

Jakob Matthias Stammler

Dynamic Multi-Mode Scheduling and Deployment with Early Exit Neural Network Integration

Dynamic Multi-Mode Scheduling and Deployment with Early Exit Neural Network Integration

Zur Erlangung des akademischen Grades eines

DOKTORS DER
INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

angenommene

Dissertation

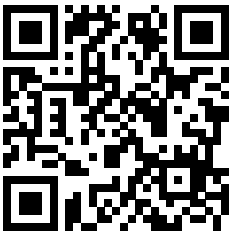
von

M.Sc.

Jakob Matthias Stammler

geboren in Heidelberg

Tag der mündlichen Prüfung:	06.10.2026
Hauptreferent:	Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent:	Prof. Arvind Easwaran



Dynamic Multi-Mode Scheduling and Deployment with Early Exit Neural Network Integration

First edition: October 2026
DOI: 10.5445/IR/1000197794
Copyright © Jakob Matthias Stammler, 2026

The latest digital edition of this work is available at
dx.doi.org/10.5445/IR/1000197794.

This page intentionally left blank

This page intentionally left blank

Abstract

Four trends can be observed in the architecture of modern automotive vehicles. These trends are the centralization of system architectures, the increasing software-centricity of the automotive domain, the growing use of hardware-abstraction layers to facilitate this shift, and the increasing deployment of machine learning-based methods. The evolution of hardware centralization is projected to progress from a distributed architecture through a domain-centralized architecture toward a vehicle-centralized architecture. In such architectures, domain controllers are used to manage automotive subsystems such as the drivetrain, the infotainment system, and body-control functions with increasing complexity. This architectural trend is the primary focus of this thesis. It is accompanied by increasingly powerful and complex hardware platforms, which require sophisticated software stacks to fully utilize their capabilities. These powerful hardware platforms also enable the deployment of machine learning-based methods. To exploit the multicore nature of the employed hardware platforms while simultaneously fulfilling the real-time requirements of the automotive domain, a safe scheduling method is required, which also satisfies functional requirements and provides Safety of the Intended Functionality guarantees to the system designer. Such a scheduling algorithm must be capable of scheduling task sets that include both machine learning-based methods and periodic tasks for the control functionality of domain-centric architectures. The underlying task model must support the modeling of complex execution platforms with peripherals, multiple cores, and hardware accelerators. Domain-centric architectures also require dynamic task execution when subsystems experience faults or anomalies or when functional components are activated or deactivated.

Existing work addresses subsets of these requirements, such as supporting machine learning-based tasks, providing a descriptive task model, or enabling dynamic execution. This thesis aims to combine these features in a holistic scheduling approach. To argue for safe execution, a correct-by-construction argument is employed within a time-triggered scheduling approach. This considers periodic tasks and models execution times as estimates which remain valid, provided that no delays, caused by contention for shared resources,

such as memory or peripherals, occur. Given a contention-free schedule, tasks complete within their estimated execution times. The integration of machine learning-based tasks into this scheduling framework is achieved through an approach that introduces a dedicated model, adds early exits to the used networks, and provides a scheduling algorithm capable of scheduling those early exits should constraint violations be predicted. The resulting networks are referred to as Anytime neural networks. Although this approach is designed for use within the time-triggered scheduling framework, a stand-alone version is also provided. The dynamic aspect of the otherwise static scheduling is introduced through a mode-change mechanism which enables mode changes during runtime with corresponding task sets and automatically generated transitions. Since the complexity of the resulting schedule graph grows exponentially with the number of required transitions, a method for reducing the number of schedules through the use of schedule variants is presented. In addition to user-defined mode changes, the mode-change mechanism can also be triggered by an automated anomaly-detection mechanism, which this approach automatically adds to the system.

The evaluation is conducted using two case studies that demonstrate the applicability of the proposed approaches to real-world scenarios. The first case study deploys two neural networks on a dedicated hardware accelerator. The second case study is based on the RoboRacer platform and deploys an autonomous-driving task set on both a Jetson Nano and an AMD KR260 platform. Generalizability is assessed through a component-wise evaluation of the scheduling approaches using synthetic task sets with varying parameters. These task sets are evaluated with respect to scheduling success rate and algorithm runtime. The highest runtime was observed for a task set of 100 tasks, a utilization of 100 %, a deadline slack of 0.0, and eight execution units, resulting in a runtime of approximately 100 min. As expected, runtime increases with the number of tasks. An increase in utilization, in the proportion of multi-version tasks, and the proportion of tasks with precedence constraints had limited effect. Because the most complex task set can be scheduled within a reasonable amount of time, the proposed scheduling approach is applicable to real-world scenarios. In addition, the Anytime neural network approach is evaluated with respect to its stand-alone applicability. This demonstrates that the proposed scheduler handles a temporary utilization increase of up to 250 % with a moderate reduction in accuracy.

This thesis introduces a novel and holistic scheduling concept, which supports scheduling machine learning-based tasks within periodic systems and establishes a correct-by-construction argument through a time-triggered scheduling framework with mode-change capabilities.

This page intentionally left blank

This page intentionally left blank

Kurzfassung

Vier zentrale Trends prägen die Architektur moderner Kraftfahrzeuge. Diese Trends sind die Zentralisierung der Fahrzeugarchitektur, der verstärkte Einsatz von Software-defined Vehicles, der steigende Einsatz von Hardware-Abstraction-Layern sowie die Nutzung von Machine-Learning-Methoden. Die Entwicklung des Trends zur Hardwarezentralisierung verläuft von einer verteilten Architektur über eine domänenzentrierte Architektur hin zu einer Fahrzeug-zentrierten Architektur. In der domänenzentralisierten Architektur werden Domain Controller eingesetzt, um Subsysteme wie den Antriebsstrang, das Infotainmentsystem oder die Karosserie zentralisiert zu steuern. Dieser Trend bilded das Umfeld dieser Dissertation. Leistungsfähigere und komplexere Hardwareplattformen fördern diesen Trend, was wiederum komplexere Software-Stacks erfordert und den Einsatz von Machine-Learning-Methoden ermöglicht. Um die Mehrkerneigenschaften der eingesetzten Plattformen auszunutzen und gleichzeitig die Anforderungen des Automotive-Bereichs zu erfüllen, wird ein Schedulingverfahren benötigt, welche funktionale Anforderungen berücksichtigt und dem Entwickler die Safety of the Intended Functionality gewährleistet. Ein solcher Algorithmus muss in der Lage sein, Task Sets zu schedulen, die sowohl Machine-Learning-Methoden als auch periodische Tasks für Steuerungsfunktionen zentralisierter Architekturen umfassen. Weiterhin muss das Modell Plattformen mit Peripherie, mehreren Kernen sowie Hardwarebeschleunigern unterstützen. Domain Controller weisen zudem dynamische Betriebsmodi auf, zwischen welchen bei Fehlern oder Anomalien oder bei Aktivieren von Komponenten umgeschaltet wird.

Bestehende Arbeiten betrachten jeweils nur einen Teil der genannten Anforderungen, etwa die Unterstützung von Machine-Learning-basierten Tasks, ein aussagekräftiges Task-Modell oder die dynamische Ausführung von Tasks. Diese Dissertation verfolgt das Ziel, diese Anforderungen in einem holistischen Ansatz zu vereinen. Zur Argumentation für eine sichere Ausführung wird innerhalb eines Time-Triggered-Scheduling-Ansatzes ein Correct-by-Construction-Argument verwendet. Der Algorithmus modelliert die Ausführungszeit unter der Annahme, dass keine Verzögerungen durch zeitgleiche Zugriffe auf Speicher oder Peripherie auftreten. Ist diese Annahme er-

füllt, werden die Tasks innerhalb ihrer vorhergesagten Ausführungszeit ausgeführt. Die Einbindung von Machine-Learning-Methoden in diesen Schedulingansatz erfolgt durch das Hinzufügen von Early Exits und einen Scheduling-Algorithmus, der diese Early Exits ausführt, sobald Constraint-Verletzungen vorhergesagt werden. Die resultierenden Neuronalen Netze werden dann als Anytime Neural Networks bezeichnet und können sowohl innerhalb des Time-Triggered-Verfahrens als auch eigenständig eingesetzt werden. Der dynamische Aspekt des eigentlich statischen Time-Triggered Schedulings wird durch einen Mode-Change-Mechanismus unterstützt, der die Ausführung verschiedener Task Sets zur Laufzeit über entsprechende Modi und zusätzlich generierte Übergangs-Schedules erlaubt. Da die Komplexität des resultierenden Schedule-Graphen mit der Anzahl der Modiübergänge exponentiell steigt, wird ein Verfahren zur Reduktion der Scheduleanzahl mittels Varianten bereits generierter Schedules eingeführt. Neben benutzerdefinierten Modiübergängen kann der Mode-Change-Mechanismus auch durch einen Anomaly-Detection-Mechanismus ausgelöst werden, der dem System automatisch zur Designzeit hinzugefügt wird.

Die Evaluierung erfolgt durch zwei Use Cases. Der erste enthält zwei neuronale Netze, die auf einem dedizierten Hardwarebeschleuniger in einem Mehrkernprozessor ausgeführt werden. Der zweite Use Case basiert auf einer RoboRacer-Plattform und schedult ein Task Set für autonomes Fahren sowohl auf einem Jetson Nano als auch auf einem AMD KR260. Die Evaluation der Generalisierbarkeit erfolgt komponentenweise. Synthetische Task Sets mit variierenden Parametern werden gescheduled und der Algorithmus hinsichtlich der Erfolgsrate sowie der Laufzeit evaluiert. Die höchste Laufzeit liegt bei 100 min für 100 Tasks, eine Auslastung von 100 %, einen Deadline-Slack von 0,0 sowie acht Execution Units. Dieses komplexeste Task Set wurde innerhalb einer angemessenen Zeit gescheduled und zeigt, dass der Ansatz für reale Anwendungsszenarien geeignet ist. Zusätzlich wird der Anytime-Neural-Network-Ansatz hinsichtlich seiner eigenständigen Einsetzbarkeit evaluiert. Die Ergebnisse zeigen, dass der vorgestellte Scheduler einen temporären Anstieg der Utilization auf 250 % bei einer Absenkung der Accuracy auf das Niveau des frühesten Early Exits bewältigen kann.

Diese Dissertation präsentiert einen holistischen Schedulingansatz, der Machine-Learning-Methoden, periodische Tasks und dynamische Betriebsmodi vereint sowie einen Correct-by-Construction-Ansatz für die sichere Ausführung von Tasks mittels Time-Triggered Scheduling mit Mode-Change-Mechanismus unterstützt.

This page intentionally left blank

This page intentionally left blank

Acknowledgements

While this journey is a team marathon, everybody has to run their own race, and individual support for everyone is essential in addition to the team spirit. I would like to express my deepest gratitude to all who have supported me throughout this journey.

First, I would like to thank my main supervisor, Prof. Dr.-Ing. Dr. h. c. Jürgen Becker, for the opportunity to work in his research group and to complete this doctoral thesis under his guidance. The amount of freedom and trust he places in his doctoral students is exceptional, and I think that I was given even more opportunities and trust than most. When traveling to conferences and project meetings, I knew that I could always rely on his support when representing the research group. I would also like to thank him for the freedom to choose and shape the research topic and direction of this thesis rather independently, which allowed me to explore my own ideas and interests. I am grateful for the many discussions we had, which helped me develop my own ideas and the skills to express them. Second, I would like to thank my co-supervisor, Prof. Arvind Easwaran, for the invitation and the opportunity to visit his research group at Nanyang Technological University in Singapore. While Jürgen allowed me to explore my own ideas and gave me many freedoms, Arvind guided me toward a thorough and structured approach to research during my stay in Singapore. His feedback on and critique of my work and this thesis helped me greatly improve the quality of my research as well as my writing. With this thesis, I am happy to have been able to combine both approaches, and I would like to thank both of them for their individual support of me personally as well as of my work.

While in Singapore, I was also able to meet and work with the Cyber-Physical Systems Research Group of Arvind, and I would like to thank the PhD students for their support and the discussions we had. I would like to thank Gao Chuanchao, Mohit Prashant, Subrat Prasad Panda, Eduardo de Conto, Xinwei Gao, Manoj Kumar Lenka, Soumya Ranjan Sahoo, Michael Yuhas, and all others for their support and for the great time they gave me during my stay in Singapore. I wish them all the best, wherever in the world their research and lives may take

them. Having written a PhD thesis with Arvind prepares them for all the challenges they will face in their academic careers.

A warm thank-you goes to my colleagues in my research group at home and in the sister group at FZI, where Jürgen's team is a great place to work, discuss ideas, and collaborate. A special thank-you is warranted for all my predecessors, on whose experience I could rely and who helped me find my place in the research group: Dr.-Ing. Leonard Masing, Dr.-Ing. Steffen Bähr, Dr.-Ing. Victor Pazmino, Dr.-Ing. Nidhi Anantharajaiah, Dr.-Ing. Tim Hotfilter, Dr.-Ing. Florian Schade, Dr.-Ing. Fabian Kempf, Dr.-Ing. Johannes Pfau, Dr.-Ing. Tobias Dörr, Dr.-Ing. Fabian Kreß, Dr.-Ing. Fabian Lesniak, Dr.-Ing. Vladimir Sidorenko, Dr.-Ing. Marc Neu, and Dr.-Ing. Julian Höfer. When hearing the words of thanks from my predecessors, I always hoped to have supported them as much as they had impacted me. With this dissertation, I happily pass the torch to my successors and also extend a great thank-you for their support and for the opportunity to hopefully pass some of my experience on to them. These are Alexey Serdyuk, Annina Gutermann, Christian Karle, Felix Frombach, Foivos Paraskevas, Hella Toto Kiesa, Henrik Scheidt, Iuliia Topko, Jona Scholz, Kai Unger, Patrick Schmidt, and Georgios Sotiropoulos as well as Brian Pachideh, Sven Nitzsche, Marius Kreutzer, Maximilian Kirschner and Konstantin Dudzik. Without the help of Dr.-Ing. Tanja Harbaum and Dr.-Ing. Jens Becker as both team lead and administrative support, the research group would not be able to function, and I would like to thank them for their support and assistance with all administrative and postdoctoral matters.

The greatest thanks go to my family, who supported me throughout this journey and who always believed in me. I would like to thank my mother, Bettina Bachmann-Frank, and her husband, Dr. Oliver Frank, for their emotional support during this journey and beyond. I would like to thank my father, Dr. Gerd Stammler, for his support, as well as for the many times he shared his opinion on research matters, despite having completed his doctorate in a very different field, and his wife, Claudia Malkmus-Stammler. Many thanks go to my sister, Julia Stammler, who always listened to my complaints and who always had a place for me to stay when I was traveling to Munich for meetings. The greatest thanks of all go to my girlfriend, An Phan, who was there for me throughout the entire PhD journey, from Karlsruhe to Singapore to Heidelberg, and who supports me in all matters, both private and professional. I would like to thank her for her support and her love, which helped me finish this dissertation and handle all the challenges that I have happily taken on in the past and will continue to take on in the future.

This page intentionally left blank

This page intentionally left blank

Contents

Abstract . . . . . **i**

Kurzfassung . . . . . **v**

Acknowledgements . . . . . **ix**

1 Introduction **1**
1.1 Execution Unit Centralization 1
1.2 Software Defined Vehicle 4
1.3 Hardware Abstraction Layer 6
1.4 Motivation 8
1.5 Used Example 10
1.6 Problem Formulation and Thesis Structure 13

2 Fundamentals **17**
2.1 Multi-Core and Many-Core Systems 17
2.1.1 Components of Multi-Processor Systems on Chip . . 17
2.1.2 Contention and Phased Execution Models 18
2.1.3 Isolation of Access towards Shared Resources 20
2.2 Artificial Neural Networks 20
2.2.1 Artificial and Convolutional Neural Networks 20
2.2.2 Hardware Accelerators for Artificial Neural Networks . 26
2.2.3 Structure of Anytime Neural Networks 29
2.2.4 Safety and Reliability Concerns in Neural Networks . 32
2.3 Real-Time Scheduling Approaches and Task Design Concepts 33
2.3.1 General Terms for Real-Time Tasks and Scheduling . . 33
2.3.2 Event-Triggered Scheduling Concepts 34
2.3.3 Time-Triggered Scheduling Concepts 36
2.3.4 Hypervisors and Real-Time Operating Systems 38
2.3.5 Task Design Concept and Implementation Guidelines 40

3 State of the Art 43
3.1 Time-Triggered and Mode Change Scheduler 43
3.1.1 Schedulers for Mitigating Shared Resource Contention 43
3.1.2 Mode-Change Approaches for Embedded Systems . . 47
3.2 Anytime Neural Network Scheduling and Design 50
3.2.1 Design of Anytime Neural Networks 50
3.2.2 Deadline-aware Scheduling of Neural Networks . . . 53
3.3 Anomaly Detection in ML-based Systems 56

4 Time-Triggered Mode-Change Scheduler 59
4.1 Model for Time-Triggered and Multi-Version Tasks 59
4.2 Time-Triggered Scheduling Approaches for Periodic Tasks . . 71
4.2.1 Polyhedron-based Time-Triggered Scheduling 72
4.2.2 Simulated Annealing Time-Triggered Scheduling . . . 79
4.3 Mode-Change Task Scheduler Extensions 89
4.3.1 Model Extensions to facilitate Mode-Changes 89
4.3.2 Integration of Mode-Change Scheduling 94
4.3.3 Schedule Adaptation to Reduce Schedule Complexity 102
4.4 Anomaly Detection Components to Trigger Mode Changes . 107
4.4.1 Hardware-based Anomaly Detection Components . . 108
4.4.2 Software Components for Anomaly Detection 118

5 Dynamic Scheduler for Anytime Neural Networks 131
5.1 Anytime Neural Network Model and Constraints 131
5.2 Exit Addition to Existing Backbone Networks 138
5.2.1 Positioning of Early Exits Start Layers 138
5.2.2 Training Strategies for Added Early Exits 143
5.2.3 Testing and Evaluation of the Added Early Exits 146
5.3 Addition of Anomaly Detection to ANNs 147
5.3.1 Layer Tracing for Anomaly Detection Dataset Creation 149
5.3.2 Configuration of Added Anomaly Detection Modules 150
5.4 Runtime Estimation Strategies 159
5.4.1 Design-Time Runtime Estimation Strategies 159
5.4.2 Online Runtime Estimation During Execution 162
5.5 Added Checkpoints for Mid-Term Scheduling of ANNs 164
5.5.1 Component Structure of the Checkpoints 164
5.5.2 Mid-Term Scheduling Algorithm for Exit Choice . . . 165
5.6 Scheduling Algorithm for ANNs with Checkpoints 167
5.6.1 Task Release Loop as the Mid-Term Scheduling 167
5.6.2 Checkpoint Loop as the Short-Term Scheduler 169

6 System Integration of Time-Triggered and ANN Scheduling . . . 173
6.1 Model Representation of Anytime Neural Networks 173
6.2 Scheduling Early Exits as Mode Changes 181
6.3 Runtime Environment and Platform Deployment 185
6.3.1 Abstract Deployment by Petri net Generation 185
6.3.2 Dedicated Management Cores 190
6.3.3 Real-time Operating System-based Deployment . . . 197

7 Evaluation 203
7.1 Ablation Studies 203
7.1.1 Safety Factor for ANN Runtime Estimation 204
7.1.2 Weight Distribution for ANN Runtime Estimation . . 209
7.1.3 Parameters of the Simulated Annealing Scheduler . . 213
7.2 Scheduling Success Rate of Polyhedron-Based Approach . . . 217
7.3 Runtime of Time-Triggered Scheduling Algorithm 220
7.3.1 Runtime of Polyhedron-based Scheduling Approach . 221
7.3.2 Runtime Evaluation for Different Starting Solutions . 223
7.3.3 Runtime of Simulated Annealing-based Scheduling . 225
7.3.4 Evaluation of Cost Function Characteristic 233
7.4 Accuracy, Equivalence and Placements of ANN Components 236
7.4.1 Placement Evaluation of Added Early Exits 236
7.4.2 Accuracy and Equivalence of Added Early Exits 239
7.4.3 Accuracy and Runtime of Anomaly Detection 240
7.5 Runtime Overhead of ANN Checkpoints 246
7.6 Anytime Neural Network Behavior with Utilization Increase . 248
7.6.1 Chosen Exit for Utilization Increase 249
7.6.2 Accuracy Drop for Utilization Increase 251
7.6.3 Deadline Success Rate for Utilization Increase 253

8 Case Studies 257
8.1 Deployment of ML-based Tasks on FleXNNgine-based HESoC 257
8.1.1 Motivation of the FleXNNgine-featuring Case Study . 258
8.1.2 Used Execution Platform and ML-based Task Set . . . 258
8.1.3 Modeling and Runtime Estimation of Case Study Tasks 259
8.1.4 Multi-Mode Scheduling of the Case Study Task Set . . 260
8.1.5 Demonstrator Deployment of the Case Study Task Set 260
8.2 RoboRacer Task Set as a Collaboration in HAL4SDV 261
8.2.1 Context of the Case Study in the HAL4SDV Project . . 262
8.2.2 RoboRacer Tasks Description and Execution Platform 262
8.2.3 Deployment onto the Execution Platform 265

9 Conclusion and Outlook 269
9.1 Summary and Novelty of the Proposed Methods 269
9.1.1 Summary of Multi-Version Time-Triggered Scheduling 270
9.1.2 Novelty of Multi-Version Time-Triggered Scheduling . 271
9.1.3 Summary of the Checkpoint-based ANN Scheduler . 274
9.1.4 Novelty of the Checkpoint-based ANN Scheduler . . . 276
9.2 Future Work 278

Bibliography 281

Publications 299

Student Theses 303

Figures 305

Tables 309

Glossary and Acronyms 313

This page intentionally left blank

This page intentionally left blank

Chapter 1

Introduction

This work is positioned in the context of the trend of centralization of software functionality inside modern automotive vehicles. The trend towards central car servers (CECASs), software defined vehicles (SDVs) and hardware abstraction layers (HALs) is researched in European-funded projects [1] as well as federally funded projects [2, Sch24a] and provides the main motivation. This chapter introduces this motivation in Sections 1.1 to 1.3. Section 1.4 condenses the findings into the motivation. This thesis uses one example to guide the reader through the chapters, which is introduced in Section 1.5. Section 1.6 introduces the problem statement and gives an outline of the rest of this thesis.

1.1 Execution Unit Centralization

Studies predicted a shift in automotive electric / electronic architecture and continue this prediction into the future. This shift is organized into five generations [3]. Past E/E architectures of generation one and two consisted of one electronic control unit (ECU) per assigned dedicated functionality in the domains of e.g. chassis, powertrain, infotainment or comfort. In generation two, a loose collaboration within one functional domain exists. ECUs communicate via dedicated bus systems, but communication between functional domains was limited. The third generation introduced stronger collaboration between functional domains and a central gateway with cross-functional connections, allowing the vehicle to handle complex functions like adaptive cruise control. The electrical/electronic (E/E) architecture of the first three generations are classified as distributed architectures [3]. The fourth generation of the E/E architecture is characterized as domain centralized. Here, a central domain control unit (DCU) consolidates functionality, which was previously executed on multiple dedicated ECUs. Each DCU is connected to a central

communication gateway and acts as the bridge between localized ECUs. This fourth generation is classified as domain centralized. Studies position the current state of the E/E architecture at the transition between generation three and four, i.e. at the transition between distributed and domain centralized architectures [3]. The forth generation and the shift towards DCUs is researched in federally funded projects like EMDRIVE [Sch24a]. The fifth generation is expected to remove dedicated ECUs and DCUs in favor of centralized high performance execution platforms. Here, functional domains are expected to be realized virtually instead of on dedicated execution units. Functionality is expected to be moved to such centralized compute units. Actors and sensors are expected to be connected via a time-sensitive networking (TSN) backbone to the central compute unit [4]. Additionally, ECUs, DCUs as well as communication systems are projected to be increasingly standardized, with a shift to software defined functionality making use of standardized application programming interfaces (APIs) [3]. Figure 1.1 shows examples for this trend. Changes between generations are marked red, blue components mark consistent components. In the fifth generation, compute units are expected to be consolidated into high performance heterogeneous systems-on-chip (HESoCs) [5]. These are referred to in multiple research projects as central car servers (CECASs) [2, 6]. This shift towards CECASs is driven by multiple factors, which fit into two categories: cost optimization and technical requirements.

Cost optimizations are projected to be observable by consolidating ECU into DCU and later into CECAS reduces the size and weight of necessary communication hardware in the form of wiring harnessed and is projected to result in a decrease in the cost per vehicle in the B segment from 894 $ in 2025 to 822 $ in 2030 for execution units. For premium cars the same cost is expected to decrease from 3,371 $ in 2025 to 3,148 $ in 2030 and for purpose-built autonomous vehicles this cost is expected to decrease from 8,508 $ to 6,227 $ [3].

This cost reduction is achieved by reducing the complexity of software via a clear functional split between software modules, which in turn reduces the development and validation cost and improves upgradability of software components [3]. Additionally, vehicle variants for different market segments increasingly share software as well as comfort functions. Driversand rely on automated support provided by advanced driver assistance systems (ADAS). This trend advances the emergence of software defined vehicles (SDVs), which are introduced in Section 1.2. Producing dedicated ECUs for each vehicle variant would increase the development and validation cost. Here centralizing the E/E architecture from ECUs into DCUs allows for a streamlined development

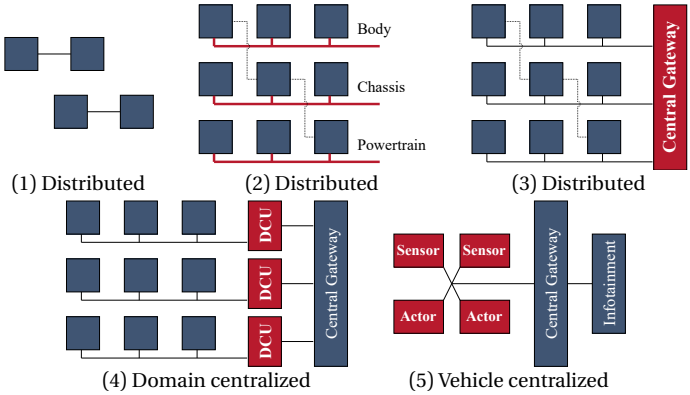


Figure 1.1: Overview of architectures in the shift towards a centralized car architecture. Changes between architectures are marked in blue. Consisted parts are marked blue, changes are marked red. Gen. 1 to Gen. 3 are classified as distributed architectures, Gen. 4 is classified as domain centralized and Gen. 5 is classified as centralized. Adapted from [3].

process and decreased number of software releases [5]. For an abstracted development of automotive applications as well as a unified deployment, a hardware abstraction layer (HAL) is projected to be deployed in CECASs on future SDVs to fulfill the below stated technical requirements. HALs are introduced in Section 1.3.

Technical requirements for more centralized compute units like DCUs are motivated largely by the executed software [3, 5] and accompanying standards like ISO 26262-5 [7] or ISO 26262-6 [8]. First, an increase in tasks employing machine learning (ML)-based methods for image classification [9], driver fatigue detection [10] or traffic sign transcriptions [11] necessitates the deployment of dedicated hardware components, able to execute this functionality fast and efficiently [5]. To facilitate this, an increase in the use of dedicated hardware accelerators is observable [12, 13]. Including these accelerators in CECASs also tightly integrates ML-based functionality with more traditional software and allows for a tight coupling of software functionality. Second, advanced driver assistance systems (ADASs) and self-driving functionality are projected to require a computing power of up to 10 TOPS. Sources call this embedded high performance computing (EHPC) and necessitate CECASs as execution units [14].

ISO 26262-5 guides the product development at the hardware level and spec-

ifies requirements for the hardware design or the specification of hardware safety requirements [7]. Parallel, ISO 26262-6 guides the software development and introduces requirements for the software architectural design, the software unit design and implementation and software integration and verification among others [8]. Additionally, the ISO 26262 standard introduces Automotive Safety Integrity Level (ASIL) and each software application is assigned an ASIL level from ASIL-A (e.g. non-breaking tail light) to ASIL-D (e.g. electric power steering) depending on the safety assurance requirements of the component [15]. With the centralization of the automotive E/E architecture, control software for components with different ASIL levels are increasingly executed on the same DCU or CECAS, resulting in mixed-criticality systems. A mixed-criticality system executes applications with different criticalities (e.g. ASIL levels) or real-time requirements on the same execution platform [16]. This necessitates isolation mechanisms between applications to, as per ISO 26262-1, eliminate "cascading failures between two or more elements that could lead to the violation of a safety requirement" [15]. ISO 26262-6 lists three interference categories [8]: 1. Timing and execution; 2. Memory; and 3. Exchange of information. This work focuses on eliminating interference of category 1) between applications by scheduling these applications to prevent simultaneous execution or access towards shared resources.

1.2 Software Defined Vehicle

A Software defined vehicle (SDV) contrasts itself from standard vehicles by a focus of the design on the software experience. This necessitates a hardware design, which is focussed on providing an adequate execution platform for the given software [17]. Software defined vehicles (SDVs) are characterized by the shift of the development life cycle towards methods native to software development. This includes a software architecture which is deployed on a centralized execution platform. Additionally, the support for over-the-air (OTA) updates is characteristic for SDVs [3, 17]. This decoupling of software and hardware architecture leads to a whitening of the functional black box [17], where the tight coupling of software functionality to electronic control units (ECUs) is weakened in favor of a more transparent software and hardware architecture [17]. This allows original equipment manufacturers (OEMs) to control the E/E architecture and architecture of the central car servers (CECASs), because of the independence from suppliers in terms of ECUs.

Automotive software can then to be sourced from more diverse suppliers. This

allows for a diversification of the marked beyond established tier-1 suppliers and allows smaller software suppliers easier access to the market. Because of the separation of software and hardware development, software portability is necessary for business development and profitability [3]. Integrating software of different suppliers and facilitating communication between software components in one centralized platform necessitates isolation and scheduling mechanisms [8]. This isolation and scheduling is facilitated via virtualization by hypervisors [18] or real-time operating systems [19]. Furthermore, software portability must be supported by hypervisors and real-time operating systems by providing APIs and interfaces. A software development, which is characterized by analysts as a *smartphone on wheels*, where the centralized hardware platform is developed by the OEM and further software features are brought in from third companies, is the result of these trends. This industry shift and OTA updates further necessitate a specialized life cycle management for software defined vehicles [20], turning the development of software defined vehicles into a continuous process, where software is continuously updated and improved over the lifetime of the vehicle. This moves the development of automotive software away from past development modes using *waterfall* or *V-model* development processes towards a more agile development process, which is characterized by continuous integration and continuous deployment [20].

In addition to OTA updates, software defined vehicles are expected to support the integration of cloud availability and cloud offloading of software functionality [3]. This offloading of software functionality to cloud servers is expected to be used for non-critical software functionality, because of the potential for higher latency, lower availability as well as privacy concerns [3]. Additionally, the offloading of software functionality to cloud servers is expected to be most beneficial for ML-based tasks, since the execution of ML-based tasks is expected to be more efficient on cloud servers, because of the higher computational power and the ability to use more powerful hardware accelerators [3].

The change towards a software focus also includes the requirements from users to be served with features previously known from software products and from personal smartphones. This includes the ability to install and uninstall software features, the ability to update software features and the ability to customize software features [20]. These software features are expected to include ML-based features, which are expected to be deployed on the centralized execution platform of software defined vehicles. This necessitates a scheduling framework, which is able to schedule ML-based tasks in a mixed-criticality environment, where tasks with different criti-

calities and real-time requirements are executed on the same execution platform [1].

With the shift towards SDVs, the European Union has identified a need for this standardized software architecture and a standardized software development process and provides support for this shift in its research projects [21]. This rise in funding and research support further reinforces the trend towards SDVs and the corresponding shifts via projects like HAL4SDV [1]. A complex and powerful middle layer, which provides standardized application programming interfaces and interfaces to the software layer and provides support for OTA updates is the result of this industry shift. This middle layer is often referred to as an HAL [1].

1.3 Hardware Abstraction Layer

As introduced above, the proliferation of software defined vehicles (SDVs) and central car servers (CECASs) necessitate the introduction of a hardware abstraction layer (HAL). This definition encompasses multiple aspects. An abstraction layer can be positioned either between the middleware and the operating system layer or between the operating system layer and the central compute hardware [1]. The European-funded project HAL4SDV identified the following features of a HAL inside automotive vehicles [1].

As shown in Figure 1.2, projects like HAL4SDV aim to create a comprehensive hardware abstraction layer as a broad term, which includes the layers between and including the middleware layer, the platform service and driver layer, the operating system layer, the virtualization layer and the hardware layer. The HAL is expected to provide a standardized interface to the software layer, which allows for the integration of software from different suppliers and the interchangeability of hardware and software components. The HAL is expected to provide the following features [1].

The first feature of a HAL is the hardware virtualization. This virtualization provides a unified interface in both directions, from underlying layers towards higher layers and from higher layers towards underlying layers. The European-funded project HAL4SDV provides abstraction from the hardware towards the virtualization layer and vice versa. This aims to provide a unified instruction set architecture (ISA) for the drivers and hypervisors and a virtualized set of instructions, which needs to be supplied by the hardware. In HAL4SDV

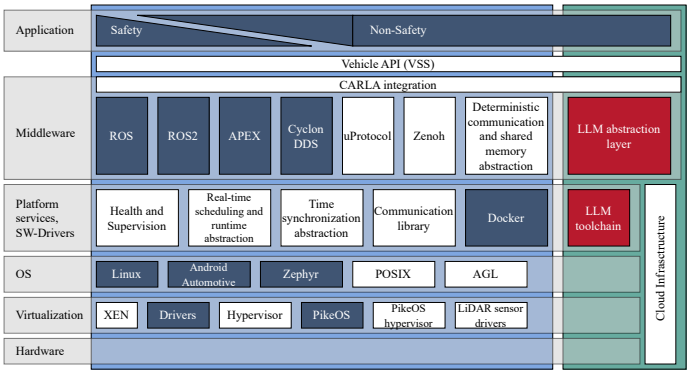


Figure 1.2: Hardware Abstraction Layer as presented in the HAL4SDV project, adapted from [1, 22]. The blue background shows on-board components, the green background shows off-board components. White components are improved in the project [1], dark blue components are commercial off-the-shelf (COTS) components, red components are new components in the HAL4SDV project.

and other European projects like Rigoletto, RISC-V is identified as a potential candidate for this ISA [1]. Further abstraction is provided between the operating and virtualization layer towards the middleware layer and vice versa. This abstraction is expected to provide functionality for the middleware like memory management, access towards isolation components and runtime scheduling from the operating system and hypervisor layer. The operating system and hypervisor layer expects to receive a unified set of instructions from the middleware layer, which is then translated into the underlying hardware and virtualization layer. This then allows for the interchangeability of the used hypervisors and provides requirements, which need to be implemented by the hypervisors [1]. Lastly the middleware layer is expected to provide a unified set of instructions to the software layer, which is then translated into the underlying operating system and hypervisor layer. This then allows for the interchangeability of the used operating systems and provides requirements, which need to be implemented by the operating systems. This furthermore aims to allow the software vendors to develop software for a unified interface, present on multiple hardware platforms inside one vehicle and across multiple vehicles, which are based on different hardware platforms [1, 3, 22].

In conjunction with the software defined vehicle aspect of the trends in automotive E/E architecture, the HAL is expected to provide functionality to

update the hypervisors and operating systems in addition to the applications of the vehicle over-the-air. This is necessary for seamlessly closing security vulnerabilities, while not changing guarantees provided by the hypervisors and operating systems to the applications. This functionality of allowing over-the-air (OTA) updates for hypervisors and operating systems needs to be supported by the hardware as well as by the hypervisors and operating systems and the HAL [1]. Because of the heterogeneous nature of modern automotive architectures and the inclusion of machine learning (ML)-based tasks, the HAL also needs to support handling of ML-based tasks and drivers for hardware accelerators. This includes the ability to schedule ML-based tasks required by the application layer without interfering with the execution of other tasks on the same execution platform and without presenting the software layer with the underlying hardware and software architecture, because the HAL is expected to provide a unified interface for these tasks. This is necessary, because the architecture of the hardware accelerators can be thought of as proprietary and different between vendors and automotive platforms. Providing the application developers with a unified interface for these tasks allows for the development of software, which is portable across different automotive platforms and hardware accelerators [1].

Lastly, the offloading of functions to cloud servers is expected to be supported by the automotive platforms and deployed hardware platforms, which also necessitate the support of the HAL. This offloading of functions to cloud servers is expected to be used for non-critical software functionality, because of the potential for higher latency, lower availability as well as privacy concerns [3]. Additionally, the offloading of software functionality to cloud servers is expected to be most beneficial for ML-based tasks, since the execution of ML-based tasks is expected to be more efficient on cloud servers, because of the higher computational power and the ability to use more powerful hardware accelerators [3]. Ideally, the deployed software is agnostic to the execution platform and can be executed on the cloud or on the local execution platform, depending on the availability of resources and the requirements of the software. This necessitates a unified interface for ML-based tasks, which is provided by the HAL [1].

1.4 Motivation

With the context provided in Sections 1.1 to 1.3, the motivation for this work lies in the intersection of the three trends in automotive E/E architecture: the shift

towards software defined vehicles (SDVs), the shift towards central car servers (CECASs) and the shift towards a standardized hardware abstraction layer (HAL). All trends necessitate a scheduling framework, which is able to schedule mixed-criticality task sets on heterogeneous systems-on-chip (HESoCs) in a centralized platform. Because the near future of automotive E/E architecture is expected to be described by the 4th generation of automotive centralized computing platforms, which are expected to encompass domain control units (DCUs) on the transition towards central car servers (CECASs), the main use case for this scheduling framework lies on the scheduling of mixed-criticality workloads, executed on domain control units (DCUs) at the center of the domains, like body, chassis, or powertrain.

Because of the centralized nature of domain control units (DCUs), the scheduling framework is expected to be centralized as well. With the shift from dedicated sensors per feature to centralized and shared sensors, which different features can access at need, the scheduling framework is expected to be able to dynamically schedule tasks and switch between execution modes depending on the current system state. Depending on the current system state, different execution versions need to be chosen.

These execution versions differ in their inputs, their execution time and execution units. Tasks, which are executed on domain control units (DCUs), are expected to be executed in a time-triggered manner, with predictable and periodic execution. Still, a dynamic execution is expected, because of the dynamic nature of required outputs by the driver or more centralized units.

These observations show the need to develop a scheduling framework, which is able to schedule mixed-criticality workloads on multicore heterogeneous systems-on-chip (HESoCs) in a centralized manner. The main use case for this scheduling framework lies in the 4th generation of automotive centralized automotive computing platforms, which are expected to encompass heterogeneous system-on-chip (HESoCs) with mixed-criticality workloads, which include machine learning (ML)-based tasks. The centralization of functionality in domain control units (DCUs) provides the need for a scheduling framework, with predictable and periodic execution, for which a time-triggered approach is the right choice. A more centralized approach, as expected in the 5th generation of automotive centralized computing platforms, is then expected to further increase the dynamic requirements and is thus expected to build upon the scheduling approach developed in this work or move towards a more dynamic operating system. Nevertheless, the focus of this thesis lies on the 4th generation of automotive centralized computing platforms, which are expected to en-

compass domain control units (DCUs) with mixed-criticality workloads, which include machine learning (ML)-based tasks.

1.5 Used Example

Example 1: Task Set Description

Table 1.1: Description of the example task set used in this thesis.

