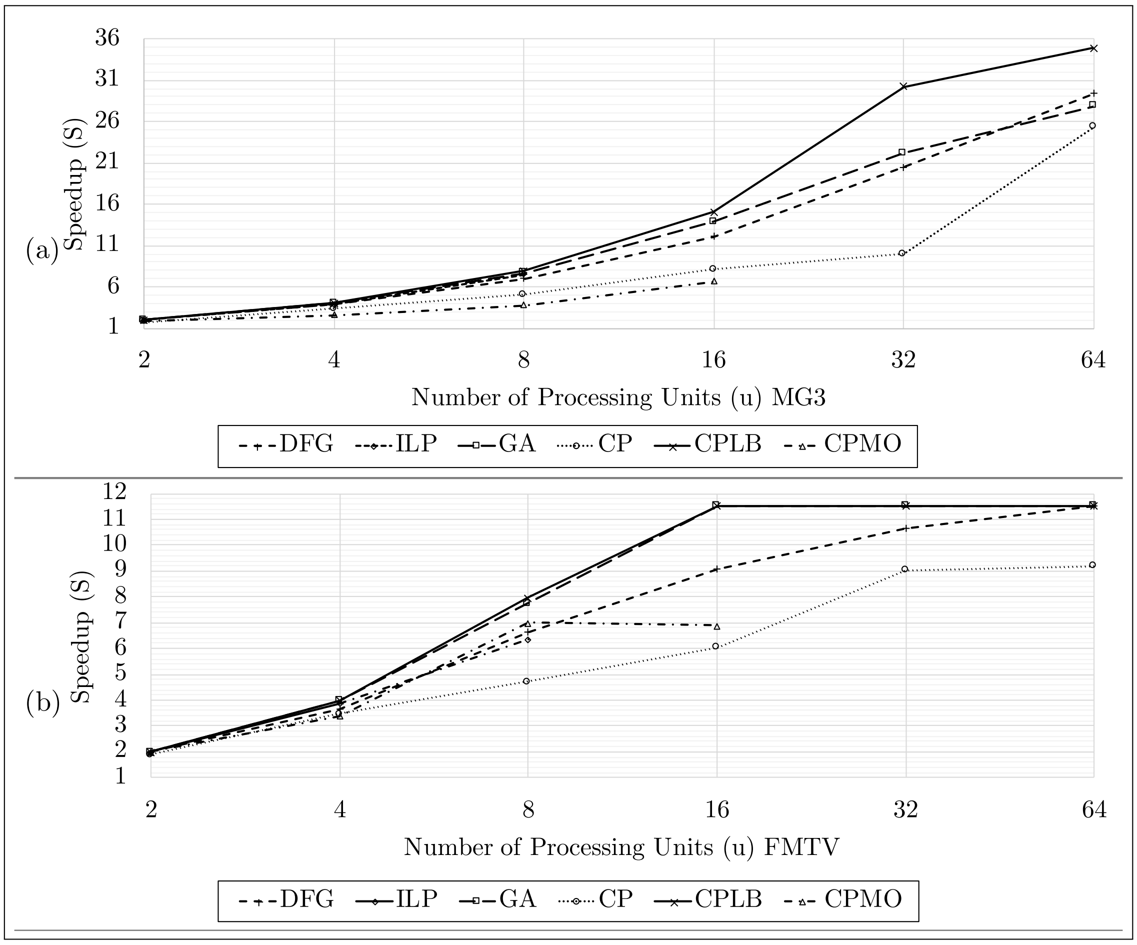


Figure 7.8: *Speedup*(u) of different DSE results for MG3 and Fmtv models

to communication costs and varying task sizes.

With the Fmtv model (cf. Figure 7.8 (b)), such speedup is saturated already at 16 PUs because there is a single task that cannot be subdivided further and consequently forms the lower bound on schedule length. To avoid saturation, the model would have to provide fewer dependencies and more homogeneous tasks regarding their sum of instruction costs. While the difference between GA and CPLB is smaller in (b) than in (a), CPLB still provides the best results for all number of PUs. As mentioned before, the DFG approach produces worse results in (b), whereas CP achieves better results than in (a). Other than that, results are similar to the MG3 model in (a).

Figure 7.9 presents the resolution time of different DSE approaches for an increasing number of tasks for (a) the Democar and (b) the Fmtv model. Results show that the multi-objective constraint programming approach CPMO scales worst with the number of tasks as it is the only approach with multi-objective optimization. The CPLB approach also does not scale very well, but the measurements often show that reasonable solutions are found nearly as quick as the GA approach does, and the rest of the resolution time is used to search the entire problem space. Furthermore, as soon as a single task defining the lower bound on the maximal PU utilization, i.e., a comparably large task that contains a high amount of instruction and is consequently mapped to the fastest PU, CPLB runs quicker than ILP, DFG, or GA. If tasks' instructions are balanced, CPLB scales worse than ILP with the number of tasks but better with the number of PUs. It is important to note that CPLB is always able to find at least a valid solution, while the ILP solver fails, e.g., for eight PUs and more (cf. Figure 7.10), even with hours of resolution time. Concerning larger models, the single objective constraint approach (CPLB) outperforms almost every DFG, ILP, and GA result (except Figure 7.7 (e) and (f) for $|\mathcal{T}| = 512$) whereas the CPMO tends to create worse results beyond a task number of 65. The peak

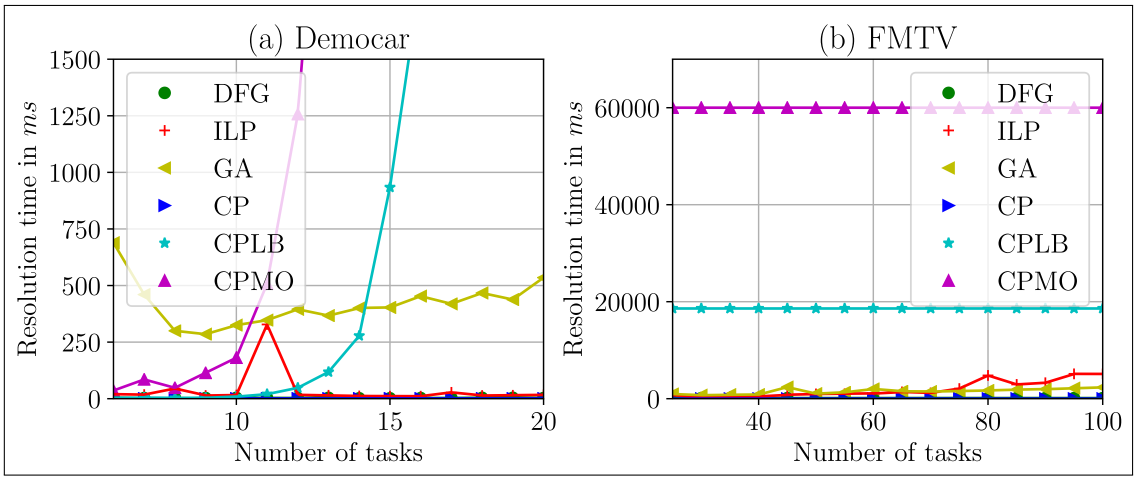


Figure 7.9: $Resolutiontime(n)$ of different DSEs for (a) Democar and (b) FMTV

for ILP at 11 tasks has been verified over multiple measurements and is probably caused by an increased number of optimal solutions by coincidence, due to the partitioning to this particular amount of tasks results in multiple equally *loaded* tasks. Figure 7.9 (a) shows that the CP approaches with optimization do not scale well with the number of tasks, and the CP and DFG approaches are yet the quickest with insignificant deviations. When applying the various DSEs to bigger models, the situation is similar: CPLB meets its limits to investigate the complete solution space at about 20 tasks. CPLB, however, finds valid solutions already at the same time the CP approach does ($< 5ms$), while ILP may not provide solutions before its resolution time at all.

The following line charts of Figures 7.9 and 7.10 provide information about the DSE's scalability, depending on the number of tasks and PUs on the X-axis as indicated. Figure 7.10 presents the resolution time t_{reso} (a) and the *ipuc* values (b) of different DSE approaches along with an increasing number of PUs u for the FMTV model. Once again, CPMO performs with its maximal runtime definition (here set to 15 minutes), and CP finds valid solutions most quickly. The GA approach performs well but still takes longer than CP for each result. The worst scaling behavior shows the ILP approach. Above eight PUs, the ILP solver did not find a valid solution at all. CPMO does also not scale well with the number of PUs and fails beyond 16 PUs for the same resolution time restriction. Interestingly, the CPLB approach starts with requiring the full defined runtime but drops to a minimum resolution time with 16 PUs and above. This resolution time reduction is due to the fact that below 16 PUs, the solution space covers a huge variety of task to PU mapping combinations resulting in different maximal PU utilizations. As soon as 16 or more PUs are available, one relatively huge task defines the lower bound of PU utilization and mapping the other tasks to other PUs does not reduce this minimal PU

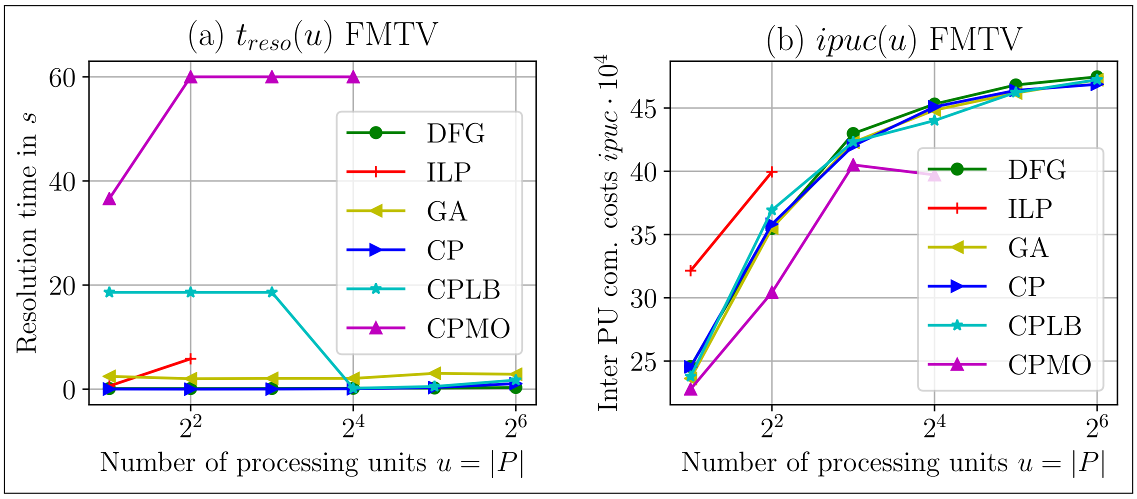


Figure 7.10: (a) Runtime(u); (b) $ipuc(u)$ of different DSEs for FMTV

utilization. Consequently, since the optimization targets only to minimize the upper PU utilization bound but not maximizing the lower bound (this would contrarily keep the CPLB resolution time high), its optimization is done, and solutions are available quickly. Figure 7.10 (b) also shows a linear increase in communication costs with the increasing number of PUs as well as the CPMO approach with the lowest *ipuc* values.

As soon as a heterogeneous structure of PU is present, the $\hat{U}^P$ metric is necessary to overcome limitations of, e.g., reducing the absolute maximal IPC per PU. Additionally, the CP solver can be configured to a specific initialization to overcome the arbitrary initial assignment values that often create an undesired homogeneous mapping along with the heterogeneous system. For example, instead of the PU utilization constraint only, the initial assignment could feature another lower bound comparable to Eq. 4.19, i.e., $\forall x : \sum_i \boldsymbol{M}_{\tau_i}^{P_x} \geq 1 \;$ with $n \geq u$. Assuming that the task number is higher than the number of PU, this equation ensures that at least one task is mapped to each PU.

Finally, Pareto front plots are presented in Figures 7.11 and 7.12 and give insights to the CPMO result sets, which are selected by lower *ipuc* values in Table 7.3, but in fact, provide a solution set, of which each entry (x marker in Figure 7.11) represents a solution that has an individual combination of (minimized) optimization parameters. The dotted trend line is added to help identify dominating solutions that have comparably good optimization value pairs, which are found below the trend line. Not all models result in a high amount of solutions for a Pareto front, especially given that CPMO does not scale well with the number of tasks.

As an intermediate summary, it is observed that various DSE provide broad flexibility for engineers facing the highly constrained problem of distributing automotive software to heterogeneous hardware with varying architectural structures and patterns. The lightweight CP approach without any optimization provides valid solutions faster than any other comparable approach, such as DFG, ILP, or GA. The single objective optimization approach CPLB provides optimal or nearly optimal solutions for most of the measurements. The CPMO approach covers multi-objective optimization with accessible Pareto fronts in an appropriate amount of time. For optimal results, however, the multi-objective CPMO approach requires significantly more time. A great benefit of using the CP

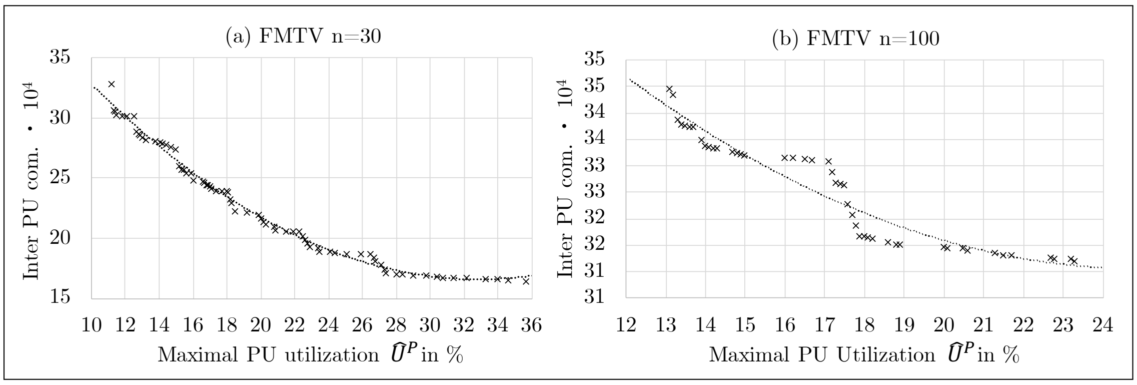


Figure 7.11: Pareto-front line charts for CPMO across the FMTV model partitioned into (a) 30 and (b) 100 tasks

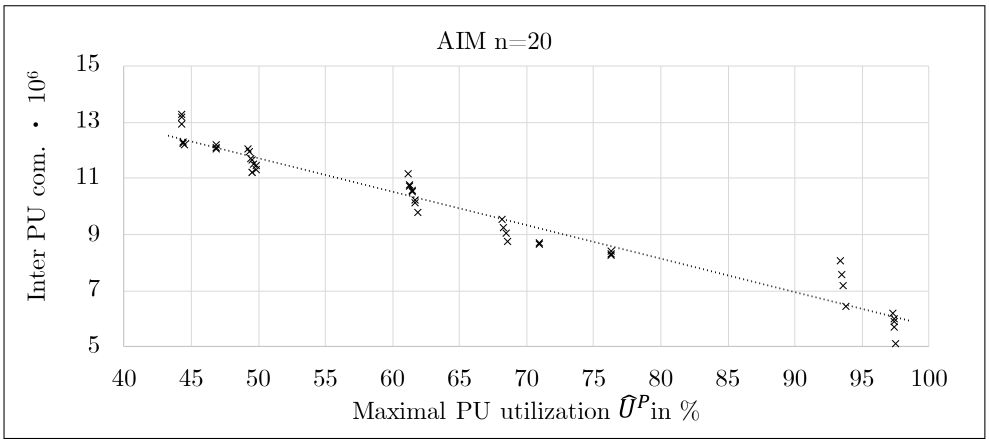


Figure 7.12: Line chart for CPMO Pareto-front of AIM model partitioned into 20 tasks

paradigm is also an automatic constraint validation that informs programmers about any contradicting or flawed model entities, variable bounds, or constraints. Such validation requires additional efforts when using different DSEs. CP applies very well to highly constrained domains consisting of combinatorial design spaces typically employed by automotive systems. It preserves the natural modeling and programming activities while providing optimal, Pareto-optimal, or near-optimal solutions in reasonable resolution time. Typical automotive constraints, consecutive constraint modeling, and solving partitioning and task mapping problems with a constraint solver are presented. Optimization goals can be easily changed or adapted without the necessity to combine or introduce new linear inequalities, which is required for ILP, or adjusting the fitness evaluation method for GA approaches. However, if resolution time is crucial and the modeling of genes, chromosomes, crossover operations, and more does not impose too much overhead, GA approaches should be in favor due to scaling better with the problem space.

7.3 CPU-GPU Response Times and Task Chain Delays

As mentioned before, the task to PU mapping problem not only requires DSE approaches for addressing optimization goal(s), but also necessitates verifying various timing properties, e.g., guaranteeing that worst-case task (chain) response times are always lower than their deadlines or covering GPU timing constraints and characteristics, which is addressed in Chapter 5.6 and evaluated in this section. As the WATERS case study model is the only one providing GPU information, this section solely considers the WATERS(2019) model and addresses the work and research in line with the challenges proposed in [30] and partially published in [19].

The WATERS challenges' AMALTHEA models often include some inconsistency on purpose, which is required to be found by the addressees. For the 2019 model, this inconsistency is constituted by the task `Planner`, which has a periodic activation of $12ms$, but an execution time higher than its period $>12ms$ for any PU. On behalf of this issue, the `Planner`'s periodic activation is increased to $15ms$ to provide feasibility.

As the previous Section 7.2 shows, the GA approach to the mapping problem including RTA is promising and hence implemented using the jenetics library [42][58]. To restrict the solution space for tasks constrained to run on either CPUs only, GPUs only, or both, instead of decoding a single chromosome with multiple integer genes, each task mapping is encoded within a dedicated chromosome consisting of a single integer gene. Consequently, genes can have different domains, which is impossible when encoding multiple genes within the same chromosome. The implemented GA includes timing verification in the form of RTA for CPU and GPU task sets and can be configured to optimize the following metrics.

I **RTSO** = Response Time Sum Optimized includes:

  i the sum over all tasks' worst-case response times across CPUs and GPUs, i.e., minimize $R^+_\mathcal{T} = \sum_i R^+_i$ that involves equations 5.37, 5.58, and H.19,

  ii the total memory access latency (cf. Eq. 5.72),

  iii the total CE latency (cf. Eq. 5.54), and

  iv the total task contention (cf. Eq. 5.55).

II **TCO** = Task Chain latency sum Optimized minimizes the sum of all task chain latency values using either of Eq. 5.44, 5.45, 5.46, 5.47, 5.48, or 5.49 depending on whether (a) worst or (b) best task chain (c) reaction or (d) age delay for (e) implicit or (f) LET communication paradigms is targeted.

III **LBO** = Load Balancing Optimized minimizes the maximal PU utilization (minimize $\hat{U}^P = \max_x U^P_x$ and U^P_x given in Eq. 5.8).

IV **EV** = No optimization at all - similar to the CP-based DSE approach of Section 7.2, but verification of all constraints, namely pairings, separations, sequencing, mapping, capacity, and deadlines (cf. Table 5.9), which also hold for all above GA implementation approaches.

V **SYN** = Response time sum optimized, but under synchronous offloading[59], i.e., an offloading task's execution time is increased by the offloaded task's response time (only used in and valid for measurements of Section 7.3.3).

[58]Many other similar libraries exist, but jenetics has been chosen due to its compliance to Eclipse's IP policies and its Java-based implementation.

[59]The **SYN** mapping can also be executed asynchronously, which is denoted **ASYN**

Combinations of the above metrics are possible in general and RTA as well as constraint consideration is integrated within the GA's fitness function and the modeling of genes and chromosomes.

7.3.1 Time Slice Derivation for GPU WRR Scheduling

During the investigation of time slice lengths and weights for the WRR scheduling on the GPU, ticks of the `Detection` task were reduced to one-tenth of its original values, i.e., $C^{GPU,+}_{Dectection*} = C^{GPU,+}_{Dectection} \cdot \frac{1}{10}$, for enabling three tasks running on the GPU without exceeding the GPU's capacity, i.e., having a feasible task set of three tasks on the GPU. These three tasks are `Detection`, `Localization`, and `SFM`. Figure 7.13 shows average normalized task slack times derived from Eq. 7.3 along with different base time slices θ (x-axis) and weights, i.e., individual time slice lengths per task derived from the base time slice θ.

$$\overline{\zeta}' = \sum_i \left(\frac{T_i - R_i^+}{T_i}\right) \cdot \frac{1}{|\mathcal{T}_{GPU}|} \tag{7.3}$$

More precisely, Figure 7.13 compares equal weights, i.e., same θ time slice lengths, priority weights derived from periods, whereas the highest priority has the highest value π_i, utilization weights of Eq. 5.3 multiplied with the number of tasks mapped to the respective GPU, and the utilization weights only.

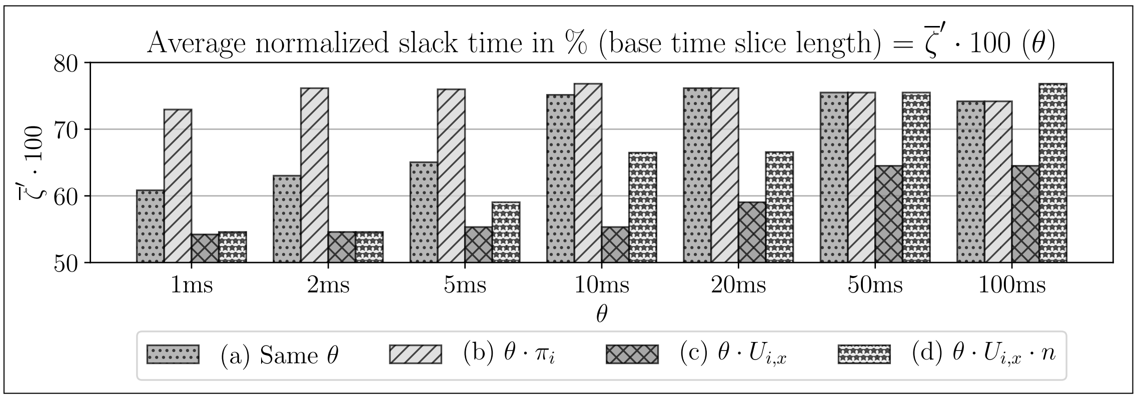


Figure 7.13: Influence of time slice derivation methods and different base time slice lengths (θ) on slack times: (a) equal, (b) priority, (c) utilization· n, and (d) utilization-based time slices

For all time slice base values except 100ms, the priority-based time slice derivation shows the highest (best) average normalized slack times. Using the task utilization results in the lowest, i.e., worst results. The equal time slice length method is outperformed by the priority-based one, although results converge with increasing base time slice lengths towards the same results. Finally, the utilization multiplied with the number of tasks method only outperforms others at 100ms due to time slice lengths start exceeding the actual task execution times and hence a near non-preemptive scheduling emerges. As a consequence, some tasks finish execution using only a single RRT and others require more turns. However, a valuable assessment can be made if not only average slack times are addressed, but also the standard deviation across all GPU task's slack times. Therefore, Figure 7.14 provides insights into each time slice weighting approach's standard deviation across all base time slice lengths. Even though equal and priority-based time

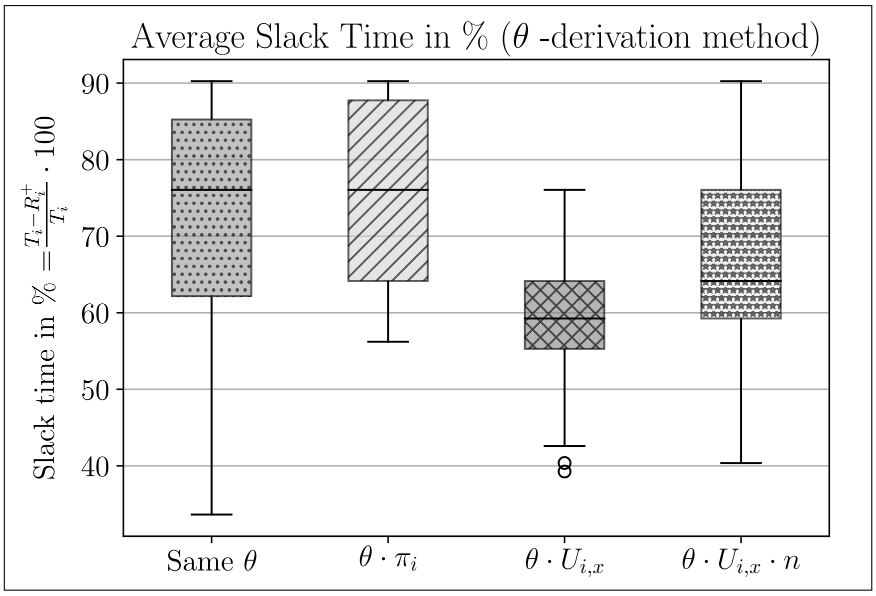


Figure 7.14: Average slack time deviations of different time slice derivation methods

slice methods show nearly the same 76% slack time medians, lower whisker as well as lower and upper quartiles are higher and hence better for the priority-based approach, making it the time slice derivation in favor of the others. This fact is also supported by the slack time bar plot in Figure 7.13, which also shows the best slack times except for the 100ms base times slice length. In general, it is not recommended to choose too large time slice lengths since the round robin fashion, i.e., sharing the computing resource across multiple entities continuously, reduces when time slices reach towards execution times or even periods. In fact, since the `SFM` task with a period of 66ms is mapped to the GPU in the measurements, choosing 100ms can potentially cause deadline misses, even though WRR is work conserving, because a higher priority task uses the entire time slice before `SFM`'s time slice is scheduled. The measurements shown in Figures 7.13 and 7.14 are only schedulable due to the mentioned assumption of reducing the `Detection` execution time and using rate monotonic priorities. It can be concluded that the priority-based time slice derivation method outperforms others presented here and is chosen consequently for the timing verification measurements of the next sections.

7.3.2 Task Chain Latency Analyses

This section applies the methodologies of Section 5.5 to the WATERS19 case study model. There are no explicit task chains given in the challenge model, but the following task chains can be derived based on the information flow of Figure 6.1. Abbreviations of each task chain are given in brackets aligned right in the following list.

$\gamma_1 = \{\text{Lane Detection} \rightarrow \text{Planner} \rightarrow \text{DASM}\}$ (LD-P-DA)

$\gamma_2 = \{\text{SFM} \rightarrow \text{Planner} \rightarrow \text{DASM}\}$ (SF-P-DA)

$\gamma_3 = \{\text{CAN polling} \rightarrow \text{Planner} \rightarrow \text{DASM}\}$ (CA-P-DA)

$\gamma_4 = \{\text{CAN polling} \rightarrow \text{EKF} \rightarrow \text{Planner} \rightarrow \text{DASM}\}$ (CA-EK-P-DA)

$\gamma_5 = \{\text{CAN polling} \rightarrow \text{Localization} \rightarrow \text{EKF} \rightarrow \text{Planner} \rightarrow \text{DASM}\}$ (CA-Lo-EK-P-DA)

$$\gamma_6 = \{\text{Lidar Grabber} \rightarrow \text{Localization} \rightarrow \text{EKF} \rightarrow \text{Planner} \rightarrow \text{DASM}\} \quad \text{(Li-Lo-EK-P-DA)}$$
$$\gamma_7 = \{\text{Lidar Grabber} \rightarrow \text{Planner} \rightarrow \text{DASM}\} \quad \text{(Li-P-DA)}$$
$$\gamma_8 = \{\text{Detection} \rightarrow \text{Planner} \rightarrow \text{DASM}\} \quad \text{(D-P-DA)}$$

Based on those task chains $\gamma_1 - \gamma_8$, various task chain latency values shown in Table 7.4 with corresponding equation references can be calculated. Results when applying those equations to the WATERS model across mappings EV–LBO are shown in Table 7.5.

Description	Notation	Equation
Worst-case task chain reaction delay implicit	ρ_ι^+	5.46
Best-case task chain reaction delay implicit	ρ_ι^-	5.44
Worst-case task chain reaction delay LET	ρ_λ^+	5.47
Best-case task chain reaction delay LET	ρ_λ^-	5.45
Worst-case task chain aging delay implicit	α_ι^+	5.48
Worst-case task chain aging delay LET	α_λ^+	5.49

Table 7.4: Task chain latency types overview, notation, and equation reference

Best-case aging delays are omitted in Table 7.4, since no reason could be found that such metrics can be of relevance. All measurements are based on asynchronous offloading except the 'SYN' approach shown in Table 7.5, which is further explained in the following Section 7.3.3. In general, task chain aging delays are much higher than reaction delays, which can also be seen in Figure H.2 due to considering two consecutive task chain instances instead of properties of a single instance only. All LET task chain delays are independent of response or execution times since only periods are taken into account. As a consequence, LET-based task chain delays are also independent of the mapping under the schedulability assumption and consequently outlined once, denoted as '*ALL*' for all mappings in Table 7.5. Although LET makes latency estimation much easier, delays are much longer than implicit results. It is observed that the load balancing approach results in the worst task chain delays. This supports the statement of memory operations, blocking, and contention significantly affecting response times and hence task chain latency, too. Even the early valid mapping outperforms the load balancing approach. In turn, the range distribution between best- and worst-case task chain delays is the lowest for the load balancing approach. Since the TCO approach's target is to minimize exactly the worst-case implicit task chain reaction sum, Table 7.5 shows the lowest values for TCO accordingly. In total, the worst-case task chain delays are approximately 10%, 25%, and 9% higher for the worst LBO result compared with the TCO results for $\rho_\iota^+, \rho_\iota^-$, and α_ι^+, respectively. The asynchronous offloading approach exceeds the synchronous one by 35.8% and TCO by ≈39%. As an intermediate summary, it can be derived that synchronous offloading has a disadvantageous effect for task chain latency delays and that valid task to PU mappings effects task chain latency delays by up to 25% for the WATERS19 model.

In addition to the task chain results of Table 7.5, the concrete task mappings, response times, s-, and pi-blocking delays are given along with periods and execution times for all tasks in Table 7.7, further extended towards contention, CE delays, accumulated label access costs, and normalized task response time in Table 7.6.

Task chain name	Worst-case implicit reaction delay ρ_ι^+							ρ_λ^+	ρ_λ^-	α_λ^+
	EV	RTSO	TCO	LBO	ASYN	SYN		ALL	ALL	ALL
LD-P-DA	75.37	84.76	84.36	75.37	99.92	99.92		106	86	167
SF-P-DA	135.88	122.19	131.65	148.98	126.30	126.30		238	152	299
CA-P-DA	31.14	31.55	26.91	31.14	32.45	32.45		50	30	55
CA-EK-P-DA	50.15	53.17	46.32	50.15	51.46	51.46		80	45	85
CA-Lo-EK-P-DA	754.46	758.41	697.57	742.85	715.96	1039.01		1680	845	1685
Li-Lo-EK-P-DA	823.93	822.63	759.58	812.47	768.24	1091.28		1733	883	1761
Li-P-DA	63.61	56.15	51.51	63.76	47.72	47.72		73	53	101
D-P-DA	439.54	357.46	406.66	505.62	414.50	575.50		640	420	835
SUM	2374.07	2286.33	2204.57	2430.35	2256.56	3063.65		4600	2514	4988

Task chain name	Best-case implicit reaction delay ρ_ι^-					Worst-case implicit age delay α_ι^+				
	EV	RTSO	TCO	LBO	SYN	EV	RTSO	TCO	LBO	SYN
LD-P-DA	55.37	64.76	64.36	55.37	79.92	136.37	145.76	145.36	136.37	160.92
SF-P-DA	49.88	36.19	45.65	62.98	40.30	196.88	183.19	192.65	209.98	187.30
CA-P-DA	16.14	16.55	11.91	16.14	17.45	36.14	36.55	31.91	36.14	37.45
CA-EK-P-DA	20.15	23.17	16.32	20.15	21.46	55.15	58.17	51.32	55.15	56.46
CA-Lo-EK-P-DA	309.46	313.41	252.57	297.85	270.96	759.46	763.41	702.57	747.85	720.96
Li-Lo-EK-P-DA	340.93	339.63	276.58	329.47	285.24	851.93	850.63	787.58	840.47	796.24
Li-P-DA	43.61	36.15	31.51	43.76	27.72	91.61	84.15	79.51	91.76	75.72
D-P-DA	219.54	137.46	186.66	285.62	194.50	634.54	552.46	601.66	700.62	609.50
SUM	1055.07	967.33	885.57	1111.35	937.56	2762.07	2674.33	2592.57	2818.35	2644.56

Table 7.5: Task chain latency analysis results in ms

In line with the challenge description [30], an offloading task is omitted (zero execution time), if the triggered task is executed on a CPU. All tasks meet their deadlines, whereas the worst normalized response time is surprisingly contained in the RTSO mapping for the Planner task. This is caused by the optimization goal targeting the response time sum, which indeed is the lowest for RTSO, instead of the mNRT metric. The local pi-blocking occurs scarce and yields low delays. This fact is due to the rather rare occasion of a task with lower priority runs on the same PU and accesses the same label(s) and due to some tasks accessing labels that are not shared at all (especially tasks that can potentially run on the GPU). For Table 7.7, Core0 and Core1 are Denver PUs and Core2–Core5 represent the ARM PUs in line with the naming of the WATERS19 Amalthea model.

	Contention				CE costs			
Task	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	-	-	-	-	-	-	-	-
LidarGrabber	8.7502	8.7502	8.7502	8.7502	-	-	-	-
DASM	0.009	0.009	0.0019	0.009	-	-	-	-
CANbuspolling	0.0005	0.0026	0.0026	0.0029	-	-	-	-
EKF	0.023	0.023	0.023	0.0259	-	-	-	-
Planner	4.4086	4.0078	0.7615	4.4086	-	-	-	-
PRESFMgpuPOST	-	2.8238	-	-	-	-	-	-
PRELocalizationgpuPOST	-	-	1.5981	-	-	-	-	-
PRELanedetectiongpuPOST	-	-	-	-	-	-	-	-
PREDetectiongpuPOST	12.0313	2.5781	-	12.0313	-	-	-	-
SFM	8.75	-	8.75	8.75	-	2.7309	-	-
Localization	0.7991	0.7991	-	0.7991	-	-	2.5413	-
Lanedetection	0.9376	0.9376	4.3756	0.9376	-	-	-	-
Detection	-	-	-	-	1.375	2.6402	2.2506	1.375

	Label Access Costs				$\frac{R_i^+}{T_i}$			
Task	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	-	-	-	-	0.7	0.5	0.732	0.524
LidarGrabber	1.0938	1.0938	1.0938	1.0938	0.844	0.606	0.606	0.849
DASM	0.0013	0.0013	0.0005	0.0013	0.262	0.262	0.21	0.262
CANbuspolling	0.0001	0.0003	0.0003	0.0003	0.04	0.04	0.04	0.04
EKF	0.0029	0.0029	0.0029	0.0029	0.267	0.442	0.294	0.267
Planner	0.4008	0.4008	0.1603	0.4008	0.962	0.989	0.697	0.962
PRESFMgpuPOST	1.8825	0.753	1.8825	1.8825	-	0.136	-	-
PRELocalizationgpuPOST	0.9401	0.9401	0.376	0.9401	-	-	0.028	-
PRELanedetectiongpuPOST	0.5001	1.2502	0.5001	0.5001	-	-	-	-
PREDetectiongpuPOST	1.7188	0.6875	0.6875	1.7188	0.479	0.031	0.071	0.809
SFM	1.2652	0.2108	1.2652	1.2652	0.517	0.167	0.517	0.716
Localization	0.376	0.376	0.1567	0.376	0.723	0.726	0.562	0.694
Lanedetection	0.5001	0.5001	1.2502	0.5001	0.6	0.737	0.801	0.6
Detection	0.2864	0.2864	0.2864	0.2864	0.54	0.575	0.805	0.54

Table 7.6: WATERS contention, CE costs, label access costs, and R_i^+/T_i results in ms

	Input Details							Mapping $M^P_{\tau_i}$						
Task	T_i	$C^+_{i,ARM}$	$C^-_{i,ARM}$	$C^+_{i,Denver}$	$C^-_{i,Denver}$	$C^+_{i,GPU}$	$C^-_{i,GPU}$	Given	EV	RTSO	TCO	LBO	(A)SYN	
OSOverhead	100	50	50	50	50	-	-	Core0	Core2	Core4	Core4	Core3	Core2	
LidarGrabber	33	13.66	10.16	10.87	9.79	-	-	Core1	Core4	Core2	Core3	Core5	Core1	
DASM	5	1.86	1.3	1.3	1.05	-	-	Core0	Core4	Core5	Core1	Core2	Core4	
CANbuspolling	10	0.6	0.4	0.6	0.4	-	-	Core0	Core0	Core3	Core4	Core3	Core4	
EKF	15	4.76	3.98	4.43	4.09	-	-	Core4	Core2	Core5	Core4	Core5	Core2	
Planner	15	13.24	9.62	12.44	9.54	-	-	Core3	Core5	Core3	Core0	Core4	Core5	
PRESFMgpuPOST	66	7.9	6.33	6.71	5.41	-	-	Core0	Core2	Core1	Core5	Core3	Core3	
PRELocalizationgpuPOST	400	17.64	7.34	14.52	6.12	-	-	Core0	Core2	Core2	Core1	Core5	Core4	
PRELanedetectiongpuPOST	66	8.23	6.79	7.63	6.08	-	-	Core5	Core1	Core5	Core0	Core1	Core1	
PREDetectiongpuPOST	200	4.71	4.01	4.09	3	-	-	Core5	Core2	Core0	Core0	Core5	Core3	
SFM	66	29.5	24.14	27.81	22.18	7.90	7.05	GP10B	Core3	GP10B	Core2	Core2	Core0	
Localization	400	387.42	366.52	294.81	276.71	124.00	117.00	GP10B	Core0	Core0	GP10B	Core1	GP10B	
Lanedetection	66	51.04	47.84	42.24	38.44	27.33	24.50	GP10B	Core1	Core1	Core5	Core0	Core0	
Detection	200	-	-	-	-	116.00	108.00	GP10B	GP10B	GP10B	GP10B	GP10B	GP10B	

	Worst-Case Response Times $R^+_{i,x}$						s-Blocking				pi-Blocking			
Task	EV	RTSO	TCO	LBO	ASYN	SYN	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	70.03	50	73.25	52.42	70.03	70.03	-	-	-	-	-	-	-	-
LidarGrabber	27.87	20	20	28.02	11.98	11.98	1.2813	0.3438	0.5156	0.3438	-	-	0.4688	8.7502
DASM	1.31	1.31	1.05	1.31	1.31	1.31	0.0006	0.0006	0.0003	0.0006	-	-	-	0.009
CANbuspolling	0.4	0.4	0.4	0.4	1.71	1.71	0.0013	0.0011	0.0004	0.0014	0.0001	0.0003	-	0.0005
EKF	4.01	6.63	4.41	4.01	4.01	4.01	0.0026	0.0053	0.002	0.0028	0.0003	-	0.0003	0.023
Planner	14.43	14.84	10.46	14.43	14.43	14.43	0.6865	0.3945	0.2154	0.6867	-	0.0938	-	4.4086
PRESFMgpuPOST	-	8.99	-	-	-	-	0.64	0.1129	0.6355	0.64	-	-	-	-
PRELocalizationgpuPOST	-	-	11.25	-	24.5	347.55	1.1269	0.1919	1.0189	0.1896	-	-	-	-
PRELanedetectiongpuPOST	-	-	-	-	-	-	0.0001	0.2501	0.6251	0.2501	0.25	-	-	-
PREDetectiongpuPOST	95.8	6.27	14.15	161.87	17.76	178.76	0.3776	0.3776	0.1432	0.3776	-	-	-	12.0313
SFM	34.14	11.05	34.14	47.24	24.56	24.56	1.89	0.7605	1.8855	1.89	-	-	-	8.75
Localization	289.31	290.23	225	277.69	225	225	1.8811	1.8814	1.3166	1.8814	-	-	-	0.7991
Lanedetection	39.63	48.62	52.85	39.63	64.18	64.18	0.0001	1.2502	0.5001	0.5001	-	-	-	0.9376
Detection	108	115.05	161	108	161	161	1.9531	0.9219	0.7813	1.9531	-	-	-	-

Table 7.7: WATERS mapping, response times, and blocking results in ms

Tables 7.7 and 7.6 refer to BCET, the adjusted sync model, i.e., task `SFM` being periodically executed every 66 instead of 33 ms (cf. Section 7.3.3), and corresponding mappings to form consistent measurements throughout this chapter. The appendix also provides measurements for WCET, the original model, and the various mapping optimization approaches at Section H.8.

Memory contention based on the given challenge equations [30] is comparably pessimistic and can cause high delays for memory-intensive tasks such as the offloading operators. However, these delays are unlikely to occur in a real scenario due to the GPU's dedicated CE and the rather low probability of concurrent access to the high amount of memory required by the GPU tasks. In most cases, the GPU tasks use this memory in a dedicated fashion. Thus, the memory contention calculation used here (cf. Eq. 5.25) is much less pessimistic than the one provided in [30] by considering the precise labels and access delays across PUs rather than assuming that every accessed cache line can be subject to contention by all PUs in the system. According to local pi-blocking, measured values are in the range of nanoseconds and microseconds, but the delays caused by global s-blocking go beyond a millisecond. Hence, Section 7.4 investigates to reduce such s-blocking times along with TDRR measurements.

7.3.3 Synchronous and Asynchronous Offloading

The process of a CPU task triggering a GPU task and actively waiting for its finalization describes synchronous offloading. Therewith, the triggering task occupies the PU significantly more due to active waiting such that its response time increases correspondingly. Since the WATERS2019 challenge model features already highly utilized PUs, setting the synchronous offloading to `true` is infeasible with the given model since no mapping can be found, which is free from exceeding a PU's capacity across all PUs. Especially GPU resources are limited because the task `Detection` must execute on the GPU, derived from ticks being only modeled explicitly for the GPU, but not for any other PU. However, when considering best-case execution times and increasing the periodic activation of the task '`PRE_SFM_gpu_POST`' from 33 to 66ms, a valid result, that guarantees all task deadlines being met, can be found for the response time sum optimization indicated as SYN and the diagonal lines in Figure 7.15. To be able to compare results comprehensively, this model adaptation is used for measurements in the following if not indicated otherwise. Having two tasks `Detection` and `Localization` mapped to the GPU in this scenario, their triggering task's response times increase from approximately 17.76 to 178.76ms and from 24.50 to 347.55ms respectively (cf. Table 7.7), which is approximately 10 and 14.2 times of their asynchronous response times. This change results in a total response time sum increase from 620.47 to 1104.51ms, which is a factor of ≈ 1.78 compared with the asynchronous response time sum. Consequently, latency and RTA measurements consider asynchronous task offloading in the following if not mentioned otherwise.

Figure 7.15 presents task chain latency results of the eight outlined task chains according to the EV, RTSO, TCO, LBO, and (A)SYN mapping results for best-case execution times and the increased activation for task '`PRE_SFM_gpu_POST`'. The scenario is chosen as it provides the only schedulable synchronous result, which is presented as SYN. ASYN denotes the same mapping, but under asynchronous offloading. Additionally, Figure 7.15 shows the summed up delays of all task chains for each mapping in the lower right bar chart with additional information about the sum of best-case task chain reaction delays for implicit

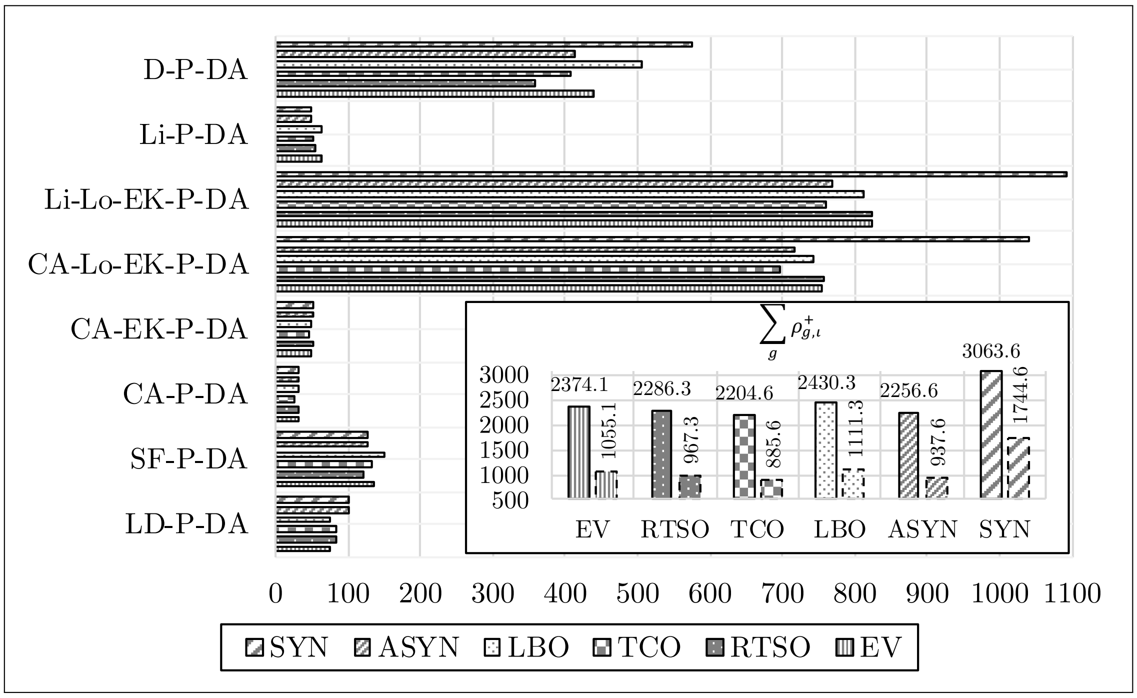
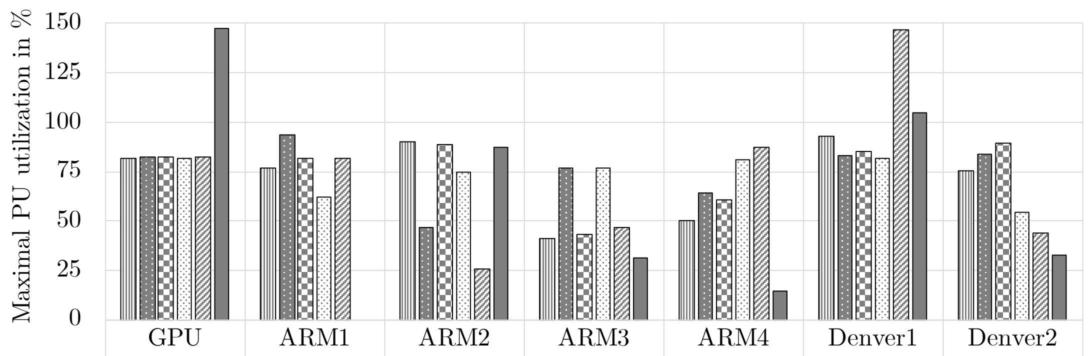


Figure 7.15: Implicit worst-case task chain reaction delays for BCET and different mappings in *ms*, and additional bar chart (lower right corner) of the same values accumulated, and compared with implicit best-case task chain reaction delays (bars with dashed lines)

communication in dashed bars. Analyzing any mapping for synchronous offloading results in violating at least one deadline except for the SYN mapping due to the significant increase of CPU cycles caused by actively waiting for the GPU task to finish. Since only three task chains include tasks being executed on the GPU, namely D-P-DA, Li-Lo-EK-P-DA, and CA-Lo-EK-P-DA, SYN, and ASYN solutions are the same for all other task chains in Table 7.5. Deviations between task chain delays are mainly caused by contention, blocking, and scheduling interference, but obviously, the highest differences are within the synchronous (SYN) and asynchronous (all other mappings) offloading mechanisms. Task chains always include the offloading (`Pre...Post`) tasks, which are required to be executed if a task chain's task is mapped to the GPU. By measuring the same mapping (SYN and ASYN) for both offloading mechanisms, an increase of the implicit worst-case task chain reaction delay of $\frac{3063.6}{2256.6} - 1 \approx 35.8\%$ is derived.