	Name	Runtime	Description
(1)	Weather Classification	CPU: 202 ms, ML: 8.22 ms	Reading in data from the image capturing task and classifying the weather.
(2)	Traffic Sign Classification	CPU: 192 ms, ML: 6.9 ms	Classification of traffic signs.
(3)	Image Input	200 µs	Reading data from an external camera.
(4)	Network Tx	50 µs	Sending data over the network.
(5)	Network Rx	50 µs	Receiving data over the network.
(6)	Decision-Making	1 ms	Decision-making based on weather, traffic signs and further external input.
(7)	Statistics Task	2 ms	Tracking statistics of the weather and external input.
(8)	Cryptography	CPU: 50 ms, Accel: 10 µs	Data encryption.
(9)	Display	100 µs	Displaying the results and statistics of the decision-making.
(10)	Data Storage	10 µs	Storing into scratchpad memory.
(11)	Data Loading	10 µs	Loading from scratchpad memory.

The used example includes eleven tasks, which are shown in Table 1.1. Tasks can have different versions. Task **(6)**, *the decision-making task, has a period of* 100 ms. *Task* **(7)**, *the statistics task, is given a period of* 200 ms *and the display task* **(9)** *is given a period of* 40 ms.

Example 1 shows an example task set description, which is used as an example throughout the thesis. This task set describes a use case which includes all concepts required by this thesis, is designed to be as simple as possible and is used to show the implementation of the proposed methods to try to explain the concepts intuitively. Example 1 was used introduced in [Sta27] as well and describes an in-vehicle vision-based weather classification. It is furthermore extended by a traffic sign detection using ML-based methods. As a minimum working example, it is applicable as a case study of a centralized

part of the infotainment system. Additional case studies are introduced in Chapter 8.

Task ***(1)*** and ***(2)*** are the two main tasks, each implementing a ML-based and a CPU-based version. These tasks are implementing weather classification and traffic sign classification. Task ***(3)*** is an image input task, which reads in data from an external camera. Task ***(4)*** and ***(5)*** model network communication, which send and receive data over the network. Task ***(6)*** is a decision-making task, which is based on the weather and traffic signs. These decisions are to send a signal to turn on the windshield wiper or showing the current speed limit on the dashboard. Task ***(7)*** is a statistics task and tracks statistics of the weather and external input. Task ***(8)*** is an encryption task which is executable on a dedicated accelerator (like [23]) or on the CPU. Task ***(9)*** displays the results and statistics of the decision-making and ***(10)*** and ***(11)*** are data movement tasks, which model the move of data into and from scratchpad memory.

Example 2: Task Precedence Constraints

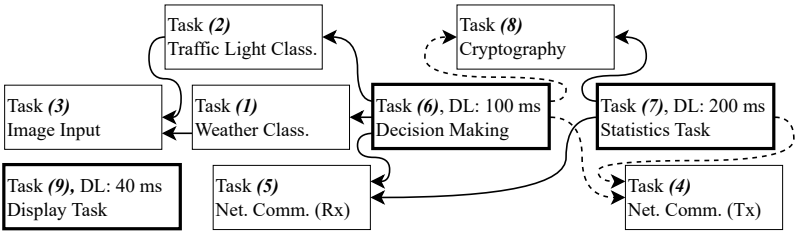


Figure 1.3: Precedence constraints of the example tasks. Precedence constraints are defined with respect to the source task. If the source task of a precedence constraint requires the completion of the target task, the arrow is solid. If the source task needs to finish before the target task can start, the arrow is dashed. Bold borders signify periodic tasks.

Tasks **(6)**, **(7)** *and* **(9)** *are periodic and released at their respective periods. Task* **(6)** *and* **(7)** *require the output by other tasks depending on Figure 1.3. Task* **(9)** *requires the output of no other task, but reads in the values from memory. Lastly, tasks* **(10)** *and* **(11)** *are non-periodic tasks, which are triggered by other tasks to move data into or loading data from the scratchpad memory.*

The tasks presented in Example 1 are connected via precedence constraints. The precedence constraints of the tasks are presented in Example 2. These tasks and precedence constraints are executed on a platform, which includes

multiple execution units and shared resources. The platform description is given in Example 3.

Example 3: Platform Description

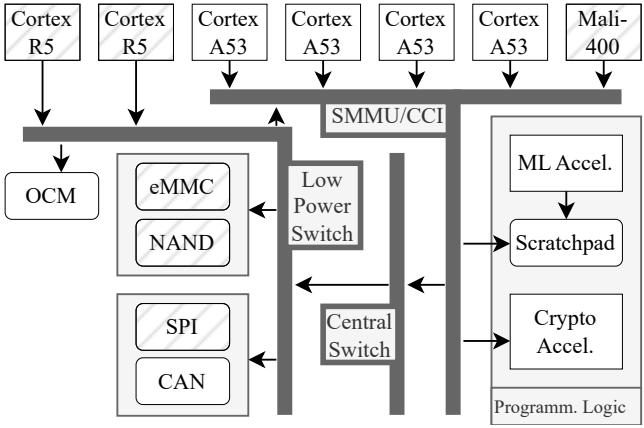


Figure 1.4: System platform adapted from [24]. Execution units are marked as boxes, the central communication interfaces are shown in gray. Shared resources are marked with rounded corners. Components not considered in this work are dashed out.

The Xilinx/AMD Ultrascale+ ZCU102 evaluation board [24] is used as the example platform. It includes a quad-core ARM Cortex-A53 CPU and a FPGA, into which a crypto module and a ML accelerator are mapped. These accelerators can execute the ML-based versions of the weather and traffic sign classification tasks as well as the encryption task. The platform includes shared resources, of which the memory interface, and the communication interfaces are shown. Some shared resources are grouped together and can be accessed via shared lines. A scratchpad memory is mapped inside the programmable logic. The Xilinx ZCU102 Ultrascale+ includes more components, which are not considered for this example. The complex interconnect is condensed. For a detailed explanation, see [24].

The combination of Example 1, Example 2 and Example 3 is used throughout this thesis to build up the system model.

1.6 Problem Formulation and Thesis Structure

The goal of this thesis is to create a holistic scheduling algorithm for periodic tasks with associated deadlines. These tasks can be both, ML-based or non-ML-based. Both types must be supported because of the increasing deployment of ML-based methods inside automotive vehicles. Safety-mechanisms must be included and timely execution, which depends on given safety levels, must be guaranteed. To support different driving modes, the introduced algorithm must support mode-change scheduling and dedicated mode-changes need to be implemented to keep task deadlines and periods. Mode changes, which are triggered by safety incidents, deadline misses or manual inputs by the driver, also need to be supported. To detect safety incidents or deadline misses, suitable monitoring components need to be included, and execution boundaries need to be defined.

To describe the given software tasks, execution modes and constraints, a mathematical model needs to be designed. This model needs to support the requirements introduced by hardware abstraction layers (HALs), software defined vehicle (SDVs) as well as the tasks introduced in Example 1. This includes a centralized hardware platform and phased execution models, which represent a proven and well-established approach in real-time scheduling. By leveraging such phased execution models, scheduling algorithms can enforce deterministic task ordering and prevent or limit interference between tasks which might be triggered due to simultaneous access to memory. This allows for predictable timing guarantees and properties, given the execution phases are adhered to [25, 26]. Because of the widespread adoption of phased execution models, the presented scheduling algorithm must align with established methods and the introduced model must make use of execution phases. Lastly, recovery mechanisms for deadline misses and safety incidents need to be described in the model and the scheduler needs to implement and support these features.

This thesis is structured as follows. Chapter 1 introduces the context, motivation and problem statement of this work. Chapter 2 reviews necessary fundamentals for multicore systems, real-time systems, hypervisors, mode change scheduling and ML-based methods. Chapter 3 presents the state of the art in static and dynamic real-time scheduling as well as anytime neural networks (ANNs) and neural network scheduling. Chapter 4 introduces the proposed scheduling method by first presenting the task model and the generation of periodic time-triggered schedules. This is then followed by the algorithm to create schedules for different execution modes and the mode

changes schedules. Afterward, the platform deployment in the form of a Petri net converter is presented. In Chapter 5, the scheduling algorithm for ANNs is presented by introducing methods to add early exits and anomaly detection methods as well as methods to introduce checkpoints and provide an upper bound for the runtime. Chapter 6 introduces how anytime neural networks are mapped onto the mode-change scheduler. Chapter 7 evaluates the proposed scheduling algorithm. This includes an ablation study, deadline success rates and algorithm runtime. For ANNs the runtime overhead, early exit choice and accuracy is evaluated. Lastly, Chapter 8 presents two case studies. These include the deployment of generated schedules onto a custom accelerator and the use of a RoboRacer task stack deployed on two evaluation platforms. These were chosen to evaluate the approach on real world use cases. Finally, Chapter 9 concludes this thesis and gives an outlook on future research.

Each chapter is structured in a way that it can be read independently. First, a concept and its modeling is introduced. Afterward this model is used in the proposed implementations. During this, main contributions are introduced using italic font and a blue | Line and examples are introduced with a red | Line . These examples all use the same context and build upon the contributions introduced in the corresponding chapters. The aim of this is to make the thesis more accessible and easier to read as well as to give the reader an opportunity to find relevant parts fast. In the evaluation, a yellow | Line is used to highlight important findings.

Summary

The work in this thesis is based on eleven first author publications. The generation of time-triggered schedules was first published in [Sta24a] and extended in [Sta25b]. The detection of safety-relevant triggers for mode changes was presented in [Sta23b] and [Sta23d]. The addition of anomaly detection methods to convolutional neural networks (CNNs) was first presented in [Sta23a] and the addition of early exits to CNN was published in [Sta24b]. The scheduling of these resulting anytime neural networks (ANNs) and early exits as well as cloud offloading was published in [Sta27, Sta25b, Sta23c].

This page intentionally left blank

This page intentionally left blank

Chapter 2

Fundamentals

Because this thesis encompasses two adjacent but distinct fields of research, a common understanding must be achieved prior to introducing the developed scheduling framework. This chapter first introduces the fundamentals of static scheduling and real-time systems, followed by the introduction of mode change scheduling. Afterwards, machine learning and artificial neural networks are introduced, which then leads to presenting of anytime neural networks.

2.1 Multi-Core and Many-Core Systems

Multi-processor systems-on-chip (MPSoCs) describe the common type of architecture used as the execution platforms in embedded systems, electronic control unit (ECUs) and domain control units (DCUs). This chapter describes architecture components found in MPSoCs, as well as the challenges, arising when scheduling tasks on such platforms.

2.1.1 Components of Multi-Processor Systems on Chip

A MPSoC architecture is defined as an architecture, which includes multiple processing elements on a single chip. A Many-Core System on Chip is a MPSoC, which includes a large number of processing elements [27]. In both cases the processing elements can include general purpose processors (GPPs) and real-time cores as well as specialized processing elements, such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs) or dedicated hardware accelerators for ML-based workloads like the FleXNNgine [28] or the Eyeriss v2 [29] accelerator as well as encryption modules like the DRACO accelerator [23]. A system, which includes heterogeneous processing elements

is called a heterogeneous system-on-chip (HESoC) [30]. One example for a HESoC is the NVIDIA Jetson AGX Orin [31], which includes a GPU and multiple central processing unit (CPU) cores as well as a dedicated deep learning accelerator. A second example for HESoCs is the Xilinx/AMD Zynq UltraScale+, which is placed on an ZCU102 development board [24]. The Zynq UltraScale+ includes a four core CPU as well as a two core real-time CPU and a FPGA.

Additionally, shared resources, which can be accessed by multiple processing elements, are included in MPSoCs. These include shared memory, which can be accessed by multiple processing elements, shared interfaces towards external devices, such as controller area network (CAN) or universal asynchronous receiver transmitter (UART), timers, configuration registers and more [30]. A trade-off exists, when accessing shared memory. Processors can access data from shared memory via dedicated caches, which are then part of the memory hierarchy and are placed close to the processing element. This usually speeds up memory access, but leads to cache coherency challenges as well as uncertainty in the access time. Introducing caches can significantly decrease the average access time, but increases the worst case access time [32]. Two approaches exist to address this issue. The first approach is to simply not use caches and to access shared memory directly. This approach leads to a higher average access time, but a lower worst case access time. The second approach is to include the caches in the execution time considerations, which leads to a higher average access time, but a lower worst case access time [33]. ***In this work, the first approach is used, and no cache is used.*** This reduces the complexity, when accessing shared memory but opens up the challenge of contention.

2.1.2 Contention and Phased Execution Models

Units simultaneously triggering access to shared resources can lead to contention, which in turn can lead to delays in accessing the shared resource or lead to data inconsistency in the case of memory access. Additionally, this then leads to non-deterministic access time to shared resources [34]. One solution to this problem is to use a phased execution model like the Predictable Execution Model (PREM) [25] or acquisition execution restitution (AER) [26]. In the AER model, the execution of tasks is divided into three phases, where during the acquisition (A) phase and restitution (R) phase, tasks can access memory, and during the execution (E) phase, tasks can only access local resources. Only one task, and with that only one unit, can be in the acquisition or restitution

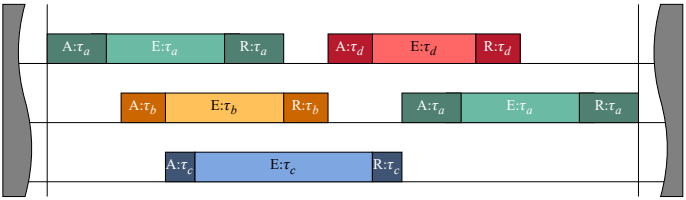


Figure 2.1: Example of the phase-based execution model AER [26]. Different tasks are shown with different colors. Darker colors represent the acquisition and restitution phases, while lighter colors represent the execution phase.

phase at any given time, which then prevents contention. The argument goes that this then allows for a predictable access time to shared resources, since the largest contribution to delays is mitigated. The AER model is described in more detail in Chapter 3 and a graphical representation is shown in Figure 2.1. ***This work uses a phased execution model to avoid contention and provide a deterministic access time to shared resources.***

Figure 2.1 shows an example of four scheduled tasks using the AER model and is adapted from [26]. No two tasks are either in the acquisition or restitution phase, if another task is in their respective acquisition or restitution phase. This prevents contention and allows for a deterministic access time to shared resources. The AER model is used in this work to avoid contention and provide a deterministic access time to shared resources. Because of the high number of different on chip components, the communication between the different components needs to be designed to handle fair access to shared resources. Communication interfaces can be implemented using a bus system, like the Advanced eXtensible Interface (AXI) bus [35], or a network on chip [36]. Accessing components either needs to be arbitrated [37] or a routing algorithm needs to be implemented [36]. In MPSoCs like the UltraScale+, the AXI bus is used as a low and high speed interconnect [24]. Since in a bus structure, only one component can access the bus at any time, an arbitration mechanism is necessary to handle access to the bus. In the UltraScale+ a round-robin arbitration mechanism is used [24]. In all cases, the communication system provides a second possibility for contention, which can lead to delays. ***Because of this, the communication systems limit the modelling of shared resources.*** All peripherals connected to the communication system must be combined into one shared resource, since accessing one potentially delays accessing others. Should multiple communication paths exist, the shared resources can be split.

2.1.3 Isolation of Access towards Shared Resources

In addition to contention, the access towards shared resources needs be isolated between cores and between tasks, to limit fault propagation in the case of failure [8]. This isolation is necessary to ensure that a fault in one core or task does not propagate to other cores or tasks, which could lead to a system failure. This isolation can be implemented via software components like hypervisors [38, 39] or via hardware mechanisms like the Xilinx Memory Protection Unit (XMPU) or Xilinx Memory Management Unit (XMMU) [24] for Xilinx / AMD platforms. Hypervisors are used to isolate the access towards shared resources between different virtual machines, which provides virtual memory addresses for each virtual machine. The XMPU and XMMU are used to isolate the access towards shared resources between different cores. The configuration of the XMPU and XMMU is done via configuration registers, where each core is configured to have access to a group of specific ranges of (potentially masked) memory addresses. Multiple methods exist, which can be used to configure these components [40, 41]. ***In this work, the isolation of access protection units is not the focus. When using hypervisors, the hypervisor is assumed to be configured correctly and the isolation of access protection units is assumed to be handled by the hypervisor. For bare metal systems, the isolation of access protection units is assumed to be static and configured during design time.***

2.2 Artificial Neural Networks

This thesis focuses on the use of anytime neural networks (ANNs), which are a type of artificial neural networks. In this work, the acronym ANN is used to refer to anytime neural networks. This chapter introduces the structure of anytime neural networks as an extension of artificial neural networks and convolutional neural networks. Additionally, hardware accelerators and safety and reliability concerns of neural networks are introduced.

2.2.1 Artificial and Convolutional Neural Networks

The general term artificial intelligence broadly describes the ability of a system to perform a task, which is generally associated with human intelligence. Machine learning (ML) is a subfield of artificial intelligence, which focuses

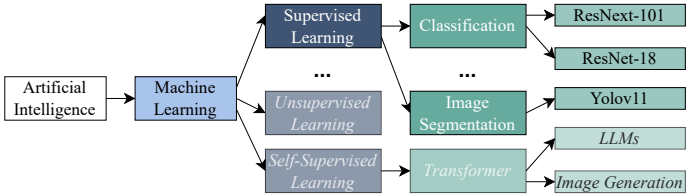


Figure 2.2: Types of machine learning workloads and application examples, adapted from [42]. Types not used in this thesis are marked in italics.

on the development of an algorithm, which is able to learn a task based on a given dataset.

Neural networks can be classified as one type of ML-based systems [42, 43]. ML-based system generally have the goal of predicting a result based on a given input. Different distinct learning types of ML-based systems can be distinguished in literature. Three examples can be given as supervised learning, unsupervised learning or self-supervised learning [42] among others. The term learning signifies the process of improving the performance of a model based on the input data, which can either be provided with labels or without labels, determining, if the learning is supervised, unsupervised, etc.

The first presented type, supervised learning, requires a dataset with known labels. The goal of supervised learning is the training of a model, which is able to predict the label of a given input. The term supervised learning is used, because the training data is labeled, and thus the training error is supervised by the training algorithm. This allows for the training of models for classification or image segmentation [43]. Examples for classification models are GoogleNet [44], ResNet [45] and ResNeXt [46] or GPUNet [47]. Additionally, YoloV11 [48] is an example for image segmentation.

Unsupervised learning defines a learning strategy, which does not require labeled input data. The goal of unsupervised learning is to find patterns in the input data which allow for a clustering of this data into different groups or to determine dimensions onto which the input data can be projected. One example for the clustering of data is the K-means algorithm [49]. For the projection of data onto a lower-dimensional space, principal component analysis (PCA) is an example [50].

Lastly, self-supervised learning gained traction with the rise of transformer-

based models [51, 52]. Self-supervised learning is a learning strategy, which first generates labels for the input data based on the input data itself. Afterwards, the generated labels are used to train a model in a supervised manner, allowing for the training of models without the need for labeled data. Example for self-supervised learning include transformer based models as well as large language models [53, 54].

This thesis focuses on models generated via supervised learning for image classification and image segmentation tasks. While imaginable, transformer architectures are not common in current automotive DCUs and are thus not the focus of this work. The introduced concepts are nevertheless transferable to these structures.

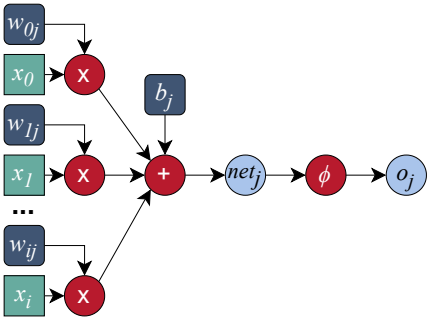


Figure 2.3: Structure of one artificial neuron, adapted from [55]. Green components represent input data, dark blue components represent learnable parameters, light blue components represent outputs.

With artificial neural networks being the most common ML-based strategy, an introduction of the basic concepts is warranted. The general structure of an artificial neural network is based on the artificial neuron, which is depicted in Figure 2.3. Given a neuron j, all inputs, x_0 to x_i are multiplied with their respective weights, w_{0j} to w_{ij}, and then summed up with a bias term b_j, resulting in the net value net_j. This value is then passed through an activation function ϕ, which provides the output o_j of the neuron. The activation function is a non-linear function, used to introduce a non-linear behavior to the neuron. Without this, all neurons could be collapsed into a single linear function. The most common activation functions are the sigmoid, hyperbolic tangent, and rectified linear unit (ReLU) functions [56]. The ReLU function is defined as $\phi(x) = \texttt{max}(0, x)$, which outputs the input directly if it is positive, otherwise it outputs zero. The hyperbolic tangent function is defined as $\sigma(x) = \texttt{tanh}(x)$, which outputs values between -1 and 1. The sigmoid function

is defined as $\phi(x) = \frac{1}{1+e^{-x}}$, which outputs values between 0 and 1. The choice of activation function can have a significant impact on the performance of the neural network, and different activation functions may be more suitable for different tasks. Together, this leads to the output equation of a neuron j as:

$$o_j = \phi\left(\sum_{i=0}^{n} w_{ij} \cdot x_i + b_j\right), \tag{2.1}$$

where n is the number of inputs to the neuron and ϕ signifies the activation function.

The neurons depicted in Figure 2.3 are then arranged into layers. One layer consists of multiple neurons. The input to each neuron are either all outputs from previous layers or values from the input data itself. The output of each neuron then acts as the input to the next layer or as the final output of the artificial neural network. Because the output of the neurons of each layer is connected to each input of the neurons of the next layer, this structure is called a fully connected layer. Because of the layered structure of artificial neural networks, further names are used, like e.g. multi-layer perceptron (MLP). With this structure a non-linear function is achieved, and the artificial neural network is assumed to learn a function $f : \mathbb{R}^n \to \mathbb{R}^m$, which maps an input vector $x \in \mathbb{R}^n$ to an output vector $y \in \mathbb{R}^m$.

This structure has to be trained to be able to perform any task. The supervised learning strategy, which was introduced above necessitates labeled input data. The input data set thus consists of a set of pairs of input vectors x_i and their corresponding output vectors y_i, where x_i represents the mapped input to the inputs of the neurons of the first layer. The output vectors y_i represent the expected output of the neural network as the values, which are expected to be present at the outputs of the neurons of the last layer. In the simplest classification example, the output value has the form of $y_i = [0, \ldots, 1, \ldots, 0]$, where the position of the 1 indicates the class of the input vector x_i [56].

The training of the neural network is performed by adjusting the weights and biases of each neuron, depending on the error between the expected output y_i and the actual output of the neural network $\hat{y}_i$. The error between both values is calculated as the loss $L(y_i, \hat{y}_i)$, which is then used to update the weights and biases of each neuron. The most common loss function for classification tasks

is the cross-entropy loss, which is calculated as:

$$L(y_i, \hat{y}_i) = -\sum_{i=0}^{m} y_i \cdot log(\hat{y}_i), \tag{2.2}$$

where m is the number of classes, y_i is the expected output for the input of the neural network with input x_i and $\hat{y}_i$ is the actual output of the neural network for the input x_i. The weights and biases are then updated using a gradient-based optimization algorithm, such as stochastic gradient descent (SGD) or Adam [57]. These methods are employed to minimize the loss function in the high dimensional space of the weights and biases. Because finding the optimal set of weights and biases is an optimization problem, which uses the gradient and the optimization step is executed once per iteration, this gradient calculation needs to be performed efficiently [58]. For this, the `backpropagation` algorithm is used, which calculates the gradient of the loss function with respect to each weight and bias in the neural network. The `backpropagation` algorithm uses the chain rule to propagate the error from the output layer back to the input layer, which allows an efficient calculation of the gradients [58]. This efficiency is a result of the splitting of the gradient calculation into

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial b_j} \cdot \frac{\partial b_j}{\partial w_{ij}}, \tag{2.3}$$

where each component can then be calculated separately and reused for the calculation of the gradients of other weights and biases. The training process is repeated for multiple iterations, until the loss function converges to a minimum value, which indicates that the neural network has learned to map the input data to the expected output data [56]. ***This work focuses on the inference of the neural network, i.e. the aggregate process of Figure 2.3 after added early exits. No significant changes to the training process are made, but an understanding of the training process is nevertheless shown.***

Convolutional neural networks (CNNs) can be understood as an extension of the above presented artificial neural networks. While fully connected layers are usually used for the last classification layer(s) of the convolutional neural network, most layers used in convolutional neural networks are convolutional layers. Convolutional layers are based on the convolution operation. In a CNN the convolution operation is used with the goal to extract features like edges, corners, textures and higher dimensional shapes from the input data. The argument goes as to what these features exactly

are and which are necessary is learned by the neural network during training [59].

The convolutional layers form the centerpiece of the CNNs. Convolutional layers are characterized by their filters, which are small matrices, slid over the input data. The filter values are learned during training. Fixed parameters are the *kernel size* as the size of the filter, the *stride* as the step size of the filter, and the *padding* as the number of pixels added to the borders of the input data. The convolution operation to calculate the values of the output matrix is performed by multiplying the filter values with the corresponding input values and summing the values before applying an activation function. The filters are usually three-dimensional and use the stride to slide over the input data in the X and Y dimensions. Furthermore, the input data can have multiple channels in the Z dimension, e.g. for RGB images. The output of the convolutional operation is then a two-dimensional matrix, which then is one *feature map* of the next input layer in the output tensor [59].

The use of convolutional layers allows for the stepwise and more explainable extraction of features from the input data. Additionally, the use of convolutional layers allows for a reduction in the number of parameters, which need to be learned during training, because of the reuse of the filter parameters [59]. This allows for the training of deeper neural networks, which can learn more complex functions. The use of convolutional layers has its focus on image processing tasks like image classification. One output value of one feature map is then calculated as:

$$o_{j,k} = \phi\left(\sum_{i=0}^{n}\sum_{u=0}^{k_x}\sum_{v=0}^{k_y} w_{i,j,u,v} \cdot x_{i,j+u,k+v}\right) + b_j, \tag{2.4}$$

where n is the number of input channels, k_x is the kernel size in the x-direction, k_y is the kernel size in the y-direction, $w_{i,j,u,v}$ is the weight of the filter for input channel i, and $x_{i,j+u,k+v}$ is the input value at position $(j+u,k+v)$ for input channel i. The bias term b_j is added to the output value after applying the activation function ϕ [56, 59].

Figure 2.4 shows one example of a convolutional operation. The input data is a two-dimensional matrix with one channel. The kernel size is 3×3 and the stride is 1. The output value is calculated by multiplying the input values with the corresponding kernel values and summing them up before adding a bias term and applying an activation function. The output value is then placed in the output matrix at the corresponding position. This process is repeated for all positions in the input matrix, resulting in a feature map of the input data.

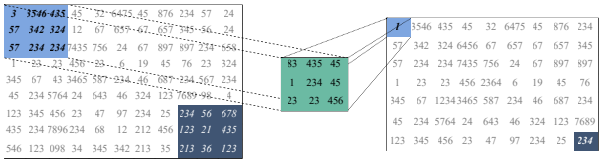


Figure 2.4: Example of one convolutional calculation. A kernel of size 3×3 is slid over the input data with a stride of 1. Input values marked in one color are translated to the output value marked in the same color. The output value is calculated by multiplying the input values with the corresponding kernel values and summing them up before adding a bias term and applying an activation function.

For an input data dimension of $X \times Y \times Z$ with $Z \neq 1$, the output dimension is $(X - k_x + 2p_x + 1) \times (Y - k_y + 2p_y + 1) \times N$, where N is the number of output channels. Here, p_x and p_y represent the padding in the x and y directions, respectively [59]. ***This work focuses on the inference of the CNNs for vision based tasks. Because the context lies in embedded systems, no batch size other than 1 is used.***

In addition to the convolutional layers, pooling layers are used in CNNs. Pooling layers are used to reduce the spatial dimensions of the input data, which allows for a reduction in the number of parameters and computations required for the neural network. Instead of learning parameters from the input data, pooling layers use a fixed operation to reduce the spatial dimensions by either taking the maximum value (max pooling) or the average value (average pooling) of a group of input values. The pooling operation is performed similar to the convolutional layers by sliding a window over the input data and applying the pooling operation to each element within the window. The output of the pooling operation is then a reduced version of the input data, which aims to retain the most important features of the input data [59].

2.2.2 Hardware Accelerators for Artificial Neural Networks

The most common mathematical computation necessary for the inference of artificial neural networks is the matrix multiplication. Inside the computation units, the matrix multiplication is performed by multiplying the input data matrix with the weight matrix of the neural network in the form of multiply and accumulate steps. This necessitates the use of Multiply-Accumulate

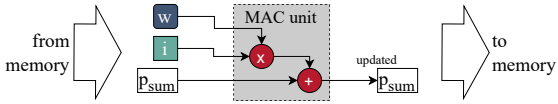


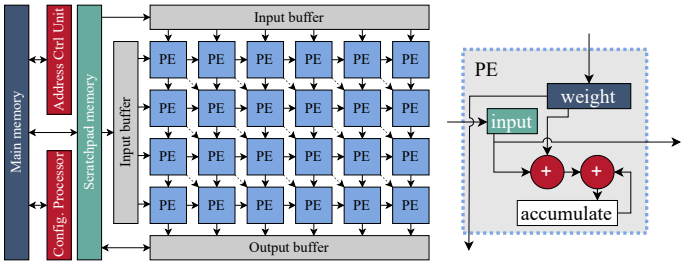
Figure 2.5: Structure of a multiply-accumulate unit, adapted from [55]. The input values i and w are multiplied and then added to the previous sum p_{sum}. The updated p_{sum} is then written back to memory.

(MAC) computation units. Figure 2.5 shows the structure of a singular MAC unit.

As shown, the MAC unit takes two input values, i and w, which are multiplied together. The result of the multiplication is then added to the previous sum, p_{sum}, which is stored in memory. Here, each weight w is multiplied with the corresponding input value i, and the result is accumulated in p_{sum}. For the biases, the same units can be used, with either the bias value being inserted with a weight of 1 or the bias value being used as a primer in the sum. The updated p_{sum} is then written back to memory, and the process is repeated for all input values and weights.

Because the singular multiply-accumulate steps of one element in the matrix multiplication are not dependent on the steps of adjacent elements, the MAC calculations are a prime candidate for parallelization. Because GPUs are designed for simple, yet parallel computation in the context of matrix multiplication in image processing, they are well suited for the inference of artificial neural networks and the included matrix multiplications [60]. For AlexNet [61], which is a simple convolutional neural network, the fully connected layers account for 58.6 million computations and parameters, since every weight is used once per inference. While the convolutional layers account for 666 million computations with 2.3 million parameters. Both types of layers showing the need for parallelization of the computations [61] and how convolutional layers reuse learned parameters. With this amount of parameters and the size of the parameter set increasing with the depth of the network, the need for more efficient parallelization is growing. Additionally, the use of GPUs is not always possible in embedded systems because of the power consumption as well as the need for predictable timing behavior.

With a more specialized approach, these accelerators usually follow the structure of systolic arrays. Systolic arrays are usually focussed on matrix multiplication and thus the multiply-accumulate steps, omitting further functionality present in GPUs. The name systolic array is derived from the step wise propagation of partial results through the array, similar to the pumping of



(a) Systolic Array (b) Processing Element

Figure 2.6: Structure of a systolic array with 4 times 6 processing elements, adapted from [28, 63]. The example processing element shows an output stationary approach.

blood through the heart [62, 63]. Many architectures are developed for the acceleration of neural networks [13, 28, 29, 64, 65], which follow a similar systolic array-like structure. This systolic array-like structure is usually a two-dimensional array of multiply-accumulate units. The output ports of each multiply-accumulate units are usually connected to the input ports of the adjacent multiply-accumulate units. With an output stationary approach, the weights and inputs travel through the processing elements of the array. In this case, the input data is fed into the array from one side, while the weights travel through the array from the other side. The output of the multiply-accumulate units is then accumulated in the processing elements and only written back to memory after the matrix multiplication is completed. Additionally, the strategies of weight stationary and input stationary are possible, where the weights or inputs are kept stationary in the processing elements, while the other values travel through the array. With the size of the systolic array being limited by the available resources, the matrix multiplication is usually split into smaller sub-matrix multiplications, then need to be calculated in a tiled manner [29]. These systolic array sizes are usually parameterized [28, 64] to allow the user to adapt the accelerator to the available resources as well as to the shape of the used artificial neural network.

Figure 2.6 shows an example of a 4×6 systolic array. The processing element shows an output stationary approach, where the weights and inputs travel through the processing elements, while the output is accumulated in the element and only written back to memory after the matrix multiplication. The systolic array is usually connected to a memory controller, which manages the data transfer between the memory and the systolic array. Additionally, a

control unit is used to manage the operation of the systolic array, including the configuration of the processing elements and the data transfer between the memory and the systolic array [28, 63]. Taken from the FleXNNgine, diagonal forwarding of outputs is supported in some architectures, allowing for specialized data transfer and output forwarding [28].

After starting the execution and loading data into the local scratchpad memory, the systolic array is able to execute the matrix multiplication predictably because the number of cycles necessary for the execution of each processing element is known, and the execution needs to be synchronized. This allows for a predictable execution time of the matrix multiplication, once data is loaded. The configuration of the systolic array dimensions is done during design time to be optimal for the given workload or set of given workloads and the chosen tiling of the employed neural networks. This allows for the optimization of the systolic array for the given workload. Additionally, it motivates added exits to be similar to the structure of the exits of the backbone neural network. ***The early exits of the anytime neural networks of this work follow this design choice and the structure of the exits of the backbone neural network.***

2.2.3 Structure of Anytime Neural Networks

In addition to the above presented artificial and convolutional neural networks, anytime neural networks are introduced. In the context of this work, the acronym ANN is used to refer to anytime neural networks. Anytime neural networks are a type of artificial neural networks, which are able to use additional exits to provide intermediate outputs during inference, allowing for early termination of the computation. These exits are called early exits and are added to the neural network at different starting layers [66, 67]. While some works use the more descriptive term of *early exit neural network (EENN)* [67], or more general *dynamic neural network* [66], this work uses the term anytime neural networks, which focuses on the ability to provide results at any time during inference.

Dynamic neural networks are split into three categories. The first category specifies dynamic routing. Dynamic routing allows for the selection of different paths through the neural network during inference, depending on the input data or available system resources. These paths can either be given by single layers, which are executed or skipped, by different branches of the neural network, or by choosing different sub-networks, which are executed or skipped. Essentially, dynamic routing is a generalization of the early exit

concept, which is the second category of dynamic neural networks. Early exit networks are defined as having one backbone network and two or more possible exits. These exits are also called heads or tails, depending on the publication. The backbone network specifies the network, which is executed to facilitate feature extraction. The added exits then branch of from this backbone network. This allows for a dynamic execution of the given exit. This specialization of dynamic routing usually does not consider branching inside the backbone network, only the branching off from the backbone network to execute the added exits. The last category specifies token skimming. Token skimming focuses on the reduction of the input dimension and to reduce the number of redundant input tokens to avoid redundant information. This aspect is usually employed in transformer-based architectures, which are not the focus of this work [66].

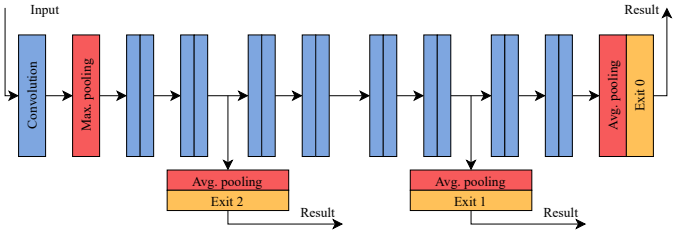


Figure 2.7: Structure of an anytime neural network with three exits. Convolutional layers are drawn blue, pooling layers are drawn red and exits are shown in yellow.

This work focuses on the second category of dynamic neural networks, which is the early exit concept. Figure 2.7 show the structure of a network with three exits. After the input data is entered into the network, the backbone network executes until after a set of convolutional layers. Then, a decision is made regarding the continuation of the execution with either the exit or the rest of the backbone network. The configuration of this decision algorithm is dependent on the given use case. Remaining execution time budget [Sta25b], available energy [68] or confidence [69] are examples for decision constraints in the given context. An anomaly detection mechanism can be employed to identify input data, which lies out of the operational design domain and choose the exit accordingly. The chosen exit can be configured in two different ways. First, the exit can have the same classes and output format as the original exit of the backbone network. This allows for the use of the identical loss function, training strategy and output format as the original backbone network. With this, the output can be used in the same task set as the original backbone

network. Second, the exit can have a different output format and / or classes as the original exit of the backbone network. This opens the potential for the use of the early exit in a different context than the original backbone network, i.e. to provide a more abstract classification of the input data than the original backbone networks exit or to classify the input according to a distinct class set [Sta24b, 66].

The training of the early exits can be either performed in a joint strategy with the backbone network by training the exits in conjunctions with the backbone network or in a split strategy, where the backbone network is trained first, and then the early exits are trained based on the intermediate feature maps of the backbone network. These intermediate feature maps then need a class assignment, which, as mentioned above, can either be the same as the original backbone network or independent. Should the decision to choose an early exit be based on attached anomaly detection mechanisms, the output type of the early exit can, for example, be used to provide a more abstract classification of the input data, which can be used to identify out-of-distribution data or be a different class set for providing a result in a different design domain [67].

The structure of the early exits should be similar to the structure of the exit of the backbone network, to allow for a joined optimization of the deployment platform. This allows for the use of the same accelerator and the same data transfer strategy for the early exits as for the backbone network. Additionally, this allows for a more efficient use of the available resources, as the early exits can be executed on the same accelerator as the backbone network, without the need for additional hardware resources [28].

Lastly, the safety and reliability concerns of the ML-based models plays an important role in deciding the use of early exits. The use of early exits can be used to provide a more reliable and safe output, by providing a more abstract classification of the input data or by providing a different class set for the early exit. Additionally, given deadlines can be met by using early exit to reduce the execution time of the neural network albeit at the cost of result confidence [66, 67]. ***This work uses the dynamic neural network concept of early exits to provide a result in the case of a detected anomaly as well as to provide a result in the case, a given deadline is expected to not be met. The term anytime neural network and the acronym ANN are used to refer to this concept. The exit structure is oriented on the structure of the backbone network, to be easily executed on given accelerators.***

2.2.4 Safety and Reliability Concerns in Neural Networks

With the EU-AI act, the European Union has introduced a legal framework for the use of artificial intelligence in high risk and limited risk applications, which includes the use of neural networks in automotive applications [70]. The EU-AI act defines high risk applications as those, which can have a significant impact on the health, safety or fundamental rights of individuals. This includes applications in the automotive domain, where neural networks are used for tasks like object detection, lane keeping and traffic sign recognition. The EU-AI act necessitates adequate risk management and mitigation strategies to be implemented for the use of neural networks in these applications. Additionally, international standards like the ISO 8800 [71] further necessitate the use of architectural redundancy patterns or anomaly detection modules to handle high risk scenarios.

Works exist to improve the reliability and safety of neural networks. One approach is to use ensemble methods, which combine multiple neural networks to improve the overall performance and robustness of the system by introducing Bayesian neural networks (BNNs) to provide redundancy to generate an uncertainty estimation for these high risk ML-based applications [72]. These approaches provide a stochastic module to the neural network, used to provide an uncertainty estimation for the output of the neural network, by adding white noise or dropouts to the input data or the intermediate feature maps of the neural network. This allows for the generation of multiple outputs for the same input data, which can then be used to estimate the uncertainty of the output by comparing the generated results. Highly varying results indicate a high uncertainty, while similar results indicate a low uncertainty. This allows for the identification of out-of-distribution data, which can be used to trigger the use of an early exit or a fallback strategy [72].