Additionally, the PU utilization results across all GA mapping results with either different or no optimization focus (only EV does not have an optimization goal), are shown in Figure 7.16 and Figure 7.17. It must be noted here that the given mapping is slightly ambiguous since task schedulers for `PRE_SFM_gpu_POST` and `PRE_Localization_gpu_POST` do not have a distinct affinity but instead, an affinity to either of the *Denver* CPUs. The following measurements assume that both of the tasks mentioned above are allocated to the Denver1 CPU. Allocating those to Denver2 could mitigate the utilization of Denver1 beyond 100%, as shown in Figure 7.16. However, the actual infeasibility issue of the GPU utilization remains. Furthermore, the published AMALTHEA model provides several task allocations with a distinct PU affinity, which is used to define the task to PU mapping, but the scheduler they are assigned to is responsible for other PUs despite their PU affinity. The latter *distraction* is though ignored in the following under the assumption that task allocation affinity primarily defines the task to PU allocation, which is also in line with the AMALTHEA model documentation in [45].



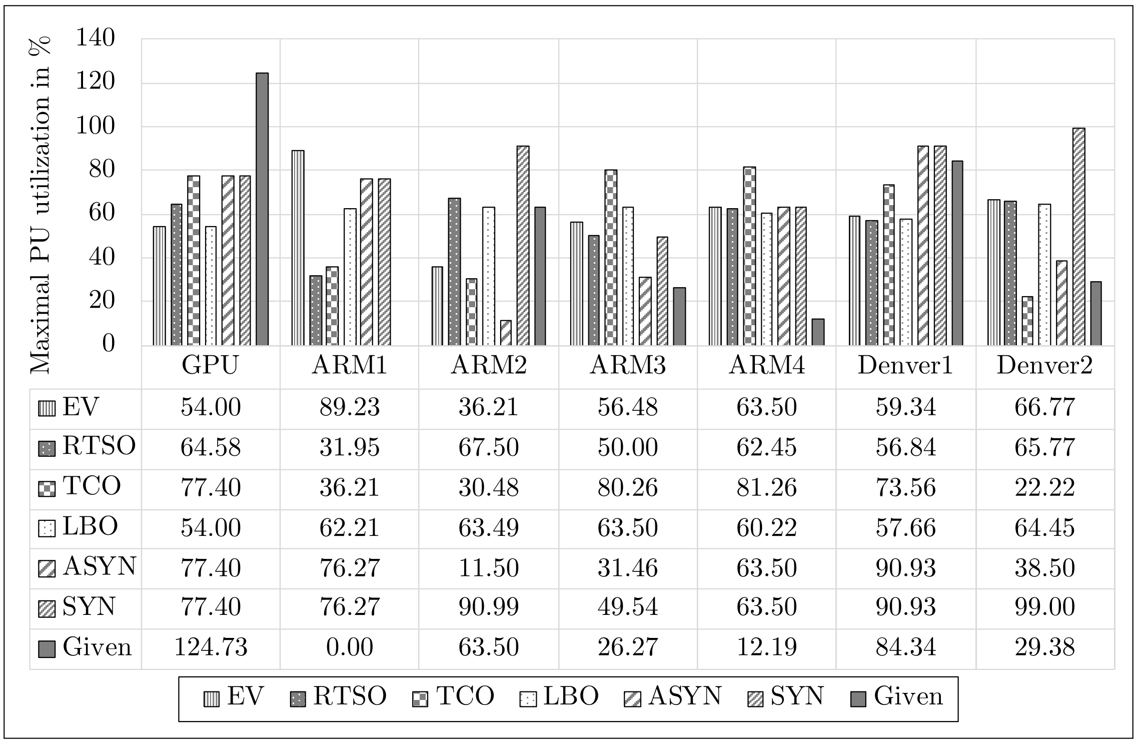
	GPU	ARM1	ARM2	ARM3	ARM4	Denver1	Denver2
▥ EV	81.70	76.57	89.83	40.98	50.00	93.03	75.36
▩ RTSO	82.80	93.76	46.98	76.57	64.44	83.44	84.13
▦ TCO	82.80	81.41	88.92	43.34	60.91	84.99	89.43
⬚ LBO	81.70	62.35	74.61	76.57	81.01	82.08	54.78
▨ SYN	82.80	81.41	26.07	46.72	87.40	146.79	44.04
■ Given	147.50	0.00	87.40	31.41	14.71	105.03	32.60

▥ EV ▩ RTSO ▦ TCO ⬚ LBO ▨ SYN ■ Given

Figure 7.16: Maximal PU utilization values $\hat{U}^P$ for EV, RTSO, LBO, TCO, and SYN GA solutions as well as the given mapping under WCET

For comparison purposes with the previous measurements, but also reasoning the more critical worst-case scenarios, both utilization diagrams are provided here according to (i) the synchronous offloading model based on BCET and an `SFM` task period increase to $66ms$ in Figure 7.17 on the one hand, and (ii) the original model based on WCET in Figure 7.16 on the other hand. According to the given mapping model, infeasibility is revealed for both cases due to the GPU being utilized by 124.73% and 147.5%, respectively. Furthermore, task instructions, contention, and blocking delays result in 105% utilization of the Denver1 PU so that deadlines are violated for tasks running on this PU for the given model mapping. Except for the given and SYN mappings, of which the latter is only (hardly, cf. 99% at Denver2 PU) feasible for BCET as shown in Figure 7.17, Figure 7.16 shows feasible utilization values (computed via Eq. 5.9), which are verified for schedulability via response times being all lower than their deadlines as shown in Table 7.7.

In addition to Figures 7.16 and 7.17, Figure 7.18 presents measurements of different metrics along with the three optimized mappings and the early valid mapping. Due to infeasibility, some of those metrics could not be calculated for the given mapping model, which is hence omitted in Figure 7.18. Interestingly, the load balancing solution shows the worst results for most metrics except the CE latency sum and LBO's minimization goal, i.e., maximal PU utilization. In contrast, the response time minimization approach forms most inner values in the radar chart 7.18 as of metrics response time sum, task contention sum, label access costs sum, and s-blocking sum, while the maximal PU utilization is



	GPU	ARM1	ARM2	ARM3	ARM4	Denver1	Denver2
▥ EV	54.00	89.23	36.21	56.48	63.50	59.34	66.77
▩ RTSO	64.58	31.95	67.50	50.00	62.45	56.84	65.77
▦ TCO	77.40	36.21	30.48	80.26	81.26	73.56	22.22
⬚ LBO	54.00	62.21	63.49	63.50	60.22	57.66	64.45
▨ ASYN	77.40	76.27	11.50	31.46	63.50	90.93	38.50
▨ SYN	77.40	76.27	90.99	49.54	63.50	90.93	99.00
■ Given	124.73	0.00	63.50	26.27	12.19	84.34	29.38

Figure 7.17: Maximal PU utilization values $\hat{U}^P$ for EV, RTSO, LBO, TCO, SYN (AYSN) GA solutions as well as the given mapping under BCET consideration

close to the LBO solution on the second-best position. By these outcomes, it is derived that minimizing response times is a more valuable optimization goal compared with load balancing optimization.

Moreover, two metrics stick out, namely the low pi-blocking for the task chain optimization and the low CE latency for both the EV and LBO mappings. The former is caused by only two tasks in total that share few labels with lower priority tasks on the same PU, namely `CANBusPolling`, which shares 1kB label `Vehicle_status_host` with the lower priority `EKF` task on the same PU and pi-blocking according to Eq. 5.23 and Eq. 5.31 is $\frac{\lceil \frac{1000}{64} \rceil \cdot 40}{2 \cdot 10^9} = 320ns$ and task `Planner`, which shares a 750kB label `Bounding_box_host` with task `Pre_Detection_gpu_Post` leading to another pi-blocking delay of $\frac{\lceil \frac{750000}{64} \rceil \cdot 16}{2 \cdot 10^9} = 93.8us$ so that the sum $\approx 94.1us$ is much lower compared with label sizes and causing pi-blocking at other tasks of other mapping solutions. The low CE latency for EV and LBO solutions is due to only offloading the `Detection` task to the GPU (cf. Table 7.7). The configuration for measurements of Figures 7.16 and 7.18 are considering BCET, asynchronous offloading, only written labels for the CE and offloaded task, and $1ms \cdot \pi_i$ for the GPU time slice derivation. The latter is outlined in the following Section 7.3.1. Following the optimization goals, TCO shows the lowest task chain latency value, RTSO the lowest response time sum, and LBO the lowest maximal PU utilization.

It can be concluded that timing verification and task mapping optimization are complex processes for designing automotive systems with lots of possibilities, metrics, and properties to both optimize or verify via constraint satisfaction. It is usually a demanding process to identify a single best solution, and various metrics must be taken into account for

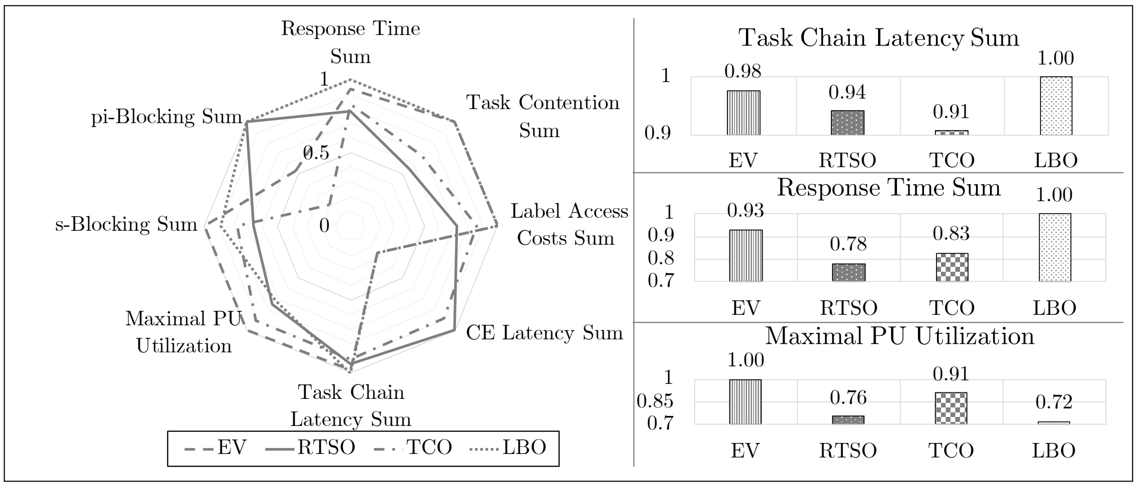


Figure 7.18: Radar and bar charts for various metric measurements in % for four different mapping results

retrieving not only valid but also constraints and requirements preserving results for multiple optimization goals. For task chain delays, it is observed that $\rho_\iota^+ < \alpha_\iota^+ < \rho_\lambda^+ < \alpha_\lambda^+$ with no standard deviation for LET, and standard deviations $\overline{\sigma}_{\rho_\iota^+} = 8.08\%$, $\overline{\sigma}_{\rho_\iota^-} = 15.34\%$, and $\overline{\sigma}_{\alpha_\iota^+} = 5,64\%$. The synchronous task to GPU offloading has shown much worse results than asynchronous offloading, even given the asynchronous offloading penalty when not actively waiting for an offloaded task's result. Measurements provided in this Section show that optimizing response times can be more valuable than balancing load only, since the vast amount of constraints and shared resource coherencies significantly interfere with execution and response times, especially in the heterogeneous multi-PU environment.

7.4 TDRR Analyses

This section presents results obtained by applying the TDRR approach of Section 5.8 to the case study models of Chapter 6. Table 7.8 gives the measured metrics, but first, the approach is exemplary outlined with the Democar model, as it provides appropriate model properties to revise TDRR's intend.

The Democar's original task configuration comprises three tasks running periodically every 5, 10, and 20 ms. This task set results in a total of 26 labels used by more than one task. Based on these labels, two release delta situations are identified, namely `Task_10ms` $\rightarrow$ `Task_20ms` and `Task_10ms` $\rightarrow$ `Task_5ms`, that potentially cause busy waiting under the assumption that these tasks are mapped to different PUs. `Task_20ms` $\rightarrow$ `Task_10ms` does not result in busy waiting, since the conflicting access interval at `Task_20ms` is located at its first and only runnable so that executing `Task_20ms` before `Task_10ms` does not affect the latter. `Task_20ms` and `Task_5ms` do not share any labels. For the two delta situations, four and 13 conflict intervals are found respectively, which can be merged into a total of five intervals, to which new runnable orders can be calculated due to the interval merging process outlined in Section 5.8.2 and Algorithm 5.2. Using these runnable orders in corresponding situations, the time spent for busy waiting can be reduced by 100% so that the worst-case task execution time can be reduced by maximal $\approx 15.4\%$ for $\delta_{\texttt{Task\_10ms} \rightarrow \texttt{Task\_5ms}}$ as shown in the Gantt chart of Figure 7.19. The calculated runnable

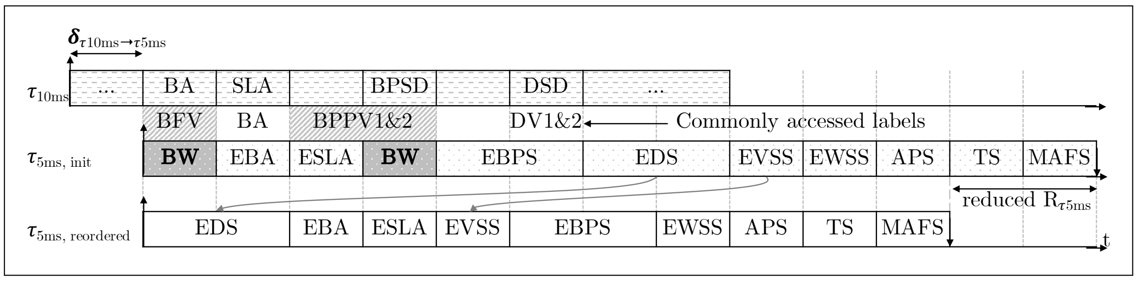


Figure 7.19: TDRR applied to the Democar model [265], case $\delta_{t_{10ms \to 5ms}} = 64 \cdot 10^3$, $\mathcal{RO}_{t_{5ms}}$ resulting in a busy waiting reduction of $16 \cdot 10^3$ instructions

orders based on Algorithm 5.3 provide conflict-free executions across all task release delta values in case of the Democar model. If, for example, `Task_10ms` is released $792 \cdot 10^3$ instructions before releasing `Task_5ms`, busy waiting occurs at `Task_5ms` for $88 \cdot 10^3$ instructions due to runnable `BrakeActuator` having *locked* label `BrakeForceVoltage`. While a busy waiting of the first runnable would as well shift all succeeding runnables, there can still occur further busy waiting (cf. Figure 7.19, second BW block). The reordering of runnables achieves reducing the execution time of task `Task_5ms` by $\frac{16 \cdot 10^3}{(88+16) \cdot 10^3} \approx 15.4\%$ such that no busy waiting occurs at all. This case is shown in Figure 7.19, which uses abbreviations in the boxes as acronyms for runnables and accessed label names, e.g., EBA=EcuBrakeActuator of the Democar model. Across all conflicting intervals, the five calculated $\mathcal{RO}$s eliminate busy waiting entirely.

The amount of $\mathcal{RO}$ calculation increases drastically as soon as conflict intervals are nested among multiple tasks, which is the case for the AIM model. For such cases, different task release delta values have to be combined as provided by the number of merged conflict intervals in Table 7.8. This combination depends on each runnable's length within the task being subject to runnable reordering, the number of runnables within the task, and their dependencies, i.e., the runnable DAG. Busy waiting reduction could not be calculated for the WATERS model due to tasks mostly running a single runnable such that no reordering and hence no busy waiting reduction is possible. Table 7.8 provides TDRR metrics of all case study models and values for the maximal execution time reduction for conflicting tasks based on Eq. 5.96. Results of Table 7.8 make use merging conflict intervals, given that observed merged interval lengths are small enough to be *covered* by reordering runnables. Especially the amount of independent runnables helps TDRR to avoid conflicts. Due to the AIM model having an immense amount of inter runnable dependencies, TDRR is further configured to allow cutting edges in a runnable DAG to be able to merge conflicts and still find appropriate runnable orders that cover replacing conflicting runnables of merged intervals.

The TDRR concept also assumes that a task's runnable order is based on an initial task configuration in form of a runnable sequence. However, (I) a task may have already been released upon a different permutation due to preliminary conflict, i.e., a conflict of at least two prior released tasks. In such situations, the predicted release delta conflicts do not match actual busy waiting periods due to shared resource access happening at different points in time than expected as a prior task was already reordered. However, flagging predecessor tasks running either in initial runnable sequence or in TDRR mode could avoid these situations. Another situation, which has not been taken into account, is (II) the interleaving of delta conflict intervals of multiple predecessor tasks. This interleaving

Model	**Number of access conflicts $\lvert ci^p \rvert$**	**Number of conflict intervals $\lvert ci^\delta \rvert$**	**Number of merged conflict intervals**	**Number of runnable orders $\lvert \mathcal{RO} \rvert$**	**Maximal task exec. time reduction in %**
Democar	17	12	12	5	15.385
FMTV	1918	2479	715	1203	3.191
AIM	15160	183461	5838	9322	15.091
MG1	5636	5746	247	5389	0.537
MG2	5031	5077	72	4959	3.624
MG3	13959	14166	529	13430	2.026
WATERS19	23	23	0	23	-

Table 7.8: TDRR results for case study models

happens in the unlikely situation of multiple predecessor tasks, running each on different PUs and not on the PU the task subject to TDRR is scheduled on, were released within a delta conflict interval before the task subject to TDRR. This situation could be tackled by prioritizing the runnable reordering, which results in a potentially higher busy waiting reduction. A throughout analysis of both situations (I) and (II) is omitted since no simulation or hardware application was conducted to estimate the probability of such situations and their obstructive effect. Based on the proposed solution concepts, busy waiting may still exist for (I) shortly subsequent TDRR delta conflicts or (II) multiple predecessor tasks being both released within delta conflict intervals. Nonetheless, these situations are expected to occur on rare occasions and still result in not entirely eliminated but still reduced execution times in the general case. TDRR's major disadvantage is (III) the assumption of a predecessor task's static runnable order, which can potentially be interrupted due to preemption, such that conflict intervals deviate from the estimated intervals. In this preemption case, a new runnable order may still result in busy waiting periods. In other words, TDRR works well for non-preemptive tasks and could be improved for preemptive scheduling. An advanced simulation of TDRR, the investigation of an online-based runnable reordering, and its adjustment towards preemptive tasks can be investigated with future work as outlined in Section 8.2.

To conclude, using TDRR to potentially execute tasks with different off-line calculated runnable orders to reduce busy waiting is successfully applied to Amalthea models. TDRR is the first approach towards having varying runnable orders within a task in Autosar while preserving precedence constraints and not accompanying significant re-validation efforts for retaining the system's software behavior. While some challenges remain to be addressed in future work, execution times can be improved by up to $\approx 15\%$, which yields better overall system performance.

7.5 Label Mapping

This section provides results of solving the data to memory mapping problem outlined in Section 5.7. Since case study models of Chapter 6 except FMTV do not provide `AccessElement` entities and no or very few memories, some model extensions were conducted as follows. The FMTV model is the only model providing all information needed to perform the label mapping. The WATERS model contains a single global memory and cache memories for all PUs but only `AccessElements` for the GRAM are available so that access delays to caches are generated. This generation is based on [31] such that accesses to global memory, omitted for the WATERS model but used for other models, take nine instructions, and cache accesses a single or eight instruction depending on whether the cache is local or remote cache. The generated models MG1–MG3 as well as the Democar and AIM models are extended to provide this memory access structure, such that a single GRAM is added, to which the access latency is always nine instructions. Local LRAM memories for every PU, to which the corresponding PU takes one access instruction, require eight instructions when being accessed by other PUs. The memory sizes are generated so that the GRAM memory can host 60% of accumulated required memory and all LRAMs can host $\frac{50\%\text{ total memory required}}{u}$. As a consequence, LRAMs memories (or Caches in case of the WATERS model) are expected to be utilized by nearly 100% and the GRAM hosts labels either rarely accessed in total or being of smaller size. The GA's fitness is implemented using Eq. 5.72.