The EU-AI act further allows for the use of anomaly detection modules to identify out-of-distribution data, which can be used to trigger the use of an early exit or another fallback strategy. Anomaly detection modules can be implemented using various techniques, but need to provide a score to determine if the computation with the original network should continue or if an exit should be used. Often, intermediate results of the neural network are used to determine the anomaly score, which can be used to identify out-of-distribution data. These intermediate results are the output of once convolutional layer and are called feature maps or layer traces. The anomaly score can be calculated using various techniques, each based on either ML-based [73] or non-ML-based technique [74].

This work uses anomaly detection components to supervise ML-based tasks in the context of automotive applications to fulfill the requirements from the EU-AI act and ISO 8800. The anomaly detection components are used to identify out-of-distribution data, which can be used to trigger the use of an early exit or to trigger a fallback strategy.

2.3 Real-Time Scheduling Approaches and Task Design Concepts

In the context of real-time systems, two main approaches exist. The first approach is the event-triggered scheduling, which is based on the assumption that tasks are triggered by events, such as interrupts or messages [75]. These events are mostly the release of tasks. The tasks can be given fixed priorities, leading to the so-called *fixed-priority scheduling* [76]. Additionally, these events might trigger the preemption of tasks, leading to *preemptive scheduling* [76]. Both approaches are not mutually exclusive. Additionally, approaches like *first-come, first-served (FCFS)* and *shortest job next (SJN)* are also used in certain scenarios. These can also be classified as subcategories of event-triggered scheduling [76]. The second approach is the time-triggered scheduling, which is based on the assumption that tasks are triggered by a periodic timer. In contrast to the event-triggered scheduling, time-triggered scheduling is based on a static assignment of task start times which are usually repeated with a given period [77].

This section first introduces the concepts of time-triggered scheduling in contrast to the event-triggered scheduling. To be effectively scheduled, tasks also follow an implementation concept. This section also introduces the task design concepts used in this thesis. Lastly, a platform deployment needs to be necessary to execute the given tasks in the chosen scheduling concept, irrespective if time-triggered or even-triggered scheduling is used. This is usually done by using a hypervisor or a real-time operating system, for which necessary concepts are introduced in the last subsection.

2.3.1 General Terms for Real-Time Tasks and Scheduling

First, general terms are introduced to provide a common understanding of the concepts used in this work. A *task* is a piece of software, which can be executed on a given platform. One executed instance of the task is called a *job*. One job,

i.e. one execution of a task can be triggered by either an event, which is then called *event-triggered*, or by a timer, which is then called *time-triggered*. The time, when a job is marked as ready to be executed is called the *release time*. The time between the release of a job and its completion is called the *response time* of the job. In the context of this thesis, a more general term *execution time* is used. One task can have a time assigned, until which the task needs to be completed. This time is called the *deadline* of the task or job. The time between the release of a job and its deadline is called the *relative deadline*. The assignment of job start times and execution units is called *scheduling*. The scheduling can either be done statically, i.e. before the execution of the tasks, or dynamically, i.e. during the execution. The scheduling can be done in a preemptive or non-preemptive manner. In the preemptive case, a task can be interrupted by another task, which is then executed first. In the non-preemptive case, a task is executed until completion, even if another task is released during its execution. The scheduling can also be done based on fixed priorities or dynamic priorities. In the fixed priority case, each task is assigned a fixed priority, which does not change during the execution of the tasks. In the dynamic priority case, the priority of a task can change during its execution based on certain criteria, such as the remaining execution time or the deadline of the task [75, 76, 77].

2.3.2 Event-Triggered Scheduling Concepts

Event-triggered scheduling is based on the assumption that tasks are triggered by incoming sensor data or messages and the resulting jobs of a task are then executed as soon as possible after receiving the data. This event, which can be an interrupt or a message, is usually generated by a sensor or another task and called a trigger event. To detect this trigger event, the incoming data or message buffer is sampled with a given sampling period [75]. If the buffer is empty, the task is not triggered and the buffer is sampled again after the sampling period. If the buffer is not empty, the task is marked as ready and triggered based on a predefined scheduling policy.

The first presented scheduling policy is the fixed priority scheduling. In this scheduling policy, each task is assigned a fixed priority. The task with the highest priority is executed before other tasks. Should the scheduling approach be preemptive, and a task with a higher priority is triggered, the currently running task is preempted, and the higher priority task is executed at the earliest possible time. This time lies between the immediate execution and the completion of the currently executed task [78]. These fixed priorities can be assigned

using a *rate-monotonic scheduling (RMS)* or *deadline-monotonic scheduling (DMS)* approaches, among others. In the rate-monotonic scheduling, the task with the shortest period is assigned the highest priority. In contrast, in the deadline-monotonic scheduling, the task with the shortest deadline is assigned the highest priority [78]. Additionally, the priorities can also be assigned to the tasks based on their criticality or to find an optimal priority assignment [79], which can be found for multi-stage and multi-resource systems [80]. In all cases, tasks are executed based on their priority. If a task is triggered and has a higher priority than the currently running task, it might preempt the currently running task. If the task has a lower priority, it is queued and executed after the currently running task is completed. This scheduling approach is then classified as the aforementioned *preemptive scheduling*. In contrast, if the task is triggered and has a lower priority than the currently running task, it is queued and executed after the currently running task is completed. Should a *non-preemptive scheduling* approach be employed, the task is queued and executed after the currently running task, still retaining its priority [76, 78]. These approaches are all considered as event-triggered scheduling.

A further scheduling concept is the *first-come first-served (FCFS)* scheduling. Here, the task triggered first is executed first, irrespective of its criticality in the system. This might be dynamically mapped to the priority of the released task with a pair-wise priority with the earlier task having a higher priority than the later task. In any case, this approach is non-preemptive, and the currently running task is executed until completion [76]. Given runtime estimations of the tasks, the *shortest job first* or *shortest job next (SJN)* scheduling approach can be employed. Here, the task with the shortest estimated execution time is executed first. This approach can be preemptive or non-preemptive, depending on the implementation. In the preemptive case, a task with a shorter estimated execution time can preempt a currently running task with a longer estimated execution time, after executing the shorter task, the preempted longer task can continue its execution [76].

Figure 2.8 shows an abstract event-triggered scheduling concept, based on the one presented in [76], where the job input is read from the disk and put into a ready queue, given an Input/Output (I/O) event, the task is put into the ready queue and scheduled based on the scheduling policy implemented in the dispatcher. The dispatcher then executes the task on the execution unit, should the job necessitate an input/output operation, the task is put into the waiting queue and waits for the input/output event to be completed. After the input/output event is completed, the task is put back into the ready queue and scheduled again based on the scheduling policy.

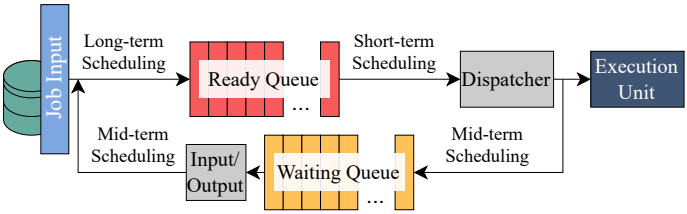


Figure 2.8: Abstract event-triggered scheduling concept, adapted from [76]. System units are drawn in blue, queues are drawn in red and yellow and schedulers are drawn in gray.

In this work, the dynamic part of the checkpoint-based scheduling concept from Section 5.5 uses this concept to schedule early exits from anytime neural networks, additionally the inclusion of this concept into the time-triggered scheduler is presented in Section 6.1.

2.3.3 Time-Triggered Scheduling Concepts

In contrast to the event-triggered scheduling concept, the time-triggered scheduling concept is based on a static assignment of job start times to specific time instants, which can be repeated with a given hyperperiod. The hyperperiod is calculated as the least common multiple of the periods of the tasks [77]. In the time-triggered case, the task period is given by the time between two consecutive releases of the task. When a task is given a period, and a result is expected with this given period, the period can be interpreted as the *implicit deadline* of this task.

This approach is based on the assumption that the execution time of the tasks is bound. This bounding of the execution time can either be measured with a profiling tool like [81] or by static analysis, should all timing behaviors be predictable [32]. For the first one tools like [82] can be used to measure the worst case interference time of the tasks, which is then used to calculate the worst case execution time of the tasks. For the second one, static analysis tools can be used to analyze the code and calculate the worst case execution time of the tasks based on possible data paths and the control flow [32]. In any case, the worst case execution time of the tasks is then one input into the scheduling algorithm to assign the start times of the tasks in a way that all tasks are completed with their given periods. These scheduling

algorithms usually create fixed schedules, which are then executed periodically.

Several assumptions are able to be taken into account, when scheduling the tasks. In contrast to the event-triggered scheduling, where tasks are marked ready when all inputs arrive and are able to be scheduled then, in the time-triggered scheduling it is possible to take these precedence constraints into account, when scheduling jobs and thus avoid unnecessarily awaking tasks. The same holds true for other communication constraints. If an I/O input is expected with a given period, the task can be scheduled to be executed after the I/O input is expected to arrive. Interference from shared resource access is the main contribution to delays in multicore systems [34, 83], either triggered by cache misses or contentious access to shared resources. When the effect is known by tools like [82], the scheduling of the tasks can be done in a way that these interferences are taken into account and to use this information to further bound the worst case execution time of the tasks. These interferences are also taken into account, when limiting the allowed access to shared resources, which is then called *temporal isolation* [83] and not only provides safety aspects but also allows to further bound the worst case execution time of the tasks.

This motivation was exploited in creating phased execution models, like the Acquisition Execution Restitution (AER) model [26] or the Predictable Execution Model (PREM) [25]. The goal there is to provide temporal isolation on the communication system to remove the effects of bus contention and to remove delays leading from simultaneous access to shared resources. Taking this into account further bounds the worst case execution time of the tasks and allows to schedule the tasks in a way that all tasks are completed with their given periods. A second method to bound the simultaneous memory access is given by limiting the available bandwidth of the memory access, with hardware components like memory bandwidth regulators [84]. This provides further possibilities to the scheduler and allows for more degrees of freedom by allowing tasks to execute shared resources simultaneously, if the combined bandwidth does not exceed 100 %. Scheduling assumes that when these constraints are met, then the worst case execution time is bounded to a lower value.

While the time-triggered scheduling concept is based on a static assignment of job start times, some approaches allow for a mode change scheduling concept, where the static assignment of job start times is changed during the execution of the tasks. In these works, the goal is usually minimizing the transition period of the schedules [85]. The transitional period is the time between

the last job of the old schedule and the first job of the new schedule. The goal is to minimize this time, while still guaranteeing that all tasks are completed with their given periods. This combination allows for the introduction of event-based behavior into the time-triggered scheduling concept, while still providing the benefits of the time-triggered scheduling concept. With a large task set, this comes at the cost of large complexity in the scheduling algorithm.

This work aims to reduce the complexity of the mode change scheduling concept by introducing a graph based scheduling approach, which dynamically changes between predefined schedules during the execution of the tasks, while still guaranteeing that all tasks are completed with their given periods.

2.3.4 Hypervisors and Real-Time Operating Systems

To execute the given jobs on a given platform under the constraints of temporal isolation and memory isolation as well as to support the scheduling concepts of time-triggered or event-triggered scheduling, a hypervisor or a real-time operating system (RTOS) is used. This subsection introduces the concepts of hypervisors and real-time operating systems and compares available commercial as well as open-source solutions.

Table 2.1: Hypervisors and real-time operating systems. The table shows the provided scheduling concepts in the considered operating systems.

	Self-description			**Supported Scheduling**	
System	Type-1	Type-2	RTOS	Time-triggered	Event-triggered
PikeOS [38]	✓	✗	✓	✓	✓
Xen [39]	✓	✓	✗	✗	✓
XtratuM [86]	✓	✗	✗	✓	✗
FreeRTOS [87]	✗	✗	✓	✗	✓
RT-Linux [88]	✗	✗	✓	✗	✓
TT-Linux [89]	✗	✗	✓	✓	✗
Zephyr [90]	✗	✗	✓	✓	✓

Operating systems (OSs) allocate shared resources, such as memory and communication systems, as well as the start times and, in multicore systems, the

chosen execution unit to the jobs of the given tasks [91]. On the operating system level, memory accesses can be virtualized, as can be the access to shared resources. Memory virtualization specifies the translation of virtual addresses in the executables to physical addresses in the memory, allowing for memory isolation and the implementation of access protection with methods like [40]. The on-platform scheduler decides the order of execution of the jobs and the assignment of the jobs to the chosen execution units. The implemented concept of the scheduler varies between different operating systems, with some providing a fixed priority scheduling, while others provide a time-triggered scheduling concept and others offering choices to the user Table 2.1 gives an overview of the provided scheduling concepts in the considered operating systems.

Hypervisors on the other hand are understood to be a layer between the hardware and the operating systems or between the hardware and bare-metal applications [92]. The difference between hypervisors and operating systems is thus fuzzy, with hypervisors providing an abstracted functionality of operating systems. A hypervisor then provides a virtual machine to the software layer above, which can either be an operating system or a bare-metal application. These virtual machines can be given a virtualized access to the shared resources and memory, allowing for temporal isolation and memory isolation. With this abstraction, the virtual machines are called guest machines and an operating system executed on one of these virtual machines is called a guest operating system. Hypervisors can be classified into two types. In type 1 hypervisors, the hypervisor is executed directly on the hardware and provides a virtual machine to the software layer above. In the second type, called type 2 hypervisor, the hypervisor is executed on top of another hypervisor or operating system and provides a virtual machine to the software layer above. Embedded systems usually make use of type 1 hypervisors, while type 2 hypervisors are usually used in desktop systems [18, 92].

Hypervisors furthermore allow for an abstraction of the underlying hardware. Not only the memory addresses and the access to shared resources can be virtualized, but also the access to the execution units can be virtualized, as can the execution units themself [18]. Virtual machines can be given a virtualized access to the execution units, allowing for temporal isolation, or given exclusive access to the execution units. A mix of these approaches is also possible. The access to shared memory and shared resources can be controlled in either a round-robin and cyclic manner or on an event-triggered basis with given priorities. Again, one hypervisor can also support both approaches for different cores [38].

In summary, hypervisors can be thought of as a generalization of operating systems, providing a virtual machine to the software layer above, which can either be an operating system or a bare-metal application. An operating system executing an application might be thought of as a type 1 hypervisor running a bare-metal application on a given virtual machine with the same characteristics as the underlying hardware. ***In the context of this work, both hypervisors and real-time operating systems are abstracted and deemed usable as the deployment platform for the scheduling concepts, as long as a time-triggered scheduling concept is provided.***

2.3.5 Task Design Concept and Implementation Guidelines

To be able to deploy tasks in the real-time domain, implementation guidelines need to be fulfilled. These tasks need to include wrapper code or be included in the scheduler in a way that the scheduler is able to schedule the tasks. Listing 2.1 shows the skeleton of the implementation of a real-time task, which is required in related works [89] to be followed to fulfill the requirements of the scheduler.

```
1  int main(int argc, char *argv[]) {
2      task_main(); // Start the main task
3      return 0; // Exit the program
4  }
5
6  void task_main(){
7      init(); // Initialize the task
8      while (true) {
9          yield_for_activation(); // Wait for activation
10         execute(); // Execute the task
11     }
12 }
13
14 void execute() {
15     memory_phase(); // Memory access code goes here
16     execution_phase(); // Task execution code goes here
17     // ... continue phased execution ... //
18     memory_phase(); // Memory access code goes here
19 }
```

Listing 2.1: Code snippet for a real-time task, adapted from [89, 93]

The main task starts when the `main` function calls the `task_main` function. Inside this `task_main` function, the task is initialized in the `init` function. Here, parameters are set, and the initial data is read in. The function then moves on to the `while` loop, which is executed until the task is suspended by the scheduler inside the operating system or the hypervisor. Inside the `while` loop, the periodic task execution is implemented by first yielding for activation in the `yield_for_activation` function. By first calling this function, the initialization of the task is separated from the first job and the task can then be set to be ready. The main functionality given by the `execute` function is then executed once the scheduler has marked the task as ready. Because the task itself is not aware of the scheduling concept, it is possible to implement the task in a way that it can be used in both time-triggered and event-triggered scheduling concepts. The `yield_for_activation` function is then implemented in a way, which sets the task to sleep and ready for execution, once awoken by the scheduler. The `execute` function then continues the execution of the task. In this work, a phased execution model is required, where the task is executed in phases, which each have a given time window. During the `memory_phase`, the task is allowed to access shared resources and memory, while during the `execution_phase`, the task is not allowed to access shared resources and memory, which follows a phased execution model like the AER model [26] or the PREM [25]. Once the execution of one job finishes, the `while` loop starts over putting the task to sleep and ready for execution, once awoken by the scheduler.

In this work, this task design is used for the scheduled tasks, which are executed in a phased execution model. This provides underlying hypervisors or real-time operating systems with a unified interface to start job execution.

This page intentionally left blank

Chapter 3

State of the Art

After the introduction of the fundamental concepts in Chapter 2, this chapter presents the state of the art using the given fundamental concepts. The state of the art is divided into three main sections. Section 3.1 introduces scheduling algorithms for time-triggered systems and presents mode change scheduling approaches. Section 3.2 presents the state of the art in creating anytime neural networks for time-constraint systems. Lastly, Section 3.3 presents the state of the art in anomaly detection to trigger mode changes in the mode-change schedules as well as the anytime neural networks. How these works compare to the presented work is then shown in the conclusion in Table 9.1 and Table 9.2.

3.1 Time-Triggered and Mode Change Scheduler

In the time-triggered domain, many scheduling methods have been proposed for varying constraints and requirements. This chapter aims to give an overview of relevant methods and their properties. First scheduling methods for time-triggered systems are presented. Then, mode change scheduling is introduced in the context of embedded systems.

3.1.1 Schedulers for Mitigating Shared Resource Contention

While many works exist in domain of embedded scheduling, this section focuses on the following relevant works: Oliviera et al. [94], Pellizoni et al. [25], Durrieu et al. [26], Aceituno et al. [95], Yip et al. [96], Roeder et al. [97] and Ueter et al. [98].

Oliveira et al. [94], present an approach for contention investigation and mitigation for low-end multicore systems. The authors contribute a interference

mitigation framework composed of a design-time tool, which automatically adds milestone monitoring components to tasks as well as a run-time tool, which monitors the execution for these milestones. Together with the interference assessment framework, the memory contention is managed. For the definition and evaluation of the contention, different evaluation scenarios, specifically for low-end multicore systems, are presented. This includes different bus-masters, platform-dependent features like caches, interference creators as well as benchmark definitions. The evaluation itself is done for a Musca-A1 platform. The contention evaluation platform is design to be adaptable for different MCU platforms by using a BSP abstraction layer. Secondly, the testing tool can be configured for different scenarios to place code and data region for each test, enable platform components, select most effective benchmarks and choose optimization options. The main part is given by the interference mitigation tool, which monitors the execution of the tasks and checks for the milestones. If the remaining slack lies below a preconfigured threshold, the mitigation tool will stall interfering cores, thus reducing the interference or raising the bus priority on the core of the task. With this work, an inclusion into time-triggered scheduling is possible, when the interference mitigation tool is used to reduce the interference of tasks on different cores [94]. *The milestones itself can be interpreted as checkpoints, which are similar to the checkpoints introduced in Section 5.6. Furthermore, the time between milestones can also be interpreted as execution phases, which are then scheduled in a time-triggered manner.*

In the predictable execution model (PREM), Pellizoni et al. [25] present a scheduling method for commercial off-the-shelf (COTS)-based embedded systems. The authors mention several challenges, which are necessary to be overcome for a predictable execution on COTS platforms. First, input/output (I/O) peripherals with bus access introduce contention capabilities onto memory and bus systems unpredictably. Second, access patterns of COTS CPUs are highly variant, and cache fetches are difficult to predict, forcing pessimistic assumptions. Third, COTS bus arbiters are designed to achieve fair arbitration instead of predictable access patterns for critical tasks. To mitigate these challenges and to achieve a predictable execution, the authors propose a scheduling method which makes use of a phased execution and the existing hardware of COTS systems. Tasks are split into two phases, in the first phase of the predictable interval, the task is allowed to access the bus system and shared memory, filling the local cache with all variables accessed within the task. In the second phase, the task is executed without any shared bus access and thus avoids interferences with other tasks. No two jobs are allowed to access the bus at the same time, thus avoiding contention. The first phase

is called memory phase and the second phase is called the execution phase. Further constraints to the implementation of tasks are given. These are the avoidance of linked list-like structures; for heap-allocated structures, the total size must be known to add correct prefetching code; the avoidance of system calls, heap-allocators or stack-allocators within loops; the avoidance of function calls; and the possibility of pointer analysis for call-graph construction. With these constraints, the compiler is able to generate the corresponding phases for each task [25].

Also using a phased execution model, Durrieu et al. [26] present a predictable flight management system for multicore processors together with the acquisition execution restitution (AER) model. The authors point out three main contributions for their improved scheduling on COTS systems. First, they specify periodic tasks being executable on dedicated cores with a time-triggered offline approach and a bare-metal execution layer. Secondly, they include aperiodic tasks on dedicated cores with a global deadline monotonic scheduling approach. Third, the communication between cores is done via a memory component as a shared resource, in which a messaging buffer area is reserved for communication. The main contribution is their execution model and the scheduling method for sporadic tasks. The scheduling model is a phased execution model, similar to the PREM. The authors define an acquire-phase, which is the first phase and fetches all variables to local memory. The execution phase comes second and executes the task without any bus access. The last phase is the restitution phase, which writes back all modified variables to the shared memory. The scheduling of periodic tasks is done offline in a time-triggered schedule. During runtime, aperiodic tasks are scheduled with a global deadline monotonic scheduling approach within gaps between the periodic tasks [26]. Both phased execution models AER and the PREM build the foundation for many related works as well as this work. ***In contrast to these works, the scheduler of this thesis supports multiple shared resources as well as an arbitrary placement of the phases within the schedule.***

Aceituno et al. [95] present a scheduling method to reduce hardware resource contention of hard real-time tasks in multi-processor system-on-chip (MPSoC)s. The authors achieve this by introducing a task model which includes the expected interference due to contention into the execution time of the tasks. It furthermore includes a phased execution model for time-triggered periodic schedules. The scheduler aims to map a task set to a set of homogeneous cores under the assumption of no allowed preemption and no migration. All tasks in the task set are considered periodic and given a worst-case execution time (WCET). The assignment of task to cores is split from the assignment

of job start times. The allocation (i.e. the assignment of tasks to cores) is done to balance the load between cores to aim for a schedulable allocation. The scheduling of tasks is the second step and done independently for all cores. Because the interference between tasks and the expected execution time increase is determined beforehand, the WCET of each task can be used to determine the schedulability of the task set and if the task set is schedulable, the scheduler will determine the start times of each job [95]. ***This approach is different to the thesis, because of the split approach for allocation and scheduling for phased execution. It provides a contrary method to the thesis, which uses a holistic approach to schedule and allocate tasks in one step.***

The mixed-criticality approach of Yip et al. [96] presents a scheduling method for scheduling tasks with three different criticality levels. The work relaxes the synchronous approach of assigning a fixed implicit deadline to all tasks by allowing a slack to occur for lower criticality tasks. This slack violates the synchronous model of communication, where each task is expected to be executed with the same frequency. The authors provide a buffering approach to allow a slack to occur for lower criticality tasks, while still guaranteeing the timing constraints of higher criticality tasks. The provided multiprocessor scheduler maximized the utilization, by distributing slack across all tasks. Three criticality levels are defined, called life, mission and non-critical. The scheduling of the time-triggered schedule is done using an integer linear programming (ILP)-based approach where the utilization is given as the objective function. A comparison towards early-release earliest deadline first (EDF) shows a wider range of schedulable task sets for the proposed approach [96]. ***This thesis uses a similar approach to model the criticality of tasks by allowing a given slack to occur for the period of the tasks. This slack is continuous in this thesis, while Yip et al. use a discrete approach to model the criticality of tasks.***

Roeder et al. [97] present a phased execution model, which schedules directed acyclic graphs (DAGs). The authors present a scheduling for tasks on heterogeneous multicore systems with a focus of scheduling tasks on CPUs and GPUs. The authors present a scheduling model which splits tasks into phases with different versions for each phase and propose an offline scheduler for these phases. Tasks are represented as DAGs, allowing for the representation of precedence constraints between the phases. In contrast to Pellizoni et al. [25] and Durrieu et al. [26], Roeder et al. do not consider memory contention but add the interference cost to the execution time of each phase. In addition to the definition of the phased execution model, Roeder et al. present a scheduling method for DAGs on their heterogeneous platform. The authors

deploy a heterogeneous earliest finish time (HEFT) approach, for which they empirically show that, in combination with their model, the schedulability is improved by 11 % up to 25 % compared to other works [97]. ***This work is relevant for this thesis, because it shows how multi-version tasks and precedence constraints are combined to create a scheduling method for heterogeneous platforms. In contrast to this work, the scheduler of this thesis supports more fine-grained phases to reduce the interference cost for access of shared resources.***

As a last related work, Ueter et al. [98] present a scheduling method for GANG tasks in a stationary environment. GANG tasks are tasks, which are executed on multiple execution units in parallel. The authors contribute a schedulability test for a stationary GANG task assignment, where the GANG tasks are deadline-constrained and sporadic. Second, Ueter et al. present a class of partitioning algorithms based on the stationary assignment that yield beneficial theoretical properties. The fixed-priority GANG scheduling needs to keep the inference with higher-priority tasks in mind. The authors identify a class of feasible GANG to processor assignment algorithms which allows for the formulation of a worst-case performance guarantee. The evaluation of their algorithm focuses on the acceptance ratio of synthetic GANG task sets in respect to the utilization and for light, moderate and heavy sporadic settings. The authors show that their proposed algorithm slightly outperforms other state-of-the-art algorithms [98]. ***This work is relevant for this thesis, because it shows how GANG tasks are scheduled on a multicore system. This thesis interpretes gang tasks as tasks which require the simultaneous start, denoted by precedence constraints.***

While these works focus on the scheduling of tasks on multicore systems, they do not consider mode changes and this thesis introduces an extension of these works to form a holistic approach. The following section focuses on scheduling methods supporting mode changes.

3.1.2 Mode-Change Approaches for Embedded Systems

While the previously presented works focus on the scheduling of phased based tasks and mitigating contention in heterogeneous systems, this section focuses on scheduling methods supporting mode changes. The following works will be presented: Bhuiyan et al. [99], Skalistis et al. [100] Negrean et al. [85], Sundar [101].

As Ueter et al. [98], Bhuiyan et al. [99] present a scheduling method for GANG tasks. The authors furthermore present a method for a mixed-criticality system, which is able to handle mode changes. The authors generalize the GANG task model for a mixed-criticality by extending the execution time as well as the parallelism degree of the GANG tasks. Furthermore, a global earliest deadline first (GEDF) scheduling method is presented, which includes a virtual deadline. Bhuiyan et al. extend the scheduler for the mixed-criticality approach and give tasks one of multiple criticality levels. An upper speedup bound for the GEDF scheduling method is presented. The evaluation is done for randomly generated task sets with different utilization and parallelism degrees. The mode change is implemented as the switch between a normal condition where all tasks must meet their deadline and a critical condition, where only the high-criticality tasks must meet their deadline. This work provides a further extension to Ueter et al. by allowing a mixed-criticality approach and introducing a mode-change possibility [99]. ***This work is relevant for this thesis, because it shows how GANG tasks are able to be scheduled in a mode-change and mixed-criticality environment. Extending Ueter et al., this thesis introduces a generalized mode-change approach with more than two modes.***

Skalistis et al. [100] present an interference-sensitive runtime adaptation method for time-triggered schedulers, effectively implementing a mode-change when the runtime interference exceeds a certain threshold. For this, the authors introduce an interference-sensitive worst case execution time estimation (*is*WCET). The proposed method extends the state of the art by adding control phases, whenever possible, which are allowed to be executed in parallel to the control phases of other tasks. Together with a response time analysis, the authors show that the execution with their interference-sensitive approach introduces no overhead and is free of timing anomalies. The task model supports a phased execution and an estimation of the WCET as well as of the precedence constraints. The input is an existing time-triggered schedule, which is adapted at runtime for this work. Non-preemptive scheduling is employed. The model extends the execution phase (from the AER model) with a ready phase and an update phase. The ready phase marks the task as ready and checks, if all dependencies are met. These dependencies are modeled as the completion of precedence constraints in this thesis and in other works. Should the task be ready, the execution commences and afterwards the update phase is executed, which notifies waiting tasks of the completion [100]. ***This work shows how a runtime environment can be used to deploy a time triggered schedule and to guarantee the timing constraints of the tasks in a near-bare-metal environment.***

The main goal of mode change scheduling is bounding the transition time between modes. Negrean et al. [85] present an approach with this goal. The main contributions are defined as eliminating restrictions regarding the communication of the tasks and to derive mode change transition latencies of distributed applications without cyclic dependencies. Determining the trigger time to change modes necessitates the bounding of the transition time between modes. These contributions are introduced for an asynchronous mode change protocol which supports the continuation of tasks of the old mode into the new mode. The system is modeled as a set of tasks mapped and executed with a priority-based preemptive scheduler on parallel execution units and with multiple shared resources. Each task is given a WCET estimation and a deadline. The mode change model includes different modes which are characterized by different associated release patterns of the tasks. Furthermore, only a subset of the tasks can be assigned to each execution mode. The mode change is triggered by a mode change request, which is sent to the scheduler. "Estimating the latency of the mode change is then given by the maximum local transition latency of the corresponding tasks maximum busy window for the tasks input activation patters" [85]. In other words, each task has a maximum busy window, which is the time between the first activation of the task and the completion of the last job of the task. The maximum busy window is determined by the input activation pattern of the tasks. The mode change transition latency is then given by the maximum local transition latency of all tasks, which is determined by the maximum busy window of each task [85]. Even though this work was introduced in 2011 for event-triggered scheduling, it is still relevant for this thesis, because it lies the foundation for determining a suitable mode change protocol and the bounding of the transition time between modes. ***The asynchronous mode-change with the continuation of tasks is relevant for this thesis, because it allows a more flexible mode-change approach. Since the minimization of the transition time is the main goal of contributions to mode-change systems, this thesis aims to minimize the transition time continuing old tasks into the transition period and start new tasks as early as possible.***

Lastly, Sundar [101] introduces a mode-change mixed-criticality system, which allows for the modeling of execution times as well as budget, release patterns and task degradation during mode changes. Together with precise rules to handle budget overruns within mode changes and during modes the algorithm computes offset values for new tasks as well as the criticality values. The mode-change algorithm is embedded in a context-aware mixed-criticality system, guided by requirements from international standards like ISO 26262 [7, 8]. The overall approach aims to keep deadlines of critical tasks while suspend-

ing or degrading lower-criticality tasks. This system extends to more than 2 criticality levels and is aimed to stay safe during mode changes and degradation of lower-criticality tasks. The scheduling is based on a graph-based mixed-criticality model, which aims to provide higher flexibility in the choice for degradation and to model functional mode change aspects [101]. ***This thesis uses a similar approach in integrating mode changes into a mixed criticality system. While Sundar uses a graph-based approach, this thesis uses a time-triggered approach to schedule the tasks and while Sundar uses a finite amount of criticality levels to degrade tasks, this thesis aims to automatically generate execution mode variants to model different degradation levels.***

3.2 Anytime Neural Network Scheduling and Design

For the related work regarding anytime neural networks, a distinction between the design of anytime neural networks and the scheduling of anytime neural networks is made. The related work regarding the design of anytime neural networks is presented in Section 3.2.1 and introduces both methods, where a given backbone network is modified to support anytime inference and methods, which generate the anytime neural network from scratch. The related work regarding the scheduling of neural networks is presented in Section 3.2.2 and features both works for anytime neural networks and for networks without early exits.

3.2.1 Design of Anytime Neural Networks

For the design of anytime neural networks, BranchyNet [68], AdaDet [102], EDANAS [103], S.E. EENNs [104], XAI-Exit [105] and E²CM [106] are presented.

First, BranchyNet [68] is an anytime neural network, which aims to reduce the inference time by being given an architecture with two early exits in addition to the original exit. The main novelties of BranchyNet were the introduction of the early exits after side branches, the use of a joint loss function to train the network and the mitigation of vanishing gradients by providing additional and more immediate gradient values. The architecture of BranchyNet is created from scratch and is not based on any given backbone network. The standard

architecture is build with the following exit branches: one early exit branch of two convolutional layers and one fully connected layer after the first convolutional layer; one early exit branch of one convolutional layer and one fully connected layer after the third convolutional layer; and the original exit branch of five early exit layers and three fully connected layers. The training of BranchyNet is performed with a joint loss function, which is the weighted sum of the loss functions of all exit branches. The decision to choose an early exit is made with a confidence value based on the entropy of the branching layer [68]. ***BranchyNet is one of the first implementations of anytime neural networks and thus introduced the concept of early exits. However, the architecture is not based on a pretrained backbone network. In this thesis, the design of anytime neural networks is based on a pretrained backbone network.***

EDANAS [103] introduces a neural architecture search (NAS) framework to determine the position for adding early exits onto existing backbone networks. Where in related work, the placement of early exits is determined manually, EDANAS uses a neural architecture search to determine the optimal placement of early exits based on the estimation of the total number of multiply-accumulate (MAC) operations. EDANAS is given an input dataset, a backbone network and the set of early exit branches. The framework then returns a set of k optimal early exit architectures. The training phase of the early exits relies on a softmax cross-entropy loss function, which implements a weighted joint loss function. The training is then undergone by choosing a trained backbone network from the set of the input networks, adding the early exits following a given set of specifications and then train the combined network with the joint loss function. Afterwards the resulting networks are evaluated based on the number of multiply-accumulate (MAC) operations and the accuracy of the network. The best performing networks are then returned as the output of EDANAS [103]. ***EDANAS is similar to this thesis in that it uses an automated approach to determine the placement of early exits. However, the lack of constraints other than the number of MAC operations makes this work less suitable for the integration into the time-triggered scheduling.***

S.E. EENNs [104] introduces the concept of single exit early exit networks, which aims to determine the configuration of the internal classifiers (called early exits in related work) of the backbone neural network. Casale et al. [104] define the internal classifiers as the early exit branch of the neural network, which can be placed after each internal layers. The classifier is trained to determine a classification in conjunction to a confidence value, which is used to determine the decision to stop inference after the given exit or to continue

the execution in the original network. The training itself is performed independently for the exit and the backbone network. The single exit is trained independently of the backbone network and with different weights and biases for each exit, i.e. only the architecture of the networks is constant between exits, the trained weights are different. Casale et al. [104] then introduce two scheduling approaches, one reactive to the current inference and one proactive given the amount of released tasks. In S.E. EENNs, it is assumed that all input data have the exact same execution time for a given exit [104]. ***S.E. EENNs is similar to this thesis in that it keeps the architecture of the used exit constant and in the way of providing both intra- and inter-inference scheduling. However, the lack of multiple networks in the scheduling approach, the lack of adaptation to anomalies and the simple timing model makes this work less suitable for the given system.***

Rahmath et al. [105] introduce XAI-Exit, which is a framework to provide interpretability to the decision of early exits in anytime neural networks. This is achieved by first, providing an early exit prediction with so-called curriculum learning which predicts the optimal exit using a confidence value as well as the interpretability and a stability metric; second, providing exit attribution maps which visualize the exit-wise attributions and aggregate information for a holistic prediction of the early exit decision; and third, providing standardized attribution techniques to benchmark the exit. The execution in the framework is split into four phases. The first phase is the training phase where a joined training of the early-exits is undergone. Second, the pre-interface phase trains both the processes' validation data and a selected subset of the training data to compute metrics for each exit. Third, the inference phase, processes and input and at each exit the data is either forwarded, or the inference exits at the given layer. Forth and last, the explanation phase generates explanations for the predictions at each exit. The choice of the given exit resembles an intra-inference type of scheduling. ***XAI-Exit is similar to this thesis in that it provides intra-inference scheduling. Even though this provides a scheduling method similar to this thesis with the decision based on anomalies, XAI-Exit lacks an inter-inference scheduling and a relation to the estimated execution time***

E^2CM [106] introduces a training method based on the sample means which does not require gradient-based training of the internal classifiers (called early exits in related work). The training starts with a backbone network and a train dataset. The configuration of the early exits is then done by tracing the intermediate results at the given layers and then computing the class mean of the class of all traced data. During inference a distance metric is used to determine the distance of the intermediate result to the class mean. If

the distance lies below a given threshold, the inference is stopped and the result is returned. Otherwise, the inference continues to the next layer [106]. ***This result is relevant because it shows that the anomaly detection methods implemented in this thesis can be used to accurately classify the intermediate results into anomaly classes and to decide on the placement of the early exits.***

3.2.2 Deadline-aware Scheduling of Neural Networks

Scheduling neural networks under given timing constraints is a well-studied field, where many works exist. The following works are comparable to this thesis and are presented here: SLEXNet [107], AdaEE [69], RT-mDL [108], Pantheon [109], LaLaRand [110] and DNN-SAM [111].

The first scheduling framework, SLEXNet [107], introduces a scheduling framework and algorithm which evaluates external factors and requirements to determine a subarchitecture to execute. These subarchitectures are previously created for a set of given speed-power characteristics for a backbone network. In contrast to other works, SLEXNet uses slimming in addition to early exits in their model. Slimming specifies the reduction of the number of channels in a given layer and reduces the number of parameters and thus the execution time of the network. The presented example network is able to handle 30 inferences per second with the full network and 60 inferences per second require the use of early exits and slimming. For 150 inferences per second, a batch size of 8 and one of the earliest exit needs to be used to keep up [107]. ***SLEXNet is similar to this thesis in that it provides a scheduling framework in combination with the adaptation of the architecture. However, the lack of multiple networks in the scheduling approach and the lack of inter-inference adaptation is an advantage of this thesis.***

AdaEE [69] introduces a scheduling framework, which adapts the threshold value for choosing early exits based on the upper confidence bound algorithm during runtime. The adaptation accounts for the known confidence at the side branches, the envisioned confidence at the side branches and the corresponding overhead. The runtime adaptation of the threshold value is done by using a multi-armed bandit approach, which calculates the expected confidence value for each exit and executes the exit, if the confidence value is above the threshold. After the confidence is calculated, the threshold is adapted based on an instantaneous reward (if the exit is chosen) or the instantaneous regret (if the exit is not chosen) after processing the exit. In contrast to other works, AdaEE does not consider the automated neural architecture search for the

placement of early exits, but rather assumes that the early exits are already placed and only adapts the threshold value for the already existing early exits. ***AdaEE is similar to this thesis in that it adapts parameters of the exit choice during runtime. In this thesis, the estimated execution time is updated during runtime.***

In contrast to the prior works, rt-mDL [108] introduces a scheduling framework for deep learning inference on embedded systems without early exits. Instead, model scaling is used to adapt the architecture of the considered network. The incoming tasks are given mixed deadlines and are considered to be periodic. The execution model itself allows for the modeling of the execution on the GPU as well as the execution on the CPU. With this combination of different neural network models, the modeling of non-negligible execution times on the CPU, mixed deadlines and the adaptation capabilities of the architecture, rt-mDL is able to provide a scheduling framework for deep learning inference on embedded systems. The System architecture includes priority-based deadline task scheduler which then scales the executed model based on the given deadline and priority. If the deadline of a higher priority task is expected to be missed, lower priority tasks are scaled down to free resources for the higher priority task. The scaling of the model is done by using a set of pre-trained models with different scaling factors. Additionally, the scheduling framework packs the CPU parts and the GPU parts to allow for a parallel execution and uses a random-forest multi-objective scheduling approach to minimize deadline misses [108]. ***The rt-mDL approach is similar to this thesis in that it provides a scheduling framework for deep learning inference on embedded systems and the splitting of the tasks into CPU and GPU parts.***

Pantheon [109] is the most similar to the work presented in this thesis. Pantheon proposes a preemption design for multi-network inference on embedded systems. The preemption mechanism is based on early exits and the scheduling framework is able to also add these early exits to existing backbone networks. Pantheon considers real-time tasks with a high priority and best-effort tasks with a lower priority. The implementation of the scheduling framework considers three priority assignment strategies. The first strategy is a rate-monotonic priority, which assigns a higher priority to the tasks with a shorter execution duration. The deadline-monotonic scheduling assigns a higher priority to tasks based on their deadline. Third, the rt-mDL priority assignment strategy can be used with a multi-objective priority assignment strategy to minimize the number of missed deadlines based on a random-forest strategy. Lastly, Pantheon is able to automatically generate early exits to allow implement a preemption strategy which provides a result [109]. ***Pan-***

theon is similar to this thesis in that it provides a scheduling framework for deep learning inference on embedded and multi-network systems. Additionally, automatically generated early exits are used to provide a preemption strategy. However, the lack of anomaly detection and the lack of inclusion in the time-triggered scheduling makes Pantheon less suitable for the integration into this thesis.

While LaLaRAND [110] does not focus on the scheduling of anytime neural networks, it provides a fixed-priority scheduling approach for a layer-level scheduling of deep neural networks between a CPU cluster and a GPU cluster. The scheduling framework takes runtime overhead as well as resources for memory into account, which are necessary for data movement in case of a reallocation between CPU or GPU during inference. The decision for allocation of layers prior to inference is based on all released tasks, their deadlines, the priority of the tasks and the estimated execution time of the layers of each task. This algorithm first assigns all layers to the GPU cluster and then iteratively reallocates layers of the highest-priority task to the CPU cluster, if the deadline of this task is possible to be met while executing tasks on the CPU cluster. Additionally, dynamic runtime reduction is possible via dynamic quantization of layers executed on the CPU cluster, which reduces the execution time at the cost of accuracy. This quantization then introduces slack, which allows for a layer reallocation. ***LaLaRAND is similar to this thesis in that it provides a scheduling framework for deep learning inference on embedded systems and by focussing on a layer-level scheduling. However, the lack of early exits, the limitation of considering two fixed execution units and the lack of inclusion in the time-triggered scheduling makes LaLaRAND less suitable for the integration into this thesis.***

Lastly, DNN-SAM [111] introduces a layer-level scheduling framework for convolutional neural networks based on a split and merge approach, which maximizes the accuracy of object detection in the safety-critical central region of the input image. The algorithm starts by determining the safety-critical central region based on additional depth and time-to-collision information provided by LiDAR and IMU sensors. This center region is then cropped and evaluated without further preprocessing with the full network. The complete image is scaled down and evaluated in parallel to the center region depending on the given deadline and utilization of the execution units. If the utilization does not allow for the execution of the (scaled down) full image, the center region remains the only evaluated part of the image. The results of the center region can then be used immediately in the next step of a given pipeline at an earlier time than the results of the full image. Should the full image be evaluated, the results of the center region are merged with the results of the

full image to provide a complete result. ***DNN-SAM is similar to this thesis in that it provides a scheduling framework, which allows for a lower quality result to be used at an earlier time than the full result. It furthermore allows for multiple versions of a network to be executed, which can be modeled in this work. However, the lack of early exits, the limitation of considering two fixed execution units and the focus on one specific use case makes DNN-SAM less suitable for the integration into this thesis.***

3.3 Anomaly Detection in ML-based Systems

Lastly, anomaly detection methods for ML-based systems are presented. FACER [73] and ODIN [74] are relevant methods, which use intermediate results of the layers of the neural network to detect anomalies [73, 74].

FACER [73] uses the intermediate results of all feature maps of a convolutional neural network to detect anomalies. The values of all feature maps are summed up and the resulting vector is then fed into a one-class classifier to detect anomalies. The classifier itself is build from a fully connected layer, which takes the summed feature maps as input and outputs a single value, which is then used to determine whether the input is an anomaly. The component itself can be trained for noise detection, unseen class detection and hardware fault detection among others. The sum of the feature maps is used to reduce the dimensionality as well as to introduce invariance to the spatial location of the features. Additionally, the vector is normalized. With this setup, the method is able to detect anomalies in a variety of scenarios, including noise detection, unseen class detection, hardware fault detection, blurring and resolution reduction. This also allows for an independent training of the anomaly detection modules from the inference. For each training data sample, the layer results are traced and then used as the training input for the anomaly detection module, allowing for a potential reduction in training time.

This idea of using ML-based classifiers on the feature maps for anomaly detection is replicated in this thesis. FACER uses all feature maps of a convolutional neural network, while the method proposed in this thesis uses only the feature maps of a subset of layers to allow for a faster decision time. Additionally, this thesis extends the possible classifiers and dimension reduction methods.

A similar approach is used in the generalized ODIN approach [74], where the anomaly detection is also based on the intermediate results of the layers of a neural network. In the non-generalized case of ODIN, the Mahalanobis distance [112] is used to detect anomalies. The Mahalanobis distance is a distance measure that takes the correlation of the data into account and is defined with:

$$D_M(x) = \sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)} \tag{3.1}$$

where x is the input vector, μ is the mean of the data and Σ is the covariance matrix of the data. The Mahalanobis distance is used to measure the distance of the input vector to the distribution of the out-of-distribution and in-distribution data. The generalized ODIN approach extends this method by not relying on the out-of-distribution data, but instead only using the in-distribution data to detect anomalies. If the traces of are to far away from the traces of the in-distribution data, the input is considered an anomaly [74]. ***This idea of using the Mahalanobis distance for anomaly detection is also used in this thesis. As with the FACER method, ODIN uses all feature maps of a convolutional neural network, while the method proposed in this thesis uses only the feature maps of a subset of layers to allow for a faster decision time.***

Summary

This section shows, that this thesis is positioned for research questions relevant for the state of the art. As with time-triggered schedules as well as for the generation and scheduling of anytime neural networks, the state of the art is extended and combined to create a holistic approach to bring ML-based tasks into mode-change schedules. The system of this thesis is completed with an extension of the state of the art in anomaly detection to trigger these mode changes.

This page intentionally left blank

Chapter 4

Time-Triggered Mode-Change Scheduler

This chapter introduces the task model as well as the scheduler to generate time-triggered schedules for HESoCs. The evaluation of these chapters are given in Section 7.1 until Section 7.3.

4.1 Model for Time-Triggered and Multi-Version Tasks

The presented task model forms the basis for the time triggered scheduling approach. This phased execution model is based on the model introduced by [Sta25a] and is extended in this work. Table 4.1 gives an overview of the sets describing tasks, units and resources of the model.

The task set is defined as

$$\mathcal{T} = \{\tau_i \mid i \in [0, N_{\mathcal{T}})\}, \tag{4.1}$$

and contains one element per task. The total amount of tasks is given by $N_{\mathcal{T}}$. Task versions are specified implicitly by further functions.

Units are collected as elements in the set $\mathcal{U}$, which is defined as

$$\mathcal{U} = \{u_j \mid j \in [0, N_{\mathcal{U}})\}, \tag{4.2}$$

and contains one element per execution unit. An execution unit represents the smallest unit, which can execute a task. In an embedded environment, this represents a CPU core, a hardware accelerator or a cloud instance to where a task could be offloaded. The total amount of execution units is given by $N_{\mathcal{U}}$.

Shared resources are represented as elements in the set $\mathcal{R}$. This set is defined as

$$\mathcal{R} = \{r_k \mid k \in [0, N_{\mathcal{R}})\}. \tag{4.3}$$

Shared resources are used to model platform components inside embedded systems, which are accessible by tasks on multiple different execution units. Shared resources are not able to execute tasks independently. Two shared resources $r_1, r_2 \in \mathcal{R}$ are considered to be different if one task $\tau_1 \in \mathcal{T}$ is able to access a shared resource $r_1 \in \mathcal{R}$ at the same time as another task $\tau_2 \in \mathcal{T} \setminus \{\tau_1\}$ accesses another shared resource $r_2 \in \mathcal{R} \setminus \{r_1\}$ without a collision. Collisions can occur at an interconnect, on-platform communication interfaces or at access registers. Should the simultaneous access result in a delay, then both accessed resources are modeled as one shared resource. The total amount of shared resources is given by $N_{\mathcal{R}}$.

Table 4.1: Sets, description and symbol of the sets of the task model.

Component	Formula	Description
Task Set	$\mathcal{T} = \{\tau_i \mid i \in [0, N_{\mathcal{T}})\}$	Tasks are elements of this set. Each task describes one dedicated software component and specifications are version-dependent and described explicitly.
Task Amount	$N_{\mathcal{T}}$	Total number of tasks.
Unit Set	$\mathcal{U} = \{u_j \mid j \in [0, N_{\mathcal{U}})\}$	Execution units are elements in this set. One execution unit represents the smallest unit, which can executed a task.
Unit Amount	$N_{\mathcal{U}}$	Total number of execution units.
Resource Set	$\mathcal{R} = \{r_k \mid k \in [0, N_{\mathcal{R}})\}$	Shared resources are elements in this set and are used to model platform components inside embedded systems, which are accessible by tasks irrespective of the execution unit.
Resource Amt.	$N_{\mathcal{R}}$	Total number of shared resources.

A subset of all tasks are required to provide a result with a given period. In this model, this is represented by an associated period. This period is modeled as an implicit deadline, which is defined as the function

$$D : \mathcal{T} \to \mathbb{Q}^+ \cup \varnothing. \tag{4.4}$$

Additionally, an allowed slack is introduced via the function

$$\gamma : \mathcal{T} \to [0,1]. \tag{4.5}$$

This slack represents the allowed deviation from the given period. A slack of $\gamma(\tau_i) = 0$ indicates that task τ_i must start exactly with the given period $D(\tau_i)$. A slack of $\gamma(\tau_i) = 0.1$ indicates that task τ_i may start the execution up to 10% later or earlier than the given period $D(\tau_i)$. The allowed starting time for task τ_i lies therefore in the interval $[(1-\gamma(\tau_i)) \cdot D(\tau_i), (1+\gamma(\tau_i)) \cdot D(\tau_i)]$.

A task $\tau_i \in \mathcal{T}$ with an associated deadline $D(\tau_i) \neq \varnothing$ is expected to start within this given interval. If a task $\tau_i \in \mathcal{T}$ has no required period, then $D(\tau_i)$ returns the empty set $\varnothing$. Tasks for which $D(\tau_i) = \varnothing$ might be scheduled according to precedence constraints (introduced in Equation (4.10)) to other tasks, for which a periodic deadline exists, or which are themselves required by a precedence constraint.

An overview of the task-dependent descriptions is given in Table 4.2. Note that these are valid for all versions of a task. Section 4.3.1 shows how the task model is extended to support different execution modes. These adaptations are introduced later and omitted here for simplicity.

Table 4.2: Components, symbols and descriptions of the task model.

Component	Formula	Description
Task Deadline	$D : \mathcal{T} \to \mathbb{Q}^+ \cup \varnothing$	Tasks are assigned a period, which is used as an implicit deadline in this model.
Task Slack	$\gamma : \mathcal{T} \to [0,1]$	Tasks are assigned an allowed slack, which indicates the allowed deviation from the given period.
Period interval	$[(1-\gamma(\tau_i)) \cdot D(\tau_i), (1+\gamma(\tau_i)) \cdot D(\tau_i)]$	This interval defines the allowed scheduling period for a task τ_i.

Two examples show the adaptation of the example platform and task set to this model. In Example 4, the execution platform, introduced in Example 3, is turned into a model representation. Example 5 shows the task set, introduced in Example 1, as model values.

Example 4: Platform Model

Table 4.3: Model values for the platform example of Example 1.

Unit	Symbol	Description
1. ARM Cortex-A53 core	$u_{cpu,1}$	GPP execution unit.
2. ARM Cortex-A53 core	$u_{cpu,2}$	GPP execution unit.
3. ARM Cortex-A53 core	$u_{cpu,3}$	GPP execution unit.
4. ARM Cortex-A53 core	$u_{cpu,4}$	GPP execution unit.
FPGA-based ML accelerator	$u_{ml,5}$	ML accelerator execution unit, executing the classification tasks.
FPGA-based crypto module	$u_{cr,6}$	Crypto module execution unit, executing the encryption and decryption tasks.

Table 4.3 gives the platform model values of the used example. This leads to $\mathcal{U} = \{u_{cpu,1}, u_{cpu,2}, u_{cpu,3}, u_{cpu,4}, u_{ml,5}, u_{cr,6}\}$. *The shared resource set is then given with Table 4.4.*

Table 4.4: Model values for resources of Example 1.

Resource	Symbol	Description
On-Chip Memory Interface	r_{mem}	Shared resource for all execution units to access the on-chip memory.
Communication Interfaces	r_{comm}	Shared resource combining all communication interfaces like CAN, SPI, UART. Modeled with one resource, because of joined on-chip communication.
Scratchpad Memory	r_{spm}	Shared resource for the scratchpad memory inside the FPGA.

Three shared resources are modeled and used in this example. The first resource models the used memory and the second resource models a shared resource, which combines all communication interfaces like CAN, SPI, UART, because of joined on-chip communication. Lastly, the scratchpad memory is modeled as a third shared resource. This leads to $\mathcal{R} = \{r_{mem}, r_{comm}, r_{spm}\}$.

The presented example platform captures the GPP cores, the ML accelerator and the crypto module as execution units. Even though, the ML accelerator and the crypto module are mapped inside the FPGA, they are modeled as separate execution units instead of the FPGA to capture the different execution capabilities of these units. Also, the scratchpad memory is mapped inside the FPGA, but is modeled as a separate shared resource to capture the different access capabilities.

Example 5: Example Task Set

Table 4.5: Model values for the task set of Example 1.

Index	Symbol	Deadline	Slack	Version Amount
(1)	τ_1	$D(\tau_1) = \varnothing$	$\gamma(\tau_1) = 0$	$N_V(\tau_1) = 2$
(2)	τ_2	$D(\tau_2) = \varnothing$	$\gamma(\tau_2) = 0$	$N_V(\tau_2) = 2$
(3)	τ_3	$D(\tau_3) = \varnothing$	$\gamma(\tau_3) = 0$	$N_V(\tau_3) = 1$
(4)	τ_4	$D(\tau_4) = \varnothing$	$\gamma(\tau_4) = 0$	$N_V(\tau_4) = 1$
(5)	τ_5	$D(\tau_5) = \varnothing$	$\gamma(\tau_5) = 0$	$N_V(\tau_5) = 1$
(6)	τ_6	$D(\tau_6) = 100\,\text{ms}$	$\gamma(\tau_6) = 0.2$	$N_V(\tau_6) = 1$
(7)	τ_7	$D(\tau_7) = 200\,\text{ms}$	$\gamma(\tau_7) = 0.2$	$N_V(\tau_7) = 1$
(8)	τ_8	$D(\tau_8) = \varnothing$	$\gamma(\tau_8) = 0$	$N_V(\tau_8) = 2$
(9)	τ_9	$D(\tau_9) = 40\,\text{ms}$	$\gamma(\tau_9) = 0.2$	$N_V(\tau_9) = 1$
(10)	τ_{10}	$D(\tau_{10}) = \varnothing$	$\gamma(\tau_{10}) = 0$	$N_V(\tau_{10}) = 1$
(11)	τ_{11}	$D(\tau_{11}) = \varnothing$	$\gamma(\tau_{11}) = 0$	$N_V(\tau_{11}) = 1$

The task set thus consists of $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}, \tau_{11}\}$. Tasks τ_6, τ_7 and τ_9 have an associated deadline and thus are expected to be scheduled with a period within the given interval. The other tasks might be scheduled according to precedence constraints or not at all. Tasks τ_1, τ_2 and τ_8 have multiple versions, which are described in the following. The other tasks have only one version.

The following task properties are version-dependent and are defined for each task version individually. A task version represents one implementation of a task. The version of the following functions and sets are denoted by the superscript υ. The number of versions for a task τ_i is given by $V_{\max}(\tau_i)$. The corresponding task $\tau_i \in T$ is denoted by the subscript i. An overview of the version-dependent task descriptions is given in Table 4.6.

The execution time of a task τ_i, version υ on an execution unit u_j is defined as

$$C_i^{\upsilon} \in \mathbb{Q}^+. \tag{4.6}$$

The allowed execution units for a task τ_i, version υ are defined as the set

$$U_i^{\upsilon} \subseteq \mathcal{U}, \tag{4.7}$$

where each element in U_i^{υ} is able to execute the version with index υ of task τ_i. Since the allowed execution units for a version is defined in Equation (4.7), an explicit definition of the execution time for the allowed execution unit via C_i^{υ} in

Equation (4.6) is omitted. The relevant execution time is thus defined via C_i^υ for the given unit U_i^υ via the same version index υ.

Generalizing existing phased execution methods like the Predictable Execution Model (PREM) [25] or the Acquisition Execution Restitution (AER) [26] model, shared resource access for a task τ_i, version υ is modeled via the function

$$A_i^\upsilon : \mathcal{R} \to \mathcal{P}(\mathbb{Q}^+), \tag{4.8}$$

where again, the subscript i denotes the task $\tau_i \in \mathcal{T}$ and the superscript υ denotes the version of the task. The function returns a set of access phase intervals for each shared resource $r_k \in \mathcal{R}$. Each access phase is defined as a tuple (a_{start}, a_{end}), where $a_{start}, a_{end} \in \mathbb{Q}^+$ and $a_{start} < a_{end}$. The value of A_i^υ for a shared resource r_k is thus a set of tuples, each describing one access window towards shared resource r_k. During the each access window, the version υ of task τ_i requires access to the shared resource r_k, given by the upper bandwidth bound B_i^υ described as

$$B_i^\upsilon : \mathcal{R} \to \mathcal{P}([0,1]). \tag{4.9}$$

This function returns a list of bandwidth bounds for each shared resource $r_k \in \mathcal{R}$. The value of $B_i^\upsilon(r_k)$ is a list of bandwidth percentages with the same amount of elements as $A_i^\upsilon(r_k)$. Each element thus corresponds to one access phase (defined in Equation (4.8)) for the same shared resource r_k. It is assumed, that shared resources are accessible from the execution units u onto which a task, requiring access to this shared resource, is scheduled. In general, $|B_i^\upsilon(r_k)| = |A_i^\upsilon(r_k)|$.

The notion of precedence constraints from existing works is extended in this model. Where existing works model precedence as simple `before` (or `after`) relations [113], P_i^υ gives an extension by specifying earliest and latest bounds between two tasks τ_i and τ_m. Precedence constraints for task τ_i, version υ towards other tasks in $\mathcal{T} \setminus \{\tau_i\}$ is thus modeled with the function

$$P_i^\upsilon : \mathcal{T} \setminus \{\tau_i\} \to \mathcal{P}(\mathbb{Q}^+), \tag{4.10}$$

where the subscript i denotes the task $\tau_i \in \mathcal{T}$ and the superscript υ denotes the version of the task. This function returns a list of start and end times of the precedence constraints for each task $\tau_m \in \mathcal{T} \setminus \{\tau_i\}$. Each time interval in the returned list is defined as a tuple with the form (p_{min}, p_{max}), where $p_{min}, p_{max} \in \mathbb{Q}^+$ and $p_{min} \leq p_{max}$. For each precedence constraint interval, the start of one job of task τ_m must be scheduled between p_{min} and

p_{max} time units relative to the start time of each job with version v of task τ_i.

As described above, jobs of task $\tau_m \in \mathcal{T}$ are scheduled, if they have an associated deadline via $D(\tau_m) \neq \emptyset$ or are required by a precedence constraint of another task $\tau_i \in \mathcal{T} \setminus \{\tau_m\}$, which in turn is required to be scheduled via precedence constraints or a deadline.

Table 4.6: Components, symbols and descriptions of task versions.

Component	Formula	Description
Execution Time	$C_i^v \in \mathbb{Q}^+$	Execution time of task τ_i, version v.
Allowed Units	$U_i^v \subseteq \mathcal{U}$	The execution units, for which version v of task τ_i is allowed to be scheduled on.
Access Phases	$A_i^v : \mathcal{R} \to \mathcal{P}(\mathbb{Q}^+)$	Access phases towards shared resources $r_k \in \mathcal{R}$ required by task τ_i, version v.
Bandwidth Bounds	$B_i^v : \mathcal{R} \to \mathcal{P}([0,1])$	Upper bandwidth bounds for each access phase towards the corresponding shared resource $r_k \in \mathcal{R}$ for task τ_i, version v.
Precedence Constraints	$P_i^v : \mathcal{T} \setminus \{\tau_i\} \to \mathcal{P}(\mathbb{Q}^+)$	Precedence constraints towards other tasks $\tau_m \in \mathcal{T} \setminus \{\tau_i\}$ for task τ_i, version v.

With these definitions, the example task set is extended by the version-dependent descriptions. Example 6 shows how the task set values for the runtime, allowed execution units and precedence constraints are turned into a model instance. In Example 7, the access windows and bandwidth bounds towards shared resources are defined. For both examples, all values must be present for a given task version, i.e. for τ_1 and version $v = 1$, the runtime C_1^1, the allowed execution units U_1^1 and the precedence constraints P_1^1 must be defined. Also, A_1^1 and B_1^1 must be defined. The same holds for $v = 2$ of task τ_1 and all other tasks and their versions.

The mapping of the example task set to the model is shown in Example 6 and Example 7. Because of the different version specifications, multiple precedence constraints and execution time estimations are present for τ_1, τ_2 and τ_8. Different execution characteristics need to be modeled for the GPP-based and the ML-based execution of these tasks. For thge other tasks, only one version is present. The bandwidth bounds of the access windows lies below 1 for a subset of the access windows.

Example 6: Runtimes Model Values of Example Tasks

Runtimes are modeled as a conservative estimate of the execution time of each task version on each execution unit, as are the precedence constraints. For an easier overview, the GPP cores are collected in the set $\mathcal{U}_{GPP} = \{u_{cpu,1}, u_{cpu,2}, u_{cpu,3}, u_{cpu,4}\}$.

Table 4.7: Example values for runtime, execution units and precedence constraints.

Task, Ver.	C_i^v	U_i^v	P_i^v
$\tau_1, \upsilon = 1$	$C_1^1 = 210 \cdot 10^3$	$U_1^1 = \mathcal{U}_{GPP}$	$P_1^1(\tau_3) = \{(-10^5, -200)\}$
$\tau_1, \upsilon = 2$	$C_1^2 = 8.22 \cdot 10^3$	$U_1^2 = \{u_{ml,5}\}$	$P_1^2(\tau_3) = \{(-10^5, -220)\}$, $P_1^2(\tau_{10}) = \{(-200, -10)\}$, $P_1^2(\tau_{11}) = \{(8300, 10^5)\}$
$\tau_2, \upsilon = 1$	$C_2^1 = 192 \cdot 10^3$	$U_2^1 = \mathcal{U}_{GPP}$	$P_2^1(\tau_3) = \{(-10^5, -100)\}$
$\tau_2, \upsilon = 2$	$C_2^2 = 6.9 \cdot 10^3$	$U_2^2 = \{u_{ml,5}\}$	$P_2^2(\tau_3) = \{(-10^5, -120)\}$, $P_2^2(\tau_{10}) = \{(-200, -10)\}$, $P_2^2(\tau_{11}) = \{(7000, 10^5)\}$
$\tau_3, \upsilon = 1$	$C_3^1 = 200$	$U_3^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_4, \upsilon = 1$	$C_4^1 = 50$	$U_4^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_5, \upsilon = 1$	$C_5^1 = 50$	$U_5^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_6, \upsilon = 1$	$C_6^1 = 10^3$	$U_6^1 = \mathcal{U}_{GPP}$	$P_6^1(\tau_1) = \{(-10^5, -C_1^{\lambda_j(\tau_1)})\}$, $P_6^1(\tau_2) = \{(-10^5, -C_2^{\lambda_j(\tau_2)})\}$, $P_6^1(\tau_4) = \{(1500, 10^5)\}$, $P_6^1(\tau_5) = \{(-10^5, 100)\}$, $P_6^1(\tau_8) = \{(1500, 10^5)\}$
$\tau_7, \upsilon = 1$	$C_7^1 = 2 \cdot 10^3$	$U_7^1 = \mathcal{U}_{GPP}$	$P_7^1(\tau_5) = \{(-10^5, 100)\}$, $P_7^1(\tau_8) = \{(-10^5, -C_8^{\lambda_j(\tau_8)})\}$, $P_7^1(\tau_4) = \{(1500, 10^5)\}$
$\tau_8, \upsilon = 1$	$C_8^1 = 5 * 10^4$	$U_8^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_8, \upsilon = 2$	$C_8^2 = 10$	$U_8^2 = \{u_{cr,6}\}$	$\varnothing$
$\tau_9, \upsilon = 1$	$C_9^1 = 100$	$U_9^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_{10}, \upsilon = 1$	$C_{10}^1 = 10$	$U_{10}^1 = \mathcal{U}_{GPP}$	$\varnothing$
$\tau_{11}, \upsilon = 1$	$C_{11}^1 = 10$	$U_{11}^1 = \mathcal{U}_{GPP}$	$\varnothing$

Table 4.7 shows the values for execution time, allowed execution units and precedence constraints for the task set of Example 5. For $P_6^1(\tau_1)$, $P_6^1(\tau_2)$ *and* $P_7^1(\tau_8)$, *the precedence constraints are defined by the runtime of the chosen version for* τ_1, τ_2 *and* τ_8, *respectively. This is denoted by* $C_1^{\lambda_j(\tau_1)}$, $C_2^{\lambda_j(\tau_2)}$ *and* $C_8^{\lambda_j(\tau_8)}$, *respectively.*

Example 7: Example Values for Shared Resource Access of Task Set

Table 4.8: Values for access windows and bandwidth bounds of Example 1.

Task	A_i^v	B_i^v
$\tau_1, v=1$	$A_1^1(r_m)=\{(i\cdot 200, i\cdot 200+10)\|i\in[0,10^3]\}$	$B_1^1(r_m)=\{(0.5)...\}$
$\tau_1, v=2$	$A_1^2(r_{spm})=\{(j\cdot 25, i\cdot 25+5)\|j\in[0,327]\}$	$B_1^2(r_{spm})=\{(1)...\}$
$\tau_2, v=1$	$A_2^1(r_m)=\{(i\cdot 200, i\cdot 200+12)\|i\in[0,960]\}$	$B_2^1(r_m)=\{(0.5)...\}$
$\tau_2, v=2$	$A_2^2(r_{spm})=\{(j\cdot 20, i\cdot 20+5)\|j\in[0,344]\}$	$B_2^2(r_{spm})=\{(1)...\}$
$\tau_3, v=1$	$A_3^1(r_{comm})=\{(0,80)\}$, $A_3^1(r_m)=\{(120,200)\}$	$B_3^1(r_{comm})=\{(1)\}$, $B_3^1(r_m)=\{(0.5)\}$
$\tau_4, v=1$	$A_4^1(r_m)=\{(0,20)\}$, $A_4^1(r_{comm})=\{(30,50)\}$	$B_4^1(r_m)=\{(0.5)\}$, $B_4^1(r_{comm})=\{(1)\}$
$\tau_5, v=1$	$A_5^1(r_{comm})=\{(0,20)\}$, $A_5^1(r_m)=\{(30,50)\}$	$B_5^1(r_{comm})=\{(1)\}$, $B_5^1(r_m)=\{(0.5)\}$
$\tau_6, v=1$	$A_6^1(r_m)=\{(0,100),(370,400),(960,10^3)\}$	$B_6^1(r_m)=\{(0.75)...\}$
$\tau_7, v=1$	$A_7^1(r_m)=\{(0,80),(1920,2000)\}$	$B_7^1(r_m)=\{(0.75)...\}$
$\tau_8, v=1$	$A_8^1(r_m)=\{(0,5),(45\cdot 10^3, 5\cdot 10^4)\}$	$B_8^1(r_m)=\{(0.25)\}$
$\tau_8, v=2$	$A_8^2(r_m)=\{(0,2),(8,10)\}$	$B_8^2(r_m)=\{(0.25)\}$
$\tau_9, v=1$	$A_9^1(r_{comm})=\{(0,10)\}$, $A_9^1(r_m)=\{(20,30),(90,100)\}$	$B_9^1(r_{comm})=\{(0.2)\}$, $B_9^1(r_m)=\{(0.75),(1)\}$
$\tau_{10}, v=1$	$A_{10}^1(r_m)=\{(0,5)\}$, $A_{10}^1(r_{spm})=\{(5,10)\}$	$B_{10}^1(r_m)=\{(1)\}$, $B_{10}^1(r_{spm})=\{(1)\}$
$\tau_{11}, v=1$	$A_{11}^1(r_{spm})=\{(0,5)\}$, $A_{11}^1(r_m)=\{(5,10)\}$	$B_{11}^1(r_{spm})=\{(1)\}$, $B_{11}^1(r_m)=\{(1)\}$

Table 4.8 shows the access windows and bandwidth bounds for each task version.

The goal of the presented model components is to describe the given tasks, such that a time triggered schedule, with hyperperiod $t_P \in \mathcal{R}^+$ can be created. This schedule needs to fulfill a set of all constraints, which are described below. Each job of a task $\tau_i \in \mathcal{T}$ is denoted by the index n of the given functions. Each scheduled job is defined by the start time $\theta_i(n)$, the chosen task version $\lambda_i(n)$ and the chosen execution unit $\phi_i(n)$.

The start time of the nth job of task τ_i is defined by

$$\theta_i : \mathbb{N} \to [0, t_P), \tag{4.11}$$

where the subscript i denotes the task $\tau_i \in \mathcal{T}$ and the n denotes the nth job of this task.

The chosen version of the nth job of task τ_i is defined by

$$\lambda_i : \mathbb{N} \to \mathbb{N}, \tag{4.12}$$

where the subscript i denotes the task $\tau_i \in \mathcal{T}$ and the n denotes the nth job of this task.

Lastly, the chosen execution unit of the nth job of task τ_i is defined by

$$\phi_i : \mathbb{N} \to \mathcal{U}, \tag{4.13}$$

where the subscript i denotes the task $\tau_i \in \mathcal{T}$ and the n denotes the nth job of this task. The collection of θ_i,ϕ_i and λ_i is defined as the schedule Σ, which results in

$$\Sigma = \{\theta_i, \lambda_i, \phi_i\}. \tag{4.14}$$

Table 4.9 gives an overview of these schedule-describing values.

Table 4.9: Components, symbols and descriptions of the schedule-describing values.

Component	Formula	Description
Job Start Times	$\theta_i : \mathbb{N} \to [0, t_P)$	Start times of task τ_i.
Task Versions	$\lambda_i : \mathbb{N} \to \mathbb{N}$	Task versions of task τ_i.
Execution Units	$\phi_i : \mathbb{N} \to \mathcal{U}$	Execution units of task τ_i.
Schedule	$\Sigma = \{\theta_i, \lambda_i, \phi_i\}$	Collection of all schedule components for.

Contribution 1: Schedule Model Definition

A schedule Σ is defined as the collection of start times $\theta_i(n)$, chosen task versions $\lambda_i(n)$ and chosen execution units $\phi_i(n)$. Together with the task set $\mathcal{T}$, the schedule Σ defines a schedule instance, which is used for the schedule creation and allows for a detailed description of multi-version tasks with precedence constraints.

A discretization of the time axis is necessary for computation. This discretization is calculated by first rounding the execution times C_i^v, precedence constraints P_i^v, and access windows A_i^v of all tasks $\tau_i \in \mathcal{T}$ and all possible versions v as well as all implicit deadlines $D(\tau_i)$ to the nearest multiple of the time resolution of the platform, onto which the task set should be scheduled. Afterwards, the final timing resolution t_δ is calculated as the greatest common divisor (GCD) of these values. With this, the solution space is discretized and made solvable with the python program. Table 4.10 gives an overview of the additional model values for the schedule creation.

A valid schedule needs to fulfill constraints to be able to be scheduled onto a given platform. These constraints are listed below. First, $N_\theta(\tau_i)$ is

Table 4.10: Components, symbols and descriptions of additional model values.

Component	Formula	Description
Hyperperiod	$t_P = \underset{\tau_i \in \mathcal{T}}{\mathtt{LCM}}(D(\tau_i))$	The least common multiple of all task deadlines.
Timing Resolution	$t_\delta = \underset{\tau_i \in \mathcal{T}}{\mathtt{GCD}}(C_i^\upsilon, P_i^\upsilon, A_i^\upsilon, D(\tau_i))$	The greatest common divisor of all execution times, precedence constraints, access windows and deadlines.

calculated as the maximum number of scheduled jobs of each task $\tau_i \in \mathcal{T}$ inside the schedule. We calculate $N_\theta(\tau_i)$ for tasks with a given deadline as

$$N_\theta(\tau_i) = \frac{t_P}{D(\tau_i)}. \tag{4.15}$$

Since t_P is calculated as the GCD including $D(\tau_i)$, $N_\theta(\tau_i) \in \mathbb{N}$.

For tasks without a given deadline, $D(\tau_i) = \varnothing$, N_θ is defined as sum of the maximum numbers of jobs, which is required to be scheduled by precedence constraints resulting from other tasks $\tau_m \in \mathcal{T} \setminus \{\tau_i\}$. This is calculated as

$$N_\theta(\tau_i) = \sum_{\tau_j \in \mathcal{T} \setminus \{\tau_i\}} \left(\max_\upsilon (|P_i^\upsilon(\tau_j)|) \cdot N_\theta(\tau_j) \right). \tag{4.16}$$

The calculation of $N_\theta(\tau_i)$ must thus be performed iteratively, starting with tasks $\tau_i \in \mathcal{T}$, for which a deadline is given via $D(\tau_i) \neq \varnothing$, followed by tasks where $D(\tau_i) = \varnothing$ and which are required by precedence constraints of already calculated tasks. This is repeated until $N_\theta(\tau_i)$ is calculated for all tasks $\tau_i \in \mathcal{T}$. It is assumed, that the chosen version of a job defines both the execution time and the shared resource access phases and bandwidths. Thus, the chosen version $\lambda_i(n)$ remains constant during one execution. The resulting constraints for a valid schedule are described in the following. Each job with a given period must be scheduled within the allowed period interval. This is defined with

$$\begin{aligned} &D(\tau_i) \cdot (1 - \gamma(\tau_i)) \leq \theta_i(n+1) - \theta_i(n) \leq D(\tau_i) \cdot (1 + \gamma(\tau_i)), \\ &\qquad \textbf{if } n \in [0, N_\theta(\tau_i) - 1), \forall \tau_i \in \{\tau \in \mathcal{T} \mid \tau_i \neq \varnothing\}, n \in [0, N_\theta(\tau_i)). \\ &D(\tau_i) \cdot (1 - \gamma(\tau_i)) \leq \theta_i(0) + t_P - \theta_i(N_\theta(\tau_i) - 1) \leq D(\tau_i) \cdot (1 + \gamma(\tau_i)), \\ &\qquad \textbf{if } n = N_\theta(\tau_i) - 1, \forall \tau_i \in \{\tau \in \mathcal{T} \mid \tau_i \neq \varnothing\}, n \in [0, N_\theta(\tau_i)). \end{aligned} \tag{4.17}$$

For every job, the chosen execution unit must be allowed for the chosen task version. This is defined with the constraint

$$\phi_i(n) \in U_i^{\lambda_i(n)}, \forall \tau_i \in \mathcal{T}, n \in [0, N_\theta(\tau_i)). \tag{4.18}$$

No two tasks are allowed to be executed on the same unit at the same time. This is ensured via the constraint:

$$\begin{aligned} \nexists \theta_i(n) \text{ s.t. } \theta_i(n) \in & \begin{cases} [\theta_m(o), \theta_m(o) + C_m^{\lambda_m(o)}) & \textbf{if } \theta_m(o) + C_m^{\lambda_m(o)} < t_P \\ [\theta_m(o), t_P) \cup & \\ [0, \theta_m(o) + C_m^{\lambda_m(o)} \bmod t_P) & \textbf{if } \theta_m(o) + C_m^{\lambda_m(o)} \geq t_P \end{cases} \\ & \forall \tau_i, \tau_m \in \mathcal{T}, n \in [0, N_\theta(\tau_i)), o \in [0, N_\theta(\tau_m)) \\ & \text{with } \phi_i(n) = \phi_m(o) \text{ and } (i,n) \neq (m,o). \end{aligned} \tag{4.19}$$

Here, the start of the first job is given by $\theta_i(n)$, and the start of the second job is given by $\theta_m(o)$. This must hold true for all possible job start times, where the respective execution units are identical, i.e. $\phi_i(n) = \phi_m(o)$. Because no two tasks can overlap with at least one start time lying in the scheduled execution time of the other task, this constraint suffices and the complete overlap is not necessary to consider.

During each time step, the total bandwidth required by all tasks towards each shared resource must not exceed the available bandwidth of 100%. This is defined with the constraint:

$$\begin{aligned} \sum_{\tau_i \in \mathcal{T}} \sum_{n=0}^{N_\theta(\tau_i)-1} & B_i^{\lambda_i(n)}(r_k) \leq 1, \forall\, r_k \in \mathcal{R}, \\ & t \in [0, t_P) \text{ s.t. } t \text{ in access window } A_i^{\lambda_i(n)}(r_k) \end{aligned} \tag{4.20}$$

Here, $B_i^{\upsilon}(r_k)$ represents the bandwidth requirements of task i with version υ for resource r_k. This constraint must hold true for every time step t in the schedule and for every shared resource r_k. The corresponding bandwidth requirements are summed up for all tasks $\tau_i \in \mathcal{T}$ and all jobs $n \in [0, N_\theta(\tau_i))$ of these tasks, which require access to the shared resource r_k at time step t. The corresponding time steps are given by $\theta_i(n) + a_{start}$ and $\theta_i(n) + a_{end}$, with $a_{start} \in A_i^{\upsilon}(r_k)$ and $a_{end} \in A_i^{\upsilon}(r_k)$.

Each job must have all of their precedence constraints fulfilled by exactly one job from the specified task. This is defined with:

$$\exists o \in [0, N_\theta(\tau_m)) \text{ s.t. } \begin{cases} \theta_o(m) \in [\theta_i(n) + p_{min}, \theta_i(n) + p_{max}) \\ \qquad \textbf{if } \theta_i(n) + p_{max} < t_P \\ \theta_o(m) \in [\theta_i(n) + p_{min}, t_P) \cup [0, \theta_i(n) + p_{max} \bmod t_P) \\ \qquad \textbf{if } \theta_i(n) + p_{min} < t_P \leq \theta_i(n) + p_{max} \\ \theta_o(m) \in [\theta_i(n) + p_{min} \bmod t_P, \theta_i(n) + p_{max} \bmod t_P) \\ \qquad \textbf{if } \theta_i(n) + p_{min} \geq t_P \end{cases}$$
$$\forall (p_{min}, p_{max}) \in P_i^{\lambda_i(n)}(\tau_m),\ \forall\, \tau_i, \tau_m \in \mathcal{T}, n \in [0, N_\theta(\tau_i)), \tag{4.21}$$

where τ_m is a task, which is required by the precedence constraint of τ_i, i.e. $P_i^{\lambda_i(n)}(\tau_m) \neq \emptyset$. The start time difference $(\theta_i(n) - \theta_m(o))$ must lie within the bounds of the precedence constraint interval (p_{min}, p_{max}), taking the periodicity into account.