The following Figure 7.20 shows a typical fitness calculation over a logarithmic time scale of base ten. With the very high amount of labels for AIM and FMTV models, the GA's engine configuration, i.e., population size, number of survivors, offset selection, and altering has a crucial effect on fitness improvement over time. For the FMTV model, a population size of 500, 10 survivors, a Truncation-based offspring selector, and a multi-point crossover altering with 90% probability at 500 positions, is found useful. This configuration provides

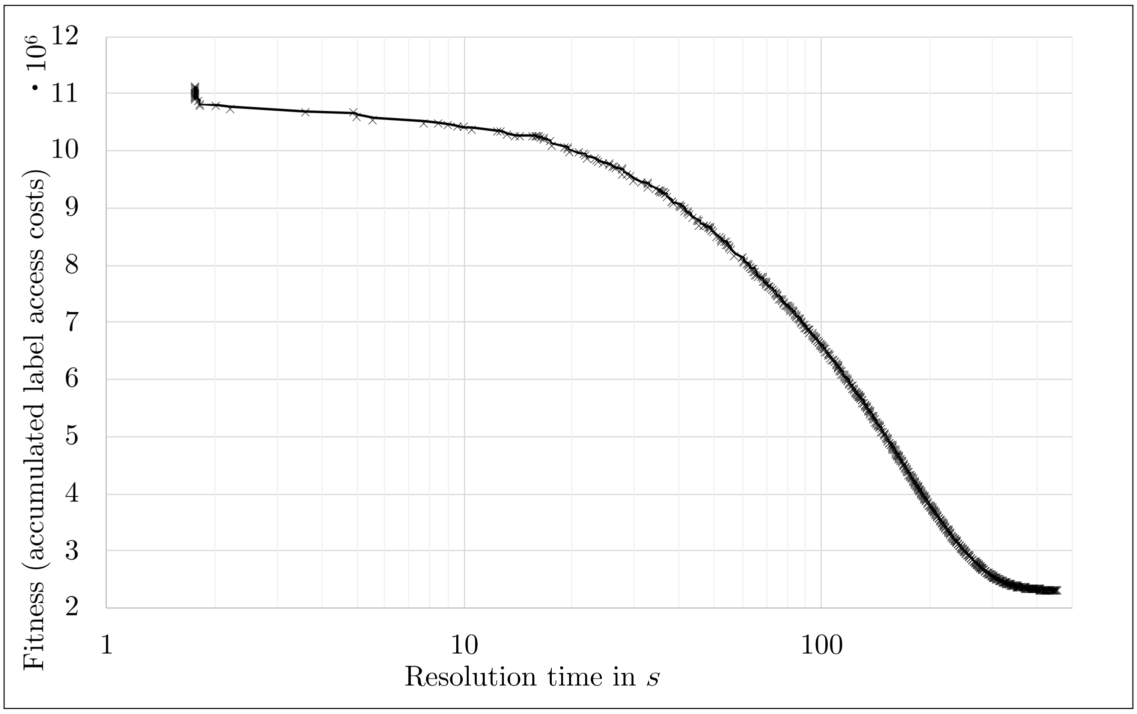


Figure 7.20: FMTV label mapping fitness value (t_{reso})

an improvement of the given FMTV label mapping access costs of 7288113 ticks after ≈ 81 seconds and to a final near-optimal fitness of 2320814 ticks after ≈ 8 minutes resolution time[60]. For the accumulated access cost metric, this result forms a reduction of 68.2% to

[60]The fitness of 2320814 did not improve over many generations so that the given result is argued to be near-optimal considering that the solution space was not investigated completely.

the original label mapping cost.

Finally, Table 7.9 shows various metrics relevant to the label mapping measurements across all case study models. Notation-wise, R-LRAM represents the remote LRAM from a different PU. For the WATERS model, specific LA delays are given as 5|40|16|1|8, which relate to GPU → GRAM, ARM → GRAM, Denver → GRAM, local cache, and remote cache.

Model	**Memory sizes**	**LA costs for GRAM only mapping**	**Final Fitness**	**Total Label Size Sum in Bits**	**LA costs per Byte GRAM| LRAM| R-LRAM**	**LA Costs Reduc. cmp. to GRAM mapping in %**
Democar	generated	136350	67650	824	9|1|8	50.39
FMTV	given	113409832	2320814	218904	9|1|8	97.95
AIM	generated	51062013	10002572	750864	9|1|8	80.41
WATERS	partially given	78824	9257	1272488192	(5|40|16)|1|8	88.26
MG1	generated	2335871	999759	1194816	9|1|8	57.2
MG2	generated	4630007	1211921	757976	9|1|8	73.82
MG3	generated	1181268	381107	608048	9|1|8	67.74

Table 7.9: Label mapping results for case study models

Due to the high availability of LRAM for the FMTV and WATERS model, comparably high access latency reductions can be achieved compared to mapping all labels to global memory, yielding in a reduction by 97.95% and 88.26%, respectively. For all other models, reduction percentages are still above 50%, which emphasizes the need for analyzing data to mapping for an efficient task execution yielding relatively low memory access delays.

The label mapping is a crucial process in timing verification. Compared with using global memory only, measurements across the case study models show that effective label mapping can reduce access times by up to 97.95%. However, given that the amount of labels goes beyond 46000 for the AIM model, careful analysis of the GA's configuration is required to efficiently and effectively traverse the solution space. Focusing on only one of the latter properties results in either a lousy convergence to a local optimum or an inappropriate increase in resolution time.

7.6 Implementation Remarks

Partitioning approaches of this thesis have been submitted open-source as contributions to the App4mc[5] platform. The timing analysis tools as well as label and task mapping approaches implemented along with this thesis are likely to be included in App4mc in the future.

When comparing different DSEs in Section 7.2, their configuration and implementation is crucial especially when comparing resolution time. Implementing constraints along with the CP solver, for instance, can be done in various ways. For example, `u[x].eq(c_ips[x].div(puCap[x] / uRes)).post()`, which seems to be a regular constraint for setting a division across parameters to a result variable, runs a magnitude slower than `u[x] = c_ips[x].div(puCap[x] / uRes).intVar()` . Moreover, the variety of model details, execution situations, people involved, i.e., programming styles and skills, and especially the vast amount of possible measurement configurations necessitate to keep track of version, configuration, and environment data for recording reasonable, reproducible, and valuable measurements.

Apart from the partitioning (Eclipse) plugin, more than 27 thousand lines of Java code are part of activities this thesis involves. The seven case study AMALTHEA models comprise $\approx$ 1.7 million lines of XML, which increases significantly, given that various adjustments and results are saved in separate AMALTHEA models.

8 Conclusions and Outlook

This chapter concludes the contributions and results of this thesis addressing challenges in the context of modern, highly constrained, distributed, heterogeneous, mixed-critical, embedded, real-time systems within the automotive industry.

Firstly, runnable **partitioning** and task mapping approaches and their benefits in developing automotive multi-PU systems are provided in Chapter 4 and 5, respectively. Different analyses, such as the cycle elimination, activation consideration, precedence constraints, or the model-based design, address various demands of the automotive industry and form a necessary phase during the exploration of parallelism based on runnables forming the atomic program parts in AUTOSAR. By using the interfaces of the AMALTHEA model, AUTOSAR compliant applications can be automatically split into concurrently executing tasks and distributed across multiple PUs. Therefore, CP-PC, ESSP, and CPP techniques for runnable WDAGs are used to form tasks either based on a predefined task number or automatically via the critical path. The evaluations of Section 7.1 show that CP-PC is the most effective approach according to speedup, schedule length / span, and slackness metrics.

Secondly, ILP, GA, DFG, and various CP-based methods are presented and compared for the **task to PU mapping** problem. The methods support finding (near) optimal task to PU allocations towards various goals such as response times, load balancing, or task chain latency. Therefore, the approaches include considering various outlined constraints according to, e.g., affinities or broader timing constraints such as deadlines[61]. Solutions are presented towards necessary response time analyses and **timing verification** challenges for high-performance automotive systems. Not only the formal outline of CPU and GPU response time analyses are provided, but also the application to AUTOSAR compliant AMALTHEA models. The analyses cover different scheduling paradigms (FPPS, RMS, FPMPS, TX2RS, and WRR), contention models, memory access types and delays, offloading patterns (synchronous vs. asynchronous), task chain latency as well as locking, queuing, and blocking delays. Results are presented along with applying the approaches to seven case study AMALTHEA models of Chapter 7 and further comparing results obtained from different configurations and optimization goals. The WRR scheduling has been investigated towards four different time slice derivation methods, of which the base time

[61] An overview of constraints is given in Table 5.9.

slice multiplied with the task priority derivation method has shown lower response times on average. Mapping results involving CPU-GPU timing verification are calculated primarily via GAs due to scaling better than ILP, CP, or other investigated heuristics. Results obtained from an advanced CPU-GPU mapping are also presented along with various configurations such as BCET or WCET consideration, synchronous or asynchronous GPU offloading, different time slice derivation methods, communication paradigms, task chain delays, and various metrics such as pi- and s-blocking, contention, CE delays, PU utilization and label access costs. Measurements are based on four primary task to PU mapping results constituted by an early valid (EV), response time optimized (RTSO), task chain optimized (TCO), and load balancing optimized (LBO) solution. For the WATERS model, each of the calculated mappings outperforms the given mapping for most metrics. In general, optimizing response times is identified as being a good trade-off across various metrics.

Thirdly, the **TDRR** approach and its algorithms are presented and applied to the same seven AMALTHEA case study models. Even though several assumptions are necessary and open issues remain unsolved, first results show reductions of busy waiting delays by 100% resulting in potential WCET reduction of up to 15%.

Fourthly, the **data to memory allocation** problem is addressed and solved via an advanced GA covering affinity constraints, many-fold heterogeneous access delays, numbers, rates, and types, as well as label sizes. Compared with allocating all data to GRAM, accumulated access delays are reduced by up to 97.95% and on average by 73.7% across the seven AMALTHEA case study models.

With APP4MC being the major open-source tool for AUTOSAR-based systems in the automotive industry, increasing research and industry interest based on citations and number of APP4MC downloads, and continuous evolution of APP4MC as a mature Eclipse project, the use of AMALTHEA and relevance of tooling provided in this dissertation are expected to grow with vehicle progression towards autonomous driving and beyond.

8.1 Summary

For a short and concise overview, two following lists summarize contributions and observed results. First of all, AMALTHEA-based approaches are given and evaluated for solving

1. the partitioning problem of allocating runnables to tasks,
2. the mapping problem of allocating tasks to PUs,
3. the mapping problem of allocating data to memory,
4. the disadvantageous behavior of busy waiting delays caused by AUTOSAR spinlocks,
5. complex timing verification considering
    (a) RTA for FPPS along with either RMS, offsets-based FPPS, or arbitrary deadlines, FPMPS, and WRR,
    (b) blocking analysis,
    (c) contention analysis,
    (d) task chain latency analyses,

(e) GPU-CPU interactions (synchronous and asynchronous offloading, CE queuing and blocking analysis),

(f) network delays (based on CAN messages), and

(g) hardware heterogeneities.

Challenges are revised with recent research, outlined with corresponding mathematical notations, and approached with various DSE heuristics as well as different case study models. Along with those case study models, evaluations show that:

I) the vast amount of constraints and requirements necessitates the use of appropriate DSE heuristics and solvers,

II) GAs form a generically good approach to large intractable problems,

III) CP is a great paradigm to naturally and flexibly approach typical automotive timing and allocation constraints,

IV) timing verification must be approached across various metrics (cf. Figure 7.18) to avoid local optima as well as major system performance bottlenecks during DSE,

V) individual challenges, i.e., challenges I–V from above, can be approached in isolation, but the immense complexity when combining those imposes significant scalability issues,

VI) asynchronous GPU task offloading results in better response times compared with synchronous offloading in most cases,

VII) mapping data across all system memories can decrease accumulated access times by up to $\approx 97.95\%$ compared with GRAM-only mapping,

VIII) TDRR can reduce worst-case task execution times by up to $\approx 15.39\%$,

IX) using task priorities for weighting time slice lengths for WRR mostly dominates equal or utilization-based approaches, and

X) CP-PC partitioning based on CP outperforms the greedy ESSP heuristic.

Statements VI–X support the thesis statement (cf. Section 1.1) by each addressing optimization along with data to memory mapping, GPU task offloading, execution time reduction, GPU timing verification, and runnable partitioning, respectively. Statements I–IV address DSE as well as timing verification and align to the thesis statement via **concluding** that CP (III) and GA-based (II) DSE approaches presented in this thesis form effective and necessary methodologies for modern automotive systems. With the contributions 1–5, this thesis addresses the required design space exploration, optimization, and timing verification approaches in order to account for, e.g., hardware accelerators, GPUs, different function domains, broader network connectivity, task chain latency, or specific communication paradigms (cf. Section 1.1).

8.2 Ongoing and Future Work

During the various implementation and evaluation activities of this dissertation, several assumptions are made that can be revised, and ideas unveiled that can be addressed in future work. Beginning with the **partitioning**, typically used, e.g., in [126, 271], metrics such as BF, WF, or FF can be implemented and compared with ESSP, CPP, or CP-PC, even though result quality is expected to be worst for either BF, WF, or FF. A promising addition to CPP, CP-PC, and ESSP is MLP, which is based on [192, 193], and can be used to, e.g., avoid comparably long tasks caused by data progression through many sequential dependencies. As stated in [90], MLP includes the *coarsening* of graphs through selecting contraction edges based on weights and a predefined threshold, to which the edge's source and target entities are merged. This approach allows identifying strongly connected entities and hence cut graphs based on a predefined threshold. These cuts are supposed to result in balanced subgraphs and relatively low inter-subgraph communication. However, since decomposing (cutting) edges is supposed to be kept as low as possible, it is only used along with the cycle decomposition in this work, but MLP could extend existing work in the future to lower task lengths.

Along with **timing verification**, a valuable investigation is the application of TDRR and CPU-GPU RTA all along with various metric measurements to either a simulation tool or an actual hardware platform combined with a tracing tool to validate analytical results on the one hand and further explore the distribution between BCRTs and WCRTs or average response times on the other hand. This could also be accompanied with, e.g., the PARSEC Benchmark suite outlined in [272] to run different programs stressing memory and the PUs, or to derive further models in addition to the case study models of Chapter 6. Such application could be, for instance, valuable for the AIM model, since actual software implementation could be used for verifying measurement results presented with analyses across Chapter 7. This topic was problematic during the course of this research since real-world applications are subject to strict IP policies and the implementation of tasks executing runnables with varying runnable orders further requires not only knowledge about the implemented software, but also adaptation of, e.g., the scheduler, which can be a complex process given that most of the AUTOSAR BSW is generated using verified and commercial tools. Furthermore, hybrid SA/CP/GA could be investigated to further improve efficiency and well known scheduling approaches such as EDF or resource protocols such as MSRP could be further integrated. In terms of CPU-GPU RTA, a combination of WRR and TX2RS, and the consideration of cooperative or non-preemptive tasks on the GPU are open challenges worth for being investigated and approached in the future. As mentioned in the corresponding section, TX2RS can also be investigated for worst-case run queue orders to achieve more accurate response time bounds, or for hyper-periodic kernel sets using priority-based run-queue ordering. In fact, TX2RS RTA could be approached with the help of CP similar to CP-PC but with varying task heights based on a kernel's block size to overcome the current algorithms greedy basis.

Three major limitations to **TDRR** are already discussed in Section 7.4 and indicated by (I)–(III). TDRR could further be investigated for using data conflict graphs [273] or runnable interaction graphs [274] for calculating $\mathcal{RO}$s. Additionally, determining overheads caused by $\mathcal{RO}$ look-ups during execution time and the memory required to store $\mathcal{RO}$s and the task release times is an open topic. Also, the effect of frequent task preemptions on conflicts can be investigated along with simulation or tracing tools. Another interesting study is the direct comparison of spinlock-based TDRR with semaphore based approaches disregarding AUTOSAR. This advancement could also lead to utilizing TDRR in other OSs to schedule sub-tasks more efficiently. Moreover, the greedy decision within TDRR to fill a conflict interval by choosing a runnable, of which its instructions are closest to the access conflict interval, could be replaced with a combination of runnables getting closer to filling the interval via, e.g., using a bin packing algorithm.

Finally, generic challenges that seem to be interesting research directions are a) the use runnables being called more than once in a task or across tasks, resulting in multi-sets, b) the investigation of different priority assignment approaches revised at, e.g., [64], the incorporation of CRPD analyses, or c) policy selection and platform minimization as potential optimization goals.

Appendices

A
List of Figures

B List of Tables

C

List of Algorithms

D

List of Examples

E
List of Definitions

F Bibliography

[21] Robert R. Schaller. “Moore’s Law: Past, Present, and Future.” In: *IEEE Spectr.* 34.6 (June 1997), pp. 52–59. ISSN: 0018-9235. DOI: 10.1109/6.591665.

[22] A. O. Caldeira and A. J. Leggett. “Influence of Dissipation on Quantum Tunneling in Macroscopic Systems.” In: *Phys. Rev. Lett.* 46 (4 Jan. 1981), pp. 211–214.

[23] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201575949.

[24] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. “A Survey and Comparative Study of Hard and Soft Real-Time Dynamic Resource Allocation Strategies for Multi-/Many-Core Systems.” In: *ACM Comput. Surv.* 50.2 (Apr. 2017). ISSN: 0360-0300. DOI: 10.1145/3057267.

[25] Jianjiang Ceng, Jeronimo Castrillon, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, T. Isshiki, and H. Kunieda. “MAPS: An Integrated Framework for MPSoC Application Parallelization.” English. In: *45th Design Automation Conference (DAC ’08).* Anaheim, CA, USA: ACM, June 2008, pp. 754–759. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391663.

[26] Christos Baloukas et al. “Mapping Embedded Applications on MPSoCs: The MNEMEE Approach.” In: vol. 105. Jan. 2011, pp. 165–179. DOI: 10.1007/978-94-007-1488-5_10.

[27] Christof Ebert and Capers Jones. “Embedded Software: Facts, Figures, and Future.” In: *Computer* 42.4 (Apr. 2009), pp. 42–52. ISSN: 0018-9162. DOI: 10.1109/MC.2009.118.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[29] Georg von der Brüggen. “Realistic Scheduling Models and Analyses for Advanced Real-Time Embedded Systems.” PhD thesis. Technical University Dortmund, 2019.

[30] Arne Hamann, Dakshina Dasari, Falk Wurst, Ignacio Sanudo, Nicola Capodieci, Paolo Burgio, and Marko Bertogna. “WATERS Industrial Challenge 2019.” In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS) co-located with the Euromicro Conference on Real-Time Systems (ECRTS)* (2019). Online at https://bit.ly/2RhDkMQ, visited 09.2020.

[31] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. “WATERS Industrial Challenge 2017.” In: *8th International Workshop an Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) co-located with the Euromicro Conference on Real-Time Systems (ECRTS)* (2017). Online at `https://bit.ly/2ReOQZ8`, visited 09.2020.

[32] Juan M Rivas, J Javier Gutiérrez, Julio L Medina, and Michael González Harbour. “Comparison of Memory Access Strategies in Multi-core Platforms Using MAST.” In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems* (2017).

[33] Johannes Schlatow, M Mischa, and Rolf Ernst. “Compositional Analysis of the WATERS Industrial Challenge 2017.” In: *WATERS workshop of the Euromicro Conference on Real-Time Systems (ECRTS) 2017* (2017).

[34] Jorge Martinez, Ignacio Sa, Paolo Burgio, and Marko Bertogna. “End-To-End Latency Characterization of Implicit and LET Communication Models.” In: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (2017).

[35] Alexander Wieder and Björn B. Brandenburg. “On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks.” In: *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium*. RTSS ’13. USA: IEEE Computer Society, 2013, pp. 45–56. ISBN: 9781479920068. DOI: `10.1109/RTSS.2013.13`.

[36] Razvan Racu, Li Li, Rafik Henia, Arne Hamann, and Rolf Ernst. “Improved Response Time Analysis of Tasks Scheduled Under Preemptive Round-Robin.” In: *Proceedings of the Int. Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS. ACM, 2007, pp. 179–184. ISBN: 978-1-59593-824-4. DOI: `10.1145/1289816.1289861`.

[37] W. Liu, J. Yi, M. Li, P. Chen, and L. Yang. “Energy-Efficient Application Mapping and Scheduling for Lifetime Guaranteed MPSoCs.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018), pp. 1–1. ISSN: 0278-0070. DOI: `10.1109/TCAD.2018.2801242`.

[38] J. P. Lehoczky. “Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines.” In: *Real-Time Systems Symposium, 1990. Proceedings., 11th*. RTSS. IEEE, Dec. 1990, pp. 201–209. ISBN: 0-8186-2112-5. DOI: `10.1109/real.1990.128748`.

[39] K. Traore, E. Grolleau, A. Rahni, and M. Richard. “Response-Time Analysis of Tasks with Offsets.” In: *Proceedings of the Conference on Emerging Technologies and Factory Automation*. Sept. 2006, pp. 1–8. DOI: `10.1109/ETFA.2006.355182`.

[40] Ingo Stierand, Philipp Reinkemeier, Sebastian Gerwinn, and Thomas Peikenkamp. “Computational Analysis of Complex Real-Time Systems - FMTV 2016 Verification Challenge.” In: *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, Toulouse, France*. WATERS. 2016.

[41] Lukas Krawczyk, Carsten Wolff, and Daniel Fruhner. “Automated Distribution of Software to Multi-core Hardware in Model Based Embedded Systems Development.” In: *Proc. of the 21st Int. Conf. on Information and Software Technologies (ICIST)*. Springer Int., 2015, pp. 320–329. ISBN: 978-3-319-24770-0. DOI: `10.1007/978-3-319-24770-0_28`.

[42] F. Wilhelmstötter and Various Contributors. *Jenetics is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, written in modern day Java.* Online at: `http://jenetics.io/`, visited 04.2020.

[43] Barak Naveh and Contributors. *JGraphT - A free Java Graph Library.* Online at `http://jgrapht.org/`, visited 04.2020.

[44] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation.* Online at: `http://www.choco-solver.org`, visited 04.2020. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[45] AMALTHEA / AMALTHEA4public Consortium and Contributors. *AMALTHEA Data Models - Online Documentation.* Online at: `https://www.eclipse.org/app4mc/documentation/`, visited 04.2020.

[46] Freescale Semiconductor, Inc. (by 2020: NXP Semiconductors). *Embedded Multicore: An Introduction.* Online at: `https://bit.ly/2t61QrT`, visited 01.2020. 2009.

[47] AUTOSAR Consortium. *AUTOSAR - Specification of Operating System; Release 4.4.0.* Online available at: `https://www.autosar.org/standards/`, visited 02.2020. Oct. 2018.

[48] James H. Anderson, Vasile Bud, and Umamaheswari C. Devi. "An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems." In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05)* (2005). DOI: `10.1109/ECRTS.2005.6`.

[49] Dirk Müller. "Schedulability Tests for Real-Time Uni- and Multiprocessor Systems." Habilitation. TU Chemnitz, 2014.

[50] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms." In: *19th Euromicro Conference on Real-Time Systems (ECRTS'07)* (2007), pp. 247–258. DOI: `10.1109/ECRTS.2007.81`.

[51] AUTOSAR Consortium. *AUTOSAR - Specification of RTE Software; Release 4.4.0.* Online available at: `https://www.autosar.org/standards/`, visited 03.2020. Oct. 2018.

[52] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System." In: *The Computer Journal* 29.5 (Jan. 1986), pp. 390–395. ISSN: 0010-4620. DOI: `10.1093/comjnl/29.5.390`.

[53] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: `10.1145/321738.321743`.

[54] Alan Burns. "Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach." In: *Advances in Real-Time Systems.* USA: Prentice-Hall, Inc., 1995, pp. 225–248. ISBN: 0130833487.

[55] Alessandro Biondi, Scuola Superiore, Sant Anna, Marco Di Natale, Scuola Superiore, Sant Anna, Giorgio Buttazzo, Scuola Superiore, and Sant Anna. "Response-Time Analysis for Real-Time Tasks in Engine Control Applications." In: *ICCPS* (2015), pp. 120–129. DOI: `10.1145/2735960.2735963`.

[56] Mircea Negrean and Rolf Ernst. "Response-time Analysis for Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources." In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)* (2012), pp. 191–200. DOI: `10.1109/SIES.2012.6356585`.