Lastly, a task cannot have a deadline and be required by precedence constraints at the same time. This is defined with the constraint

$$\nexists\, \tau_i \in \mathcal{T} \text{ s.t. } D(\tau_i) \neq \emptyset \wedge \exists \tau_m \in \mathcal{T} \setminus \{\tau_i\} \text{ s.t. } P_i^v(\tau_m) \neq \emptyset, \tag{4.22}$$

where τ_i is a task, which has a given deadline via $D(\tau_i) \neq \emptyset$ and is required by precedence constraints of another task $\tau_m \in \mathcal{T} \setminus \{\tau_i\}$, i.e. $P_i^v(\tau_m) \neq \emptyset$.

Contribution 2: Valid Schedule Constraints

Together with the schedule definition, the constraints for a valid schedule are defined. These constraints ensure that the created schedule is valid and can be scheduled onto a given platform. The constraints include deadlines, allowed execution units, shared resource constraints, precedence constraints and bandwidth constraints.

4.2 Time-Triggered Scheduling Approaches for Periodic Tasks

Given the task model introduced in Section 4.1, a time triggered scheduler is introduced in this section. This scheduler aims to generate a static and periodic schedule $\Sigma = \{\theta_i(n), \lambda_i(n), \phi_i(n)\}$ for all tasks $\tau_i \in \mathcal{T}$ under the given constraints. This schedule is defined by the start times $\theta_i(n)$, chosen task versions $\lambda_i(n)$

and chosen execution units $\phi_i(n)$ for all jobs $n \in [0, N_\theta(\tau_i))$ of these tasks. The constraints for a valid schedule, as given in the last chapter, are dependent on the chosen task version and thus non-linear. Non-linear constraints make an (integer) linear programming formulation impossible without extensive translation of the constraints into a linear form. Neither a simplex algorithm [114] nor a branch and bound solver [115] is used for schedule creation due to this. Because the constraints are also not continuous, the use of gradient-based approaches, which require a continuous Hessian matrix or Jacobian matrix, is not available for this problem. Optimization methods for unknown gradients (e.g. [116, 117]) or customized heuristics are thus necessary. Two scheduling algorithms are presented. The first algorithm is a non-optimal but fast heuristic based on a polyhedral approach, while the second algorithm uses a generalized simulated annealing approach [117, 118] to generate an optimal schedule, at the cost of a longer runtime. Both algorithms are used to generate valid schedules adhering to the constraints.

4.2.1 Polyhedron-based Time-Triggered Scheduling

The polyhedron-based approach to generate time triggered schedules is based on the publication [Sta24a]. The algorithm starts by generating a 3D space with the dimensions of $(t_P/t_\delta) \times N_\mathcal{U} \times (N_\mathcal{R} + 1)$. This space represents the time triggered schedule, with the first dimension representing the time dimension, the second dimension denoting the execution unit and the third dimension representing execution units in the layer with index 0 and shared resources for indexes ≥ 1. This space is visualized with unit cubes with an edge length of 1. Additionally, N_τ spaces of dimensions $(t_P/t_\delta) \times N_\mathcal{U} \times 1$ are created. These will represent viable starting points for all tasks and are called viability spaces. Figure 4.1 shows these spaces.

After the 3D spaces are generated, $N_\theta(\tau_i)$ sets of polyhedrons are generated for each task τ_i, according to the steps shown in Figure 4.2. The goal is to generate a basket of polyhedrons for all tasks and later pack these polyhedrons into the 3D space, while adhering to the constraints for a valid schedule. The generation of one polyhedron one task is done in *four* steps.

First, C_i^υ/t_δ cubes are concatenated to create a rectangle of dimension $(C_i^\upsilon/t_\delta) \times 1 \times 1$. This rectangle represents the execution of task τ_i for a given version υ on the execution units.

Second, each access window towards shared resources is turned into a rectangle with the dimensions of $(A_i^\upsilon(r_j)[n][1] - A_i^\upsilon(r_j)[n][0])/t_\delta \times N_\mathcal{U} \times B_i^\upsilon(r_j)[n]$. The

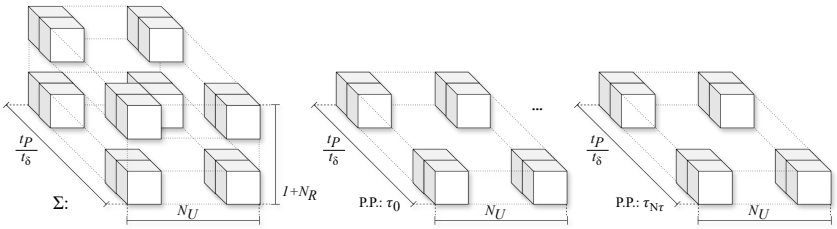


Figure 4.1: Creation of the 3D space for the polyhedral approach. The first dimension represents the time dimension, the second dimension represents the execution units and the third dimension represents the shared resources. On the left, the space for the schedule is shown, to the right of this, the viability spaces.

length of this rectangle in the time dimension is determined by the length of the access window $((A_i^\upsilon(r_j)[n][1] - A_i^\upsilon(r_j)[n][0]))$ with index n. The width is determined by the number of execution units N_u and the height is determined by the memory bound of the corresponding access window, which is determined by $B_i^\upsilon(r_j)[n]$. This step is repeated once for every access window with index n of every shared resource r_j.

Third, this rectangle is placed above the first rectangle at the time step of the start time of the access window $A_i^\upsilon(r_j)[n][0]$ parallel to the time axis and execution unit axis. The distance from the ground plane is given by the index j of the shared resource r_j. Both steps, *two* and *three* are repeated for each access window.

Forth, a rectangle of the dimensions $(2 \cdot \gamma(\tau_i) \cdot D(\tau_i))/(t_\delta)$ generated and will be placed in front of each polyhedron in the time dimension, should a deadline be given to the task τ_i. This represents one viability rectangle for tasks with a given deadline. These four steps are repeated once for each unit $u \in U_i^\upsilon$ that can execute the given task version υ, with the first rectangle representing the execution of the task being placed at the width according to the index of the execution unit u. These polyhedrons represent the execution of one job of the given version υ of the task.

Each of the generated polyhedrons is added to the basket of polyhedrons $N_\theta(\tau_i)$ times, allowing the 3D packing algorithm to select the same polyhedron shape for every job of task τ_i.

The generation of these polyhedrons is then repeated for each task version υ of the same task τ_i until no more task versions are left. After this, the same

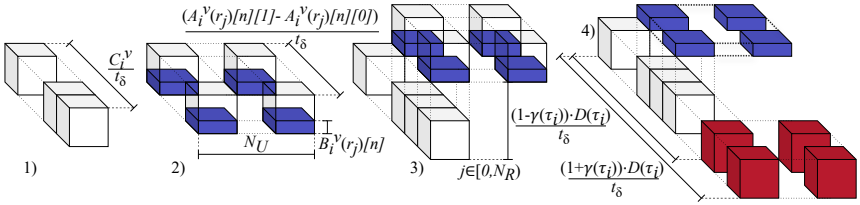


Figure 4.2: Creation of one polyhedron out of the task description for the polyhedral approach. Blocks modelling the execution on execution units are shown in white, blocks modelling access to shared resources are shown in blue. The filling of these blocks is determined by the memory bound of the corresponding access window. Red rectangles signify possible starting points for the next job of the same task.

process is repeated for each task $\tau_i \in \mathcal{T}$ until no more tasks are left. With this, a basket of polyhedrons is generated, out of which the 3D packing algorithm selects fitting polyhedrons to build the schedule, as shown in Figure 4.3.

For the placement of tasks with precedence constraints, a set of polyhedrons is created. This set holds the viability rectangles, which define the placement of tasks τ_j which are required by τ_is precedence constraints, i.e. $\tau_j \in \mathcal{T}$ s.t. $P_i^v(\tau_j) \neq \emptyset$. These rectangles define possible placements for the starting block of the nth job of task τ_j which is required by the precedence constraints of task τ_i. Similar to the viability rectangles generated for tasks with deadlines, they have the dimensions of $(P_i^v(\tau_j)[n][1] - P_i^v(\tau_j)[n][0]) \times N_u \times 1$ and will be placed on the bottom layer of the 3D grid. These rectangles are collected seperately and will be associated with the corresponding task viability space.

The polyhedral algorithm to generate a schedule is split into two parts. It is adapted from the works of [Sta24a]. First, a starting solution is created. Should it be necessary, this solution is then improved by iteratively selecting polyhedrons to switch with the aim of improving the schedule until a valid schedule is found. The starting solution is generated using the following iterative algorithm.

In the *first* iteration step, the next polyhedron is selected from the basket of polyhedrons. The polyhedron with the earliest possible start time is chosen. For each task τ_i with a given deadline $D(\tau_i) \neq \emptyset$, the earliest possible placement is either at the start of the schedule, should there be no other job of the current

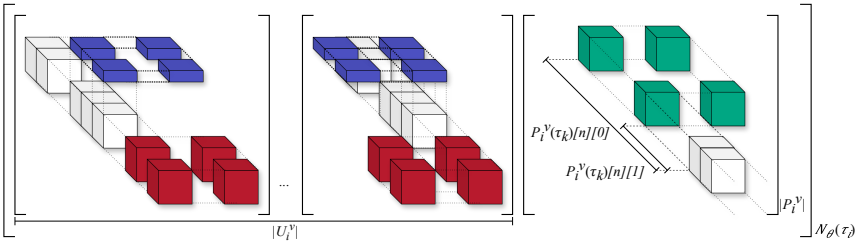


Figure 4.3: Creation of the polyhedron basket for one task version. Blocks modelling the execution on execution units are shown in white, blocks modelling access to shared resources are shown in blue. Red rectangles signify the viability rectangles of possible starting points for the next job of the same task. Viability rectangles signifying the start of required tasks from precedence constraints are shown in green.

task, or within $(\gamma(\tau_i) \cdot D(\tau_i))/t_\delta$ after the start of the previous job. For a task τ_j which is required by precedence constraints of another task τ_i, the earliest placement lies in one of the viability rectangles, marked in the corresponding viability space. This space is either filled by viability rectangles based on deadlines or based on precedence constraints. These rectangles also wrap around the start of the schedule onto the end of the hyperperiod. A polyhedron can only be placed, if the starting block lies inside a viable rectangle of the corresponding task and if no polyhedron of the same task is present on the schedule.

The placement of a polyhedron is possible, if no cube of the execution layer (index 0 in the third dimension), overlaps with a scheduled cube. For access windows in the corresponding layers, the total amount of rectangles within the index of the cube is not allowed to exceed 1, i.e. the sum of all memory bounds at each time step is not allowed to exceed 1, as shown in Equation (4.20). Should the placement of a task exceed this limit, the polyhedron is not able to be placed at the corresponding time step.

Should there be a tie between the earliest possible polyhedron, polyhedrons required by precedence constraints are chosen. Should there still be a tie, the longer polyhedron is placed. If both polyhedrons have the same length, the polyhedron with less other polyhedrons of the same task is placed. If there is still a tie, the unit with more scheduled polyhedron on it is chosen. Finally, the task with the lower index is chosen.

Second, the selected polyhedron is placed at the earliest possible time step on

the schedule, marking all cubes in the 3D space filled by the task as occupied. Additionally, on set polyhedrons of the same task is removed from the basket of possible polyhedrons.

Third, the corresponding viability rectangles for the next job of the same task and for tasks required by precedence constraints are placed in the corresponding 3D spaces, denoting the possibility for the placement of the next polyhedrons.

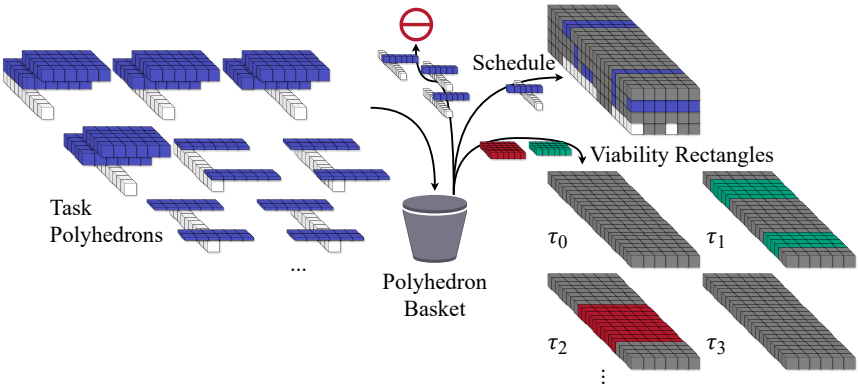


Figure 4.4: Overview of the polyhedral-based scheduling algorithm. At the start, the polyhedrons are created and put into the basket. Out of these, the schedule are built. For these viability rectangles such as the precedence and next job rectangles are used. One set of the remaining polyhedrons of the same task is removed per job.

Figure 4.4 gives an overview of the polyhedral-based scheduling algorithm again. The generated polyhedrons are shown in the left part of the figure and put into the polyhedron basket. Out of this basket, the corresponding polyhedrons are taken and put onto the schedule. Furthermore, associated viability rectangles are placed. One set of unused polyhedrons of the same task are discarded.

The algorithm ends in one of two cases. Either a schedule is found, which results in no polyhedrons left to schedule. In this case, the generated schedule is a valid schedule, as per the constraints. The simulated annealing-based scheduler might then be used for optimization but is not necessary.

Example 8: Polyhedrons for Two Example Tasks

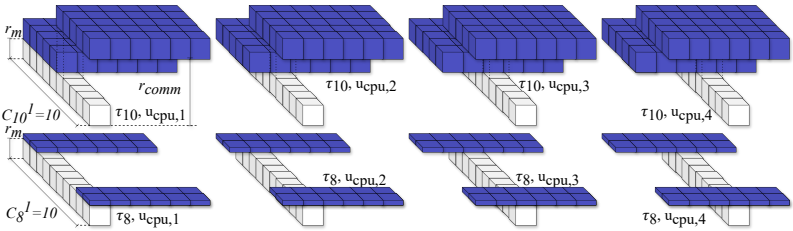


Figure 4.5: Example polyhedron sets for tasks τ_8 and τ_{10}. Access windows are shown with blue blocks. The execution of the tasks is shown with white boxes.

Per task, four polyhedrons are generated since tasks τ_8 and τ_{10} have only one version, which is executable on $\mathcal{U}_{GPP} = \{u_{cpu,1}, u_{cpu,2}, u_{cpu,3}, u_{cpu,4}\}$. The rectangle in white is shifted based on the execution unit, which is assigned for the corresponding polyhedron. Task τ_{10} accesses two shared resources, r_m and r_{comm} with full memory bounds. Task τ_8 accesses r_m during two windows with a memory bound of 0.25.

If the algorithm is unable to select a next polyhedron out of the polyhedron basket, i.e. the constraints are not adhered to, either an iterative improvement of the schedule must be taken. This improvement can either be done using the modelling of tasks as polyhedrons (as above) or the simulated annealing-based approach is used to improve this starting solution until a valid solution is found, or a stopping criterion is reached, respectively. A dedicated evaluation of this polyhedral approach with respect to runtime and scheduling success rate can be found in Section 7.2.

Example 8 shows an example for the generated polyhedrons for task τ_8 and τ_{10} of the example task set values as introduced in Example 6. Note that these polyhedrons are entered into the polyhedron basket $N_\theta(\tau_8)$ and $N_\theta(\tau_{10})$ times each.

The iterative improvement of the schedule aims to move polyhedrons within the given constraints fit more tasks into the schedule. This movement of polyhedrons is based on the polyhedrons still left in the basket and tries to make space by shifting the currently scheduled polyhedrons. The algorithm iteratively checks how much each polyhedron might need to be shifted to facilitate the scheduling of a second polyhedron out of the polyhedron basket and chooses the shift with the least amount of necessary shifting and includes all

necessary and possible shifts to facilitate the scheduling of the polyhedron out of the polyhedron basket. This process facilitates an underlying cost function which adds 1 for each block of size $1 \times 1 \times 1$ which is shifted in the first plane of the 3D space. The cost function also adds a penalty of one for each free block between the execution of the candidate polyhedron and scheduled polyhedron on the first plane on the same execution unit. For an easier definition, the functions $\tau^{\uparrow}(t,u)$ and $\tau^{\downarrow}(t,u)$ are defined, which return the task index of the next scheduled job on execution unit u after time t and the task index of the last scheduled job on execution unit u before time t, respectively. The functions $\theta^{\uparrow}(t,u)$ and $\theta^{\downarrow}(t,u)$ are defined, which return the start time of the next scheduled job on execution unit u after time t and the start time of the last scheduled job on execution unit u before time t, respectively. The function $\lambda^{\uparrow}(t,u)$ and $\lambda^{\downarrow}(t,u)$ are defined in the same way, but return the version of the next and last scheduled job on execution unit u after and before time t, respectively.

Additionally, the functions $\theta_i^{\dagger}(n)$ and $\theta_i^{\ddagger}(n)$ are defined, which return the start time of the nth job of task τ_i before and after the shift, respectively. This results in the cost function

$$F_{\text{cost}} = F_{\text{shift}} + F_{\text{gap}}, \tag{4.23}$$

with

$$F_{\text{shift}} = \sum_{i=0}^{N_{\mathcal{T}}-1} \sum_{n=0}^{N_{\theta}(\tau_i)} \left| \theta_i^{\dagger}(n) - \theta_i^{\ddagger}(n) \right|, \tag{4.24}$$

resulting in the sum of all shifts of all polyhedrons associated with the tasks and

$$\begin{aligned} F_{\text{gap}} = &\left| \theta_i(n) - \theta^{\downarrow}(\theta_i(n), \phi_i(n)) + C_{\tau^{\downarrow}(\theta_i(n),\phi_i(n))} \right| \\ &+ \left| \theta^{\uparrow}(\theta_i(n), \phi_i(n)) - \left(\theta_i(n) + C_i^{\lambda_i(n)} \right) \right|, \end{aligned} \tag{4.25}$$

summing up all gaps left by the insertion of a given task. If a shift moves a polyhedron of a task τ_j, version m for which P_j^m is not empty, the algorithm additionally shifts the associated polyhedrons. Should a polyhedron be scheduled, one set of polyhedrons of the same task are removed from the basket. Should no polyhedron from the polyhedron basket be selectable, the iterative improvement has not finished it's work and further improvement by simulated annealing-based solutions is necessary. A dedicated evaluation of this polyhedral approach with respect to runtime and scheduling success rate can be found in Section 7.2.

Contribution 3: Polyhedron-based Scheduler

The polyhedron-based scheduler is a heuristic algorithm to generate time triggered schedules and is based on modeling tasks as 3D objects in a 3D space. This allows for a geometric interpretation of the scheduling problem and an illustrative way to generate schedules.

4.2.2 Simulated Annealing Time-Triggered Scheduling

The simulated annealing-based approach to generate time triggered schedules is based on the first-author publications [Sta25a, Sta26] and is extended in this work. The presented algorithm generates valid schedules as given in Contribution 3 by iteratively improving a given solution vector $\vec{x}$, representing the schedule that needs to be generated. This solution vector $\vec{x}$ is defined as

$$\vec{x} = \left[-\vec{x}_\theta- \,|\, -\vec{x}_\lambda- \,|\, -\vec{x}_\phi-\right], \tag{4.26}$$

with the first third of the vector representing the start times in a start time vector $\vec{x}_\theta$, the second third representing the version vector as $\vec{x}_\lambda$ and the last third representing the execution unit vector $\vec{x}_\phi$. These vectors are defined as follows:

$$\vec{x}_\theta = \left[\theta_0(0),\dots,\theta_0(N_\theta(\tau_0)),\dots,\theta_{N_\mathcal{J}-1}(0),\dots,\theta_{N_\mathcal{J}-1}(N_\theta(\tau_{N_\mathcal{J}-1}))\right]. \tag{4.27}$$

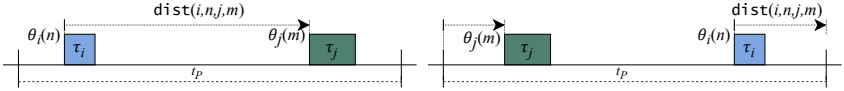
Here, the first element is the start time of the first job task τ_0, while the last element is the last start time of the last task $\tau_{N_\mathcal{J}-1}$. The second element represents the second start time of the first task τ_0. The vector represents all tasks with an implicit deadline first and all tasks without an implicit deadline afterwards. As a helper function, we define the function $\texttt{index}(i,n)$, which returns the index of the start time $\theta_i(n)$ of job n of task τ_i in the start time part of the solution vector $\vec{x}$. This function is defined as

$$\texttt{index}(i,n) = \sum_{k=0}^{i-1} N_\theta(\tau_k) + n, \tag{4.28}$$

and allows for a condensed definition of later functions. Additionally, the functions $\texttt{dist}(i,n,j,m)$ and $\texttt{prec_index}(i,n,j)$ are defined. The function

$$\texttt{dist}(i,n,j,m) = \begin{cases} |\theta_j(m)-\theta_i(n)|, \textbf{ if } \theta_i(n) \le \theta_j(m) \\ t_P - |\theta_i(n)-\theta_j(m)|, \textbf{ else} \end{cases} \tag{4.29}$$

returns the distance from the nth job of τ_i to the mth job of τ_j in the forward direction, assuming that the schedule is periodic with a period of t_P. The return value assumes that τ_i comes before τ_j in the schedule, also wrapping around the period. Should the distance from $\theta_j(m)$ to $\theta_i(n)$ be calculated, the tasks are just entered in the opposite order, i.e. $\texttt{dist}(j,m,i,n)$, which results in the distance from $\theta_j(m)$ to $\theta_i(n)$ in the forward direction. Figure 4.6 shows an illustration of the distance definition.



(a) $\theta_i(n) < \theta_j(m)$ (b) $\theta_i(n) \geq \theta_j(m)$

Figure 4.6: Distance function definition for the function used for this chapter.

The function $\texttt{prec_index}(i,n,j)$ returns the index of the job of task τ_j which is required by the precedence constraints of the nth job of task τ_i.

The version vector $\vec{\mathrm{x}}_\lambda$ and the execution unit vector $\vec{\mathrm{x}}_\phi$ are ordered in the same way with $\vec{\mathrm{x}}$ representing the chosen version $\lambda_i(n)$ and $\vec{\mathrm{x}}_\phi$ representing the chosen execution unit $\phi_i(n)$. Thus, the version vector $\vec{\mathrm{x}}_\lambda$ and the execution unit vector $\vec{\mathrm{x}}_\phi$ are defined as

$$\vec{\mathrm{x}}_\lambda = \left[\lambda_0(0),\dots,\lambda_0(N_\theta(\tau_0)),\dots,\lambda_{N_\mathcal{J}-1}(0),\dots,\lambda_{N_\mathcal{J}-1}(N_\theta(\tau_{N_\mathcal{J}-1}))\right]. \tag{4.30}$$

and

$$\vec{\mathrm{x}}_\phi = \left[\phi_0(0),\dots,\phi_0(N_\theta(\tau_0)),\dots,\phi_{N_\mathcal{J}-1}(0),\dots,\phi_{N_\mathcal{J}-1}(N_\theta(\tau_{N_\mathcal{J}-1}))\right]. \tag{4.31}$$

respectively. The total length of the solution vector $\vec{\mathrm{x}}$ is equal to $3 \cdot \sum_{i=0}^{N_\mathcal{J}-1} N_\theta(\tau_i)$ elements. The chosen version and chosen execution unit for a given job start time can then be indexed by adding $\sum_{i=0}^{N_\mathcal{J}-1} N_\theta(\tau_i)$ for the index in the version vector and $2 \cdot \sum_{i=0}^{N_\mathcal{J}-1} N_\theta(\tau_i)$ for the index in the unit vector, respectively. Extracting the start times θ, the task versions λ and the execution units ϕ from the values from the solution vector $\vec{\mathrm{x}}$ is consequently done using the helper function, which mainly filter out unused values resulting from removing job indexes which are not scheduled (because of potentially not necessary precedence

constraints). Additionally, the start times are sorted in ascending order for each task. These functions are defined as:

$$\theta_i = \texttt{sort}(t_s \in \vec{\mathrm{x}}[\texttt{index}(i,n)] | \forall n \in [0, N_\theta(\tau_i)) \text{ with } t_s \neq -1), \tag{4.32}$$

$$\lambda_i(n) = \vec{\mathrm{x}}[\texttt{index}(i,n) + \sum_{i=0}^{N_\mathcal{T}-1} N_\theta(\tau_i)], \tag{4.33}$$

$$\phi_i(n) = \vec{\mathrm{x}}[\texttt{index}(i,n) + 2 \cdot \sum_{i=0}^{N_\mathcal{T}-1} N_\theta(\tau_i)], \tag{4.34}$$

and directly translate to the schedule $\Sigma = \{\theta_i(n), \lambda_i(n), \phi_i(n)\}$.

Contribution 4: Solution Vector

The solution vector $\vec{\mathrm{x}}$ represents an input to an optimization algorithm used to generate schedules. With the given definitions and functions to interpret the indexes, the solution vector can be translated into a schedule $\Sigma = \{\theta_i(n), \lambda_i(n), \phi_i(n)\}$, which can then be optimized or deployed on the system.

The simulated annealing-based scheduling algorithm starts with an initial solution vector, which can either be randomly generated, created with simple starting values or a heuristic approach like the polyhedral-based scheduler introduced in Section 4.2.1[1].

An upper and a lower bound is calculated for the solution vector $\vec{\mathrm{x}}$. The lower bound $\vec{\mathrm{x}}_{\text{lb}}$ is defined as

$$\vec{\mathrm{x}}_{\text{lb}} = \left[-\vec{\mathrm{x}}_{\theta,\text{lb}} - | -\vec{\mathrm{x}}_{\lambda,\text{lb}} - | -\vec{\mathrm{x}}_{\phi,\text{lb}} - \right]. \tag{4.35}$$

Again, the bound is split into three parts, where each part represents the lower bound for the start time vector $\vec{\mathrm{x}}_\theta$, the version vector $\vec{\mathrm{x}}_\lambda$ and the execution unit vector $\vec{\mathrm{x}}_\phi$, respectively. The lower bound for the start time vector $\vec{\mathrm{x}}_{\theta,\text{lb}}$ is defined as:

$$\vec{\mathrm{x}}_{\theta,\text{lb}} = [\ \underbrace{0,\dots,0}_{\sum_{\tau_i \in \{\tau \in \mathcal{T} \mid D(\tau) \neq \emptyset\}} N_\theta(\tau_i)}\ ,\ \overbrace{-1,\dots,-1}^{\sum_{\tau_j \in \{\tau \in \mathcal{T} \mid D(\tau) = \emptyset\}} N_\theta(\tau_j)}\ ], \tag{4.36}$$

which is equal to 0 for all tasks $\tau_i \in \mathcal{T}$ with an implicit deadline $D(\tau_i) \neq \emptyset$ and -1 for all tasks $\tau_j \in \mathcal{T}$ without an implicit deadline $D(\tau_j) = \emptyset$. A value of -1 indicates that the given job is not scheduled. With the first part of $\vec{\mathrm{x}}_{\theta,\text{lb}}$ assigned to tasks τ_i for which an implicit deadline is defined, i.e.

[1] An evaluation of the strategy to generate starting vectors is given in Section 7.3.2.

$D_i \neq \emptyset$, and the second part of $\vec{\mathrm{x}}_{\theta,\mathrm{lb}}$ reserved for tasks τ_j without an implicit deadline $D_j = \emptyset$, the vector $\vec{\mathrm{x}}_{\theta,\mathrm{lb}}$ consists of a set of `0`s followed by a set of `-1`s.

The lower bound for the version vector $\vec{\mathrm{x}}_{\lambda,\mathrm{lb}}$ is defined as

$$\vec{\mathrm{x}}_{\lambda,\mathrm{lb}} = [0,\dots,0], \tag{4.37}$$

where the lowest possible version index for all tasks is 0 and the lower bound for the execution unit vector $\vec{\mathrm{x}}_{\phi,\mathrm{lb}}$ is defined as

$$\vec{\mathrm{x}}_{\phi,\mathrm{lb}} = [0,\dots,0], \tag{4.38}$$

with the lowest possible execution unit index for all tasks being 0. Thus, the total lower bound $\vec{\mathrm{x}}_{\mathrm{lb}}$.

The upper bound $\vec{\mathrm{x}}_{\mathrm{ub}}$ is defined similarly as

$$\vec{\mathrm{x}}_{\mathrm{ub}} = \left[-\vec{\mathrm{x}}_{\theta,\mathrm{ub}}- \,|\, -\vec{\mathrm{x}}_{\lambda,\mathrm{ub}}- \,|\, -\vec{\mathrm{x}}_{\phi,\mathrm{ub}}-\right], \tag{4.39}$$

with

$$\vec{\mathrm{x}}_{\theta,\mathrm{ub}} = [t_P,\dots,t_P], \tag{4.40}$$

describing the upper bound for the start time vector $\vec{\mathrm{x}}_{\theta}$, which is equal to the total schedule period t_P for all tasks $\tau_i \in \mathcal{T}$.

The upper bound for the version vector $\vec{\mathrm{x}}_{\lambda,\mathrm{ub}}$ is defined as

$$\vec{\mathrm{x}}_{\lambda,\mathrm{ub}} = \left[V_{\max}(\tau_0),\dots,V_{\max}(\tau_{N_{\mathcal{T}}-1})\right], \tag{4.41}$$

where the highest possible version index for all tasks is equal to the maximum number of versions for the given task $V_{\max}(\tau_i)$.

The upper bound for the execution unit vector $\vec{\mathrm{x}}_{\phi,\mathrm{ub}}$ is defined as

$$\vec{\mathrm{x}}_{\phi,\mathrm{ub}} = \left[N_{\mathcal{U},d}(\tau_0),\dots,N_{\mathcal{U},d}(\tau_{N_{\mathcal{T}}-1})\right]. \tag{4.42}$$

Here, the number of distinct unit indexes $N_{\mathcal{U},d}(\tau_i)$ is used, instead of the total number of execution units $N_{\mathcal{U}}$, because not all task versions are compatible with all execution units. Thus, the upper bound for the execution unit vector $\vec{\mathrm{x}}_{\phi}$ is defined as the maximum number of distinct execution units that can execute any version of the given task τ_i. The number of distinct unit indexes $N_{\mathcal{U},d}$ is calculated as

$$N_{\mathcal{U},d}(\tau_i) = \left|\{u \in \mathcal{U} \mid u \in U_i^v \;\forall\, v \in [0, V_{\max}(\tau_i))\}\right|. \tag{4.43}$$

Additionally, a translation of local unit indexes to global unit indexes needs to be implemented to make use of this more efficient indexing of the unit solution vector $\vec{\mathrm{x}}_{\phi}$. Should the global unit index be used for all tasks and all jobs, then the space of invalid solutions would be larger, since a global unit index might be chosen for a candidate solution in $\vec{\mathrm{x}}$, which might not be valid for the given task. With this optimization, the solution space is constraint to accelerate the search for valid solutions.

Because of the reasons stated above, fulfilling the scheduling constraints for a valid schedule, as defined in Contribution 6, must be ensured. This is facilitated via a given cost function $F_{\text{cost}}(\vec{\mathrm{x}})$ instead of fixed optimization constraints. The given schedule constraints are thus implemented via penalty terms $p_{\dagger}(\vec{\mathrm{x}})$, which increase the cost of a solution vector $\vec{\mathrm{x}}$ if a constraint is violated. Additionally, bonus terms $p_{*}(\vec{\mathrm{x}})$ are introduced to give a bonus to the cost function, should optimization goals be met. In general, the cost function is defined as

$$F_{\text{cost}}(\vec{\mathrm{x}}) = \sum_{\dagger} w_{\dagger} \cdot p_{\dagger}(\vec{\mathrm{x}}) - \sum_{*} w_{*} \cdot p_{*}(\vec{\mathrm{x}}), \tag{4.44}$$

where $w_{\dagger}$ and w_{*} are defined as weights to scale the penalty and bonus terms and $p_{\dagger}(\vec{\mathrm{x}})$ and $p_{*}(\vec{\mathrm{x}})$ are the penalty and bonus terms, respectively. To ensure that the generated schedule is valid, these penalty terms are mapped to one schedule constraint each.

To ensure that scheduled tasks adhere to their implicit deadline, the penalty term $p_{\text{DL}}(\vec{\mathrm{x}})$ is introduced. This penalty term is defined as

$$p_{\text{DL}}(\vec{\mathrm{x}}) = \sum_{i=0}^{N_{\mathcal{T}}-1} \sum_{n=0}^{N_{\theta}(\tau_i)-1} \begin{cases} 0, \textbf{if } \texttt{dist}(i,n,i,n+1 \texttt{ mod } N_{\theta}(\tau_i)) \leq (1 \pm \gamma(\tau_i)) \cdot D(\tau_i) \\ \texttt{dist}(i,n,i,n+1 \texttt{ mod } N_{\theta}(\tau_i)), \textbf{else} \end{cases} \tag{4.45}$$

A penalty is given, should the time between two subsequent jobs $\theta_i(n)$ and $\theta_i(n+1)$ or $\theta_i(0)+t_P-\theta_i(n)$ lie outside the given period slack of $(1 \pm \gamma(\tau_i))$ of the tasks implicit deadline $D(\tau_i)$. Here, the periodicity of the schedule needs to be is considered.

To ensure that no two tasks are scheduled to execute on the same execution unit at the same time, the penalty term $p_{\text{OL}}(\vec{\mathrm{x}})$ is introduced. This penalty term is given with

$$p_{\mathrm{OL}}(\vec{\mathrm{x}}) = \sum_{i=0}^{N_{\mathcal{J}}-1} \sum_{j=0}^{N_{\mathcal{J}}-1} \sum_{n=0}^{N_\theta(\tau_i)-1} \sum_{\substack{m=0,\\ m\neq n}}^{N_\theta(\tau_j)-1} \begin{cases} \mathtt{max}\Big(0, \mathtt{min}\big(\theta_i(n) + C_i^{\lambda_i(n)}, \\ \quad \theta_j(m) + C_j^{\lambda_j(m)}\big) - \mathtt{max}\big(\theta_i(n), \theta_j(m)\big)\Big) \\ \quad \textbf{if } \phi_i(n) = \phi_j(m) \\ 0, \textbf{ else} \end{cases} \tag{4.46}$$

The penalty is then proportional to the amount of overlap between two tasks τ_i and τ_j on the same execution unit $\phi_i(n) = \phi_j(m)$. Should there be no overlap, or the tasks are scheduled on different execution units, no penalty is given.

For two tasks τ_i and τ_j with a defined precedence constraint $P_i^{\lambda_i(n)}(\tau_j) \neq \emptyset$, the penalty term $p_{\mathrm{prec}}(\vec{\mathrm{x}})$ is introduced. We furthermore define the function $\mathtt{prec_index}(i,n,j)$, which returns the indexes of the jobs of task τ_j, which is scheduled due to a precedence constraint between τ_i and τ_j, resulting from in $P_i^{\lambda_i(n)}(\tau_j) \neq \emptyset$. While there might be an assigned index for the mth job of task τ_j, this job might not be scheduled because the scheduled task version $\lambda_i(n)$ of the given job n of τ_i does not require task τ_j. The given penalty term $p_{\mathrm{prec}}(\vec{\mathrm{x}})$ is defined as

$$p_{\mathrm{prec}}(\vec{\mathrm{x}}) = \sum_{i=0}^{N_{\mathcal{J}}-1} \sum_{n=0}^{N_\theta(\tau_i)-1} \sum_{j=0}^{N_{\mathcal{J}}-1} \sum_{\substack{m\in \\ \mathtt{prec_index}(i,n,j)}} \begin{cases} \mathtt{dist}(i,n,j,m), \\ \quad \textbf{if } \mathtt{dist}(i,n,j,m) \notin [p_{min}, p_{max}) \\ 0, \textbf{ else} \end{cases} \tag{4.47}$$

where p_{min} and p_{max} are the minimum and maximum values for the given precedence constraint between the two tasks, i.e. $(p_{min}, p_{max}) \in P_i^{\lambda_i(n)}(\tau_j)$. A penalty value of `1` is given for every time step over or under the given precedence constraints $P_i^{\lambda_i(n)}(\tau_j)$ between task τ_is nth job and the task τ_j, of which the job with the index $\mathtt{prec_index}(i,n,j)$ is considered.

To ensure that the access towards every shared resource $r_k \in \mathcal{R}$ is limited to a bandwidth of 100 %, the penalty term $p_{\mathrm{acc}}(\vec{\mathrm{x}})$ is introduced. This penalty term

is calculated as:

$$p_{\text{acc}}(\vec{\text{x}}) = \sum_{t=0}^{t_P-1} \sum_{r_k \in \mathcal{R}} \begin{cases} 0, \textbf{ if } \sum_{i=0}^{N_{\mathcal{T}}-1} \sum_{n=0}^{N_\theta(\tau_i)-1} B_i^{\lambda_i(n)}(r_k) < 1 \\ \qquad \forall \theta_i(n) \text{ s.t. } t \in [\theta_i(n)+a_{start}, \theta_i(n)+a_{end}) \\ \sum_{i=0}^{N_{\mathcal{T}}-1} \sum_{n=0}^{N_\theta(\tau_i)-1} B_i^{\lambda_i(n)}(r_k) \textbf{ else} \\ \qquad \forall \theta_i(n) \text{ s.t. } t \in [\theta_i(n)+a_{start}, \theta_i(n)+a_{end}) \end{cases} \tag{4.48}$$

with $(a_{start}, a_{end}) \in A_i^{\lambda_i(n)}(r_k)$ signifying the absolute start and end times of an access windows. A penalty value is given, proportional to the amount by which the access bandwidth towards a shared resource r_k is exceeded.

Bonus terms allow the shaping of the cost function to fulfill additional tasks. It is possible to give a bonus for occupying less unique execution units with bonus function of defined as

$$p_{\text{occ}} = N_{\mathcal{U}} - |\{u_o \in \mathcal{U} \mid u_o \in \phi_i(n) \,\forall\, i \in [0, N_{\mathcal{T}}-1), n \in [0, N_\theta(\tau_i)-1)\}|, \tag{4.49}$$

where the occupied execution unit amount, given by

$$|\{u_o \in \mathcal{U} \mid u_o \in \phi_i(n) \,\forall\, i \in [0, N_{\mathcal{T}}-1), n \in [0, N_\theta(\tau_i)-1)\}|$$

is subtracted from the total amount of execution units $N_{\mathcal{U}}$.

One bonus term is always given to improve how close the task period is to the given periodic deadline. This bonus term is defined as

$$p_{\text{timely}}(\vec{\text{x}}) = \sum_{\substack{\tau_i \in \mathcal{T} \text{ s.t.} \\ D(\tau_i) \neq \emptyset}} \sum_{n=0}^{N_\theta(\tau_i)} |D(\tau_i) - \texttt{dist}(i, n, i, n+1 \bmod N_\theta(\tau_i))| \tag{4.50}$$

Additional bonus terms are feasible. These could be used to calculate a viable selection and deployment of Chiplets, or to allow the creation of the transitional schedules for mode-change scheduling, as explained in Section 4.3.

Contribution 5: Periodic Schedule Constraints

The view on the schedule and the corresponding solution vector as well as functions allow for a periodic schedule description, which is used to generate the time triggered schedules.

A cost function value of $F_{\text{cost}} \leq 0$ is defined as having achieved a valid schedule. For this, it is necessary, that the largest possible bonus term is smaller than the smallest possible penalty term. For this, the weights for the bonus terms w_* and the weights for the penalty terms $w_\dagger$ need to be dimensioned to prevent a cost function value below 0 for breached constraints. With the penalty and bonus terms returning natural numbers, the minimum for each penalty weight needs to be at least a given value $w_{\dagger,\min}$ for all penalty weights $w_\dagger$. For the presented bonus terms p_{occ} and p_{timely}, it must be ensured that

$$w_{\dagger,\min} \cdot 1 \geq w_{\text{occ}} \cdot \max_{\vec{x}}(p_{\text{occ}}(\vec{x})) + w_{\text{timely}} \cdot \max_{\vec{x}}(p_{\text{timely}}(\vec{x})). \tag{4.51}$$

Rearranging this to isolate for $\frac{w_{\dagger,\min}}{w_{\text{occ}} \cdot w_{\text{timely}}}$ then yields

$$\frac{w_{\dagger,\min}}{w_{\text{occ}} \cdot w_{\text{timely}}} \leq \frac{\max_{\vec{x}}(p_{\text{occ}}(\vec{x}))}{w_{\text{timely}}} + \frac{\max_{\vec{x}}(p_{\text{timely}}(\vec{x}))}{w_{\text{occ}}}. \tag{4.52}$$

Expanding this with

$$\max_{\vec{x}}(p_{\text{occ}}(\vec{x})) = N_{\mathcal{U}} - 1 \text{ and} \tag{4.53}$$

$$\max_{\vec{x}}(p_{\text{timely}}(\vec{x})) = |\{\tau_i \in \mathcal{T} \mid D(\tau_i) \neq \emptyset\}| \cdot \frac{t_P}{t_\delta} \tag{4.54}$$

then leads to

$$\frac{w_{\dagger,\min}}{w_{\text{occ}} \cdot w_{\text{timely}}} \leq \frac{N_{\mathcal{U}} - 1}{w_{\text{timely}}} + \frac{|\{\tau_i \in \mathcal{T} \mid D(\tau_i) \neq \emptyset\}| \cdot \frac{t_P}{t_\delta}}{w_{\text{occ}}}. \tag{4.55}$$

for the given case. If this holds true for every penalty weight, then a cost function value of $F_{\text{cost}} \leq 0$ guarantees a valid schedule. Changing the bonus terms necessitates the inclusion of the further bonus weights w_* and the new calculation of $w_{\dagger,\min}$.

The last step before using the solver is to introduce a discretization strategy, since start times θ_i are discretized to t_δ, task versions λ_i are discretized in 1-wide steps until $V_{\max}(\tau_i)$ and execution units are discretized in 1 wide steps up

to a total of $N_{\mathcal{U}}$. In the simulated annealing solver the floor function is used, always rounding non-integer values to the lower integer. With this, every value has the same probability.

In conclusion, this creation of the solution vector $\vec{x}$, the bounds $\vec{x}_{\text{lb}}$ and $\vec{x}_{\text{ub}}$ as well as the cost function $F_{\text{cost}}(\vec{x})$ and the subsequent optimization of the solution vector using a simulated annealing-based solver can be summarized into a function taking in the task descriptions and returning a valid schedule.

Contribution 6: Simulated Annealing Scheduler

The simulated annealing-based scheduler takes in the task descriptions and returns a valid schedule via the given solution vector. While other optimizers like non-dominated sorting genetic algorithm (NSGA) could be used, the simulated annealing-based solver is chosen for this work because it is used in related work for scheduling problems and provides an easily extensible framework.

Once the cost function and the solution vector is set up, the schedule can be created using many optimization methods. In this work, a simulated annealing-based solver is chosen. An ablation study determining hyperparameters is presented in Section 7.1. An evaluation of the success rate in finding valid schedules is given in Section 7.2, the algorithm runtime evaluation is given in Section 7.3 for varying constraint values of A_i^v, B_i^v, C_i^v and $N_{\mathcal{T}}$ as well as $N_{\mathcal{U}}$.

Figure 4.7 in Example 9 shows an excerpt of a time diagram of a generated schedule from the overarching example. The example displays the periodic schedule for the time around $t = 0$ and was generated using the simulated annealing-based solver using the task model values from Table 4.7 as an input. The tasks scheduled in Figure 4.7 are given their values in Table 4.11. This example shows that the scheduling algorithm is able to find a valid schedule for the given task set in the case only one execution mode exists. However, another main contribution is the ability to find valid schedules and transition schedules for multiple execution modes, which is shown in the next chapters.

Example 9: Timing Diagram of Generated Schedule

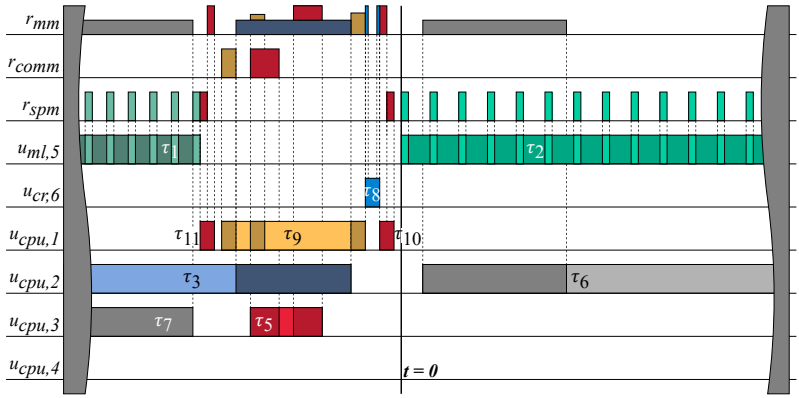


Figure 4.7: Diagram of schedule created from the example use case. Tasks are marked with the index and a corresponding color. Access windows are marked darker.

The bandwidth information allows for a stacking of access windows on the main memory r_{mm}, if a platform allows for this. Because the developer needs to have platform information beforehand and needs to model the resources anyway, this restriction is manageable. Since the point of the value of $t = 0$ *is arbitrary, another randomized start is feasible. The following values are determined for the displayed jobs.*

Table 4.11: Scheduling values of the example task set.

Task	Jobs	Job Start	Unit	Version
τ_1	$N_\theta(\tau_1) = 2$	$\theta_1(1) = t_P - 7040$	$\phi_1(1) = u_{ml,5}$	$\lambda_1(1) = 1$
τ_2	$N_\theta(\tau_2) = 2$	$\theta_2(0) = 0$	$\phi_2(0) = u_{ml,5}$	$\lambda_2(0) = 1$
τ_3	$N_\theta(\tau_3) = 4$	$\theta_3(3) = t_P - 1035$	$\phi_3(3) = u_{cpu,2}$	$\lambda_3(3) = 0$
τ_4	*Not scheduled in example excerpt.*			
τ_5	$N_\theta(\tau_5) = 6$	$\theta_5(5) = t_P - 105$	$\phi_5(2) = u_{cpu,3}$	$\lambda_5(2) = 0$
τ_6	$N_\theta(\tau_6) = 2$	$\theta_6(0) = 15$	$\phi_6(0) = u_{cpu,2}$	$\lambda_6(0) = 0$
τ_7	$N_\theta(\tau_7) = 1$	$\theta_7(0) = t_P - 2145$	$\phi_7(0) = u_{cpu,3}$	$\lambda_7(0) = 0$
τ_8	$N_\theta(\tau_8) = 4$	$\theta_8(3) = t_P - 25$	$\phi_8(3) = u_{cr,5}$	$\lambda_8(3) = 1$
τ_9	$N_\theta(\tau_9) = 5$	$\theta_9(4) = t_P - 125$	$\phi_9(4) = u_{cpu,1}$	$\lambda_9(4) = 0$
τ_{10}	$N_\theta(\tau_{10}) = 4$	$\theta_{10}(3) = t_P - 15$	$\phi_{10}(3) = u_{cpu,1}$	$\lambda_{10}(3) = 0$
τ_{11}	$N_\theta(\tau_{11}) = 4$	$\theta_{11}(3) = t_P - 140$	$\phi_{11}(3) = u_{cpu,1}$	$\lambda_{11}(3) = 0$

4.3 Mode-Change Task Scheduler Extensions

Creating one time triggered schedule in central car servers (CECASs) is not sufficient for the complex tasks sets deployed in modern CECAS. These use case include mode changes [119, 120], for which the main optimization is reducing the transitional time between the old mode and the new mode [85]. In this chapter the scheduling model from Section 4.1 is used together with the scheduling approaches from Section 4.2 to automatically generate one schedule for each of the different execution modes. In addition to that, transition schedules executed once between the periodic schedules are generated to facilitate a continuation of continuing tasks with the given period. Furthermore, precedence constraints must be adhered to during the transition period. Multiple mode-change schedulers exist [121, 122, 123, 124]. But these are not tightly integrable into this automated scheduling scheme, which covers an integrated end-to-end approach as presented in this thesis.

The mode change scheduler uses the existing task model, which is presented in Section 4.1. It extends the time triggered scheduling approach presented in Section 4.2. The complete workflow necessitates the introduction of the following model elements.

4.3.1 Model Extensions to facilitate Mode-Changes

A mode specification M is added to define the deadline of tasks in an execution mode as well as to list available task versions in this execution mode. For this, the collection of execution modes is defined as:

$$M = \{m_\mu \mid \mu \in [0, N_M)\}, \tag{4.56}$$

where one execution mode is defined as $m^\mu = \{V^\mu, D^\mu, \gamma^\mu\}$, with

$$V^\mu : \mathcal{T} \to \mathcal{P}(\mathbb{N}) \tag{4.57}$$

defined as a function assigning the viable task versions to the given tasks,

$$D^\mu : \mathcal{T} \to \mathbb{Q}^+ \cup \varnothing, \tag{4.58}$$

as defined in Equation (4.4), assigns the given deadlines to the tasks τ_i in a given execution mode m^μ and

$$\gamma^\mu : \mathcal{T} \to [0,1], \tag{4.59}$$

gives an allowed slack to the deadline D^μ. N_M defines the amount of execution modes. The result of the function of possible indexes of task versions V^μ has the form of a list of indexes for $V^\mu(\tau_i)$ as

$$V^\mu(\tau_i) = \{\upsilon \mid \upsilon \in [0, V_{\max}(\tau_i)) \wedge \upsilon \text{ in execution mode } m^\mu\}, \tag{4.60}$$

where υ are indexes of the version of task τ_i, which are available in execution mode m^μ. Table 4.12 gives an overview of the added model components, which describe the execution modes

Table 4.12: Model component descriptions for execution modes.

Component	Formula	Description
Mode Set	$M = \{m_\mu \mid \mu \in [0, N_M)\}$	Set of execution modes which need to be scheduled.
Execution Mode	$m^\mu = \{V^\mu, D^\mu\}$	An execution mode is defined by the viable task versions and the deadlines of the tasks in this execution mode.
Viable Task Versions	$V^\mu : \mathcal{T} \to \mathcal{P}(\mathbb{N})$	Function assigning the viable task versions to the given tasks in a given execution mode.
Continuing Tasks	$\mathcal{T}^{\mu,\nu}_{\text{cont}} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) \neq \emptyset \wedge D^\nu(\tau) \neq \emptyset\}$	Set of tasks, which are present in both execution modes m^μ and m^ν.
New Tasks	$\mathcal{T}^{\mu,\nu}_{\text{new}} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) = \emptyset \wedge D^\nu(\tau) \neq \emptyset\}$	Set of tasks, which are only present in the new execution mode m^ν.
Old Tasks	$\mathcal{T}^{\mu,\nu}_{\text{old}} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) \neq \emptyset \wedge D^\nu(\tau) = \emptyset\}$	Set of tasks, which are only present in the old execution mode m^μ.
Task Deadlines	$D^\mu : \mathcal{T} \to \mathbb{Q}^+ \cup \emptyset$	Function assigning deadlines to the given tasks in a given execution mode.
Deadline Slack	$\gamma^\mu : \mathcal{T} \to [0,1]$	Function assigning a deadline slack to the given tasks in a given execution mode.

The goal is to create a network of time triggered schedules, represented by a directed graph $G_\Sigma = \{V_\Sigma, E_\Sigma\}$, where each vertex is a periodic schedule and each edge is either a transition schedule or a transition without a given schedule. The input is a directed specification graph $G_M = \{M, M_T\}$. This chapter thus introduces the function

$$S : G_M \to G_\Sigma, \tag{4.61}$$

which turns the specification G_M into a graph of connected schedules G_Σ.

In the specification graph G_M, each vertex represents an execution mode $m^\mu \in M$, and each directed edge $e^{\mu,\nu} = (v^\mu, v^\nu) \in M_T$ represents a possible mode change from execution mode m^μ to execution mode m^ν. This also includes the possibility for the introduction of schedules, which are only executed once (like start up schedules). If a schedule σ_μ is periodic (i.e. can be executed after itself), then $e^{\mu,\mu} \in M_T$. Each vertex $m^\mu \in M$ contains the description of the viable task versions V^μ and deadlines D^μ of the tasks for the respective execution mode m^μ, which the resulting vertex in G_Σ must adhere to. An edge between two vertices $e^{\mu,\nu} \in M_T$ represents a possible mode change from execution mode m^μ to execution mode m^ν. As these edges do not contain additional information or weights and are defined as simple directed edges. This puts the need to create and adhere to constraints in the hands of the scheduling functions for schedules $S_{\Sigma,*}$.

The schedule graph G_Σ contains a vertex $\Sigma^\mu \in \Sigma_P$ for each execution mode $m^\mu \in M$, which represents a periodic time triggered schedule Σ^μ, created for the respective execution mode configuration m^μ. Each directed edge $e^{\mu,\nu} = (s_\mu, s_\nu) \in M_C$ represents the possibility of a mode change from schedule Σ^μ to schedule Σ^ν. This is reflected in Σ_T. Each edge represents either a transition schedule $\Sigma^{\mu,\nu}$ or a direct mode change from Σ^μ to Σ^ν. In case a transition schedule, which is created to change from schedule Σ^μ to schedule Σ^ν, is needed, adhering to the given constraints is still necessary.

While $S_{\Sigma,0}$, as given in Contribution 6, creates a schedule for one independent execution mode, the mode change scheduler needs a scheduling approach to adhere to the added constraints given by the transition between two execution modes. Thus turning the graph G_M into G_Σ requires an addition to the scheduling function $S_{\Sigma,0}$, resulting in two new scheduling functions. These are defined as $S_{\Sigma,1}$ and $S_{\Sigma,2}$. $S_{\Sigma,1}$ creates a schedule for one execution mode $m^\mu \in M$, while adhering to the constraints given by a previous schedule Σ^ν for execution mode $m^\nu \in M$. $S_{\Sigma,2}$ creates a transition schedule between two given schedules Σ^μ and Σ^ν for execution modes m^μ and m^ν, while adhering to the constraints given by both schedules. Continuing tasks, i.e. periodic tasks, which are both present in the old schedule and the new schedule need to continue their period into the new schedule. For easier definition, these tasks are collected in the set

$$\mathcal{T}^{\mu,\nu}_{\text{cont}} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) \neq \emptyset \wedge D^\nu(\tau) \neq \emptyset\} \tag{4.62}$$

for each mode change from M^μ to M^ν. Additionally, tasks which are only present in the new schedule Σ^ν are collected in the set

$$\mathcal{T}^{\mu,\nu}_{\text{new}} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) = \emptyset \wedge D^\nu(\tau) \neq \emptyset\} \tag{4.63}$$

and tasks which are only present in the old schedule Σ^μ are collected in the set

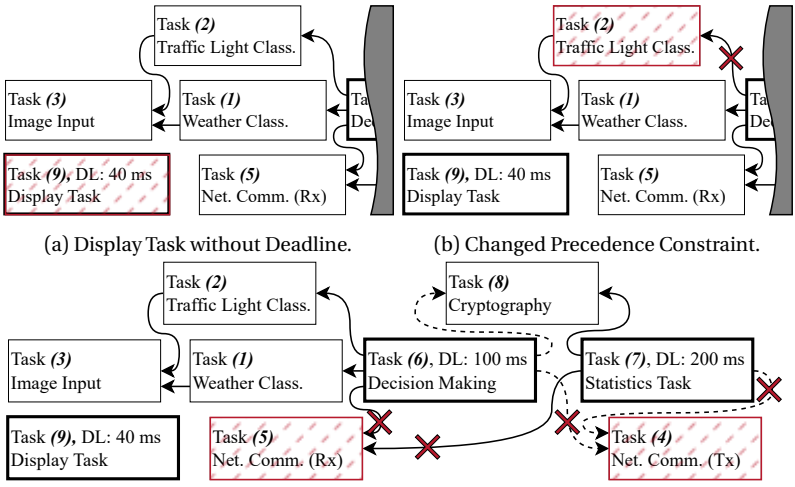
$$\mathcal{T}_{\text{old}}^{\mu,\nu} = \{\tau \in \mathcal{T} \mid D^\mu(\tau) \neq \emptyset \wedge D^\nu(\tau) = \emptyset\}. \tag{4.64}$$

Table 4.13: Model component descriptions for mode change schedules.

Component	Formula	Description
Specification Graph	$G_M = \{M, M_T\}$	Graph specifying execution modes M (see Table 4.12) and possible mode changes.
Scheduling Graph	$G_\Sigma = \{\Sigma_P, \Sigma_T\}$	Graph containing the generated schedules for the execution modes and the transition schedules for the mode changes.
Schedule for Execution Mode	$\Sigma^\mu \in \Sigma_P$	A time-triggered schedule for execution mode m^μ.
Transition Schedule	$\Sigma^{\mu,\nu} \in \Sigma_T$	transition schedule between schedule Σ^μ and schedule Σ^ν.
Mode Specific Schedule Values	$\theta_i^\mu(n)$, $\lambda_i^\mu(n)$, $\phi_i^\mu(n)$, $N_\theta^\mu(\tau_i)$	Schedule values for task τ_i in schedule Σ^μ.
General Scheduling Function S	S	Function to create a complete scheduling graph G_Σ from the specification graph G_M.
Scheduling Function $S_{\Sigma,0}$	$S_{\Sigma,0}$	Standard scheduling function to create a schedule without any constraints given by previous schedules.
Scheduling Function $S_{\Sigma,1}$	$S_{\Sigma,1}$	Function to create a schedule for one execution mode m^μ while adhering to model constraints and job start constraints given by a previous schedule Σ^ν.
Scheduling Function $S_{\Sigma,2}$	$S_{\Sigma,2}$	Scheduling function to create one transition schedule between two given schedules Σ^μ and Σ^ν.

To illustrate different execution modes, Example 10 lists three examples, in which the precedence constraints between modes differ. Changing between these modes while still meeting deadlines is the goal of the mode change scheduler.

Example 10: Different Task Versions for Execution Modes



(a) Display Task without Deadline. (b) Changed Precedence Constraint.

(c) Unavailable Network Connection.

Figure 4.8: The execution modes of the given example task set can change based on availability or user input. Starting from Figure 1.3, some functions might become unavailable. These are marked in red.

Figure 4.8 shows three examples of different execution modes which require different schedules. Should the user turn off the display, the task does not need to be scheduled. Should no traffic light be detected, no classification is necessary and should the network connection be unavailable, no sending or receiving of data is necessary. These are reflected in the model values shown in Table 4.14.

Table 4.14: Example values adapted for mode changes.

Mode	Changed Task Set Values
M_0	*No changes to Example 5, all versions available*
$M^{(a)}$	$D^{(a)}(\tau_9) = \emptyset$
$M^{(b)}$	*Addition of execution mode to* τ_6*, where* $P_6^1(\tau_2) = \emptyset, P_6^1(\tau_1) = P_6^0(\tau_1)$ *and* $P_6^1(\tau_5) = P_6^0(\tau_5)$ *but* $V^{(b)}(\tau_6) = \{1\}$
$M^{(c)}$	*As above* $P_6^2(\tau_4) = P_6^2(\tau_5) = \emptyset, P_6^2(\tau_1) = P_6^0(\tau_1), P_6^2(\tau_2) = P_6^0(\tau_2),$ $P_6^2(\tau_5) = P_6^0(\tau_5)$ *but* $V^{(c)}(\tau_6) = \{2\}$*. Additionally:* $P_7^1(\tau_4) = P_7^1(\tau_5) = \emptyset,$ $P_7^2(\tau_8) = P_7^0(\tau_8)$ *and* $V^{(c)}(\tau_8) = \{1\}$

As the example shows, different execution modes are characterized by different available task versions and deadlines. Precedence constraints and other task

characteristics are defined via the task versions. This allows for the modelling of tasks which omit the input of certain tasks via task versions with and without precedence constraints. Additionally, the availability of certain task versions can be used to model the unavailability of certain functions in an execution mode. Should a task not be necessary in an execution mode, its deadline is set to $\varnothing$. These examples are used to model the different execution modes in Example 10.

Contribution 7: Mode-Change Schedule Model

The addition of the mode specifications as assigned deadlines and valid task version for each execution mode allows for the modelling of different execution modes in a unified and simple way. It furthermore allows the software components to remain unchanged once modelled.

4.3.2 Integration of Mode-Change Scheduling

The scheduling functions use an extension of the cost function from Section 4.2.2, F_{cost}, by adding extra penalty terms. The same simulated annealing solver is chosen. The penalty terms are defined to implement the following constraints.

For easier definition, the $\texttt{dist}^{\mu,\nu}(i,n,j,m)$ function is defined as the distance of the start time of job n of task τ_i in schedule Σ^ν to the start time of job m of task τ_j in schedule Σ^μ. This is formalized as:

$$\texttt{dist}^{\mu,\nu}(i,n,j,m) = |t_P^\nu - \theta_j^\mu(m) + \theta_i^\nu(n)|. \tag{4.65}$$

Given an already existing schedule Σ^μ, if a job n of a task τ_i has a start time $\theta_i(n)$ and the task has a periodic deadline of $D^\nu(\tau_i)$ in the new schedule configuration, then one job of this task needs to be scheduled between $D^\nu(\tau_i)\cdot(1-\gamma^\nu(\tau_i))$ and $D^\nu(\tau_i)\cdot(1+\gamma^\nu(\tau_i))$ after the last start time in the old schedule. This is formalized as:

$$\begin{aligned}\theta_i^\nu(0) \geq &\begin{cases}0, \textbf{ if } \theta_i^\mu(N_\theta(\tau_i)-1)+D^\nu(\tau_i)\cdot(1-\gamma^\nu(\tau_i)) < t_P^\mu \\ \theta_i^\mu(N_\theta(\tau_i)-1)+D^\nu(\tau_i)\cdot(1-\gamma^\nu(\tau_i)) \ \texttt{mod}\ t_P^\mu \textbf{ otherwise}\end{cases} \\ \wedge\, \theta_i^\nu(0) < &\ \theta_i^\mu(N_\theta(\tau_i)-1)+D^\nu(\tau_i)\cdot(1+\gamma^\nu(\tau_i)) \ \texttt{mod}\ t_P^\mu\end{aligned} \tag{4.66}$$

where $\theta_i^\mu(N_\theta(\tau_i)-1)$ is the start time of the last job of task τ_i in the old schedule Σ^μ, $D^\nu(\tau_i)$ is the period of task τ_i in the new schedule Σ^ν and $\theta_i(n)$ is the start time of job n of task τ_i in the new schedule Σ^ν. t_P^μ is the hyperperiod of the

old schedule Σ^{μ}. Note that the new schedule Σ^{n+1} is generated using these added constraints. Since a periodic schedule has no clear start or end, the new schedule can start at any point in time. Still, scheduled tasks need to be adapted to the phase of the old schedule.

This constraint is translated into the penalty term p_{inc} by calculating the distance of each job start time $\theta_i^{\lambda_i(n)}$ to the allowed interval and summing these distances up for all continuing tasks $\tau_c \in \{\tau \in \mathcal{T} \mid D^{\mu}(\tau) \neq \varnothing \wedge D^{\nu}(\tau) \neq \varnothing\}$. If a job start time is within the allowed interval, no penalty is added for this job. The penalty term is thus defined as:

$$p_{\text{inc}}(\vec{\text{x}}) = \sum_{\tau_i \in \mathcal{T}_{\text{cont}}^{\mu,\nu}} \begin{cases} 0, \textbf{ if } \texttt{dist}^{\mu,\nu}(i,0,i,N_{\theta}^{\mu}(\tau_i)-1) \\ \qquad \in [D^{\nu}(\tau_i)\cdot(1-\gamma^{\nu}(\tau_i)), D^{\nu}(\tau_i)\cdot(1+\gamma^{\nu}(\tau_i))) \\ \texttt{dist}^{\mu,\nu}(i,0,i,N_{\theta}^{\mu}(\tau_i)-1) - D^{\nu}(\tau_i) \textbf{ otherwise} \end{cases} \tag{4.67}$$

where $\mathcal{T}_{\text{cont}}^{\mu,\nu}$ is the set of continuing tasks between the two schedules. This penalty term is added to the cost function F_{cost} using a weighting factor w_{inc} to control the influence of this penalty term on the overall cost. The rest of the penalty terms stay the same as in Section 4.2.2.

Tasks which are scheduled with a start time of $\theta_i^{\mu}(N_{\theta}) \geq t_P - C_i^{\lambda_i(N_{\theta})}$, i.e. tasks which would not be able to finish their execution before the end of the period in the old schedule, need to be scheduled at the beginning of the new schedule with the corresponding execution time left, this is done by scheduling the last job of the new schedule, so that the execution time is preserved, when periodically executing the new tasks. Additionally, these tasks need to be scheduled on the same execution unit as in the old schedule to avoid migration overhead for the change of execution units. This constraint is formalized as:

$$\begin{aligned} &\lambda_i^{\nu}(N_{\theta}^{\nu}(\tau_i)-1) = \lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) \\ &\phi_i^{\nu}(N_{\theta}^{\nu}(\tau_i)-1) = \phi_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) \\ &\qquad \forall\, \tau_i \in \mathcal{T}^{\mu} \text{ s.t. } \theta_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu} \end{aligned} \tag{4.68}$$

This constraint is translated into the penalty term $p_{\text{in-mig}}$ by checking if the last job of the new schedule Σ^{ν} is scheduled on the same execution unit as the last job of the old schedule Σ^{μ} for each task, which has a continuing execution over the end of the old schedule. If this is not the case, a penalty is added. Additionally, the task version must stay constant under the same

conditions. The penalty term $p_{\text{in-mig}}$ is added to avoid migration of execution units for continuing tasks over the starting point of the new schedule. For each continuing task, which has a continuing execution over the end of the old schedule, the execution unit $\phi_i^\mu(N_\theta(\tau_i)-1)$ of the last job in the old schedule Σ^μ must be equal to the execution unit of the continuation of the job in the new schedule Σ^ν. Otherwise, a penalty is added. The penalty term is thus defined as:

$$p_{\text{in-mig}} = \sum_{\tau_i \in \mathcal{T}^\mu} \begin{cases} 1, \textbf{ if } \phi_i^\mu(N_\theta^\mu(\tau_i)-1) \neq \phi_i^\nu(N_\theta^\nu(\tau_i)-1) \\ \qquad \wedge \theta_i^\mu(N_\theta^\mu(\tau_i)-1) + C_i^{\lambda_i^\mu(N_\theta^\mu(\tau_i)-1)} > t_P^\mu \\ 0, \textbf{ otherwise} \end{cases} \tag{4.69}$$

$$+ \sum_{\tau_i \in \mathcal{T}^\mu} \begin{cases} 1, \textbf{ if } \lambda_i^\mu(N_\theta^\mu(\tau_i)-1) \neq \lambda_i^\nu(N_\theta^\nu(\tau_i)-1) \\ \qquad \wedge \theta_i^\mu(N_\theta^\mu(\tau_i)-1) + C_i^{\lambda_i^\mu(N_\theta^\mu(\tau_i)-1)} > t_P^\mu \\ 0, \textbf{ otherwise} \end{cases} \tag{4.70}$$

where $\phi_i^\mu(N_\theta(\tau_i)-1)$ is the execution unit of the last job of task τ_i in the old schedule Σ^μ and $\phi_i^\nu(N_\theta(\tau_i)-1)$ is the execution unit of the continuation of this job in the new schedule Σ^ν. Consequently, this is also the last execution unit of task τ_i in the new schedule Σ^ν. Here, $C_i^{\lambda_i^\mu(N_\theta^\mu(\tau_i)-1)}$ specifies the execution time of the chosen task version in the old schedule Σ^μ. Additionally, the task version must stay constant under the same conditions.

These penalty terms are added to the cost function F_{cost} using corresponding weighting factors $w_\dagger$ to control the influence of this penalty term on the overall cost. The rest of the penalty terms stay the same as in Section 4.2.2. Table 4.15 show the used penalty terms for the given position in the scheduling graph. For all scheduling algorithms.

Lastly, the cost functions for the penalty terms for the generation of transition schedules are introduced. Here, a few changes need to be implemented, because the transition schedule is not periodic. A transition schedule needs to be created to account for the transition between two already existing schedules Σ^μ and Σ^ν.

Thus, the penalty term $p_{\text{in-mig}}$ needs to be changed to not assume periodicity of the schedule. For this, the lower bound $\vec{x}_{\theta,\text{lb}}$ is reduced to $t_P^\mu - \theta_i^\mu(N_\theta^\mu(\tau_i)-1)$, i.e. the start time of the last job of the old schedule Σ^μ relative to the start of the new schedule Σ^ν. This allows the first job "in" $\Sigma^{\mu,\nu}$ to begin before the schedule, thus simulating the continuation of the job from the old schedule

Σ^{μ} into the transition schedule $\Sigma^{\mu,\nu}$. The upper bound stays constant. The last job of a new task or continuing task, $\tau_i \in \mathcal{T}^{\mu,\nu}_{\text{cont}} \cup \mathcal{T}^{\nu}_{\text{new}}$, thus needs to be scheduled in such a way that the execution time is preserved. This new version of $p_{\text{in-mig}}$ is formalized as:

$$p_{\text{in-mig}}(\vec{\text{x}}) = \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \phi_i^{\mu,\nu}(0) \neq \phi_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) \\ \qquad \wedge \theta_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.71}$$

$$+ \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \lambda_i^{\mu,\nu}(0) \neq \lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) \\ \qquad \wedge \theta_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.72}$$

$$+ \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \theta_i^{\mu,\nu}(0) \neq \theta_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) \\ \qquad \wedge \theta_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.73}$$

Additional, the last job of a new task or continuing task, $\tau_i \in \mathcal{T}^{\mu,\nu}_{\text{cont}} \cup \mathcal{T}^{\mu,\nu}_{\text{new}}$, needs to be scheduled in such a way that the execution time is preserved. This is done by scheduling the last job of the new schedule, so that the execution time is preserved, when periodically executing the new tasks. This constraint is implemented with the penalty term $p_{\text{out-mig}}$, defined as:

$$p_{\text{out-mig}}(\vec{\text{x}}) = \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \phi_i^{\mu,\nu}(N_{\theta}^{\nu}(\tau_i)-1) \neq \phi_i^{\nu}(0) \\ \qquad \wedge \theta_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu,\nu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.74}$$

$$+ \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \lambda_i^{\mu,\nu}(N_{\theta}^{\nu}(\tau_i)-1) \neq \lambda_i^{\nu}(0) \\ \qquad \wedge \theta_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu,\nu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.75}$$

$$+ \sum_{\tau_i \in \mathcal{T}^{\mu}} \begin{cases} 1, \textbf{ if } \theta_i^{\mu,\nu}(N_{\theta}^{\nu}(\tau_i)-1) \neq \theta_i^{\nu}(0) \\ \qquad \wedge \theta_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1) + C_i^{\lambda_i^{\mu,\nu}(N_{\theta}^{\mu}(\tau_i)-1)} > t_P^{\mu,\nu} \\ 0, \textbf{ otherwise} \end{cases} \tag{4.76}$$

Lastly, the deadlines for all tasks, which are only present in the new schedule Σ^{ν} need to be met at the end of the transition schedule $\Sigma^{\mu,\nu}$ backwards, starting from the beginning of the new schedule, given with:

$$p_{\text{out}}(\vec{x}) = \sum_{\tau_i \in \mathcal{T}^{\mu,\nu}_{\text{new}}} \begin{cases} 0, \textbf{ if } \texttt{dist}^{\mu,\nu}(i, N^{\mu,\nu}_{\theta}(\tau_i) - 1, i, 0) \\ \qquad \in [D^{\nu}(\tau_i) \cdot (1 - \gamma^{\nu}(\tau_i)), D^{\nu}(\tau_i) \cdot (1 + \gamma^{\nu}(\tau_i))) \\ \texttt{dist}^{\mu,\nu}(i, N^{\mu,\nu}_{\theta}(\tau_i) - 1, i, 0) - D^{\nu}(\tau_i) \textbf{ otherwise} \end{cases} \tag{4.77}$$

The penalty term p_{inc} is adapted accordingly. These penalty terms are added to the cost function F_{cost} using the weighting factors $w_{\dagger}$ to control the influence of this penalty term on the overall cost. The rest of the penalty terms stay the same as in Section 4.2.2. Table 4.15 gives an overview of the used penalty terms, depending on the scheduling function.

Table 4.15: Used cost functions of the respective scheduling algorithms.

Term	p_{DL}	p_{OL}	p_{prec}	p_{acc}	p_{inc}	$p_{\text{in-mig}}$	p_{out}	$p_{\text{out-mig}}$	p_{occ}	p_{timely}	p_{trans}
$S_{\Sigma,0}$	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
$S_{\Sigma,1}$	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗
$S_{\Sigma,2}$	✓[a]	✓	✓	✓	✓[a]	✓[a]	✓	✓	✓	✓	✓

[a]Adapted to not assume periodicity.

The standard scheduling function $S_{\Sigma,0}$ uses a cost function with the penalty terms p_{DL}, p_{OL}, p_{prec}, and p_{acc}. Additionally, the bonus terms p_{occ} and p_{timely} are added. This function is used for single-mode scheduling as well as for the first schedule in the multimode scheduling. For the second scheduling function $S_{\Sigma,1}$, the penalty terms p_{inc} and $p_{\text{in-mig}}$ are added to account for old tasks, continuing execution into the new execution mode. For creating transition schedules, the third scheduling function $S_{\Sigma,2}$ is used, which adds the penalty terms p_{out} and $p_{\text{out-mig}}$ to account for tasks, which continue execution into the next schedule. The bonus term p_{trans} is added to reduce the transitional time between the two schedules. The rest of the penalty and bonus terms stay the same for all three scheduling functions. Lastly, the penalty terms of p_{inc} and $p_{\text{in-mig}}$ are adapted to not assume periodicity, because transition schedules are only executed once.

Contribution 8: Mode Change Scheduler

The mode change scheduler uses additional penalty terms to create schedules for different execution modes but otherwise stays the same as the periodic scheduler. The extension itself is used to create schedules which can be periodically executed after other schedules of other modes. Additionally, transition schedules can be created.

The generated traditional schedule is not periodic, as it is only executed once to transition from one periodic schedule to another. The schedule length for the transition schedule is also defined as t_P, even though the transition schedule is not periodic. It is calculated as the least common multiple of the phase differences of all continuing tasks $\tau_i \in \{\tau \in \mathcal{T} \mid D^{\mu}(\tau) \neq \emptyset \wedge D^{\nu}(\tau)\}$ as $t_P = \text{lcm}(\{|D^{\mu}(\tau_i) - t_P|, |D^{\nu}(\tau_i) - t_P| \mid \tau_i \in \mathcal{T}_{\text{cont}}\})$. This ensures that a phase adaptation can take place between both schedules. Because p_{DL} encompasses the allowed scheduling slack, the phase adaptation is reflected in the cost function.

Figure 4.9 shows the timing definitions for the mode change scheduler using a three unit example. Tasks $\tau_{\text{old},1}$ and $\tau_{\text{old},2}$ of the old schedule Σ^{old} are marked in blue and continue into the transition schedule $\Sigma^{\text{old,new}}$. New tasks $\tau_{\text{new},1}$ and $\tau_{\text{new},2}$ of the new schedule Σ^{new} are marked in red and start within the given transition schedule. The continuing task τ_{cont} is marked in yellow and continues execution from the old schedule via the transition schedule and into the new schedule. The transitional period t_R starts at the start time of the jobs, which is the earliest of the latest jobs of all old tasks. It ends at the earliest start time of all jobs of all new tasks.

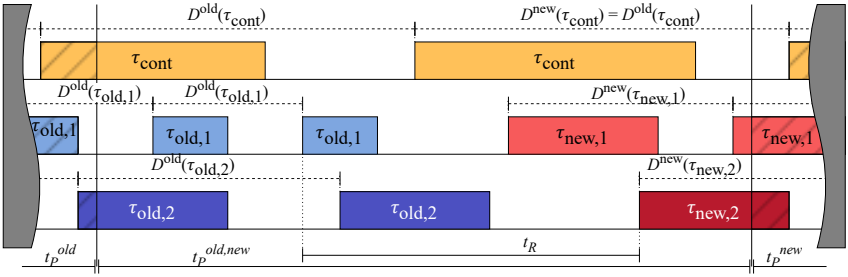


Figure 4.9: Transitional period and deadline definitions for the mode change scheduler. Old tasks are marked in blue, new tasks are marked in red and continuing tasks are marked in yellow.

Old tasks and new tasks are continued to be scheduled during the transi-

tion as long as possible to reduce the effect of the transitional period. This transitional period t_R is defined as the time between the first old task finishing the execution and the last new task starting the execution. This adds a new bonus term p_{trans} to the cost function F_{cost} of $S_{\Sigma,2}$, which is defined as:

$$p_{\text{trans}}(\vec{x}) = \min_{\tau_i \in \mathcal{T}_{\text{new}}^{\text{old,new}}} (\theta_i^{\text{old,new}}(n)) - \max_{\tau_j \in \mathcal{T}_{\text{old}}^{\text{old,new}}} (\theta_i^{\text{old,new}}(m) + C_i^{\lambda_i^{\text{old,new}}(m)}). \quad (4.78)$$

Here, $\mathcal{T}_{\text{old}}^{\text{old,new}}$ is the set of old tasks, which are still scheduled in the transition schedule $\Sigma^{\text{old,new}}$ and $\mathcal{T}_{\text{new}}^{\text{old,new}}$ is the set of new tasks, which are already scheduled in the transition schedule $\Sigma^{\text{old,new}}$. The nth job of task τ_i is given as the last scheduled job of task τ_i, where $n = \text{argmax}_k(\texttt{index}(i,k) \neq z)$, where z gives the value of not scheduled jobs. The mth job of task τ_j is given as the first scheduled job of task τ_j, where $m = \text{argmin}_k(\texttt{index}(j,k) \neq z)$.