[57] Reinder J. Bril, Johan J. Lukkien, and Wim F.J. Verhaegh. "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred

preemption." In: *Real-Time Systems* 42.1-3 (2009), pp. 63–119. ISSN: 09226443. DOI: 10.1007/s11241-009-9071-z.

[58] Yun Wang and M. Saksena. "Scheduling Fixed-Priority Tasks with Preemption Threshold." In: *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*. Dec. 1999, pp. 328–335. DOI: 10.1109/RTCSA.1999.811269.

[59] Giorgio Buttazzo, Marco Bertogna, and Gang Yao. "Limited Preemptive Scheduling for Real-Time Systems. A Survey." In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 3–15. ISSN: 1941-0050. DOI: 10.1109/TII.2012.2188805.

[60] Z. Dong, C. Liu, S. Bateni, K. Chen, J. Chen, G. v. d. Brüggen, and J. Shi. "Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-Based Partitioning Approaches." In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2018, pp. 164–176. DOI: 10.1109/RTAS.2018.00026.

[61] Robert I. Davis. "Burns Standard Notation for Real-Time Scheduling." In: *In Real-Time Systems: The Past , the Present, and the Future* (2013), pp. 1–4.

[62] Ignacio Sanudo, Paolo Burgio, and Marko Bertogna. "Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System." In: *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, Toulouse, France*. WATERS. 2016.

[63] N.C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. Tech. rep. YCS-164. Department of Computer Science, University of York, 1991.

[64] Robert I. Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. "A Review of Priority Assignment in Real-Time Systems." In: *J. Syst. Archit.* 65.C (Apr. 2016), pp. 64–82. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2016.04.002.

[65] Robert I. Davis, A. Zabos, and A. Burns. "Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems." In: *IEEE Transactions on Computers* 57.9 (2008), pp. 1261–1276. DOI: 10.1109/TC.2008.66.

[66] Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. "Improved Multiprocessor Global Schedulability Analysis." In: *Real-Time Systems* 46.1 (2010), pp. 3–24. DOI: 10.1007/s11241-010-9096-3.

[67] Piotr Dziurzanski, Amit Kumar Singh, Leandro Soares Indrusiak, and Björn Saballus. "Hard Real-Time Guarantee of Automotive Applications during Mode Changes." In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS '15. Lille, France: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450335911. DOI: 10.1145/2834848.2834859.

[68] Michael L. Dertouzos and A. K. Mok. "Multiprocessor Online Scheduling of Hard-Real-Time Tasks." In: *IEEE Trans. Softw. Eng.* 15.12 (Dec. 1989), pp. 1497–1506. ISSN: 0098-5589. DOI: 10.1109/32.58762.

[69] Michael L. Dertouzos. "Control Robotics: The Procedural Control of Physical Processes." In: *IFIP Congress*. 1974, pp. 807–813. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Dertouzos74.

[70] Sanjoy K. Baruah and John Carpenter. "Multiprocessor Fixed-Priority Scheduling with Restricted Interprocessor Migrations." In: *J. Embedded Comput.* 1.2 (Apr. 2005), pp. 169–178. ISSN: 1740-4460. DOI: 10.1109/EMRTS.2003.1212744.

[71] James H. Anderson and H. Leontyev. “A Unified Hard/Soft Real-Time Schedulability Test for Global EDF Multiprocessor Scheduling.” In: *2013 IEEE 34th Real-Time Systems Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2008, pp. 375–384. DOI: 10.1109/RTSS.2008.15.

[72] Nathan Fisher, Joël Goossens, and Sanjoy K. Baruah. “Optimal Online Multiprocessor Scheduling of Sporadic Real-time Tasks is Impossible.” In: *Real-Time Systems* 45.1-2 (2010), pp. 26–71. DOI: 10.1007/s11241-010-9092-7.

[73] Kecheng Yang and James H. Anderson. “Optimal GEDF-based Schedulers that allow Intra-Task Parallelism on Heterogeneous Multiprocessors.” In: *12th IEEE Symposium on Embedded Systems for Real-time Multimedia, ESTIMedia 2014, Greater Noida, India, October 16-17, 2014*. IEEE, 2014, pp. 30–39. DOI: 10.1109/ESTIMedia.2014.6962343.

[74] Xu Jiang, Nan Guan, Di Liu, and Weichen Liu. “Analyzing GEDF Scheduling for Parallel Real-Time Tasks with Arbitrary Deadlines.” In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. Ed. by Jürgen Teich and Franco Fummi. IEEE, 2019, pp. 1537–1542. DOI: 10.23919/DATE.2019.8714859.

[75] Anand Srinivasan. “Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors.” PhD thesis. University of North Carolina at Chapel Hill, 2003, p. 200.

[76] Michael Deubzer. “Robust Scheduling of Real-Time Applications on Efficient Embedded Multicore Systems.” PhD thesis. Technische Universität München, Lehrstuhl für Informationstechnik im Maschinenwesen, 2011.

[77] Mircea Florin Negrean. “Performance Analysis of Multi-Core Multi-Mode Systems with Shared Resources - Principles and Application to AUTOSAR -.” PhD thesis. Technischen Universität Braunschweig, 2016.

[78] Andreas Abel et al. “Impact of Resource Sharing on Performance and Performance Prediction: A Survey.” In: *CONCUR 2013 – Concurrency Theory*. Springer Berlin Heidelberg, 2013, pp. 25–43. ISBN: 978-3-642-40184-8. DOI: 10.7873/DATE.2014.109.

[79] Peter Greenhalgh. *Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7 - Improving Energy Efficiency in High-Performance Mobile Platforms*. White Paper, Online: https://bit.ly/2GHP8Xa, visited: 09.2020. 2011.

[80] Nvidia Corporation. *Jetson AGX Xavier and the New Era of Autonomous Machines*. Online at: https://bit.ly/2RNTGMP, visited 01.2020.

[81] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. “Model-Based Engineering in the Embedded Systems Domain: An Industrial Survey on the State-of-Practice.” In: *Softw. Syst. Model.* 17.1 (Feb. 2018), pp. 91–113. ISSN: 1619-1366. DOI: 10.1007/s10270-016-0523-3.

[82] Juan M Rivas, J Javier Gutiérrez, Julio L Medina, and Michael González Harbour. “Calculating Latencies in an Engine Management System Using Response Time Analysis with MAST.” In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (2016).

[83] Alessio Balsini, Alessandra Melani, Pasquale Buonocunto, and Marco Di Natale. “FMTV 2016: Where is the Actual Challenge?” In: *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, Toulouse, France*. WATERS. 2016.

[84] Junchul Choi, Donghyun Kang, and Soonhoi Ha. “A Novel Analytical Technique for Timing Analysis of FMTV 2016 Verification Challenge Benchmark.” In: *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, Toulouse, France*. WATERS. 2016.

[85] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. “Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores.” In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (2017).

[86] Claire Pagetti and Onera Enseeiht Tuhh. “WATERS Industrial Challenge 2017 with Prelude.” In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems* (2017).

[87] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, Giorgio Buttazzo, and Marco Di Natale. “Addressing Analysis and Partitioning Issues for the WATERS 2019 Challenge.” In: *WATERS Workshop of the ECRTS Conference* (2019).

[88] Alexander Diewald, Simon Barner, and Selma Saidi. “Combined Data Transfer Response Time and Mapping Exploration in MPSoCs.” In: *WATERS Workshop of the ECRTS Conference* (2019).

[89] Steffen Vaas, Peter Ulbrich, Marc Reichenbach, and Dietmar Fey. “Application-Specific Tailoring of Multi-Core SoCs for Real-Time Systems with Diverse Predictability Demands.” In: *Journal of Signal Processing Systems* 91.7 (July 2019), pp. 773–786. ISSN: 1939-8018. DOI: 10.1007/s11265-018-1389-0.

[90] Julian Kienberger. “Systematic and Methodical Analysis, Validation and Parallelization of Embedded Automotive Software for Multiple-IEU Platforms.” PhD thesis. University of Augsburg, 2019, p. 196.

[91] Martin Lowinski. “Parallelization of Legacy Automotive Control Software for Multi-Core Platforms.” PhD thesis. Technische Universität Berlin, 2019, p. 168.

[92] Martin Lowinski, Dirk Ziegenbein, and Sabine Glesner. “Partitioning Embedded Real-Time Control Software based on Communication Dependencies.” In: *Proc. of the Int. Workshop on Modelling in Automotive Software Engineering*. 2015, pp. 2–11.

[93] Martin Lowinski, Dirk Ziegenbein, and Sabine Glesner. “Splitting Tasks for Migrating Real-Time Automotive Applications to Multi-Core ECUs.” In: *11th IEEE Symp. on Industrial Embedded Systems*. 2016. DOI: 10.1109/SIES.2016.7509418.

[94] AUTOSAR Consortium. *AUTOSAR Classic Platform v4.4.0: Specification of Timing Extensions*. Online available at: https://www.autosar.org/standards/, visited 01.2020. Oct. 2018.

[95] Wenhao Wang, Sylvain Cotard, Fabrice Gravez, Yael Chambrin, and Benoit Miramond. “Optimizing Application Distribution on Multi-Core Systems within AUTOSAR.” In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. TOULOUSE, France, Jan. 2016.

[96] Arne Hamann. *Real-time Systems Engineering @ Bosch*. Slides published along with a guest lecture at Friedrich-Alexander-Universität Erlangen-Nürnberg, Department of Computer Science 4 (Distributed Systems and Operating Systems), Echtzeitsysteme. Online at: https://bit.ly/36hrE1x, visited 01.2020.

[97] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.6*. Online at: https://www.omg.org/spec/SysML/1.6, visited 01.2020.

[98] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language, Third edition*. 3rd ed. MK/OMG Press. Morgan Kaufmann, Oct. 2014. ISBN: 978-0-12-800202-5.

[99] Tim Weilkiens. *SYSMOD - The Systems Modeling Toolbox - Pragmatic MBSE with SysML, 2nd Edition*. Dec. 2016. ISBN: 9783981787580.

[100] J.-L Voirin. *Model-based System and Architecture Engineering with the Arcadia Method*. Nov. 2017, pp. 1–368. ISBN: 9781785481697.

[101] Murray Cantor. "Rational Unified Process for Systems Engineering: Part 1." In: *Journal of Systems Architecture - JSA* (Jan. 2003).

[102] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hnninger. *Advanced Model-Based Engineering of Embedded Systems: Extensions of the SPES 2020 Methodology*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319480022.

[103] International Council on Systems Engineering (INCOSE). *System Engineering Vision 2025*. Online at: https://bit.ly/37pVYsd, visited 01.2020. July 2014.

[104] Object Management Group. *XML Metadata Interchange (XMI) Specification*. Online at: http://www.omg.org/spec/XMI/, visited 01.2020.

[105] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Vol. 2050. Lecture Notes in Computer Science. Springer, 2001. ISBN: 3-540-42184-X.

[106] Jonas Diemer, Philip Axer, and Rolf Ernst. "Compositional Performance Analysis in Python with pyCPA." In: *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (2012). Online at http://retis.sssup.it/waters2012/accepted/102_Final_paper.pdf, visited 09.2020.

[107] Kai Richter. "Compositional Scheduling Analysis Using Standard Event Models." In: (2005), p. 236.

[108] Martijn Hendriks. "Model Checking Timed Automata." PhD thesis. Radboud Universiteit Nijmegen, 2006. ISBN: 9090203788.

[109] Christian Bradatsch. "Multicore-Entwicklungsplattform für den Automobilbereich." PhD thesis. Universität Augsburg, Fakultät für Angewandte Informatik, 2016.

[110] Florian Kluge. *tms-sim – Timing Models Scheduling Simulation Framework Release 2016-07*. July 2016. DOI: 10.13140/RG.2.1.2544.0246.

[111] Michael Deubzer, Martin Hobelsberger, Juergen Mottok, Frank Schiller, Reiner Dumke, Markus Siegle, Ulrich Margull, Michael Niemetz, and Gerhard Wirrer. "Modeling and Simulation of Embedded Real-Time Multicore Systems." In: *Proceedings of the 3rd Embedded Software Engineering Congress* (2010), pp. 228–241.

[112] Matthias Freier. "Analysis of Real-Time Capabilities of Dynamic Scheduled System Applications." PhD thesis. Karlsruher Institut für Technologie (KIT), 2016.

[113] Timon Kelter. "WCET Analysis and Optimization for Multi-Core Real-Time Systems." PhD thesis. Dortmund University, 2015.

[114] VDA QMC Working Group 13 / Automotive SIG. *Automotive SPICE Process Assessment / Reference Model; Version 3.1*. Online at: https://bit.ly/3asMZbA, visited 01.2020; Revision ID 656. Nov. 2017.

[115] Marie-Agnés Peraldi-Frati, Arda Goknil, Julien DeAntoni, and Johan Nordlander. "A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2." In: *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems* (2012), pp. 230–239. DOI: 10.1109/ICECCS20050.2012.6299218.

[116] Shuai Li, Matteo Morelli, Ansgar Radermacher, Jérémie Tatibouët, Pauline Deville, Arnault Lapitre, Sébastien Gérard, and Chokri Mraidha. *Polygraph Tool Suite: Configuration and Conformity Validation for Data Flow Based Real-Time Systems.* 2019.

[117] Michael Gonzalez Harbour, Jose Carlos Palencia Gutierrez, Jose Javier Gutierrez Garcia, and Juan Maria Rivas Concepcion. *Modelling and Analysis Suite for Real Time Applications (MAST 1.5.1) - Analysis Techniques used in MAST.* https://mast.unican.es/mast_analysis_techniques.pdf, visited 02.2020. 2015.

[118] Martin Hillenbrand. "Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen." German. PhD thesis. 2012. 398 pp. ISBN: 978-3-86644-803-2. DOI: 10.5445/KSP/1000025616.

[119] Henrik Kaijser, Henrik Lönn, Matthias Tichy, Wenjing Yuan, and Saimir Baci. "Tool Assisted Model Based Multi Objective Analyses of Automotive Embedded Systems." In: *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, 7th of July 2015, Lund, Sweden.* 2015.

[120] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. "Timed Hazard Analysis of Self-healing Systems." In: *Assurances for Self-Adaptive Systems.* Ed. by Rogério de Lemos Javier Camara, Carlo Ghezzi, and Antonia Lopes. Vol. 7740. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2013, pp. 112–151. DOI: 10.1007/978-3-642-36249-1_5.

[121] James H. Anderson. "Having Fun Experimenting with Hardware Management and Mixed Criticality on Multicore." In: *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) 2015, co-located with the Euromicro Conference on Real-Time Systems (ECRTS)* (2015). Keynote Slides, available at: https://waters2015.inria.fr/files/2015/06/Waters2015-Keynote.pdf.

[122] Florian Leitner-Fischer, Stefan Leue, and Sirui Liu. "Automated Freedom from Interference Analysis for Automotive Software." In: *CARS 2016 : Critical Automotive applications : Robustness & Safety.* Ed. by Matthieu Roy. Villeurbanne: CCSD, 2016. URL: https://hal.archives-ouvertes.fr/CARS2016/hal-01375597.

[123] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. "Mapping on Multi Many Core Systems: Survey of Current and Emerging Trends." In: *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE* (2013), pp. 1–10. ISSN: 0738-100X. DOI: 10.1145/2463209.2488734.

[124] Vincenzo Bonifaci, Björn B. Brandenburg, Gianlorenzo D'Angelo, and Alberto Marchetti-Spaccamela. "Multiprocessor Real-Time Scheduling with Hierarchical Processor Affinities." In: *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016.* 2016, pp. 237–247. DOI: 10.1109/ECRTS.2016.24.

[125] Hye Churn Jang and Hyun Wook Jin. "MiAMI: Multi-core aware Processor Affinity for TCP/IP over Multiple Network Interfaces." In: *Proceedings - Symposium on the High Performance Interconnects, Hot Interconnects* (2009), pp. 73–82. ISSN: 15504794. DOI: 10.1109/HOTI.2009.19.

[126] Owen R. Kelly, Hakan Aydin, and Baoxian Zhao. "On Partitioned Scheduling of Fixed-Priority Mixed-Criticality Task Sets." In: *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing*

and Communications. TRUSTCOM '11. USA: IEEE Computer Society, 2011, pp. 1051–1059. ISBN: 9780769546001. DOI: 10.1109/TrustCom.2011.144.

[127] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. "Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors." In: *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium.* RTSS '09. USA: IEEE Computer Society, 2009, pp. 469–478. ISBN: 9780769538754. DOI: 10.1109/RTSS.2009.51.

[128] Arne Hamann, Dirk Ziegenbein, Simon Kramer, and Martin Lukasiewycz. "FMTV 2016 Verification Challenge." In: *7th International Workshop an Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) co-located with the Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2016). Online at https://bit.ly/3horif4, visited 09.2020.

[129] Marek Jersak. "Compositional Performance Analysis for Complex Embedded Applications." PhD thesis. Technical University of Braunschweig, 2005. DOI: 10.1504/ijes.2005.008807.

[130] Jan Henrik Weinstock, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, and Laura Tosoratto. "Time-Decoupled Parallel SystemC Simulation." English. In: *Proceedings of the Conference on Design, Automation & Test in Europe (DATE).* European Design and Automation Association, 2014. DOI: 10.7873/DATE.2014.204.

[131] N. Navet, S. Louvart, J. Villanueva, S. Campoy-Martinez, and J. Migge. "Timing Verification of Automotive Communication Architectures using Quantile Estimation." In: *Embedded Real-Time Software and Systems (ERTS)* (2014).

[132] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. "System Level Performance Analysis - the SymTA/S Approach." In: *IEE Proceedings Computers and Digital Techniques* (2005). DOI: 10.1049/ip-cdt:20045088.

[133] Gerd Behrmann and Kim G. David Alexandre and Larsen. "A Tutorial on Uppaal." In: *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures.* Ed. by Marco Bernardo and Flavio Corradini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-30080-9. DOI: 10.1007/978-3-540-30080-9_7.

[134] Sergio Yovine. "KRONOS: A Verification Tool for Real-time Systems." In: *International Journal on Software Tools for Technology Transfer* 1.1 (Dec. 1997), pp. 123–133. ISSN: 1433-2779. DOI: 10.1007/s100090050009.

[135] Didier Lime, Oliver H. Roux, Charlotte Seidner, and Louis Marie Traonouez. "Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches." In: *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), Lecture Notes in Computer Science* (2009).

[136] K. Tindell and J. Clark. "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems." In: *Microprocessing & Microprogramming* 50 (Apr. 1994), pp. 117–134. DOI: 10.1016/0165-6074(94)90080-9.

[137] Traian Pop, Petru Eles, and Zebo Peng. "Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems." In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign.* CODES '02.

Estes Park, Colorado: Association for Computing Machinery, 2002, pp. 187–192. ISBN: 1581135424. DOI: 10.1145/774789.774828.

[138] J. C. Palencia Gutierrez, J. J. Gutierrez Garcia, and M. Gonzalez Harbour. “Best-case Analysis for Improving the Worst-case Schedulability Test for Distributed Hard Real-time Systems.” In: *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*. June 1998, pp. 35–44. DOI: 10.1109/EMWRTS.1998.684945.

[139] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. “Model Checking and the State Explosion Problem.” In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[140] Claire Maizaa, Hamza Rihani, Juan M. Rivas, Joel Goossens, Sebastian Altmeyer, and Robert I. Davis. “A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems.” In: *ACM Comput. Surv. 1, 1, Article 01* 1.1 (2019), p. 46. DOI: 10.1145/3323212.

[141] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. “Cache-Related Preemption and Migration Delays : Empirical Approximation and Impact on Schedulability.” In: *Proceedings of Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. 2010, pp. 33–44.

[142] Daniel Alexander Cordes. “Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models.” PhD thesis. Technische Universität Dortmund, 2013.

[143] Gerardine Immaculate Mary, Z. C. Alex, and Lawrence Jenkins. “Response Time Analysis of Messages in Controller Area Network: A Review.” In: *Journal of Computer Networks and Communications* (2013). ISSN: 2090-7141. DOI: 10.1155/2013/148015.

[144] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Transactions on computers* 39.9 (1990), pp. 1175–1185. DOI: 10.1109/12.57058.

[145] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised.” In: *Real-Time Systems* 35.3 (2007), pp. 239–272. ISSN: 09226443. DOI: 10.1007/s11241-007-9012-7.

[146] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. “Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions.” In: Dec. 2018. DOI: 10.1109/RTSS.2018.00056.

[147] Martin Stigge, Nan Guan, and Wang Yi. “Refinement-Based Exact Response-Time Analysis.” In: *Proceedings of the 2014 Agile Conference*. AGILE ’14. USA: IEEE Computer Society, 2014, pp. 143–152. ISBN: 9781479957989. DOI: 10.1109/ECRTS.2014.29.

[148] Björn B. Brandenburg. *Multiprocessor Real-Time Locking Protocols: A Systematic Review*. Online at: https://arxiv.org/abs/1909.09600, visited 12.2019. 2019. arXiv: 1909.09600 [cs.DC].

[149] Ruslan Sadykov and Laurence A Wolsey. “Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with

Deadlines and Release Dates.” In: *INFORMS Journal on Computing* 18.2 (2006), pp. 209–217. ISSN: 1091-9856. DOI: 10.1287/ijoc.1040.0110.

[150] J. N. Hooker. “A Hybrid Method for the Planning and Scheduling.” In: *Constraints Journal* 10.4 (2005), pp. 385–401. ISSN: 13837133. DOI: 10.1007/s10601-005-2812-2.

[151] Ze-Wei Chen, Hang Lei, Mao-Lin Yang, Yong Liao, and Jia-Li Yu. “Improved Task and Resource Partitioning under the Resource-Oriented Partitioned Scheduling.” In: *Journal of Computer Science and Technology* 34 (4 2019), pp. 839–853. DOI: 10.1007/s11390-019-1945-5.

[152] Zaid Al-bayati, Youcheng Sun, Haibo Zeng, Marco di Natale, Qi Zhu, and Brett Meyer. “Task Placement and Selection of Data Consistency Mechanisms for Real-time Multicore Applications.” In: *21st IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)* (2015), pp. 172–181. DOI: 10.1109/RTAS.2015.7108440.

[153] R Govindarajan. “On-Chip Memory Architecture Exploration Framework for DSP Processor- Based Embedded System on Chip On-Chip Memory Architecture Exploration of Embedded System on Chip.” In: (2008). DOI: 10.1145/2146417.2146422.

[154] Laurent Perron. “Operations Research and Constraint Programming at Google.” In: *Principles and Practice of Constraint Programming – CP 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23786-7.

[155] Jean Charles Régin. “Global constraints: A survey.” In: *Springer Optimization and Its Applications* 45 (2011), pp. 63–134. ISSN: 19316836. DOI: 10.1007/978-1-4419-1644-0_3.

[156] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Boloni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, and Bin Yao. “A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems.” In: *Journal of Parallel and Distributed Computing* 61 (2001), pp. 810–837. DOI: 10.1006/jpdc.2000.1714.

[157] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. “Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.6 (June 2010), pp. 911–924. ISSN: 1937-4151. DOI: 10.1109/TCAD.2010.2048354.

[158] Dhananjay R. Thiruvady, Irene Moser, Aldeida Aleti, and Asef Nazari. “Constraint Programming and Ant Colony System for the Component Deployment Problem.” In: *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*. Vol. 29. 2014, pp. 1937–1947. DOI: 10.1016/j.procs.2014.05.178.

[159] Meng X., Liu Y., Gao X., and Zhang H. “A New Bio-inspired Algorithm: Chicken Swarm Optimization.” In: *Advances in Swarm Intelligence ICSI, Lecture Notes in Computer Science* 8794 (2014), p. 5. DOI: 10.1007/978-3-319-11857-4_10.

[160] Pradip Kumar Sahu, Tapan Shah, Kanchan Manna, and Santanu Chattopadhyay. “Application Mapping Onto Mesh-Based Network-on-Chip Using Discrete Particle Swarm Optimization.” In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22 (2014), pp. 300–312. DOI: 10.1109/TVLSI.2013.2240708.

[161] A. Alagarsamy and L. Gopalakrishnan. “SAT: A New Application Mapping Method for Power Optimization in 2D - NoC.” In: *2016 20th International Symposium on VLSI Design and Test (VDAT)*. May 2016, pp. 1–6. DOI: 10.1109/ISVDAT.2016.8064880.

[162] Jian-Jun Han, Dakai Zhu, Xiaodong Wu, Laurence T Yang, and Hai Jin. “Multiprocessor Real-time Systems with Shared Resources: Utilization Bound and Mapping.” In: *IEEE Transactions on Parallel and Distributed Systems* 25 (11 2014), pp. 2981–2991. DOI: 10.1109/TPDS.2013.302.

[163] Jiazheng Li, Guozhi Song, Yue Ma, Cheng Wang, Baohui Zhu, Yan Chai, and Jieqi Rong. “Bat Algorithm Based Low Power Mapping Methods for 3D Network-on-Chips.” In: *Theoretical Computer Science*. Springer Singapore, 2017, pp. 277–295. ISBN: 978-981-10-6893-5.

[164] Aravindhan Alagarsamy, Lakshminarayanan Gopalakrishnan, Sundarakannan Mahilmaran, and Seok Bum Ko. “A Self-Adaptive Mapping Approach for Network on Chip with Low Power Consumption.” In: *IEEE Access* 7 (2019), pp. 84066–84081. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2925381.

[165] Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. “Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?” In: *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*. IEEE Computer Society, 2008, pp. 342–353. DOI: 10.1109/RTAS.2008.27.

[166] Florian Kluge, Chenglong Yu, Jörg Mische, Sascha Uhrig, and Theo Ungerer. “Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor.” In: *Proc. of th 12th Int. Workshop on Software and Compilers for Embedded Systems*. Nice, France: ACM, 2009, pp. 33–42. ISBN: 978-1-60558-696-0. DOI: 10.1145/1543820.1543828.

[167] Rajib Mall. *Real-Time Systems: Theory and Practice*. Dorling Kindersley (India) Pvt. Ltd, 2007. ISBN: 978-81-317-0069-3.

[168] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. “Real-Time Synchronization Protocols for Multiprocessors.” In: *Proc. of the 9th IEEE Real-Time Systems Symp. (RTSS'88)*. 1988, pp. 259–269. DOI: 10.1109/REAL.1988.51121.

[169] Ragunathan Rajkumar. “Real-time Synchronization Protocols for Shared Memory Multiprocessors.” In: *10th International Conference on Distributed Computing Systems*. 1990. DOI: 10.1109/ICDCS.1990.89257.

[170] Renata Martins Gomes, Fabian Mauroner, and Marcel Baunach. “Collaborative Resource Management for Multi-Core AUTOSAR OS.” In: *Informatik aktuell: Betriebssysteme und Echtzeit*. Ed. by Wolfgang A. Halang and Olaf Spinczyk. Springer-Verlag Berlin Heidelberg, 2015, p. 136. ISBN: 978-3-662-48611-5. DOI: 10.1007/978-3-662-48611-5_11.

[171] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Scuola Superiore, Sant Anna, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. “A Comparison of MPCP and MSRP when Sharing Resources in the Janus Multiple-Processor on a Chip Platform.” In: *IEEE Real Time Technology and Applications Symp.* 2003, pp. 1–10. DOI: 10.1109/RTTAS.2003.1203051.

[172] Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. “A Flexible Real-Time Locking Protocol for Multiprocessors.” In: *Int. Conf.*

on Embedded and Real-Time Computing Systems and Applications. 2007. ISBN: 0769529755. DOI: 10.1109/RTCSA.2007.8.

[173] Andreas Sailer, Stefan Schmidhuber, Maximilian Hempe, and Michael Deubzer. "Distributed Multi-Core Development in the Automotive Domain – A Practical Comparison of ASAM MDX vs . AUTOSAR vs . AMALTHEA." In: *ARCS 2016; 29th International Conference on Architecture of Computing Systems*. 2016, pp. 4–7. ISBN: 9783800741571.

[174] AMALTHEA / AMALTHEA4public Consortium and Contributors. *Eclipse* APP4MC *- Application Platform Project for Multicore; IDE*. APP4MC Tool Platform available for download at: https://www.eclipse.org/app4mc/downloads/, visited 04.2020.

[175] Xian-He Sun and Dawei Wang. "Concurrent Average Memory Access Time." In: *Computer* 47.5 (May 2014), pp. 74–80. ISSN: 0018-9162. DOI: 10.1109/MC.2013.227.

[176] Christian Ferdinand and Reinhold Heckmann. "aiT: Worst-Case Execution Time Prediction by Static Program Analysis." In: *Building the Information Society* 156 (2004), pp. 377–383. DOI: 10.1007/978-1-4020-8157-6_29.

[177] Rapita Systems Ltd. *RapiTime Worst-Case Execution Time Analysis Tool kit.* Online at: https://bit.ly/2sUJ3j9, visited 12.2019.

[178] Tidorum Ltd. Niklas Holsti. "Bound-T Execution Time Analyzer." In: (2004). http://www.bound-t.com/, visited 04.2020.

[179] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. "Chronos: A Timing Analyzer for Embedded Software." In: *Science of Computer Programming 69.1-3 (12/2007)* (2007), pp. 56–67. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.01.014.

[180] Adrian Prantl, Markus Schordan, and Jens Knoop. "TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis." In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (2008). http://costa.tuwien.ac.at/papers/wcet08-tubound.pdf, visited 12.2019, pp. 141–148.

[181] Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. "Towards an Open Timing Analysis Platform." In: *11th International Workshop on Worst-Case Execution Time Analysis* (July 2011).

[182] Raimund Kirner. "The WCET Analysis Tool Calcwcet167." In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies*. ISoLA-12 Part II (2012), pp. 158–172. DOI: 10.1007/978-3-642-34032-1_17.

[183] TRACES Team. "OTAWA - Open Tool for Adaptive WCET Analysis." In: (2014). http://www.otawa.fr/, visited 04.2020.

[184] Björn Lisper. "SWEET - A Tool for WCET Flow Analysis (Extended Abstract)." In: *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*. ISoLA 8803 (2014). Ed. by Margaria T. and Steffen B. DOI: 10.1007/978-3-662-45231-8_38.

[185] Aloysius Ka-Lau Mok. "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment." PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1983. URL: http://hdl.handle.net/1721.1/15670.

[186] Sophie Quinton, Mircea Negrean, and Rolf Ernst. "Formal Analysis of Sporadic Bursts in Real-Time Systems." In: *Proceedings of the Conference on Design,*

Automation and Test in Europe. DATE ’13. Grenoble, France: EDA Consortium, 2013, pp. 767–772. ISBN: 9781450321532. DOI: 10.7873/DATE.2013.163.

[187] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. “Real World Automotive Benchmarks For Free.” In: *6th International Workshop an Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) co-located with the Euromicro Conference on Real-Time Systems (ECRTS)* (2015).

[188] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Springer Publishing Company, Incorporated, 2011. ISBN: 1461406757.

[189] The TIMMO-2-USE Consortium. “TIMMO-2-USE: Timing Model – Tools, Algorithms, Languages, Methodology, USE Cases; Deliverable D11.” In: (Aug. 2012). Online at https://bit.ly/2Rg2489, visited 03.2020.

[190] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989. ISBN: 0262691302.

[191] Sebastian Kehr, Milos Panic, Eduardo Quinones, Bert Boddecker, Jaume Abella, Francisco J. Cazorla, and Günter Schäfer. “RunPar : An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores.” In: *Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis*. ACM, 2014, 29:1–29:10. ISBN: 9781450330510. DOI: 10.1145/2656075.2656096.

[192] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Uçar. “Multi-Level Direct K-Way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices.” In: *J. Parallel Distrib. Comput.* 68.5 (May 2008), pp. 609–625. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2007.09.006.

[193] Robert Preis, Burkhard Monien, and Stefan Schamberger. “Approximation Algorithms for Multilevel Graph Partitioning.” In: *Handbook of Approximation Algorithms and Metaheuristics*. Ed. by Teofilo F. Gonzalez. Chapman and Hall/CRC, 2007. DOI: 10.1201/9781420010749.ch60.

[194] Sanjoy K. Baruah and N. Fisher. “The Partitioned Multiprocessor Scheduling of Sporadic Task Systems.” In: *26th IEEE International Real-Time Systems Symposium (RTSS’05)*. 2005, 9 pp.–329. ISBN: 0-7695-2490-7. DOI: 10.1109/RTSS.2005.40.

[195] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin Marteil. “An ILP Approach for Mapping AUTOSAR Runnables on Multi-Core Architectures.” In: *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO ’15. Amsterdam, Holland: Association for Computing Machinery, 2015. ISBN: 9781605586991. DOI: 10.1145/2693433.2693439.

[196] F. Khenfri, K. Chaaban, and M. Chetto. “A Novel Heuristic Algorithm for Mapping AUTOSAR Runnables to Tasks.” In: *2015 International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*. 2015, pp. 1–8.

[197] Peng Deng, Fabio Cremona, Qi Zhu, Marco Di Natale, and Haibo Zeng. “A Model-Based Synthesis Flow for Automotive CPS.” In: *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*. ICCPS ’15. Seattle, Washington: Association for Computing Machinery, 2015, pp. 198–207. ISBN: 9781450334556. DOI: 10.1145/2735960.2735972.

[198] Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. “Multi-Source Software on Multicore Automotive ECUs.” In: *IEEE Transactions*

on Industrial Electronics 59.10 (2012), pp. 3934–3942. DOI: 10.1109/TIE.2012.2185913.

[199] Tae-Young Choe. “Task Scheduling Algorithm to Reduce the Number of Processors using Merge Conditions.” In: *International Journal on Computer Science and Engineering (IJCSE)* (Feb. 2012).

[200] K. Kanoun, D. Atienza, N. Mastronarde, and M. van der Schaar. “A Unified Online Directed Acyclic Graph Flow Manager for Multicore Schedulers.” In: *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. 2014, pp. 714–719. DOI: 10.1109/ASPDAC.2014.6742974.

[201] Gregory S. Hornby, Lukas Sekanina, and Pauline C. Haddow. “Evolvable Systems: From Biology to Hardware.” In: *8th International Conference on Evolvable Systems, ICES 2008*. Springer, 2008.

[202] J.L.Szwarcfiter and P.E.Lauer. “Finding the Elementary Cycles of a Directed Graph in O(n+m) per Cycle.” In: *Technical Report Series* 60 (May 1974). DOI: 10.1007/BF01931370.

[203] Robert Tarjan. “Enumeration of the Elementary Circuits of a Directed Graph.” In: *SIAM Journal on Computing* 2.3 (Sept. 1973), pp. 211–216. ISSN: 0097-5397. DOI: 10.1137/0202017.

[204] Paola Festa, Panos M. Pardalos, and Mauricio G.C. Resende. “Feedback Set Problems.” In: *Encyclopedia of Optimization*. Boston, MA: Springer US, 2009, pp. 1005–1016. ISBN: 978-0-387-74759-0. DOI: 10.1007/978-0-387-74759-0_178.

[205] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. “Introduction to Algorithms.” In: vol. 3. MIT Press, 2009. Chap. 21, 24 and 27.

[206] Yu-Kwong Kwok and Ishfaq Ahmad. “Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors.” In: *Parallel and Distributed Systems, IEEE Transactions on* 7.5 (1996), pp. 506–521. DOI: 10.1109/71.503776.

[207] Tao Yang and Apostolos Gerasoulis. “DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors.” In: *IEEE Transactions on Parallel and Distributed Systems* 5 (1993), pp. 951–967. DOI: 10.1109/71.308533.

[208] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[209] Guoqi Xie, Yuekun Chen, Yan Liu, Yehua Wei, Renfa Li, and Keqin Li. “Resource Consumption Cost Minimization of Reliable Parallel Applications on Heterogeneous Embedded Systems.” In: *IEEE Transactions on Industrial Informatics* 13.4 (2016), pp. 1629–1640. ISSN: 1551-3203. DOI: 10.1109/TII.2016.2641473.

[210] Renata Melo e Silva de Oliveira and Maria Sofia F. Oliveira de Castro Ribeiro. “Comparing Mixed & Integer Programming vs. Constraint Programming by Solving Job-Shop Scheduling Problems.” In: *Independent Journal of Management & Production* 6.1 (2015), pp. 211–238. ISSN: 2236-269X. DOI: 10.14807/ijmp.v6i1.262.

[211] Kamol Limtanyakul. “Scheduling of Tests on Vehicle Prototypes Using Constraint and Integer Programming.” In: *Operations Research Proceedings 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 421–426. ISBN: 978-3-540-77903-2.

[212] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. “Communication Centric Design in Complex Automotive Embedded Systems.” In:

29th Euromicro Conference on Real-Time Systems (ECRTS 2017). Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:20. ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.10.

[213] Frank Hannig, João M P Cardoso, Thilo Pionteck, Wolfgang Schröder-Preikschat, and Jürgen Teich. "Architecture of computing systems – ARCS 2016: 29th international conference Nuremberg, Germany, April 4-7, 2016 Proceedings." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9637. 2016, pp. 173–184. ISBN: 9783319306940. DOI: 10.1007/978-3-319-30695-7.

[214] Lukas Krawczyk, Mahmoud Bazzal, Ram Prasath Govindarajan, and Carsten Wolff. "An Analytical Approach for Calculating End-to-End Response Times in Autonomous Driving Applications." In: *WATERS Workshop of the ECRTS Conference* (2019).

[215] Robert Hilbrich and Michael Behrisch. "Improving the Efficiency of Dislocality Constraints for an Automated Software Deployment in Safety-Critical Systems." In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018)*. Workshop on Software Engineering for Applied Embedded Real-Time Systems, SEERTS'18. ceur-ws.org, Mar. 2018, pp. 90–95.

[216] Yiannis Papadopoulos and Christian Grante. "Evolving Car Designs using Model-based Automated Safety Analysis and Optimisation Techniques." In: *Journal of Systems and Software* 76.1 (2005), pp. 77–89. ISSN: 0164-1212. DOI: 10.1016/j.jss.2004.06.027.

[217] Carlos M. Fonseca and Peter J. Fleming. "Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization." In: *Proceedings of the Fifth International Conference on Genetic Algorithms* (July 1993), pp. 416–423. ISSN: 14639076.

[218] Stefan Stattelmann, Sebastian Ottlik, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. "Combining Instruction Set Simulation and WCET Analysis for Embedded Software Performance Estimation." In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES)* (2012), pp. 295–298. DOI: 10.1109/SIES.2012.6356600.

[219] Sanjoy K. Baruah, A. Burns, and Robert I. Davis. "Response-Time Analysis for Mixed Criticality Systems." In: *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*. RTSS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 34–43. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.12.

[220] Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. "Response-Time Analysis of Engine Control Applications Under Fixed-Priority Scheduling." In: *IEEE Transactions on Computers* 67.5 (May 2018), pp. 687–703. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2777826.

[221] Maciej Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009, pp. i–xiii, 1–386. ISBN: 978-1-84882-309-9.

[222] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z Chen. "Task Scheduling and Voltage Selection for Energy Minimization." In: *Proceedings of the 39th annual Design Automation Conference*. ACM. 2002, pp. 183–188. DOI: 10.1109/DAC.2002.1012617.

[223] Donghyun Kang, Junchul Choi, and Soonhoi Ha. "Worst Case Delay Analysis of Shared Resource Access in Partitioned Multi-Core Systems." In: *Proceedings of*

the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia. ESTIMedia '17. Seoul, Republic of Korea: Association for Computing Machinery, 2017, pp. 84–92. ISBN: 9781450351171. DOI: 10.1145/3139315.3139322.

[224] Björn B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems." PhD thesis. USA, 2011. ISBN: 9781267256188.

[225] U. Keskin, R.J. Bril, and J.J. Lukkien. "Exact Response-time Analysis for Fixed-priority Preemption-Threshold Scheduling." English. In: *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010, Bilbao, Spain, September 13-16, 2010).* United States: Institute of Electrical and Electronics Engineers, 2010, pp. 1–4. ISBN: 978-1-4244-6848-5. DOI: 10.1109/ETFA.2010.5640984.

[226] M. Saksena and Yun Wang. "Scalable Real-Time System Design Using Preemption Thresholds." In: *Proceedings 21st IEEE Real-Time Systems Symposium*, pp. 25–34. DOI: 10.1109/REAL.2000.895993.

[227] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Åke Jönsson. "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics." In: *RTSS 2008.* 2008.

[228] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. "End-to-End Timing Analysis of Cause-Effect Chains in Automotive Embedded Systems." In: *Journal of Systems Architecture* 80.Supplement C (Oct. 2017). URL: http://www.es.mdh.se/publications/4877-.

[229] Tomasz Kloda, Antoine Bertout, and Yves Sorel. "Latency analysis for data chains of real-time periodic tasks." In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA* (2018), pp. 360–367. ISSN: 19460759. DOI: 10.1109/ETFA.2018.8502498.

[230] Johannes Schlatow and Rolf Ernst. "Response-Time Analysis for Task Chains with Complex Precedence and Blocking Relations." In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017). ISSN: 1539-9087. DOI: 10.1145/3126505.

[231] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. "Period Optimization for Hard Real-Time Distributed Automotive Systems." In: *Proceedings of the 44th Annual Design Automation Conference.* DAC '07. San Diego, California: Association for Computing Machinery, 2007, pp. 278–283. ISBN: 9781595936271. DOI: 10.1145/1278480.1278553.

[232] Saad Mubeen, Thomas Nolte, Mikael Sjödin, John Lundbäck, and Kurt-Lennart Lundbäck. "Supporting Timing Analysis of Vehicular Embedded Systems through the Refinement of Timing Constraints." In: *Softw. Syst. Model.* 18.1 (Feb. 2019), pp. 39–69. ISSN: 1619-1366. DOI: 10.1007/s10270-017-0579-8.

[233] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Raj Rajkumar. "Fractional GPUs : Software-based Compute and Memory Bandwidth Reservation for GPUs." In: *Proceedings of the Real-Time and Embedded Technology and Applications Symposium.* RTAS. 2019, pp. 29–41. ISBN: 9781728106786. DOI: 10.1109/RTAS.2019.00011.

[234] Jalil Boudjadar and Simin Nadjm-Tehrani. "Schedulability and Memory Interference Analysis of Multicore Preemptive Real-time Systems." In: *Proceedings of the Int. Conference on Performance Engineering.* ICPE. L'Aquila, Italy: ACM, 2017, pp. 263–274. ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030233.

[235] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. "Deadline-Based Scheduling for GPU with Preemption Support." In: *2018 IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2018, pp. 119–130. DOI: 10.1109/RTSS.2018.00021.

[236] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed." In: *2017 IEEE Real-Time Systems Symposium (RTSS)* (2017), pp. 104–115. DOI: 10.1109/RTSS.2017.00017.

[237] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems." In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Ed. by Sebastian Altmeyer. Vol. 106. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 20:1–20:21. ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.20.

[238] Roberto Cavicchioli and Nicola Capodieci and Marko Bertogna. "Memory Interference Characterization Between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms." In: *Proceedings of the Int. Conference on Emerging Technologies and Factory Automation*. ETFA. 2017, pp. 1–10. DOI: 10.1109/ETFA.2017.8247615.

[239] J. Bakita, N. Otterness, James H. Anderson, and F. D. Smith. "Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs." In: *OSPERT* (2018), p. 49.

[240] D. Mukunoki, T. Imamura, and D. Takahashi. "Automatic Thread-Block Size Adjustment for Memory-Bound BLAS Kernels On GPUs." In: *IEEE 10th international Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)* (2016), pp. 377–384. DOI: 10.1109/MCSoC.2016.32.

[241] R. Lim, B. Norris, and A. Malony. "Autotuning GPU Kernels via Static and Predictive Analysis." In: *46th International Conference on Parallel Processing (ICPP)* (2017), pp. 523–532. DOI: 10.1109/ICPP.2017.61.

[242] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. "Understanding the Impact of CUDA Tuning Techniques for Fermi." In: *International Conference on High Performance Computing Simulation* (2011), pp. 631–639. DOI: 10.1109/HPCSim.2011.5999886.

[243] J. Kurzak, S. Tomov, and J. Dongarra. "Autotuning GEMM Kernels for The Fermi GPU." In: *IEEE Transactions on Parallel and Distributed Systems* 23.11 (Nov. 3), pp. 2045–2057. DOI: 10.1109/TPDS.2011.311.

[244] NVIDIA Corporation. *NVIDIA CUDA C: Best Practices*. Version 10.2.89 [Online] available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, visited: 03-06-2020. Nov. 2019.

[245] NVIDIA Corporation. *NVIDIA CUDA C: Programming Guide*. Version 10.2.89 [Online] available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, visited: 03-06-2020. Nov. 2019.

[246] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. *The Case for FIFO Real-time Scheduling*. Tech. rep. Online: http://hdl.handle.net/10993/24935. Fakultät für Angewandte Informatik, 2016, p. 12.

[247] Karthik S. Lakshmanan. "Scheduling and Synchronization for Multi-core Real-time Systems." PhD thesis. Carnegie Mellon University, 2011, p. 237.

[248] T.S. Rajesh Kumar, R. Govindarajan, and C.P. Ravikumar. "On-chip Memory Architecture Exploration Framework for DSP Processor-based Embedded System on Chip." In: *ACM Trans. Embed. Comput. Syst.* 11.1 (Apr. 2012), 5:1–5:25. ISSN: 1539-9087. DOI: 10.1145/2146417.2146422.

[249] Ernesto Wandeler and Lothar Thiele. *Real-Time Calculus (RTC) Toolbox*. Online at http://www.mpa.ethz.ch/Rtctoolbox. 2006.

[250] Jörn Schneider. "Why Current Memory Management Units are not Suited for Automotive ECUs." In: *Automotive - Safety & Security*. Vol. 210. LNI. GI, 2012, pp. 99–114.

[251] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. "Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective." In: *Proceedings of the 5th International Workshop on OpenMP*. IWOMP '09. DresdenG, Germany: Springer-Verlag, 2009, pp. 79–92. ISBN: 978-3-642-02284-5. DOI: 10.1007/978-3-642-02303-3_7.

[252] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport." In: *High Performance Computing - HiPC 2006*. 2006, pp. 338–352. ISBN: 978-3-540-68040-6. DOI: 10.1007/11945918_35.