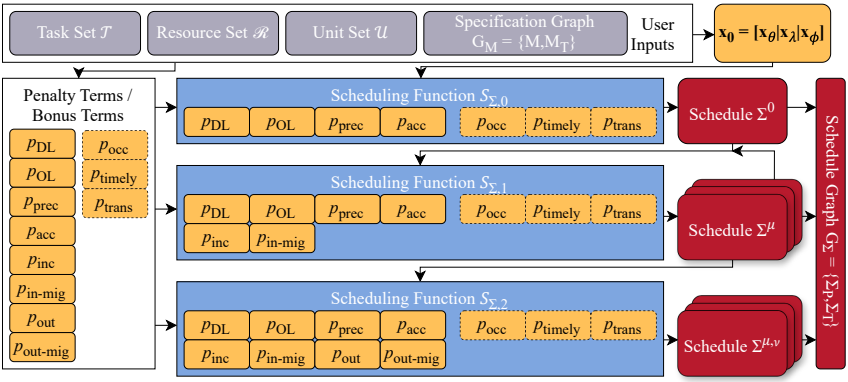


Figure 4.10: The algorithm components for scheduling mode changes between different execution modes. Algorithm components are marked with squared corners, generated data and input data is marked with boxes with rounded corners. User input is marked in gray. Generated intermediate components are marked in yellow. Schedules are marked in red. Scheduling functions are marked in blue.

This algorithm iterates over all vertexes v^μ in the specification graph G_M, and starts by generating a schedule for the execution mode m^μ, represented by the vertex v^μ, which has the least incoming edges. For this, we can use scheduling function $S_{\Sigma,0}$, as there are no constraints given by a previous schedule. The

created schedule Σ^{μ} is then added as a vertex s_{μ} to the scheduling graph G_{Σ}.

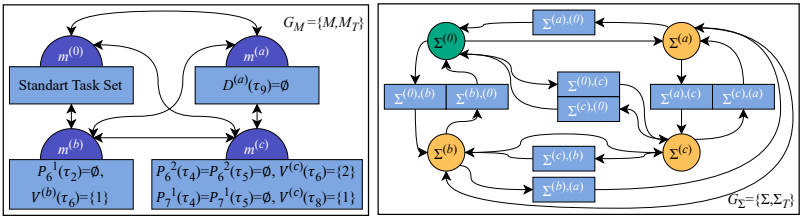
Afterwards, the algorithm iterates over all vertexes v^{μ} in the specification graph G_M, which have incoming edges from vertexes that already have a schedule in the scheduling graph G_{Σ} and the vertex with the least amount of incoming edges is selected for scheduling. Until all vertexes in G_M have been processed. Should a vertex have multiple incoming edges, one incoming edge $e^{\mu,\nu}$ is selected arbitrarily to generate the constraints for using $S_{\Sigma,1}$ for the scheduling of the respective execution mode m^{μ}.

This lead to all vertexes in G_M being processed and scheduled in G_{Σ}. Additionally, one incoming edge $e^{\mu,\nu}$ is selected for each vertex (apart from the first vertex) with incoming edges to create the constraints for scheduling the respective execution mode m^{μ}. Afterwards, a list of edges remains, which represent mode changes not yet possible in G_{Σ}, but specified in G_M. For each of these edges $e^{\mu,\nu} = (v^{\mu}, v^{\nu}) \in E_M$, the corresponding schedules Σ^{μ} and Σ^{ν} from the scheduling graph G_{Σ} are retrieved. The scheduling function $S_{\Sigma,2}$ is then used to create the transition schedule $\Sigma^{\mu,\nu}$ for the mode change from execution mode m^{μ} to execution mode m^{ν}, adhering to the constraints given by both, the incoming and the outgoing, schedules Σ^{μ} and Σ^{ν}. The created transition schedule $\Sigma^{\mu,\nu}$ is added as an edge $e^{\mu,\nu}$ to the scheduling graph G_{Σ}.

The result is a scheduling graph G_{Σ}, which contains a time triggered schedule for each execution mode defined in the specification graph G_M. Some schedules lead into other schedules without the need for a transition schedule. Additionally, transition schedules for the rest of the schedules are generates as defined in G_M.

Should all schedules be created with a cost function value of $F_{\text{cost}} \leq 0$, the scheduling graph G_{Σ} is valid and can be used for execution on the given platform. If a scheduling fails, i.e. no valid schedule is found within a given amount of iterations, the algorithm can either stop and return an error or continue with the next scheduling step. In the latter case, the failed schedule is marked as invalid in the scheduling graph G_{Σ}. Should the respective schedule be generated with $S_{\Sigma,1}$, it is regenerated with $S_{\Sigma,0}$ and a transition schedule is created. This allows generating a scheduling graph G_{Σ}, even if some schedules could not be created. This graph can then be deployed onto the given execution platform or generalized into other existing works.

Example 11: Specification and Schedule Graph for Task Set



(a) Specification Graph. (b) Schedule Graph.

Figure 4.11: Specification graph and scheduling graph for example task set. Nodes in the specification graph are marked in blue and shown the changes to the original execution mode $M^{(0)}$. Nodes in the scheduling graph are marked in green, if scheduling function $S_{\Sigma,0}$ is used, yellow, if scheduling function $S_{\Sigma,1}$ is used and blue, if scheduling function $S_{\Sigma,2}$ is used.

Figure 4.11 shows the specification graph $G_M = \{M, M_T\}$ and the scheduling graph G_Σ for the example task set. The specification graph contains the execution modes $M^{(0)}, M^{(a)}, M^{(b)}$, and $M^{(c)}$ as vertexes and a full connection as edges, i.e. $M_T = [1]^{4\times4}$. The scheduling graph contains the generated schedules $\Sigma^{(0)}, \Sigma^{(a)}$, $\Sigma^{(b)}$, and $\Sigma^{(c)}$ as vertexes. The edges for the mode changes from $\Sigma^{(0)}$ to $\Sigma^{(a)}$, from $\Sigma^{(0)}$ to $\Sigma^{(b)}$, and from $\Sigma^{(b)}$ to $\Sigma^{(c)}$ are simple. All other mode changes require a transition schedule, which are represented with a schedule specification on the given edge.

Example 11 shows the specification graph and the scheduling graph for the example task set. The specification graph contains four nodes, each representing an execution mode. The scheduling graph also contains four nodes, each representing a schedule for the respective execution mode. The edges between the schedules represent the possible mode changes. Three mode changes can be executed without a transition schedule, while all others require a transition schedule, which is represented by the schedule specification on the edge.

4.3.3 Schedule Adaptation to Reduce Schedule Complexity

As the number of transition schedules grows very large, as seen in Example 11, a strategy to reduce the number of transition schedules is to adapt existing sched-

ules to execute multiple execution mode specifications. If the only difference between two schedules Σ^μ and Σ^ν is for a deadline of a task τ_i as $D^\mu(\tau_i) \neq \emptyset$ and $D^\nu(\tau_i) = \emptyset$, the schedule Σ^μ can be adapted to include the execution of Σ^ν by two different strategies, providing a tradeoff between schedule quality and flexibility of changing between execution modes.

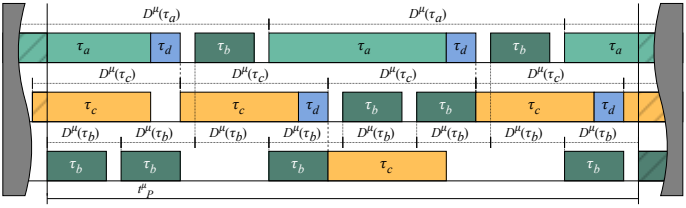
Flexible Mode-Change Schedule Adaptation

The first strategy is achieved by simple removing all jobs of task τ_i from the schedule. With this, every task, which is required (directly or indirectly) by a precedence constraint of τ_i also needs to be removed. This flexible adaptation allows for a switch between both (or more) schedules at arbitrary points during the scheduling period.

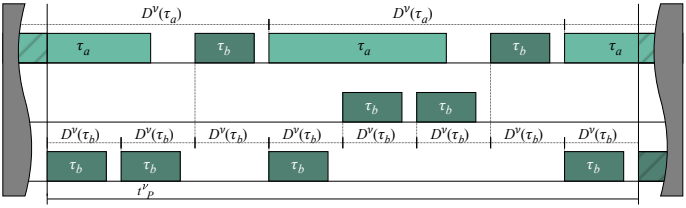
One schedule can have multiple adapted versions, depending on the number of tasks with changed deadlines. For example, if two tasks τ_i and τ_j have a deadline in Σ^μ but not in Σ^ν, there are three adapted versions of Σ^μ for Σ^ν: one with only the jobs of τ_i removed, one with only the jobs of τ_j removed and one with the jobs of both tasks removed.

Since the rest of the schedule stays the same, a switch between two schedules, where one schedule is an adapted version of the other, can be executed during the schedule period and without the need for a transition schedule. This adaptation is included in the scheduling graph G_Σ with schedule Σ^μ retaining all specified edges and the adapted schedule Σ^ν only having edges from and to Σ^ν, should these edge exist in the specification graph G_M. Should only one edge between Σ^μ and Σ^ν exist in G_M, then the transition schedules from Σ^ν to other schedules are created, but only transition schedules towards Σ^ν, which are not also available via Σ^μ, are created. This allows for a reduction of the number of transition schedules, while still allowing for all mode changes specified in G_M to be executed.

Figure 4.12 shows the adaptation of schedules for multiple execution modes using the flexible adaptation. Four tasks, $\{\tau_a, \tau_b, \tau_c, \tau_d\}$ are present in the first schedule, from which $\{\tau_a, \tau_b\}$ are also present in the second schedule. Task τ_c requires the execution of τ_d. In the second schedule, τ_c has no assigned deadline and thus, both τ_c and τ_d are removed from the schedule.



(a) Unchanged Schedule.



(b) Flexible Adapted Schedule.

Figure 4.12: Tasks in both modes are marked green, tasks only in the first mode are yellow and tasks scheduled via precedence constraints in the first mode but not in the second mode are marked in blue.

Optimized Mode-Change Schedule Adaptation

The second option represents an optimized adaptation. Here, again, to generate the adapted schedule Σ^{ν} from a given schedule Σ^{μ}, all tasks τ_j with $D^{\nu}(\tau_j) = \emptyset$ and precedence constraints originating from τ_j are removed from the based schedule Σ^{μ}. Additionally, the first starting times $\theta_i^{\mu}(0)$ and last starting times $\theta_i^{\mu}(N_{\theta,\mu}(\tau_i))$ for all tasks with $D^{\mu}(\tau_i) \neq \emptyset$ are fixed. Starting times of jobs which are required by precedence constraints are not marked as fixed.

Afterwards, the scheduling function $S_{\Sigma,1}$ is called again with the configured solution vector as well as penalty and bonus terms. This allows the scheduler to optimize the schedule by potentially choosing different task versions or moving the starting times slightly to achieve a better timeliness. Additionally, bonus terms to occupy less execution units might yield a better result for the schedule. Since the starting times of the tasks are not fixed, a constraint in the possibility to switch between execution modes needs to be introduced. A switch between execution modes is only possible at the time steps $t \in [0, t_P)$, if

$$t - \theta_i^{\mu}(n) = t - \theta_i^{\nu}(n) \wedge \lambda_i^{\mu}(n) = \lambda_i^{\nu}(n) \wedge \phi_i^{\mu}(n) = \phi_i^{\nu}(n) \tag{4.79}$$

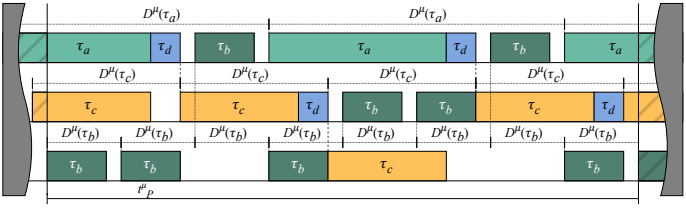
holds true for all execution units u and jobs n of the corresponding tasks τ_i, which is executed on the given unit $u = \phi_i^{\mu}(n)$. Additionally, the chosen version needs to be identical between both schedules. Because one differing value would require a migration between versions or execution units or would lead in a mismatch of assigned time in the schedule and given execution time, a change in these values is not supported. Additionally, the amount of executed jobs of each task must be identical before and after the given time step t. Otherwise, one job of a task might be skipped or executed multiple times.

Both schedules are then interchangeable, should these constraints be fulfilled. A switch between them is allowed during the hyperperiod and the secondary schedule Σ^{ν} is then only connected to the node of schedule Σ^{μ} in the scheduling graph. As above, the schedule G_{Σ} retains all specified edges and the adapted schedule Σ^{ν} only has the edges from and to Σ^{ν}, should these edges exist in the specification graph G_M. Should only one edge between Σ^{μ} and Σ^{ν} exist in G_M, then the transition schedules from Σ^{ν} to other schedules are created, but only transition schedules towards Σ^{ν}, which are not also available via Σ^{μ}, are created.

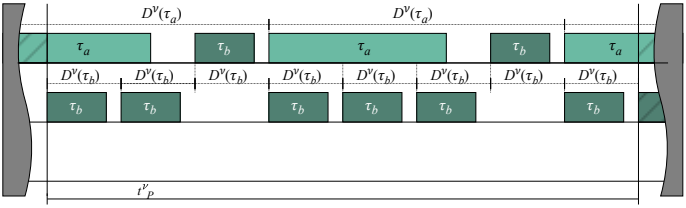
Figure 4.13 shows the adaptation of schedules for multiple execution modes using the optimized adaptation. Four tasks, $\{\tau_a, \tau_b, \tau_c, \tau_d\}$ are present in the first schedule, from which $\{\tau_a, \tau_b\}$ are also present in the second schedule. Task τ_c requires the execution of τ_d. In the second schedule, τ_c has no assigned deadline and thus, both τ_c and τ_d are removed from the schedule. The optimization using the scheduling function $S_{\Sigma,1}$ allows for moving the jobs of task τ_b such that one execution unit stays unoccupied.

Input into the Scheduling Functions

Both steps are included into the scheduling functions at the same step. Schedules Σ^{ν}, which can be marked as adapted from another schedule Σ^{μ}, are marked as such, if the only difference is $D^{\mu}(\tau_i) \neq \emptyset$ and $D^{\nu}(\tau_i) = \emptyset$ for given tasks τ_i. For each combination, an adapted schedule will be created. First, each corresponding execution mode $m^{\nu} \in M$ is collected in a separate set $M^{\mu} \subset M$, representing all modes which will be associated with the schedule Σ^{μ}. This is repeated for each schedule. Afterwards, the creation of the schedule graph G_{Σ}



(a) Unchanged Schedule.



(b) Optimized Adapted Schedule.

Figure 4.13: Tasks in both modes are marked green, tasks only in the first mode are yellow and tasks scheduled via precedence constraints in the first mode but not in the second mode are marked in blue.

out of the specification graph is undergone with $G_M = \{M \setminus \{M^{\mu}, ...\}, M_T\}$ for all schedules, which are represented via other schedules, which are collected in $\{M^{\mu}, ...\}$.

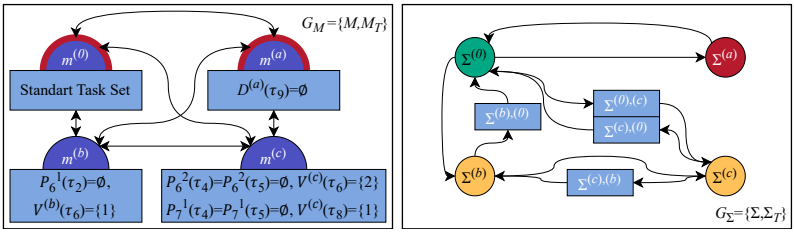
After the creation of this preliminary schedule graph, each adapted schedule Σ^{ν} is created using the corresponding configuration $m^{\nu} \in M^{\mu}$ using scheduling function $S_{\Sigma,1}$. Because the start and end times of these schedules are the same, a mode change can be undergone at the end of each hyperperiod using a transition period of $t_R = 0$. Transitions towards a schedule Σ_x, given as $m_{\nu,x} \in M_T$, which are not available as $m_{\mu,x}$ are generated using scheduling function $S_{\Sigma,2}$. The same holds true for schedules towards Σ^{ν}. All other transitions are available as the same as $\Sigma^{\mu,x}$ and $\Sigma^{x,\mu}$ for both schedules with the difference of removing tasks τ_i from the transition schedules $\Sigma^{x,\nu}$ and $\Sigma^{\nu,x}$ for which $D^{\nu}(\tau_i) = \varnothing$.

Contribution 9: Changing Execution Modes during Hyperperiod

The presented method allows for a change of the schedule during the hyperperiod, as long as given constraints are fulfilled. This allows for a reduction of the number of transition schedules and a simpler system description.

Example 12 shows the adaption of schedules for the example task set. The specification graph is the same as in Example 11, but the scheduling graph contains adapted schedules. The schedule $\Sigma^{(a)}$ is an adapted version of $\Sigma^{(0)}$ for the execution mode $M^{(a)}$. This allows the number of transition schedules to be reduced from nine to four, since all transitions to $\Sigma^{(a)}$ are represented by transitions to $\Sigma^{(0)}$.

Example 12: Specification and Adapted Schedule Graph for Task Set



(a) Specification Graph. (b) Schedule Graph.

Figure 4.14: Specification graph and scheduling graph for the example task set. Nodes in the specification graph are marked in blue and shown the changes to the original execution mode $M^{(0)}$. Nodes in the scheduling graph are marked in green, if scheduling function $S_{\Sigma,0}$ is used, yellow, if scheduling function $S_{\Sigma,1}$ is used and blue, if scheduling function $S_{\Sigma,2}$ is used. Red nodes represent adapted schedules.

Figure 4.14 shows the specification graph $G_M = \{M, M_T\}$ and the scheduling graph G_Σ using the method to adapt schedules. In comparison to Example 11, the scheduling graph contains only four transition schedules. The schedule $\Sigma^{(a)}$ is an adapted version of $\Sigma^{(0)}$ for the execution mode $M^{(a)}$. This allows the number of transition schedules to be reduced from nine to four, since all transitions to $\Sigma^{(a)}$ are represented transitions to $\Sigma^{(0)}$.

4.4 Anomaly Detection Components to Trigger Mode Changes

Deciding on the change in execution mode can either be triggered via a user input or automatically to account for detected anomalies. To support this, a model-based approach to determine necessary monitoring components is used. This approach encompasses the creation of anomaly detection code

inside software modules (adapted from [Sta23b]), as well as the creation of hardware monitoring modules for the heterogeneous systems-on-chip (HESoCs) platform (adapted from [Sta23d]). These modules both target the monitoring of two similar but distinct system layers. While [Sta23b] is used to generate monitors inside given software tasks, which can be seen as white boxes, [Sta23d] targets the hardware deployment of the communication of these tasks and assumes black boxes. The latter also generates hardware descriptions of the generated monitor components and includes the possibility to describe external tasks running on other ECUs and DCUs inside the same automotive vehicle system. Section 4.4.1 describes the adaptation strategy for hardware components and Section 4.4.2 describes the adaptation strategy for software component. Both are necessary to be deployed inside the given environment for automatically triggering mode changes.

4.4.1 Hardware-based Anomaly Detection Components

The hardware components to monitor communication systems are generated using a three tiered model to abstract the underlying hardware platform. This allows for the generation of the hardware monitors to be done automatically, after the time-triggered scheduling (TTS) has been undergone. The used model is adapted from [Sta23d] and consists of the following parts.

Model Description

The first part describes the underlying hardware platform and is called *platform architecture*. Here, a description of the used HESoC platform is given. This includes the execution units $\mathcal{U}$, adapted from Section 4.1. Additionally, a description of the communication links is defined with the set

$$\mathcal{L} = \{l_i \mid [0, N_{\mathcal{L}})\}, \tag{4.80}$$

where $N_{\mathcal{L}}$ is the number of communication links. The function

$$\varphi_{\mathcal{L}} : \mathcal{L} \to \mathcal{P}(\mathcal{U}), \tag{4.81}$$

with $\mathcal{P}(\mathcal{U})$ being the power set of $\mathcal{U}$, describes which execution units are connected to which communication links. Lastly, the function

$$\varphi_{\Gamma_R} : \mathcal{L} \rightarrow \{0,1\} \tag{4.82}$$

describes the reachability of a communication link. If a communication link l_i allows for the addition of monitoring components, $\varphi_{\Gamma_R}(l_i)$ returns 1, otherwise it returns 0.

This architecture description encompasses the hardware platform and the monitoring capability. Table 4.16 summarizes this first model part.

Table 4.16: Components and descriptions of the platform architecture.

Component	Formula	Description
Execution Units	$\mathcal{U} = \{u_i \mid [0, N_\mathcal{U})\}$	Set of $N_\mathcal{U}$ execution units.
Comm. Links	$\mathcal{L} = \{l_i \mid [0, N_\mathcal{L})\}$	Set of $N_\mathcal{L}$ communication links.
System Elements	$\mathcal{U} \cup \mathcal{L}$	Union of units and communication links.
Link Connections	$\varphi_\mathcal{L} : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{U})$	Mapping of communication links to connected execution units.
Reachability	$\varphi_{\Gamma_R} : \mathcal{L} \rightarrow \{0,1\}$	Describing if a communication link allows for monitoring components.

The second part of the model describes the *functional implementation*. Here, an abstract representation of the communication and information flow between the software tasks are given. The set of tasks $\mathcal{T}$ is included from Section 4.1[2]. Two sets of information flows are defined. The first set describes unintended information flow and is defined as:

$$I_S \subseteq \mathcal{T} \times \mathcal{T}, \tag{4.83}$$

and specifies information flow between two tasks which is not intended by the system designer. An unintended information flow is added to I_S if a task $\tau_i \in \mathcal{T}$ might be able to send data to another task $\tau_j \in \mathcal{T}$ without this being intended by the system designer. Should that happen, an anomaly detection component needs to alert the system of this unintended communication. While the system itself is designed with a whitelist-based approach, defining these

[2]These are called features in [Sta23d].

unintended information flows allows for a specified filtering of the detection of anomalies.

The second set describes intended information flow and is defined as:

$$I_N \subseteq \mathcal{T} \times \mathcal{T}. \tag{4.84}$$

I_N specifies information flow between two tasks which is intended by the system design, $I_S \cup I_N = \emptyset$. An intended information flow is added to I_N if a task $\tau_i \in \mathcal{T}$ is allowed to send data to another task $\tau_j \in \mathcal{T}$. Looking back to Section 4.1, this is not to be confused with precedence constraints, which define temporal dependencies between tasks. Communication between tasks might occur without any temporal dependencies e.g. via shared memory space.

Since not all information flow is suspicious or normal under all circumstances, a set of rules Γ_R is defined to describe under which conditions information flow should be considered for these sets. These rules can be temporal (e.g. only during certain time windows), or based on other task communication. These rules are mapped to I_S and I_N via the function

$$\varphi_{\Gamma_R} : \Gamma_R \to \mathcal{P}(I_S \cup I_N). \tag{4.85}$$

Additionally, the function $\varphi_i : \mathcal{T} \to \mathcal{U}$ maps each job of a task to the scheduled execution unit, adapted from Section 4.1. An overview of the components of the *functional implementation* is given in Table 4.17.

Lastly, the *information flow* part of the model describes concrete information flow and how aggregate system behavior and states are monitored. For this, the set of system properties $\mathcal{S}$ is defined as:

$$\mathcal{S} = \{s_i \mid [0, N_{\mathcal{S}})\}, \tag{4.86}$$

where each property s_i describes a measurable system property, e.g. the current vehicle speed or the current temperature. Additionally, the set of conditions Γ_C is defined as:

$$\Gamma_C = \{c_i \mid [0, N_{\Gamma_C})\}, \tag{4.87}$$

Table 4.17: Components and descriptions of the functional implementation.

Component	Formula	Description
Software Tasks	$\mathcal{T} = \{\tau_i \mid [0, N_\mathcal{T})\}$	Software tasks[a].
Unintended Information Flow	$I_S \subseteq \mathcal{T} \times \mathcal{T}$	Set of unintended (suspicious) information flow between tasks.
Intended Information Flow	$I_N \subseteq \mathcal{T} \times \mathcal{T}$	Set of intended (normal) information flow between tasks.
Rules	$\Gamma_R = \{r_i \mid [0, N_{\Gamma_R})\}$	Rules describing information flow variability.
Rule Mapping	$\varphi_{\Gamma_R} : \Gamma_R \to \mathcal{P}(I_S \cup I_N)$	Rule assignment to information flow.
Unit Mapping	$\phi_i : \mathbb{N} \to \mathcal{U}$	Unit assignment of task τ_i, job n[a].

[a]See Section 4.1 for detailed explanation.

where each condition c_i describes the circumstances, under which a property is marked as suspicious. The function

$$\varphi_{\Gamma_C} : \Gamma_C \to \mathcal{P}(\mathcal{S}) \tag{4.88}$$

maps conditions to one or more properties. The function

$$\varphi_V : \mathcal{S} \to \mathcal{P}(\mathcal{T}) \tag{4.89}$$

assigns each property to one or more tasks, which produce the property data. Properties $\mathcal{S}$ can be view analogous to suspicious or normal information flow specifications I_S and I_N, while conditions Γ_C can be viewed analogous to rules Γ_R. A summary of the *information flow* components is given in Table 4.18.

Table 4.18: Components and descriptions of the information flow.

Component	Formula	Description
System Properties	$\mathcal{S} = \{s_i \mid [0, N_\mathcal{S})\}$	Measurable system properties.
Conditions	$\Gamma_C = \{c_i \mid [0, N_{\Gamma_C})\}$	Set of conditions under which properties are suspicious.
Condition Mapping	$\varphi_{\Gamma_C} : \Gamma_C \to \mathcal{P}(\mathcal{S})$	Condition to properties mapping.
Property Mapping	$\varphi_V : \mathcal{S} \to \mathcal{P}(\mathcal{T})$	Properties to tasks mapping.

The generation of the hardware monitoring components can commence, when the following requirements are met. First, at least one link is marked

as reachable, i.e. $\exists\ l_i \in \mathcal{L} : \varphi_{\Gamma_R}(l_i) = 1$. Second, suspicious information flows and normal information flows are disjoint, i.e. $I_S \cap I_N = \emptyset$. Third, every unit is connected to every other unit by a unique series of links.

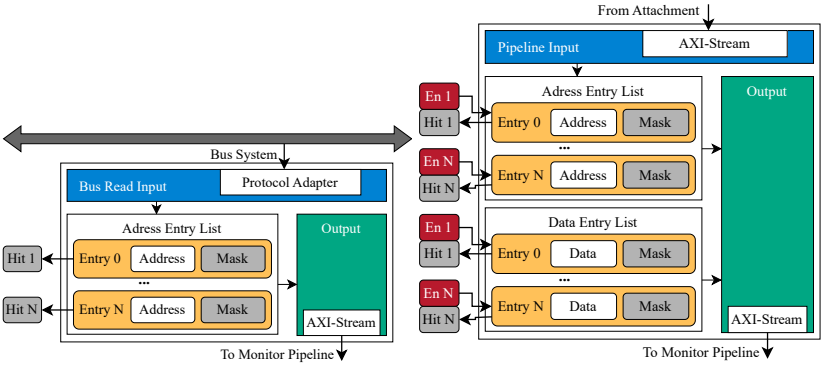
Hardware Monitor Library

The monitoring is facilitated by a library of (hardware) components, which can be instantiated and connected to the reachable links, should such possibilities exist. In embedded systems, these components can either be instantiated inside FPGAs or implemented as software components, which monitor the corresponding memory addresses. Inside automotive vehicles, a connection to controller area network (CAN) buses or to a time-sensitive networking (TSN) interface might be needed, should only these links be reachable. The hardware library is split into three different categories of components. All components are connected via an AXI-stream interface and form a monitor pipeline.

> ***Contribution 10: Monitoring Pipeline***
>
> *The monitoring pipeline is defined as the series of monitoring components connected via an AXI stream interface. Hardware descriptions are automatically generated for the components based on a given platform architecture and functional implementation. The monitoring pipeline consists of the following components: 1. Link Attachment, 2. Data Filter, 3. Evaluation Processor.*

The first component is a *link attachment*. This component is instantiated for each reachable link l_i with $\varphi_{\Gamma_R}(l_i) = 1$. It connects to the link and reads in the transmitted data. It records the sent data as well as the source address and sink address. The *link attachment* contains a list of address entries and address masks, which can be used to filter the recorded data. Each entry additionally contains a component output signaling a hit flag for the corresponding entry. This hit flag is set, when data matching the address entry and mask is recorded. The recorded data and the source and sink addresses are forwarded inside of the monitor pipeline, if they match on of the masked address entries. Figure 4.15a shows the structure of the *link attachment* component. The hit flag is marked with arrow and the AXI-based output interface is marked accordingly. The attachment to the communication link is marked as well and facilitates the reading of data from the link. It needs to be implemented accordingly for each link type, e.g. CAN or TSN inside automotive vehicles or AXI inside HESoC.

The second hardware component is the *data filter*. This component is instantiated once after the attachment and has a similar structure. Instead of the adaptation to a communication link, it contains an input interface from the AXI stream of the *link attachment*. Similar to the *link attachment*, it contains a list of address entries and masks, which are used to filter the incoming data. The filter additionally contains a list of data entries and masks, which are used to detect and filter the incoming monitoring data and variables for rules Γ_R and conditions Γ_C. Each address and data entry has an enable flag as input and a hit flag as an output. Every rule and condition is represented by one address entry or one data entry respectively. The enable flag activates these filters according to events from the attachments or permanently by constantly setting it to 1. As for the *link attachment*, the recorded data as well as the source and sink addresses are forwarded inside the monitor pipeline. If an entry is matched, the corresponding hit flag is set.



(a) Link Attachment Component (b) Data Filter Component

Figure 4.15: The structure of the attachment and filter components. Read in components are marked in blue, entries in lists are marked in yellow and output components are marked green. Enable flags, which influence the usage of the list entries are marked in red and hit flags are marked in gray.

Figure 4.15b shows the structure of the *data filter* component. The hit flags are marked with arrows and the AXI-based input and output interfaces are marked accordingly. The filtering of data and addresses is shown as well.

The third component is realized inside of a MicroBlaze [125] or MicroBlaze-V softcore processor [126]. Other architectures might be possible but were not considered during implementation. This CPU implements the *evaluation*

processor. It reads in the forwarded data from the *data filter* and evaluates the hit flags for rules Γ_R and conditions Γ_C. Inside this processor, the monitored data is evaluated and can be saved to a local memory. Inside this processor, a potential switch of the TTS mode can be triggered, should an anomaly be detected.

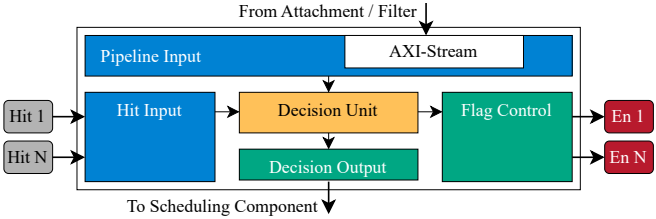


Figure 4.16: The structure of the monitor processor. Read in components are marked in blue, decision components are marked in yellow. Output components are marked in green.

Figure 4.16 shows the structure of the evaluation processor and the used interfaces. The softcore processor is connected via an AXI stream interface to the *data filter* and reads in the forwarded data. Additionally, it is connected via an AXI memory-mapped interface to a local memory, where monitored data can be stored for evaluation. The hit flags are able to be read in via a distinct interface and evaluated. Because there exist various AXI stream components to merge, forward and buffer AXI components inside the Vivado design suite [127], the connection of the monitoring pipeline is facilitated using these preexisting components. This leads to only one necessary MicroBlaze processor at the end of every pipeline.

While these components might be configured by hand for a specific use-case, the automatic generation and configuration of these components is necessary for complex systems and facilitated via the model described above. In all cases, the deployed software in the MicroBlaze is designed by the user and not included in the automatic generation.

Automatic Generation of Anomaly Detection Components

The generation of the hardware monitoring components is done itself in a three-step process, shown in Figure 4.17 and first introduced in [Sta23d]. First,

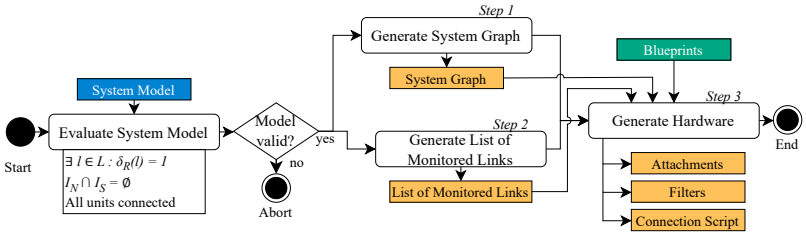


Figure 4.17: Generation process for hardware monitoring components. User provided inputs for any given platform are marked in blue, persistent inputs are marked in green and tool created artifacts are marked in yellow.

a system graph is generated based on the platform architecture and functional implementation. This graph is called the *system graph* $G_S = \{V_S, E\}$. The vertices V_S are defined with

$$V_S = \mathcal{U} \cup \mathcal{L} \tag{4.90}$$

and edges E are connected between the vertices, if there exist a connection between the corresponding execution units and communication links according to $\varphi_{\mathcal{L}}$, i.e.

$$E = \{(u_i, l_j) \mid u_i \in \varphi_{\mathcal{L}}(l_j)\}. \tag{4.91}$$

This graph needs to be acyclic for the generation to continue. Using this graph G_S, a set of paths, each denoting the connection of one job to another job is generated. This set of paths is defined as $P = \{p^{i,j}_{m,n} \mid [0, N_P)\}$ where the path $p^{i,j}_{m,n}$ gives the connection from the mth job of task τ_i to the nth job of task τ_j. The path contains the system elements in an alternating order of execution units and communication links, from the execution unit of job n of τ_i to the execution unit of job m of τ_j, i.e. it is defined as

$$p^{i,j}_{m,n} = \{u_0, l_0, \ldots, l_{N-1}, u_N\}, \tag{4.92}$$

where $u_0 = \varphi_m(\tau_i)$ and $u_N = \varphi_n(\tau_j)$ and $(\tau_i, \tau_j) \in I_S$. The path is generated by traversing the graph G_S starting at u_0 and ending at u_N. The path is unique for every pair of jobs due to the acyclic nature of the graph. The second step involves marking links for the placement of attachments. Every first and last reachable link of a path $p^{i,j}_{m,n}$ is marked for the placement of a *link attachment*. These marked links are collected in

$$L_M = \{l_i \in \mathcal{L} \mid \varphi_{\Gamma_R}(l_i) = 1, \\ \wedge (l_i = \min_{l_k \in \mathcal{L}}(l_k \in p^{i,j}_{m,n}) \vee l_i = \max_{l_k \in \mathcal{L}}(l_k \in p^{i,j}_{m,n}))\}. \tag{4.93}$$

For every system property which is not transmitted via an already marked linked, an arbitrary reachable link is added to the set L_V for the placement of an attachment, which monitors the corresponding variable. This link must lie on a path transmitting the variable data, resulting in

$$L_V = \{l_i \in \mathcal{L} \mid \varphi_{\Gamma_R}(l_i) = 1, \\ \wedge \exists v_k \in \mathcal{S}, \exists p^{i,j}_{m,n} \in P : (\{\tau_i, \tau_j\}) \in \varphi_V(v_k), l_i \in p^{i,j}_{m,n}\}. \tag{4.94}$$

Lastly, the same is repeated for rules Γ_R and conditions Γ_C as

$$L_R = \{l_i \in \mathcal{L} \mid \varphi_{\Gamma_R}(l_i) = 1, \\ \wedge \exists r_k \in \Gamma_R, \exists p^{i,j}_{m,n} \in P : (\{\tau_i, \tau_j\}) \in \varphi_{\Gamma_R}(r_k), l_i \in p^{i,j}_{m,n}\} \tag{4.95}$$

and

$$L_C = \{l_i \in \mathcal{L} \mid \varphi_{\Gamma_C}(l_i) = 1, \\ \wedge \exists c_k \in \Gamma_C, \exists p^{i,j}_{m,n} \in P : (\{\tau_i, \tau_j\}) \in \varphi_{\Gamma_C}(c_k), l_i \in p^{i,j}_{m,n}\}. \tag{4.96}$$

These links, i.e. $L_A = L_M \cup L_V \cup L_R \cup L_C$, are marked for the placement of attachments. At these places one *link attachment* will be instantiated and connected to the corresponding link.

The *link attachments*, *data filters* and *evaluation processors* are generated and the *monitoring pipeline* is formed. For every marked link $l_i \in L_A$, a *link attachment* is instantiated and connected to the corresponding link. Additionally, an address list with the addresses of the source and sink units lying on each path, which includes the given link is built. For every system property $v_k \in \mathcal{S}$, marked as suspicious by a condition $c_k \in \Gamma_C$, an address entry is added to the attachment of a link in L_V monitoring the variable. For the suspicious information flows I_S, address entries inside the filters are created. Data entries are created for conditions, which specify system properties. Lastly, the hit flags and active flags are connected such that each rule and condition activates the corresponding filters and the hit flags are forwarded to the evaluation processor. The evaluation processor is configured to evaluate the hit flags according to the rules Γ_R and conditions Γ_C, which are mapped to the

path via φ_{Γ_R} and φ_{Γ_C}. Using existing AXI stream components, the monitoring pipeline is formed by connecting the attachments, filters and evaluation processor accordingly. How the monitored information is handled is determined by the system designer via the software deployed onto the *evaluation processor*.

Example 13: Monitoring Pipeline

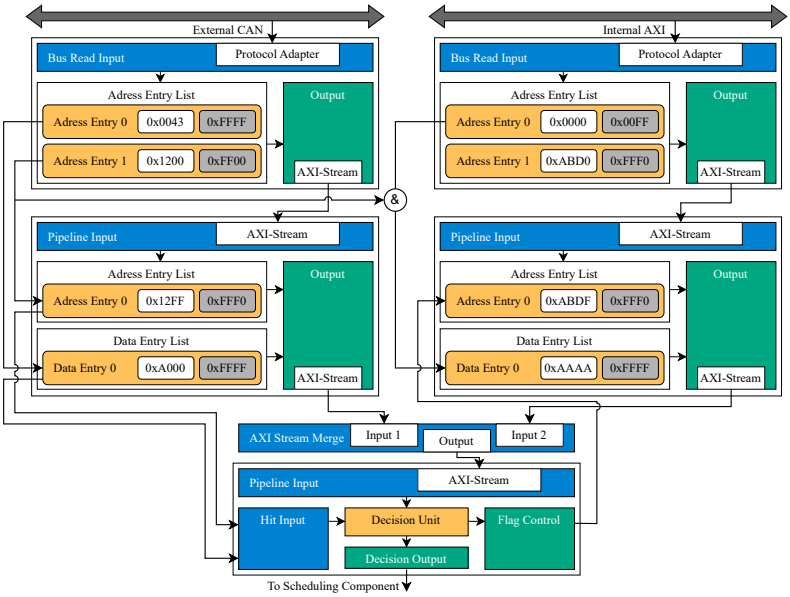


Figure 4.18: Example of a monitoring pipeline with two attachments and filters and one MicroBlaze processor.

This example shows the monitoring pipeline for a system with two reachable links. One link is an internal AXI link and the second one is an external CAN bus. The corresponding attachments read in the data from both links and issue a hit flag, should the given data or address match the masked input. This flag can then be either used to activate entries in filter components or forwarded to the evaluation processor. Similarly, enable flags can be set by the processor, an entry in an attachment or a combination of those.

Example 13 gives an example of a monitoring pipeline for a system with two reachable links, which use AXI and CAN respectively and are connected to the monitoring pipeline by one attachment each.

4.4.2 Software Components for Anomaly Detection

In contrast to the generation of hardware monitors from Section 4.4.1, this section describes the adaptation of software components to account for different safety guarantees and is based on [Sta23b]. While this model is similar to the previously presented model, the generation process adapts software components to be able to trigger exceptions or further reactions, like a mode change between TTS modes.