[253] Oren Avissar, Rajeev Barua, and Dave Stewart. "Heterogeneous Memory Management for Embedded Systems." In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '01. Atlanta, Georgia, USA: ACM, 2001, pp. 34–43. ISBN: 1-58113-399-5. DOI: 10.1145/502217.502223.

[254] K.W. Tindell, H. Hansson, and A.J. Wellings. "Analysing Real-Time Communications: Controller Area Network (CAN)." In: *Proceedings 15th IEEE Real-Time Systems Symposium, San Juan, Puerto Rico*. 1994, pp. 259–265. DOI: 10.1109/REAL.1994.342710.

[255] Karthik Lakshmanan. "AUTOSAR Extensions for Predictable Task Synchronization in MultiCore ECUs." In: *Proc. of the SAE World Congress and Exhibition* (2011).

[256] Matthias Becker, Nima Khalilzad, Reinder J Bril, and Thomas Nolte. "Extended Support for Limited Preemption Fixed Priority Scheduling for OSEK / AUTOSAR-Compliant Operating Systems." In: *10th IEEE Int. Symp. on Industrial Embedded Systems* 2 (2015). DOI: 10.1109/SIES.2015.7185062.

[257] Sebastian Kehr, Milos Panic, Eduardo Quinones, Bert Boddeker, Jorge Becerril Sandoval, Jaume Abella, Francisco J Cazorla, and Gunter Schafer. "Supertask: Maximizing Runnable-Ievel Parallelism in AUTOSAR Applications." In: *Design, Automation & Test in Europe (DATE)*. 2016, pp. 25–30. ISBN: 9783981537079.

[258] Bryan C. Ward and James H. Anderson. "Supporting Nested Locking in Multiprocessor Real-Time Systems." In: *Proc. of the 24th Euromicro Conf. on Real-Time Systems*. IEEE Computer Society, 2012, pp. 223–232. DOI: 10.1109/ECRTS.2012.17.

[259] Martin Alfranseder, Michael Deubzer, Benjamin Justus, and Christian Siemers. "An Efficient Spin-Lock Based Multi-core Resource Sharing Protocol." In: *IEEE Int. Performance Computing and Communications Conf.* (2014), pp. 1–7. DOI: 10.1109/PCCC.2014.7017090.

[260] Björn B. Brandenburg and James H. Anderson. *The OMLP family of optimal multiprocessor real-time locking protocols*. 2012. DOI: 10.1007/s10617-012-9090-1.

[261] Maolin Yang and Alexander Wieder. "Global Real-Time Semaphore Protocols : A Survey , Unified Analysis , and Comparison." In: *Real-Time Systems Symp., 2015 IEEE*. San Antonio, TX: IEEE, 2015, pp. 1–12. DOI: 10.1109/RTSS.2015.8.

[262] Björn B. Brandenburg. "Improved Analysis and Evaluation of Real-time Semaphore Protocols for P-FP Scheduling." In: *Proc. of the 19th Real-Time and Embedded Technology and Applications Symp. (RTAS)*. IEEE Computer Society, 2013, pp. 141–152. ISBN: 978-1-4799-0186-9. DOI: 10.1109/RTAS.2013.6531087.

[263] Stefan Schmidhuber, Michael Deubzer, Ralph Mader, Michael Niemetz, and Jürgen Mottok. "Towards the Derivation of Guidelines for the Deployment of Real-Time Tasks on a Multicore Processor." In: *Model-Based Safety and Assessment*. Ed. by Frank Ortmeier and Antoine Rauzy. Cham: Springer International Publishing, 2014, pp. 152–165. ISBN: 978-3-319-12214-4.

[264] Andreas Sailer. "Reverse Engineering of Real-Time System Models from Event Trace Recordings." PhD thesis. University of Bamberg Press: Otto-Friedrich-Universität Bamberg, July 2019.

[265] Patrick Frey. "A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems Dissertation." PhD Thesis. Ulm University, 2010.

[266] J. H. Kim, I. Kang, S. Kang, and A. Boudjadar. "A Process Algebraic Approach to Resource-Parameterized Timing Analysis of Automotive Software Architectures." In: *IEEE Transactions on Industrial Informatics* 12.2 (2016), pp. 655–671. DOI: 10.1109/TII.2016.2527624.

[267] J. L. Szwarcfiter and P. E. Lauer. *Finding the Elementary Cycles of a Directed Graph in O(n + m) per Cycle*. Univ. of Newcastle upon Tyne, Newcastle upon Tyne, England, May 1974.

[268] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph." In: *SIAM J. Comput.* 4.1 (1975), pp. 77–84. URL: http://dblp.uni-trier.de/db/journals/siamcomp/siamcomp4.html#Johnson75.

[269] James C. Tiernan. "An Efficient Search Algorithm to Find the Elementary Circuits of a Graph." In: *Commun. ACM* 13.12 (Dec. 1970), pp. 722–726. ISSN: 0001-0782. DOI: 10.1145/362814.362819.

[270] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. "Scheduling for Reduced CPU Energy." In: *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*. OSDI '94. Monterey, California: USENIX Association, 1994. URL: http://dblp.uni-trier.de/db/conf/osdi/osdi94.html.

[271] K. Lakshmanan, D. d. Niz, R. Rajkumar, and G. Moreno. "Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems." In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. June 2010, pp. 169–178. DOI: 10.1109/ICDCS.2010.91.

[272] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. "Holistic Resource Allocation for Multicore Real-Time Systems." In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2019, pp. 345–356. DOI: 10.1109/RTAS.2019.00036.

[273] Khalid Omar Thabit. "Cache Management by the Compiler." PhD Thesis. Rice University, Houston, USA, 1982.

[274] Hamid Reza Faragardi, Kristian Sandstr, and Thomas Nolte. “An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-core Systems.” In: *7th Int. Symp. on Telecommunications.* 2014, pp. 41–48. DOI: 10.1109/ISTEL.2014.7000667.

[275] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits- A Design Perspective.* 2ed. Prentice Hall, 2004. ISBN: 8120322576.

G Affidavit (German)

Erklärung über die Eigenständigkeit der erbrachten wissenschaftlichen Leistung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Bei der Auswahl und Auswertung folgenden Materials haben mir die nachstehend aufgeführten Personen in der jeweils beschriebenen Weise unentgeltlich geholfen.

1. Junhyung Ki hat Teile der Event-Chain Analysen am AMALTHEA Modell implementiert.
2. The Bao Bui hat Teile der Blocking und Contention Analysen am AMALTHEA Modell implementiert.
3. Daniel Paredes Zevallos hat aus konzeptueller Sicht an Teilen des TX2RS Verfahren mitgewirkt, sowie rudimentäre Implementierungen am AMALTHEA Modell umgesetzt.

Weitere Personen waren an der inhaltlichen materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder andere Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

....	....
(Ort, Datum)	(Unterschrift)

$\mathcal{H}$ Content Appendices

H.1 Power and Performance Equations

Given Symmetric Multi Processors (SMPs), the IPC $\varkappa$ metric, the number of PUs $|\mathcal{P}|$, and a PU frequency f, the system performance can be defined by Equation H.1.

$$performance = |\mathcal{P}| \cdot IPC \cdot f \tag{H.1}$$

The time a program that requires IC instructions for being executed can then be estimated via Eq. H.2.

$$time_{IC} = \frac{IC}{|\mathcal{P}| \cdot f \cdot \varkappa} \tag{H.2}$$

Here, increasing the number of PUs is reflected in the denominator such that time performance is lower, i.e., better. Finally, the operation voltage V_{CC} must be taken into account for calculating the power a system requires as shown in the power approximation Equation H.3 [275, page 210], with x denoting the PU index.

$$power = \sum_{x} \left(Capacitance_x \cdot f_x \cdot Voltage^2\right) \tag{H.3}$$

When assuming linear frequency voltage relation, which is a very simplified estimation and means that increasing the frequency causes a voltage increase by the same factor, twice the amount of PUs results in twice the power and doubling the frequency results in eight times the original power. Reflecting the question on how to efficiently increase processing performance, adding PUs is more reasonable than increasing the frequency.

H.2 Formalism on Affinity and Delay Constraints

H.2.1 Pairing

$$\begin{aligned} \Phi^{r,P} = \{\mathcal{R}(\Phi^{r,P}), \mathcal{P}(\Phi^{r,P})\} \text{ with } \mathcal{R}(\Phi^{r,P}) \subseteq \mathcal{R}; \mathcal{P}(\Phi^{r,P}) \subseteq \mathcal{P}; \\ \forall\, r_a \in \mathcal{R}(\Phi^{r,P}) : M^{P}_{r_a} = x \text{ with } P_x \in \mathcal{P}(\Phi^{r,P}) \end{aligned} \tag{H.4}$$

$$\begin{aligned}\Phi^{\tau,P} = \{\mathcal{T}(\Phi^{\tau,P}), \mathcal{P}(\Phi^{\tau,P})\} \text{ with } \mathcal{T}(\Phi^{\tau,P}) \subseteq \mathcal{T}; \mathcal{P}(\Phi^{\tau,P}) \subseteq \mathcal{P};\\ \forall\ \tau_i \in \mathcal{T}(\Phi^{\tau,P}) : M^P_{\tau_i} = x \text{ with } P_x \in \mathcal{P}(\Phi^{\tau,P})\end{aligned} \tag{H.5}$$

H.2.2 Separation

$$\begin{aligned}\Phi^{r|P} = \{\mathcal{R}(\Phi^{r|P}), \mathcal{P}(\Phi^{r|P})\} \text{ with } \mathcal{R}(\Phi^{r|P}) \subseteq \mathcal{R}; \mathcal{P}(\Phi^{r|P}) \subseteq \mathcal{P};\\ \forall\ r_a \in \mathcal{R}(\Phi^{r|P}) : M^P_{r_a} \neq x \text{ with } P_x \in \mathcal{P}(\Phi^{r|P})\end{aligned} \tag{H.6}$$

$$\begin{aligned}\Phi^{\tau|P} = \{\mathcal{T}(\Phi^{\tau|P}), \mathcal{P}(\Phi^{\tau|P})\}\text{with } \mathcal{T}(\Phi_{\tau|P}) \subseteq \mathcal{T}; \mathcal{P}(\Phi^{\tau|P}) \subseteq \mathcal{P};\\ \forall\ \tau_i \in \mathcal{T}(\Phi^{\tau|P}) : M^P_{\tau_i} \neq x \text{ with } P_x \in \mathcal{P}(\Phi^{\tau|P})\end{aligned} \tag{H.7}$$

H.2.3 Pairing - No Target

$$\Phi^{r,pair} = \mathcal{R}(\Phi^{r,pair}) \subseteq \mathcal{R} \text{ with } \forall\ r_a \in \mathcal{R}(\Phi^{r,pair}) : M^\tau_{r_a} = i \tag{H.8}$$

$$\Phi^{\tau,pair} = \mathcal{T}(\Phi^{\tau,pair}) \subseteq \mathcal{T} \text{ with } \forall\ \tau_i \in \mathcal{T}(\Phi^{\tau,pair}) : M^P_{\tau_i} = x \tag{H.9}$$

H.2.4 Separation - No Target

E.g. $\Phi^{\tau|sep} = \{\{\tau_1, \tau_2\}, \{\tau_3, \tau_4\}, \{\tau_5\}\}$ must result in $\{M^P_1, M^P_2\} \neq \{M^P_3, M^P_4\} \neq \{M^P_5\}$. A possible result is $M^P_1 = 4, M^P_2 = 4, M^P_3 = 1, M^P_4 = 3, M^P_5 = 2$. To formally define this relationship, the indexes g, h are used in Eq. H.11 as a group indexes within the constraint. Runnable indexes a belong to groups g and runnable indexes b belong to groups h. $M^P_{a,g}$ denotes the allocation of runnable r_a of the g-th group in $\Phi^{r|sep}$.

$$\begin{aligned}\Phi^{r|sep} =& \{\mathcal{R}_g(\Phi^{r|sep}), \mathcal{R}_h(\Phi^{r|sep}), ...\}; \mathcal{R}_g(\Phi^{r|sep}) \cap \mathcal{R}_h(\Phi^{r|sep}) = \emptyset\\ &\text{with } \forall\ r_a \in \mathcal{R}_g(\Phi^{r|sep}) \subseteq \mathcal{R};\\ &\quad \forall\ r_b \in \mathcal{R}_h(\Phi^{r|sep}) \subseteq \mathcal{R} : M^\tau_{r_a} \neq M^\tau_{r_b}, g \neq h\end{aligned} \tag{H.10}$$

$$\begin{aligned}\Phi^{\tau|sep} =& \{\mathcal{T}_g(\Phi^{\tau|sep}), \mathcal{T}_h(\Phi^{\tau|sep})...\}; \mathcal{T}_g(\Phi^{\tau|sep}) \cap \mathcal{T}_h(\Phi^{\tau|sep}) = \emptyset\\ &\text{with } \forall\ \tau_i \in \mathcal{T}_g(\Phi^{\tau|sep});\\ &\quad \tau_j \in \mathcal{T}_h(\Phi^{\tau|sep}) : M^P_{\tau_i} \neq M^P_{\tau_j}; g \neq h\end{aligned} \tag{H.11}$$

H.2.5 Delay Constratins

$$\Phi_\Gamma = \{\tau_0, \tau_1, w\} : \begin{cases} \text{one-to-one :} & \forall\ z \text{ with } \tau_{0,z}\ \exists!\ \tau_1(w), \#z_0 = \#z_1 \\ \text{reaction :} & \forall\ z \text{ with } \tau_{0,z}\ \exists\ \tau_1(w) \\ \text{unique-reaction :} & \forall\ z \text{ with } \tau_{0,z}\ \exists!\ \tau_1(w), \#z_0 \geq \#z_1 \end{cases} \tag{H.12}$$

Here, a window defines a starting- and end point in time $w = \{t_{lower}, t_{upper}\}$, $\tau_i(w)$ denotes the occurrence of τ_i during window w, and $\#z_0$ represents the number of occurrences (jobs) of task τ_0. The above notation is based on task events $\tau_{i,z}$, i.e. specific occurrences (jobs) of task τ_i, but in general any modeled events of the Amalthea events model can be used. For the *one-to-one* delay constraint, the number of occurrences across the source and target for the specified time window must be the same ($\#z_0 = \#z_1$). The *reaction* delay constraint allows any under- or over-sampling as long as there exists at least one target occurrence within the window w. The *unique-reaction* constraint is a combination of *one-to-one* and *reaction*, but allows over-sampling, i.e. situations that the source occurs more often than the target $z_0 \geq z_1$.

H.3 Definitions on Paths, Cycles, and DAGs

A path is defined by a list of edges that are sequentially connected, i.e., an edge's source equals its predecessor's target (cf. Eq. H.13, first condition), no source or target exists more than once since that indicates a cycle (cf. Eq. H.13, second condition), no target exists for the entry (first) source among all edges (cf. Eq. H.13, third condition), and no source exists for the exit (last) target among all edges (cf. Eq. H.13, fourth condition).

$$\begin{aligned} path = \{e_1, ...\} \quad & \text{with } \forall\ i \in [1, |path| - 1]; k \in [1, |path|]; i \neq k; \forall\ \varphi \text{ with } e_\varphi \in (\mathcal{E} \setminus path) : \\ & e^t(i) = e^s(i+1)\ ;\ e^s(i) \neq e^s(k)\ ;\ e^t(i) \neq e^t(k) \\ & e^t(\varphi) \neq e^s(1)\ ;\ e^s(\varphi) \neq e^t(|path|)\ ;\ |path| \geq 2 \end{aligned} \tag{H.13}$$

A path can also be related to a PP pp_j such that $path$ is replaced with $path_{pp_j}$ and $\mathcal{E}$ with $\mathcal{E}_{pp_j}$ in Eq. H.13. $paths(r_a)$ denotes all paths that include r_a wither as source or target runnable. Given the path definition in Eq. H.13, the finite set of paths is defined by distinct path sets as shown in the following Eq. H.14.

$$\begin{aligned} & Paths = \{path_1, ...\} \\ & \text{with } \forall\ i, j \in [1, |Paths|]; i \neq j\ :\begin{cases} path_i \setminus path_j \neq \emptyset \text{ if } |path_i| \geq |path_j| \\ path_j \setminus path_i \neq \emptyset \text{ if } |path_j| \geq |path_i| \end{cases} \\ & Paths_{\mathcal{E}} \subseteq Paths \text{ with } \forall\ i \in [1, |Paths|] : \mathcal{E} \cap path_i \neq \emptyset \end{aligned} \tag{H.14}$$

As soon as any source or target vertex occurs more than once and not adjacently as source and target within a directed sequence of vertices, the sequence contains a loop, respectively cycle. This leads to the following definition of a cycle in Eq. H.15

$$\begin{aligned} cycle = \{e_1, ...\} \quad & \text{with } \forall\ i, j \in [1, |cycle| - 1]; i \neq j : \\ & |cycle| \geq 2\ ;\ e^t(i) = e^s(i+1); \\ & e^s(i) \neq e^s(j) \cap e^t(i) \neq e^t(j)\ ;\ e^t(|cycle|) = e^s(1) \end{aligned} \tag{H.15}$$

Given the cycle definition in Eq. H.15, the finite set of cycles is defined by distinct cycle sets as given in the following Eq. H.16.

$$\begin{aligned} & Cycles = \{cycle_1, ...\} \\ & \text{with } \forall\ i, j \in [1, |Cycles|]; i \neq j\ :\begin{cases} cycle_i \setminus cycle_j \neq \emptyset \text{ if } |cycle_i| \geq |cycle_j| \\ cycle_j \setminus cycle_i \neq \emptyset \text{ if } |cycle_j| \geq |cycle_i| \end{cases} \end{aligned} \tag{H.16}$$

Eq. H.17 gives the list of runnables contained in a path.

$$\mathcal{R}(path_i) = \left(\bigcup_{\varphi} e^s_\varphi \right) \cup e^t(i, |path_i|) \text{ with } e_\varphi \in path_i \tag{H.17}$$

Eq. H.18 defines a DAG based on Eq. H.17 and Eq. H.16.

$$\begin{aligned} DAG = \{\mathcal{R}, \mathcal{E}\} \quad & \text{with } \forall r_a \in \mathcal{R}; \forall e_\varphi \in \mathcal{E}; \forall i,j \in [1, |Paths_\mathcal{E}|] \;: \\ & r_a \in \left(e^s_\varphi \cup e^t_\varphi\right) \in \mathcal{R}(DAG) \;;\; Cycles_{DAG} = \emptyset; \\ & e_\varphi \in path_j \text{ with } j \geq 1; \\ & \mathcal{R}(path_i) \cap \mathcal{R}(Paths_\mathcal{E} \setminus path_i) \neq \emptyset \text{ with } Paths_\mathcal{E} = \textstyle\bigcup_i path_i \end{aligned} \tag{H.18}$$

H.4 Offset-based RTA using Transactions

This section's formulas are derived from [39]. To retrieve a task's response time under offset consideration, all transactions' interference are added to the tasks execution time as shown in Eq. H.19.

$$R^{+,\text{offs};(n+1)}_{i,x} = C^+_{i,x} + \sum_{\Gamma_d \in \Gamma} \left(W_d \left(\tau_i, R^{+,\text{offs};(\text{n})}_{i,x} \right) \right) \;:\; R^{+,\text{offs},0}_{i,x} = C^+_{i,x} \tag{H.19}$$

For a single transaction, the maximal interference over all higher priority tasks within the transaction Γ_d on a task τ_i is shown in Eq. H.20.

$$W_d(\tau_i, t) = \max_{h \in hp(\Gamma_d, \tau_i)} \left(W_{d,h}(\tau_i, t) \right) \tag{H.20}$$

The interference of the h-th task in transaction Γ_d on task τ_i at time t is shown in Eq. H.21. T_d denotes the common period of tasks in transaction Γ_d.

$$W_{d,h}(\tau_i, t) = \sum_{j \in hp(\Gamma_d, \tau_i)} \left(\left(\left\lfloor \frac{t^*_{t,d,j,h}}{T_d} \right\rfloor + 1 \right) \cdot C_{d,j} - x_{d,j,h}(t) \right) \tag{H.21}$$

Eq. H.21 introduces two additional parameters, i.e. $t^*_{t,d,j,h}$ and $x_{d,j,h}$, which are shown in Eq. H.22 and Eq. H.24, respectively.

$$t^*_{t,d,j,h} = t - phase(\tau_{d,j}, \tau_{d,h}) \tag{H.22}$$

The phasing of two tasks with the same period but different offsets using the modulo operation % is shown in Eq. H.23.

$$phase(\tau_{d,j}, \tau_{d,h}) = (T_d + (O_{d,j} - O_{d,h})) \;\%\; T_d \tag{H.23}$$

Lastly, Eq. H.24 gives the time value that task $\tau_{d,j}$ cannot execute until t due to τ_h as $x_{d,j,h}(t)$

$$x_{d,j,h}(t) = \begin{cases} 0 & \text{for } t^*_{t,d,j,h} < 0 \\ \max(0, C^+_{d,j,x} - (t^*_{t,d,j,h} \% T_d)) & \text{otherwise} \end{cases} \tag{H.24}$$

H.5 Runnable DAG of the Democar Model

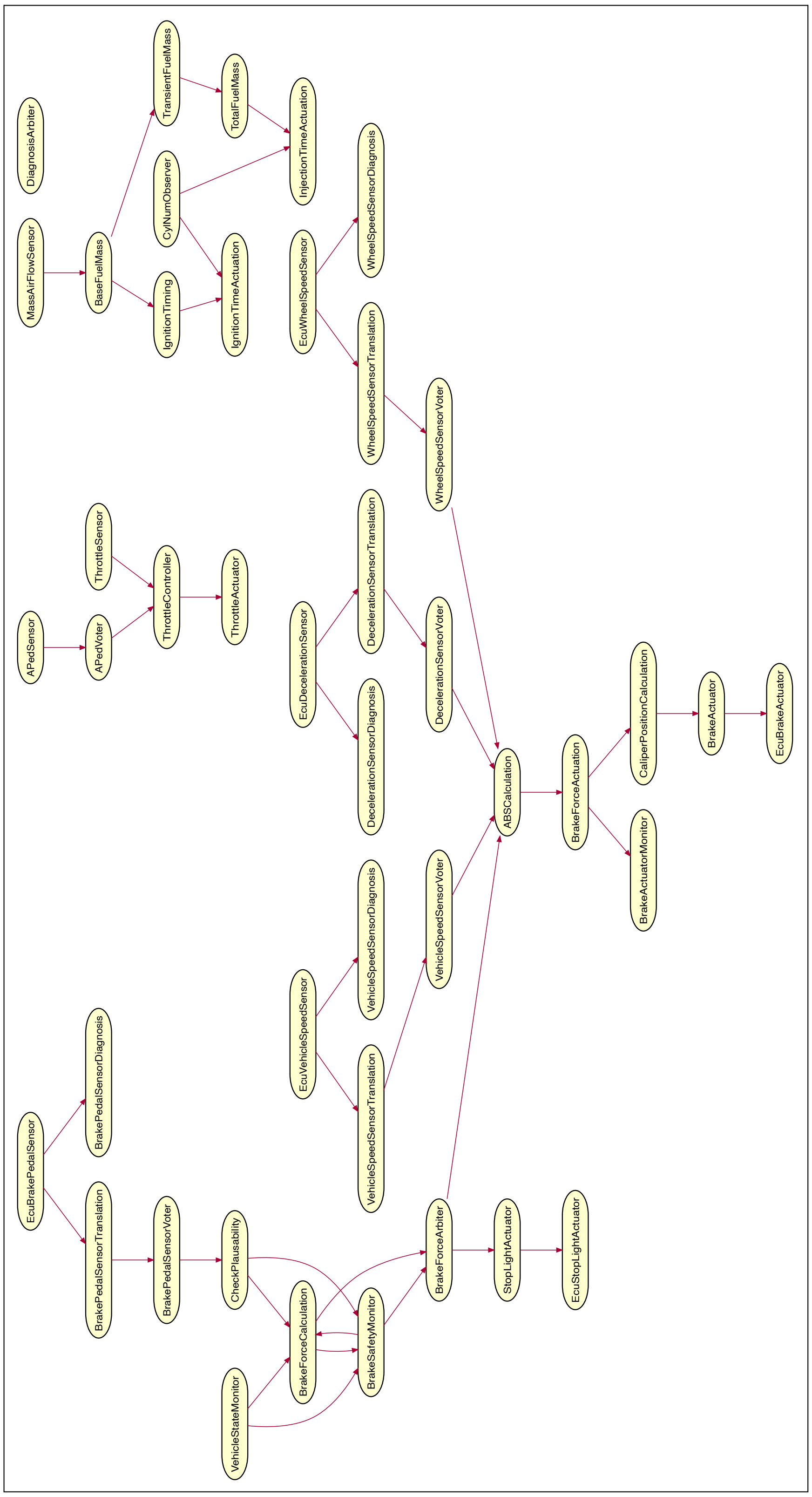
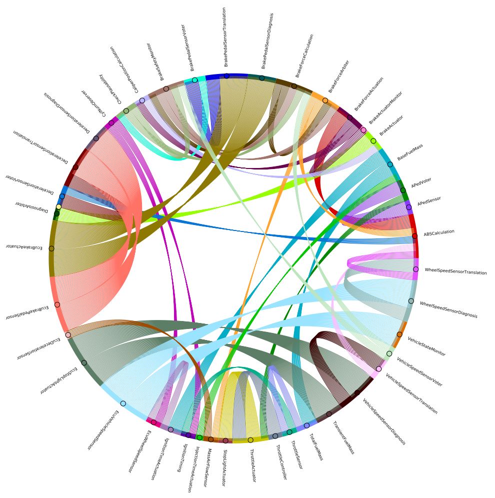
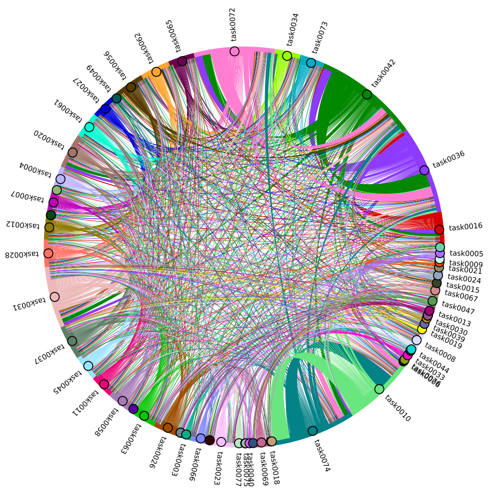


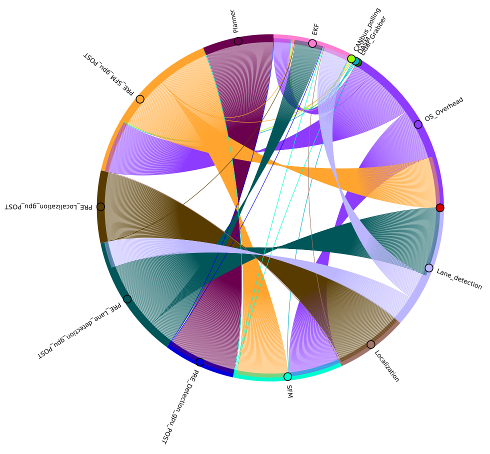
Table H.1: Democar runnable dependency graph

H.6 Model Chord Charts

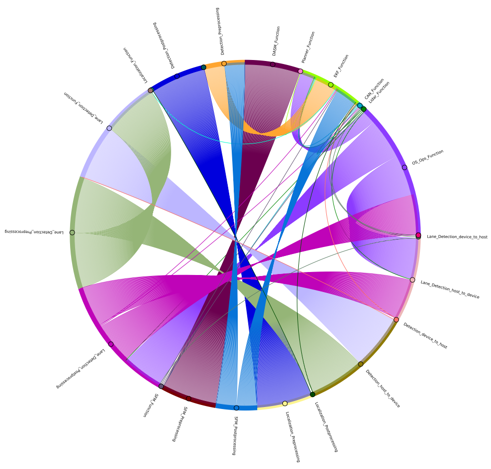

(a) Democar runnable dependencies

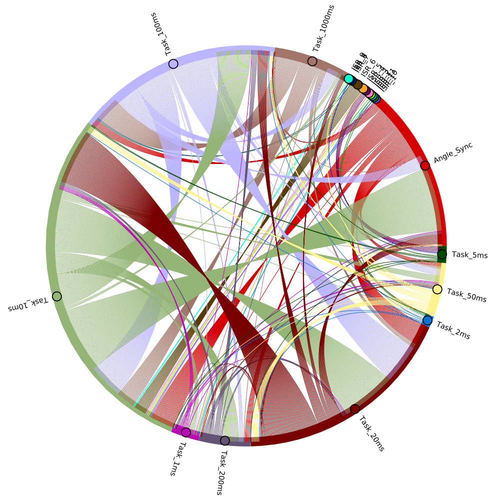


(b) AIM task dependencies



(c) WATERS19 task dependencies


(d) WATERS19 runnable dependencies



(e) FMTV task dependencies

Figure H.1: Chord charts of different models' task and runnable dependencies

H.7 Task Chain Delay Examples

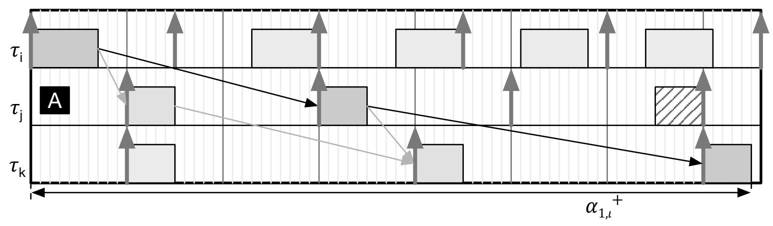


Figure H.2: Gantt chart on implicit worst-case task chain age delay example with increasing periods

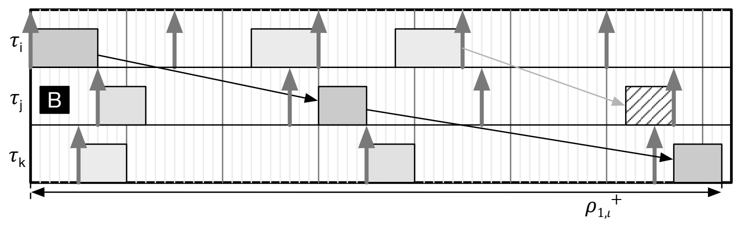


Figure H.3: Gantt chart on implicit worst-case task chain reaction delay example with increasing periods

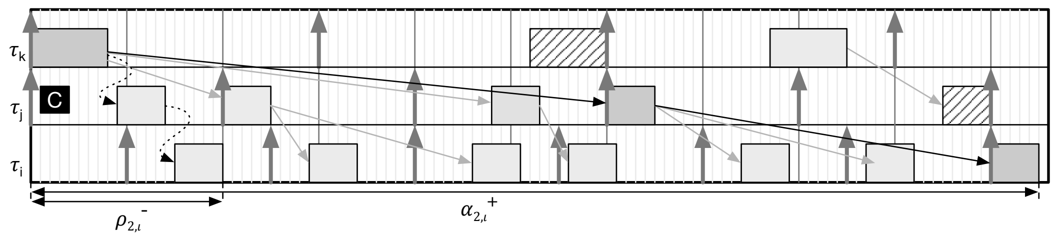


Figure H.4: Gantt chart on implicit worst-case task chain age delay and implicit best-case reaction with decreasing periods

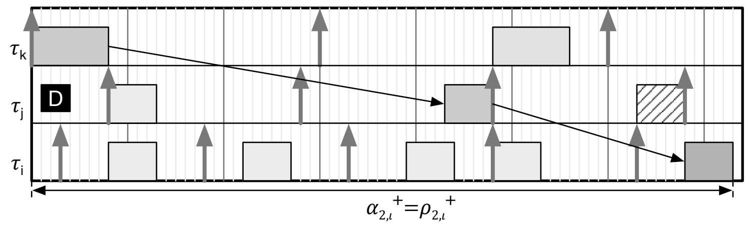


Figure H.5: Gantt chart on implicit worst-case task chain age and reaction delay example with decreasing periods

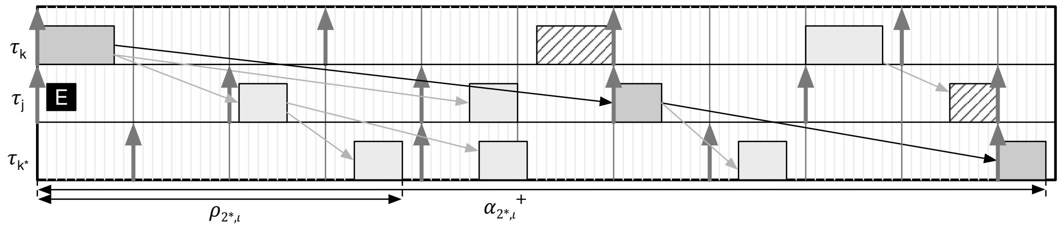


Figure H.6: Gantt chart on implicit worst-case task chain age delay example with alternating periods

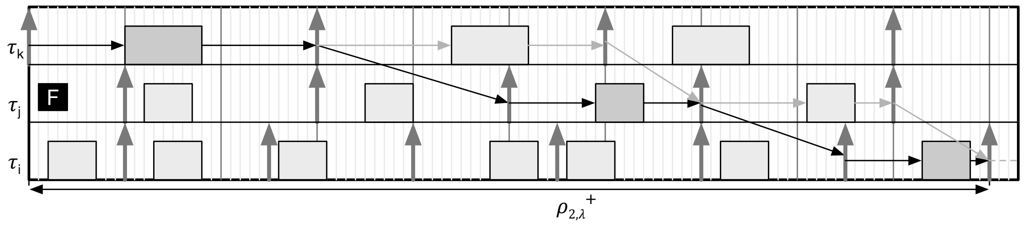


Figure H.7: Gantt chart on LET-based worst-case task chain reaction delay example with decreasing periods

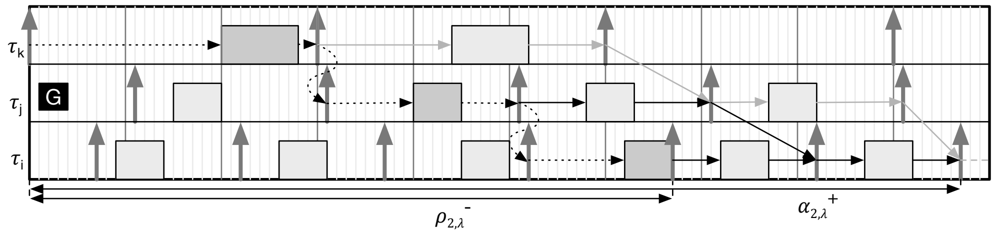


Figure H.8: Gantt chart on LET-based worst-case task chain age and best-case reaction delays example with decreasing periods

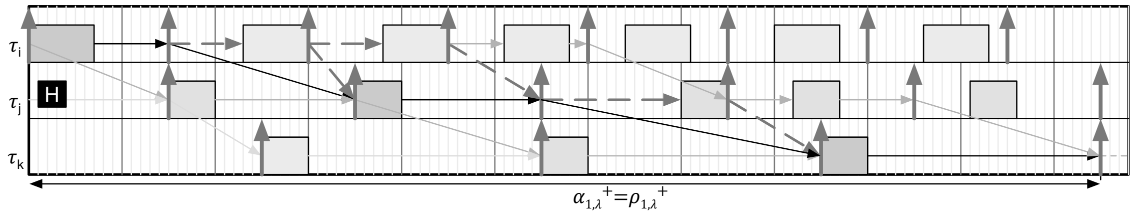


Figure H.9: Gantt chart on LET-based worst-case task chain age and reaction delays example with increasing periods

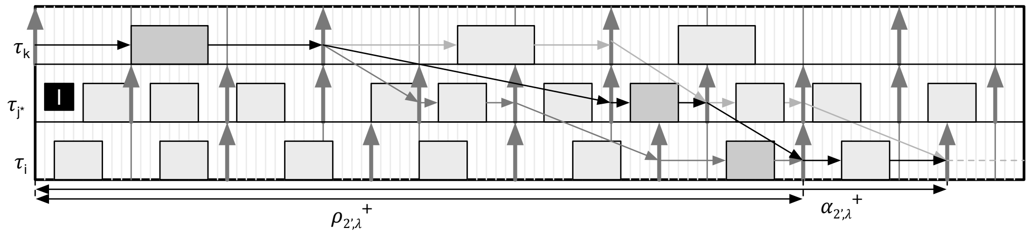


Figure H.10: Gantt Chart on LET-based worst-case task chain age and reaction delays example with alternating periods

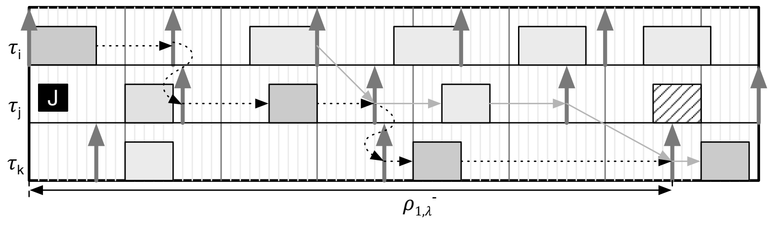


Figure H.11: Gantt Chart on LET-based best-case task chain reaction delay example with increasing periods

H.8 Various WCET Measurements for the WATERS Model

								Mapping $M^P_{\tau_i}$				
Task	T_i	$C^+_{i,ARM}$	$C^-_{i,ARM}$	$C^+_{i,Denver}$	$C^-_{i,Denver}$	$C^+_{i,GPU}$	$C^-_{i,GPU}$	Given	EV	RTSO	TCO	LBO
OSOverhead	100	50.00	50.00	50.00	50.00	-	-	Core0	Core5	Core2	Core2	Core2
LidarGrabber	33	13.66	10.16	10.87	9.79	-	-	Core1	Core4	Core3	Core4	Core1
DASM	5	1.86	1.30	1.30	1.05	-	-	Core0	Core1	Core5	Core5	Core3
CANbuspolling	10	0.60	0.40	0.60	0.40	-	-	Core0	Core0	Core3	Core1	Core3
EKF	15	4.76	3.98	4.43	4.09	-	-	Core4	Core1	Core2	Core2	Core3
Planner	15	13.24	9.62	12.44	9.54	-	-	Core3	Core0	Core1	Core0	Core0
PRESFMgpuPOST	33	7.90	6.33	6.71	5.41	-	-	Core0	Core1	Core5	Core5	Core1
PRELocalizationgpuPOST	400	17.64	7.34	14.52	6.12	-	-	Core0	Core0	Core5	Core0	Core5
PRELanedetectiongpuPOST	66	8.23	6.79	7.63	6.08	-	-	Core5	Core3	Core2	Core3	Core2
PREDetectiongpuPOST	200	4.71	4.01	4.09	3.00	-	-	Core5	Core0	Core1	Core4	Core1
SFM	33	29.50	24.14	27.81	22.18	7.90	7.05	GP10B	GP10B	Core0	Core1	GP10B
Localization	400	387.42	366.52	294.81	276.71	124.00	117.00	GP10B	Core3	GP10B	GP10B	Core5
Lanedetection	66	51.04	47.84	42.24	38.44	27.33	24.50	GP10B	Core2	Core4	Core3	Core4
Detection	200	-	-	-	-	116.00	108.00	GP10B	GP10B	GP10B	GP10B	GP10B

	Response times $R^+_{i,x}$				s-Blocking				pi-Blocking			
Task	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	50.00	73.94	73.93	50.00	-	-	-	-	-	-	-	-
LidarGrabber	23.50	25.31	22.41	13.06	0.9063	1.0782	0.5156	1.4688	-	-	-	-
DASM	1.30	1.87	1.87	1.87	0.0003	0.0003	0.0003	0.0003	-	-	-	-
CANbuspolling	0.60	0.60	0.60	2.47	0.0004	0.0011	0.0008	0.0011	0.0001	-	-	0.0003
EKF	7.04	4.79	4.79	9.13	0.0032	0.0039	0.0021	0.0046	-	-	-	-
Planner	14.52	13.36	13.36	13.32	0.2028	0.2133	0.6793	0.3019	0.0939	0.0938	0.0001	-
PRESFMgpuPOST	26.97	-	-	23.34	0.1084	0.2560	0.2560	0.1084	-	-	-	-
PRELocalizationgpuPOST	-	43.87	163.44	-	1.4085	1.0189	1.0182	0.3781	-	-	-	0.4688
PRELanedetectiongpuPOST	-	-	-	-	0.6251	0.6251	0.0000	0.6251	-	-	0.6250	-
PREDetectiongpuPOST	104.61	44.85	63.28	30.70	0.1432	0.1432	0.2370	0.2370	-	-	-	-
SFM	9.90	30.19	32.59	9.90	0.7560	1.8855	1.8855	0.7560	-	-	-	-
Localization	392.12	240.00	240.00	392.12	1.3151	1.8807	1.3164	0.3781	-	-	-	-
Lanedetection	56.05	56.05	56.05	56.05	1.2502	1.2502	0.0000	1.2502	-	-	-	-
Detection	123.90	173.00	173.00	123.90	0.7813	0.7813	1.8125	0.7813	-	-	-	-

Table H.2: WATERS mapping, response times, and blocking results in *ms*

	Contention				CE costs			
Task	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	-	-	-	-	-	-	-	-
LidarGrabber	8.7502	8.7502	7.6565	1.7500	-	-	-	-
DASM	0.0019	0.0090	0.0090	0.0090	-	-	-	-
CANbuspolling	0.0005	0.0029	0.0005	0.0026	-	-	-	-
EKF	0.0046	0.0259	0.0230	0.0230	-	-	-	-
Planner	0.7214	0.7615	0.7615	0.7214	-	-	-	-
PRESFMgpuPOST	2.8238	-	-	2.8238	-	-	-	-
PRELocalizationgpuPOST	-	8.4607	1.5981	-	-	-	-	-
PRELanedetectiongpuPOST	-	-	-	-	-	-	-	-
PREDetectiongpuPOST	-	-	12.0313	2.5781	-	-	-	-
SFM	-	1.8750	1.8750	-	2.7309	-	-	1.0121
Localization	4.2304	-	-	4.2304	-	2.5413	2.5413	-
Lanedetection	4.3756	4.3756	4.3756	4.3756	-	-	-	-
Detection	-	-	-	-	2.5757	2.2506	2.3151	1.3750

	Label Access Costs				$U_i^+ = \frac{R_i^+}{T_i}$			
Task	EV	RTSO	TCO	LBO	EV	RTSO	TCO	LBO
OSOverhead	-	-	-	-	0.500	0.739	0.739	0.500
LidarGrabber	1.0938	1.0938	1.0938	0.4375	0.712	0.767	0.679	0.396
DASM	0.0005	0.0013	0.0013	0.0013	0.260	0.374	0.374	0.374
CANbuspolling	0.0001	0.0003	0.0001	0.0003	0.060	0.060	0.060	0.247
EKF	0.0012	0.0029	0.0029	0.0029	0.469	0.319	0.319	0.609
Planner	0.1603	0.1603	0.1603	0.1603	0.968	0.891	0.891	0.888
PRESFMgpuPOST	0.7530	1.8825	1.8825	0.7530	0.817	-	-	0.707
PRELocalizationgpuPOST	0.3760	0.9401	0.3760	0.9401	-	0.110	0.409	-
PRELanedetectiongpuPOST	1.2502	1.2502	1.2502	1.2502	-	-	-	-
PREDetectiongpuPOST	0.6875	0.6875	1.7188	0.6875	0.523	0.224	0.316	0.153
SFM	0.2108	0.5061	0.5061	0.2108	0.300	0.915	0.988	0.300
Localization	0.9401	0.1567	0.1567	0.9401	0.980	0.600	0.600	0.980
Lanedetection	1.2502	1.2502	1.2502	1.2502	0.849	0.849	0.849	0.849
Detection	0.2864	0.2864	0.2864	0.2864	0.620	0.865	0.865	0.620

Table H.3: WATERS contention, CE costs, label access costs, and R_i^+/T_i results in ms

H.9 Example WRR Gantt Chart

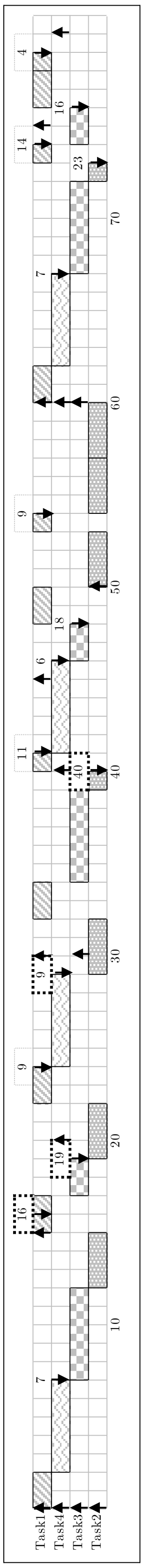


Figure H.12: Example Gantt chart on WRR scheduling based on [36] and the task set of table H.4. The entire busy period is shown, constituted by 79 instructions and 12 round robin turns, whereas ↑ denotes a task's arrival and ↓ denotes its response. Rectangles with a number inside located above a response arrow indicate the response time for the corrsponding task instance. Thick dashed response times indicate worst-case response times of a task within the busy period window.

Task	T_i	C_i	θ_i
Task1	3	15	2
Task2	10	50	3
Task3	7	30	5
Task4	5	20	5

Table H.4: Example task set based on [36]

H.10 Delay Equations Overview

Notation	Name	Considered properties	Focus	Used for	Eq.
e_c	Edge cost	Rate; nb accesses; label size; cache line; memory access cost	Edge i.e. two runnables	Partitioning; DAG; Cycle decomposition	4.6
cc_x	Communication cost	Mapping; label size; CAN Ids; busy period	CAN message	RTA; Intra-ECU communication	5.10
w_{CS}	Critical section window	PU memory access delay; label size; cache line; com. paradigm	Semaphore or label and a task	Blocking (RTA)	5.19
$L_i(com)$	Normalized access delay	Label size; cache line; memory access cost; number of accesses; com paradigm	Single task	RTA; Communication paradigms	5.30
$ipuc$	Inter PU communication	Accumulated label size and rate across labels shared between task pairs mapped to different PUs	Mapping	Memory Mapping	7.1

Table H.5: Overview of delays derived from labels and label accesses