Model Description

The model used for this generation is two tiered. In the first tier, called *component interface*, a set of inputs and outputs are defined. The inputs are defined as:

$$\mathcal{S}_{\text{in}} = \{s_{\text{in},k} \mid k \in [0, N_I)\}, \tag{4.97}$$

where each input $s_{\text{in},k}$ describes input data and input variables into a software component, like vehicle speed, steering angle or temperature. The outputs are defined similarly via

$$\mathcal{S}_{\text{out}} = \{s_{\text{out},k} \mid k \in [0, N_O)\}, \tag{4.98}$$

where each output $s_{\text{out},k}$ describes output data and output variables of a software component. The notation orients itself on Section 4.4.1, since S_{in} and S_{out} are similar in concept to the system variables $\mathcal{S}$. S_{out} is defined analogously to S_{in}. These could be the same as the inputs, but could also be derived data, e.g. a variable describing if the current vehicle speed is above a certain threshold or control variables like desired audio volume, windshield wiper speed or seating position.

Each input is given a trust guarantee, defined by the function

$$g_{\text{give}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{fix}} \to \mathbb{N}, \tag{4.99}$$

which describes the trust guarantee given by that input data or fixing variable, which will be explained later. This represents a quantitative measure of how much the system trusts the given input data, based on the providing source. Each output is given a safety requirement, defined by the function

$$r_{\text{give}} : \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{fix}} \to \mathbb{N}, \tag{4.100}$$

which describes the minimum safety requirement necessary for the output data. This represents a quantitative measure of how much safety is required for the given output data, based on the potential consequences of a failure of that output data.

The last element of the first tier is given by the intended flow of information inside the component,

$$F \subseteq \mathcal{S}_{\text{in}} \times \mathcal{S}_{\text{out}}, \tag{4.101}$$

modelling what inputs $\mathcal{S}_{\text{in}}$ are processed inside the software component to influence the outputs $\mathcal{S}_{\text{out}}$. This flow of information is fed into the model during system design. Table 4.19 gives an overview of the first model tier.

Table 4.19: Components, symbols and descriptions of the component interface.

Component	Formula	Description
Inputs	$\mathcal{S}_{\text{in}} = \{s_{\text{in},k} \mid k \in [0, N_I)\}$	Set of N_I input data and variables.
Outputs	$\mathcal{S}_{\text{out}} = \{s_{\text{out},k} \mid k \in [0, N_O)\}$	Set of N_O output data and variables.
Trust Guarantee	$g_{\text{give}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{fix}} \to \mathbb{N}$	Function mapping inputs and fixing variables to trust guarantees.
Safety Requirement	$r_{\text{give}} : \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{fix}} \to \mathbb{N}$	Function mapping outputs and fixing variables to safety requirements.
Information Flow	$F \subseteq \mathcal{S}_{\text{in}} \times \mathcal{S}_{\text{out}}$	Set of intended information flow between inputs and outputs.

The second tier of the model describes the *internal structure* of the software component. First, a set of internal inferred variables is described as

$$\mathcal{S}_{\text{inf}} = \{s_{\text{inf},k} \mid k \in [0, N_F)\}, \tag{4.102}$$

where each inferred variable $s_{\text{inf},k}$ describes an internal part of the software component, which is not directly given by the inputs but can be inferred from the given software code. This could be a component describing a routing algorithm or an object classification algorithm. Additionally, a set of fixing variables is defined as

$$\mathcal{S}_{\text{fix}} = \{s_{\text{fix},k} \mid k \in [0, N_X)\}, \tag{4.103}$$

where each fixing variable $s_{\text{fix},k}$ describes an internal part of the software component, which is not directly given the software code, but is added via code generation to fix model violations. This could be a component describing a plausibility check or a redundancy mechanism.

Out of these variables and the given inputs and outputs, a set of inferred transactions are defined as:

$$X \subseteq (\mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{inf}} \cup \mathcal{S}_{\text{fix}}) \times (\mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \cup \mathcal{S}_{\text{fix}}), \tag{4.104}$$

where each transaction describes the flow of information between inputs, outputs and inferred variables. This flow of information is inferred from the software code and describes how software components interact. With the use of these transactions, the trust guarantees and requirements are propagated through the system. The inferred guarantees are defined as

$$g_{\text{inf}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \to \mathbb{N}, \tag{4.105}$$

and the inferred requirements are defined as

$$r_{\text{inf}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \to \mathbb{N}. \tag{4.106}$$

With these, a set of trust violations can be collected in $V \subseteq X$. Table 4.20 gives an overview of the second model tier.

Instance Generation

One model instance is generated from a given software system by a process, which encompasses three steps. These are called *transaction analysis*, *forward pass* and *backward pass*. First, the *transaction analysis* extracts variables $\mathcal{S}_{\text{inf}}$, inputs $\mathcal{S}_{\text{in}}$ and outputs $\mathcal{S}_{\text{out}}$, and inferred transactions X from a given source code. Input variables into the system or read data from provided interfaces are mapped to the input set $\mathcal{S}_{\text{in}}$. Since these are usually represented by variables in the source code, one input variable $s_{\text{in},k} \in \mathcal{S}_{\text{in}}$ is created for each variable. Output variables or data sent to provided interfaces are mapped to the output set $\mathcal{S}_{\text{out}}$. These might be function return variables or variables which are sent to an output interface. Similar to this, variables inside the source code are mapped to the set $\mathcal{S}_{\text{inf}}$.

Table 4.20: Components, symbols and descriptions of the internal structure.

Component	Formula	Description
Inferred Features	$\mathcal{S}_{\text{inf}} = \{s_{\text{inf},k} \mid k \in [0, N_{\mathcal{F}})\}$	Set of internal inferred variables.
Fixing Features	$\mathcal{S}_{\text{fix}} = \{s_{\text{fix},k} \mid k \in [0, N_{\mathcal{F}})\}$	Set of added fixing variables.
Inferred Transactions	$X \subseteq (\mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{inf}} \cup \mathcal{S}_{\text{fix}}) \times (\mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \cup \mathcal{S}_{\text{fix}})$	Set of inferred transactions between inputs, outputs and inferred variables.
Inferred Guarantees	$g_{\text{inf}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \to \mathbb{N}$	Function mapping inputs, outputs and inferred variables to inferred trust guarantees.
Inferred Requirements	$r_{\text{inf}} : \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \to \mathbb{N}$	Function mapping inputs, outputs and inferred variables to inferred safety requirements.
Trust Violations	$V \subseteq X$	Set of transactions where inferred guarantees do not meet inferred requirements.

The flow of information between these variables is mapped to the set of transactions X. If one variable is used to calculate another variable, a transaction is created between the corresponding variables. Afterwards, existing variables might be able to be collapsed into a single variable, if every chain of transactions from a starting variable $s_{\text{inf},1} \in \mathcal{S}_{\text{inf}}$ ends in a single ending variable $s_{\text{inf},n} \in \mathcal{S}_{\text{inf}}$ and every transaction in the chain only leads to variables between $s_{\text{inf},1}$ and $s_{\text{inf},n}$, i.e.

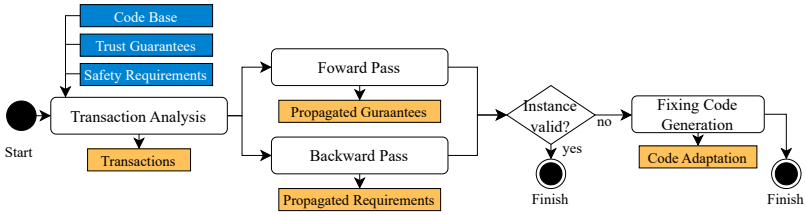


Figure 4.19: Generation process for software anomaly detection components. User provided inputs are marked in blue, tool created artifacts are marked in yellow.

$$\begin{aligned}\forall s_{\text{inf},1},\dots,s_{\text{inf},n} \in \mathcal{S}_{\text{inf}}, s_{\text{inf},1} \neq s_{\text{inf},n}\ \exists\, S_{\text{chain}} = (s_{\text{inf},1}, s_{\text{inf},2},\dots,s_{\text{inf},n}) \\ \text{s.t. } (s_{\text{inf},1}, s_{\text{inf},2}),\dots,(s_{\text{inf},n-1}, s_{\text{inf},n}) \in X \\ \wedge\, \nexists s_{\text{inf},k} \in \mathcal{S}_{\text{chain}}, s_{\text{inf},m} \in \mathcal{S}_{\text{inf}} \setminus S_{\text{chain}} \\ \text{s.t. } (s_{\text{inf},k}, s_{\text{inf},m}) \in X \vee (s_{\text{inf},m}, s_{\text{inf},k}) \in X\end{aligned} \tag{4.107}$$

With this, the internal structure of the software component is extracted and can be used for the next steps. The resulting reduced set of inferred variables is then used in the subsequent steps.

Second, the *forward pass* propagates the trust guarantees through the system. For every input $s_{\text{in},k} \in \mathcal{S}_{\text{in}}$ an out tree $T(s_{\text{in},k})$ is created with the corresponding input as the root. The children of each vertex are given by the transactions leading from that vertex up to but excluding the first output $s_{\text{out},k} \in \mathcal{S}_{\text{out}}$. The inferred guarantee of the root is given by the trust guarantee $g_{\text{give}}(s_{\text{in},k})$, while the inferred guarantees are calculated for the inferred variables as well as for the outputs. For every variable, the inferred guarantee is calculated as the minimum of the guarantees of the variables with transactions leading to it, i.e. for every variable $s_{\text{inf},i} \in \mathcal{S}_{\text{inf}}$, this is given with

$$\begin{aligned}\forall s_{\text{inf},i} \in \mathcal{S}_{\text{inf}} : g_{\text{inf}}(s_{\text{inf},i}) = min(\{g_{\text{give}}(s_{\text{in},k}) \\ \text{s.t. } s_{\text{in},k} \in \mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{inf}} \wedge s_{\text{inf},i} \in T(s_{\text{in},k})\}).\end{aligned} \tag{4.108}$$

Lastly, the requirements are propagated backwards through the system in the *backward pass.* For every output $s_{\text{out},k} \in \mathcal{S}_{\text{out}}$ an in tree $T(s_{\text{out},k})$ is created with the corresponding output as the root. The children of each vertex are given by the transactions leading to that vertex up to but excluding the first input $s_{\text{in},k} \in \mathcal{S}_{\text{in}}$. The inferred requirement of the root is given by the maximum of the trustworthiness requirements $r_{\text{give}}(s_{\text{out},k})$ of all tree roots, in which the given variable is included given by

$$\begin{aligned}\forall s_{\text{inf},i} \in \mathcal{S}_{\text{inf}} : r_{\text{inf}}(s_{\text{inf},i}) = max(\{r_{\text{give}}(s_{\text{out},k}) \\ \text{s.t. } s_{\text{out},k} \in \mathcal{S}_{\text{out}} \cup \mathcal{S}_{\text{inf}} \wedge s_{\text{inf},i} \in T(s_{\text{out},k})\}).\end{aligned} \tag{4.109}$$

Example 14 shows an example of how a given code base is turned into a model instance, which is then used to detect violations of trustworthiness requirements.

Example 14: Software Monitoring Model

```
float v_0 = 0.0, v_des = 0.0; // IN
std::vector<float> d_all = {5, 0.0}; // IN
float v_1 = 0.0, d_min = 0.0; // INTERNAL
float lvl_brk = 0.0, lvl_th = 0.0; // OUT

while(1){
    yield(); // Implement real-time best practices
    v_0 = get_v_0();
    v_des = get_v_des();
    d_all = get_d_all();

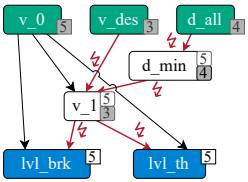
    d_min = *std::min_element(d_all.begin(), d_all.end());
    if(d_min < D_save){
        v_1 = calc_v_redux(v_0, d_min);
        lvl_th = calc_lvl_th(v_0, v_1);
        lvl_brk = calc_lvl_brk(v_0, v_1);
    }
    else{
        v_1 = v_des;
        lvl_brk = 0.0;
    }
    update_state(lvl_th, lvl_brk);
}
```

Listing 4.1: Used example of cruise control code loop.

The given code example is, together with the trust requirements and safety guarantees from Figure 4.20a turned into the model instance shown by Figure 4.20b.

SW Task	r_{inf} and r_{give}	g_{inf} and g_{give}
v_0	-	$g_{\text{give}}(\texttt{v_0}) = 5$
v_des	-	$g_{\text{give}}(\texttt{v_des}) = 3$
d_all	-	$g_{\text{give}}(\texttt{d_all}) = 4$
d_min	$r_{\text{inf}}(\texttt{d_min}) = 5$	$g_{\text{inf}}(\texttt{d_min}) = 4$
v_1	$r_{\text{inf}}(\texttt{v_1}) = 5$	$g_{\text{inf}}(\texttt{v_1}) = 3$
lvl_brk	$r_{\text{give}}(\texttt{lvl_brk}) = 5$	-
lvl_th	$r_{\text{give}}(\texttt{lvl_th}) = 5$	-

(a) Extracted guarantees and requirements.



(b) Resulting model graph

Figure 4.20: Model instance for the given code example.

Because the trustworthiness requirements and guarantees are not met for the transitions (v_des, v_1), (d_all, d_min), (d_min, v_1) *as well as* (v_1, lvl_brk) *and* (v_1, lvl_th), *a violation is detected, which needs to be handled.*

The example code calculates the current speed of the vehicle and compares it to a given minimum distance to a preceding vehicle. If the current speed is above the minimum distance, the brakes are engaged. The inputs are given by the current speed and the minimum distance, while the output is given by the brake engagement. The inferred variables are given by the calculated speed and the comparison of the speed to the minimum distance. The transactions are given by the flow of information between these variables, i.e. from the inputs to the calculated speed, from the calculated speed to the comparison and from the comparison to the brake engagement. The `update_state()` method is used as the output of the model. After modeling the software component, the model instance can be used to detect violations of trustworthiness requirements, which is shown in the next part.

Model Validation and Violation Detection

With this model instance, the given system for violations of safety requirements can be evaluated. In [Sta23b], this is done for violations of the model instance regarding trustworthiness guarantees as well as requirements, denoted by V_T and violations of the model instance regarding the unintended flow of information, denoted by V_U. A model is considered to be validated, if the set of violations $V = V_T \cup V_U$ is empty. The evaluation of the model instance for trustworthiness violations is done by checking for every transaction, if the inferred guarantee of the source variable is greater than or equal to the inferred requirement of the sink variable, i.e. if for $(s_1, s_2) \in X : g(s_1) \geq r(s_2)$. With $r()$ and $g()$ denoting the inferred requirements and guarantees r_{inf} and g_{inf} or the given requirements and guarantees r_{give} and g_{give}, respectively. This leads to the set V_T being defined as:

$$V_T = \{(s_1, s_2) \in X \mid g(s_1) < r(s_2)\}. \tag{4.110}$$

Furthermore, the generation of a model instance necessitates the detection of the model instance for unintended information flow. If there exists a path inside G from an input $s_{\text{in},k} \in \mathcal{S}_{\text{in}}$ to an output $s_{\text{out},k} \in \mathcal{S}_{\text{out}}$, which is not included in the intended flow of information F, this path is marked as a violation of the model instance. To calculate this, the set of first edges in the list of violating paths is given by W_{in} and calculated as

$$\begin{aligned} W_{in} = \{(s_{\text{in},k}, s_2) \in X \mid (s_{\text{in},k} \in \mathcal{S}_{\text{in}}) \\ \wedge (\exists s_{\text{out},k} \in \mathcal{S}_{\text{out}} : ((s_{\text{in},k}, s_2) \notin F) \wedge (s_2 \in T(s_{\text{in},k})))\}. \end{aligned} \tag{4.111}$$

After this, the set of final edges in violating paths is given by W_{out} and calculated as

$$W_{out} = \{(s_1, s_{\text{out},k}) \in X \mid (s_{\text{out},k} \in \mathcal{S}_{\text{out}}) \wedge (\exists s_{\text{in},k} \in \mathcal{S}_{\text{in}} : ((s_1, s_{\text{out},k}) \notin F) \wedge (s_1 \in T(s_{\text{in},k})))\}. \tag{4.112}$$

Since all information flow is inserted into the system with $\mathcal{S}_{\text{in}}$ and exits the system at $\mathcal{S}_{\text{out}}$, monitoring either of the two sets is sufficient, when detecting unintended information flow. The choice of whether W_{in} or W_{out} is monitored, is decided to minimize the overhead of superfluously monitored transactions. For this, the projected fraction p_{in} of all incoming information flowing through $(s_{\text{in},k}, s_2)$ is given by $p_{\text{in}}(s_{\text{in},k}, s_2)$, with

$$\sum_{s_{\text{in},k} \in \mathcal{S}_{\text{in}}, s_2 \in F \wedge s_2 \in T(s_{\text{in},k})} p_{\text{in}}(s_{\text{in},k}, s_2) = 1, \tag{4.113}$$

and $p_{\text{in}}(s_{\text{in},k}, s_2) = 0$ if $(s_{\text{in},k}, s_2) \notin X$. Analogously, the projected fraction of all outgoing information flowing through $(s_1, s_{\text{out},k})$ is given by $p_{out}(s_1, s_{\text{out},k})$, with

$$\sum_{s_{\text{out},k} \in \mathcal{S}_{\text{out}}, s_1 \in F \wedge s_1 \in T(s_{\text{out},k})} p_{out}(s_1, s_{\text{out},k}) = 1, \tag{4.114}$$

and $p_{out}(s_1, s_{\text{out},k}) = 0$ if $(s_1, s_{\text{out},k}) \notin X$. Furthermore, m_{in} denotes the fraction of all unnecessarily monitored transactions and is calculated with

$$m_{in} = \sum_{(s_{\text{in},k}, s_2) \in W_{in}} p_{\text{in}}(s_{\text{in},k}, s_2). \tag{4.115}$$

Analogously, m_{out} is calculated with

$$m_{out} = \sum_{(s_1, s_{\text{out},k}) \in W_{out}} p_{out}(s_1, s_{\text{out},k}) \tag{4.116}$$

and gives the fraction of all unnecessarily monitored transactions, when monitoring W_{out}. As the amount of incoming information flow equals the amount of outgoing information, the choice on which set of W_{in} or W_{out} to consider for V_U is give by

$$V_U = \begin{cases} W_{in} & \text{if } m_{in} \leq m_{out} \\ W_{out} & \text{otherwise} \end{cases}. \tag{4.117}$$

With this, the *trustworthiness violations* V_T and the *unintended flow violations* V_U are signaling transactions. In the context of this work, generating monitoring components, which monitor the system and enable a change in the execution mode of the TTS in case of a detected violation, is a way to react to given violations. Inside this system, the correction of these violations is modeled by inserting variables, signifying a potential solution to the monitoring components. These variables are collected in the set $\mathcal{S}_{\mathrm{V}}$. Every violating transition $v \in V$ is then split by inserting a variable $s_{\mathrm{fix},i} \in \mathcal{S}_{\mathrm{fix}}$ into the transaction, i.e.

$$\forall (s_{v,1}, s_{v,2}) \in V \ : \ X_{fixed} = \{X \setminus (s_{v,1}, s_{v,2})\} \cup \{(s_{v,1}, s_{\mathrm{fix},i}), (s_{\mathrm{fix},i}, s_{v,2})\}, \quad (4.118)$$

where $v = (s_{v,1}, s_{v,2})$ are source and sink variables of violating transaction and X_{fixed} is the set of transactions after the insertion of the variables for violation correction. The variable $s_{\mathrm{fix},i}$ defines the variable which is inserted to correct the current violation v. These variables are then given a fixed trustworthiness guarantee g_{give} and requirement r_{give}, equal to the inferred guarantee and requirement of the source and sink variables, respectively, i.e.

$$g_{\mathrm{give}}(s_{\mathrm{fix},i}) = g_{\mathrm{inf}}(s_{v,1}) \quad (4.119)$$

and

$$r_{\mathrm{give}}(s_{\mathrm{fix},i}) = r_{\mathrm{inf}}(s_{v,2}). \quad (4.120)$$

The old transaction is then removed and a pair of new transactions is inserted to connect the source and sink variables to the new variable, which represents a method to deal with potential safety violations. This method could be a monitoring component, which detects the violation and triggers a mode change in the TTS to mitigate the potential safety problem.

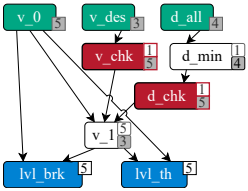
Contribution 11: Software Monitoring Model

The model for software monitoring components gives a way to extract the internal structure of a software component and evaluate it for trustworthiness violations. With the use of this model, monitoring components can be generated, which detect these violations and trigger a change in the execution mode of the mode-change scheduler.

Example 15: Fix for Violations in the Software Monitoring Model

SW Task	r_{inf} and r_{give}	g_{inf} and g_{give}
v_0	-	$g_{\text{give}}(\texttt{v_0}) = 5$
v_des	-	$g_{\text{give}}(\texttt{v_des}) = 3$
d_all	-	$g_{\text{give}}(\texttt{d_all}) = 4$
d_min	$r_{\mathbf{inf}}(\texttt{d_min}) = 1$	$g_{\mathbf{inf}}(\texttt{d_min}) = 4$
v_1	$r_{\text{inf}}(\texttt{v_1}) = 5$	$g_{\text{inf}}(\texttt{v_1}) = 3$
v_chk	$r_{\mathbf{give}}(\texttt{v_chk}) = 1$	$g_{\mathbf{give}}(\texttt{v_chk}) = 5$
d_chk	$r_{\mathbf{give}}(\texttt{d_chk}) = 1$	$g_{\mathbf{give}}(\texttt{d_chk}) = 5$
lvl_brk	$r_{\text{give}}(\texttt{lvl_brk}) = 5$	-
lvl_th	$r_{\text{give}}(\texttt{lvl_th}) = 5$	-

(a) Guarantees and requirements for fixed model.



(b) Validated model graph

Figure 4.21: Model instance fix for the given code example.

The system presented in Example 14 is evaluated in Figure 4.20b. Detected violations are marked, and new fixing variables inserted, depicted in Figure 4.21b. Guarantees and requirements are shown in Figure 4.21a, where changed values are marked. Added variables are marked in Figure 4.21b and implemented into the code as exceptions in Listing 4.2

```
1  // ... previous code
2  d_min = *std::min_element(d_all.begin(), d_all.end());
3  if(d_chk < D_save){
4      try {
5          float d_chk = check_d_min(d_min);
6      } catch (const std::exception& e) {
7          // Handle the exception
8      }
9      v_1 = calc_v_redux(v_0, d_chk);
10     // continue...
11
12 } else {
13     try {
14         float v_chk = check_v_des(v_des);
15     } catch (const std::exception& e) {
16         // Handle the exception
17     }
18     v_1 = v_chk;
19     // continue...
20 }
```

Listing 4.2: Bare-bones example of cruise control loop.

Inside the checking functions, the variables are validated and in case some checks fail, an exception is thrown, which might lead to a change in execution mode.

Summary

Chapter 4 introduced a task model for HESoCs as well as a scheduler to create time triggered schedules for these systems. The task model allows for the specification of periodic tasks with precedence constraints and multiple versions. The scheduler first uses a polyhedron-based approach followed by a simulated annealing-based approach to generate schedules which meet the specified constraints while optimizing for resource utilization and timeliness. Additionally, a mode-change scheduling algorithm was presented to generated time-triggered schedules to handle dynamic changes in system operation modes. Lastly, an automated method to generate anomaly detection components for software components was presented, which can be used to trigger mode changes in the generated schedules. The evaluation of these methods is given in Section 7.1 until Section 7.3.

This page intentionally left blank

This page intentionally left blank

Chapter 5

Dynamic Scheduler for Anytime Neural Networks

This chapter introduces the scheduling of anytime neural networks. This includes the addition of early exits and anomaly detection methods as well as the runtime estimation and checkpoint structure. It is based on the first-author publications [Sta27, Sta25b, Sta23a, Sta24b, Sta23c]. Section 5.1 introduces the model and Section 5.2 introduces a method to add early exits to existing neural networks. Section 5.3 presents an approach to add anomaly detection methods to neural networks. The runtime bounding method is described in Section 5.4, while the checkpoint structure for the execution of anytime neural networks on HESoC is given in Section 5.5. Lastly, Section 5.6 describes the scheduler for the checkpoint structure, representing a standalone scheduling method for ANNs.

The content of Chapter 4 and of this chapter are developed independently. How a combination is achieved, and the methods are introduced in Chapter 5 integrate into the combined scheduler, which is presented in Chapter 4, is described in Chapter 6.

5.1 Anytime Neural Network Model and Constraints

To generate anytime neural networks from existing neural networks, as described in the state of the art, methods to terminate execution, while still providing a valid result, are needed to be added to the existing network structure. This section introduces a model as well as an example which guides through the EE addition process presented in Section 5.2 and the subsequent

sections of this chapter. This model is designed to integrate in Section 6.1 with the task model presented in Section 4.1.

One neural network model $\mathcal{L}_i$ is defined as a partially ordered set of neural network layers

$$\mathcal{L}_i = \{l_i | i \in [0, N_{\mathcal{L}_i})\} \tag{5.1}$$

If a neural network layer l_i is executed after layer l_j, then $l_i < l_j$. The input layer of the neural network thus is denoted as l_0. The output layer is denoted as $l_{N_{\mathcal{L}_i}-1}$. All neural networks are collected in the set $\mathcal{N}$. Tasks are collected in the set $\mathcal{T}$, where one element $\tau_i \in \mathcal{T}$ is defined with the tuple $\tau_i = \{s_{i,-1}, c_i, \alpha_i, \sigma_i\}$, where $s_{i,-1}$ is the input c_i is the class label, $\alpha_i \in \mathcal{N}$ defines the assigned neural network and $\sigma_i \in \mathbb{N}$ gives the priority of the input sample. The priority is defined as an integer value, where a lower value indicates a higher priority, i.e. if $\sigma_i < \sigma_j$, then task τ_i has a higher priority than task τ_j. The set of execution units $\mathcal{U}$ represents all available processing resources, on which neural network layers can be executed.

Table 5.1: Model definition for the used variables in the neural network model.

Component	Symbol	Description
Neural Networks	$\mathcal{N} = \{\mathcal{L}_i \| i \in [0, N_{\mathcal{N}})\}$	Set of all neural networks.
Network Layers	$\mathcal{L}_i = \{l_{i,j} \| j \in [0, N_{\mathcal{L}_i})\}$	Set of all neural network layers.
Execution Units	$\mathcal{U} = \{u_k \| k \in [0, N_{\mathcal{U}})\}$	Set of available execution units[a].
Input Task Set	$\mathcal{T} = \{\tau_i \| i \in [0, N_{\mathcal{T}})\}$	Set of all input tasks[a].
Input Task	$\tau_i = \{s_{i,-1}, c_i, \alpha_i, \sigma_i\}$, $\tau_i \in \mathcal{T}$	Input task, including input sample, data class, assigned neural network and priority.
Layer output	$s_{i,j}$	Trace of the activation of task τ_i after layer l_j, input sample defined as $s_{i,-1}$.

[a]Similar as introduced in Section 4.1.

Example 16 shows how the structure of the ResNet-18 [45] is mapped onto the defined neural network model description. The input sample $s_{i,-1} \in \tau_i$ is processed by the first layers l_1 and l_2. Afterwards the residual blocks $\{l_3, \dots, l_6\}$, $\{l_7, \dots, l_{10}\}$, $\{l_{11}, \dots, l_{14}\}$ and $\{l_{15}, \dots, l_{18}\}$ are executed. Lastly, the output layer $l_{19} = e_0$ as the original exit is executed. This example will be used throughout this chapter and the introduced components will be added to this network structure.

Example 16: Mapping ResNet-18 to the Neural Network Model

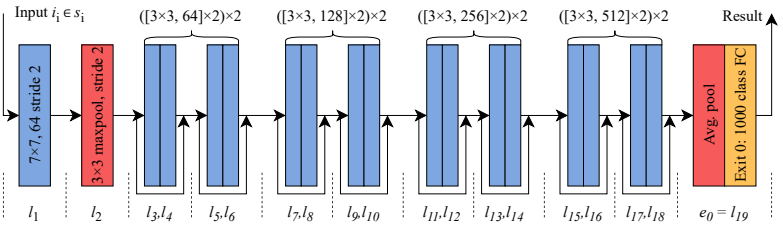


Figure 5.1: Mapping of the structure of the ResNet-18 model [45] onto the defined model. Pooling layer are marked in red, output layers are marked in orange and convolutional layers are marked in blue.

Figure 5.1 maps the structure of the ResNet-18 model onto the defined neural network model. This model is retrained in our example task for weather classification. The input sample $s_{i,-1} \in \tau_i$ is processed by the first layers l_1 and l_2. Afterwards the residual blocks are executed until a classification is undergone in layer e_0.

The timing definitions for each layer follow the same structure. A timing defined with $\hat{t}$ denotes an estimated value, while a timing defined with t (i.e. without the hat) denotes a measured value. Furthermore, a timing defined with $\bar{t}$ denotes an absolute time, while a timing defined without the overline denotes a relative time. Lastly the combination as $\hat{\bar{t}}$ denotes an estimated absolute time, like the expected task finish time. An overview of all definitions is given in Table 5.2.

First, $\hat{t}_C$ denotes the estimated execution time for a layer $l_i \in \mathcal{L}_i$ on a given execution unit $u \in \mathcal{U}$, given by the function

$$\hat{t}_C : \mathcal{L}_i \times \mathcal{U} \rightarrow \mathbb{R}^+. \tag{5.2}$$

The execution time is determined using the runtime estimation method presented in Section 5.4 and depends on the specific execution unit due to varying computational capabilities. During execution, the execution time might differ from the estimated execution time, this measured execution time on execution unit $u \in \mathcal{U}$ is denoted by

$$t_C : \mathcal{L}_i \times \mathcal{U} \rightarrow \mathbb{R}^+. \tag{5.3}$$

The task release time is defined as:

$$t_R : \mathcal{T} \rightarrow \mathbb{R}^+, \tag{5.4}$$

denoting the time, when the task is released for processing. The execution start time is defined as the start of execution of the given layer of the assigned neural network α_i and is denoted as:

$$t_S : \mathcal{L}_i \rightarrow \mathbb{R}^+. \tag{5.5}$$

The start of an execution of a task is then given by $t_S(l_0)$, where $l_0 = \min(\alpha_i)$ with α_i being the assigned neural network for task τ_i.

Given the start time of the task, the estimated absolute finish time of one task on a specific execution unit is defined as:

$$\hat{\bar{t}}_C : \mathcal{T} \times \mathcal{U} \rightarrow \mathbb{R}^+, \tag{5.6}$$

and calculated with $\hat{\bar{t}}_C(\tau_i, u) = t_S(l_0) + \max_{l \in \alpha_i}(\hat{\bar{t}}_C(l, u))$ for execution unit $u \in \mathcal{U}$ and the start time of layer $l_0 = \min(\alpha_i)$, summing the estimated execution times of all layers in the assigned neural network on that unit. The measured absolute finish time of one task on a specific execution unit is defined with:

$$\bar{t}_C : \mathcal{T} \times \mathcal{U} \rightarrow \mathbb{R}^+, \tag{5.7}$$

The task deadline is defined as:

$$t_D : \mathcal{T} \rightarrow \mathbb{R}^+, \tag{5.8}$$

denoting the relative time past the release time, when the task needs to be completed. The absolute deadline is thus defined as:

$$\bar{t}_D : \mathcal{T} \rightarrow \mathbb{R}^+, \tag{5.9}$$

and calculated as $\bar{t}_D(\tau_i) = t_R(\tau_i) + t_D(\tau_i)$. These time definitions are summarized in Table 5.2.

Note that the runtime estimation and start times are generalized for all neural network layers. For any arbitrary layer l_i the start time $t_S(l_i)$ and expected execution time $\hat{t}_C(l_i)$ are defined in respect to the given layer, irrespective of the starting layer l_0 and defined for every other layer l_i with $i \neq 0$ as the start and execution time of the current layer. Again, $\hat{t}$ indicates an estimated value, while values without the hat indicate measured values. Values with $\bar{t}$ indicate absolute times, while values without the overline indicate relative times.

Table 5.2: Runtime definitions for neural network tasks.

Component	Estimated	True	Collection
Layer Execution Time	$\hat{t}_C(l_j, u_k)$	$t_C(l_j, u_k)$	$\mathrm{T}_C(l_j, u_k)$
Layer Finish Time	$\hat{\bar{t}}_C(l_j)$	$\bar{t}_C(l_j)$	$\overline{\mathrm{T}}_C(l_j)$
Release Time	✗	$t_R(\tau_i)$	$\mathrm{T}_R(\tau_i)$
Start Time	✗	$t_S(l_j)$	$\mathrm{T}_S(l_j)$
Relative Deadline	✗	$t_D(\tau_i)$	$\mathrm{T}_D(\tau_i)$
Absolute Deadline	✗	$\bar{t}_D(\tau_i)$	$\overline{\mathrm{T}}_D(\tau_i)$

Example 17: Timing Definitions for ResNet-18

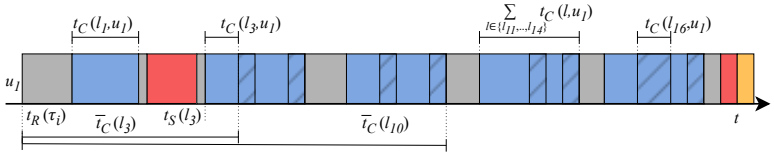


Figure 5.2: Timing example for ResNet-18. Box coloring is taken from Figure 5.1. Idle times are marked in gray.

Figure 5.2 shows timing values for layers of the ResNet-18 example, executed on execution unit u_1. The start of the task is denoted as $t_S(\tau_i)$, while the start time of a layer is denoted by $t_S(l_i)$. The execution time is marked for layer l_1, l_3 and l_{16} as well as all layers $\{l_{11}, \ldots l_{14}\}$. Note that this includes potential idle times between layers. The finish times $\hat{t}_C(l_3)$ and $\hat{t}_C(l_{10})$ are marked as well.

In addition to the existing layers of the neural network, early exits and anomaly detection modules are added to the network structure. These are modeled as additional layers $e_i \in \mathcal{L}_i$ and $a_i \in \mathcal{L}_i$. Each EE layer e_i has an associated estimated execution time $\hat{t}_C(e_i, u)$ and measured execution time $t_C(e_i, u)$ as defined in Equation (5.2) and Equation (5.3), anomaly detection layers are defined to also be layers and thus also have estimated and measured execution times. Exit layers are denoted by using the variable name e_i, where i indicates the index of the EE in the resulting neural network structure. An index of 0 defines the last possible exit layer, i.e. the original exit, while an index of $N_e - 1$ defines the earliest possible exit layer, where N_e is the total number of EE layers in the neural network. Similarly, anomaly detection layers are denoted with the subscript a_i, where i indicates the index of the anomaly detection layer in

the neural network structure. An index of 0 defines the latest possible anomaly detection layer, while an index of $N_a - 1$ defines the earliest possible anomaly detection layer, where N_a is the total number of anomaly detection layers in the neural network.

The partial order of a neural network as a set of layers $\mathcal{L}_i$ is kept to include the EE layers e_i and anomaly detection layers a_i. For all layers executed before an exit layer e_i, the relation $l_j < e_i$ holds. For all layers which would be executed, should the EE not be chosen, the relation is not defined. Similarly, for all layers executed before an anomaly detection layer a_i, the relation $l_j < a_i$ holds. For all layers which would be executed, should the anomaly detection not be chosen, the relation is not defined.

For practical use, added components need to be attached to the network at positions between block-like structures. These structures often consist of convolutional and pooling layers. Example 18 illustrates potential positions for the addition of EE layers and anomaly detection components in the ResNet-18 [45]. For all ResNet-based models, it is advantageous to add EE layers and anomaly detection components after the residual blocks, as these blocks are designed to extract features. This concept can be mirrored to the residual blocks of ResNeXt-based models [46], the dense blocks of DenseNet-based models [128] and the inception blocks of GoogLeNet-based models [44].

Example 18: Potential Component Positions in ResNet-18

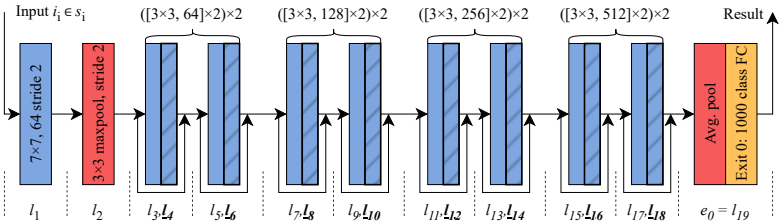
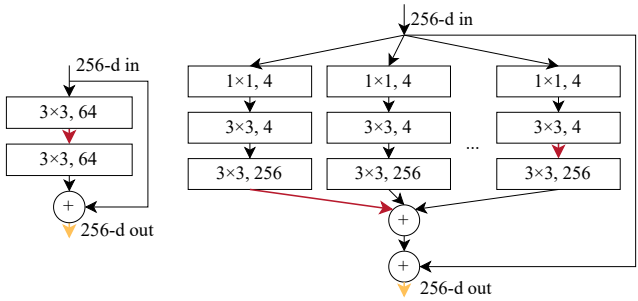


Figure 5.3: Positions where early exits and anomaly detection modules can be inserted into the ResNet-18 model. Corresponding layers are bold and marked dashed.

Figure 5.3 shows potential positions for the addition of EE layers and anomaly detection components. For ResNet-18 [45] in this model, these are given as $l_4, l_6, l_8, l_{10}, l_{12}, l_{14}, l_{16}$ *and* l_{18}.

While the intermediate result of each layer of a neural network might be used as an input into an added EE layer or anomaly detection component, using the output of the last layer of a block-like structure facilitates a more complete feature extraction. Using one convolutional layer as the attachment point inside such a block-like structure leads to an incomplete extraction and bears the danger of a significant drop in accuracy. Figure 5.4 illustrates how the output of the last layer of a block-like structure is a viable position for the addition of EE layers and anomaly detection components, while the output of one convolutional layer inside such a block-like structure is might not be a viable position due to only partial feature extraction. This concept holds true for other block-like architectures like DenseNet-based models [128] and GoogLeNet-based models [44].



(a) ResNet-18 [45] (b) ResNeXt-101 [46]

Figure 5.4: Possible attachment layers for the argument of using the output of the last layer of a block-like structure, argued via the diagrams from [45] and [46]. Viable outputs are marked in yellow, while non-viable outputs are marked in red. Blocks show filter size and number of filters.

Contribution 12: Early Exit Model for Anytime Neural Networks

The presented model and timing definitions allow for the modelling of the addition of early exits and anomaly detection components to existing neural networks. This model is designed to integrate with the task model presented in Section 4.1 and thus allows for the integration of these components into the generated schedules for HESoC.

In Figure 5.4a, the output after the final addition is marked in yellow, which signifies a viable position for the addition of components. Using the red output might lead to missed information via the skip connection. In Figure 5.4b, again the output after the final addition is marked in yellow, which signifies a viable

position for the addition of components. Using the red output right before the internal addition or the output of the internal convolutional layer on the right risks loosing information via the skip connection or via other convolutional layers.

5.2 Exit Addition to Existing Backbone Networks

This section introduces a method to add early exits to existing neural networks, based on the first-author publication [Sta24b]. While many further methods exist to add early exits to neural networks [69, 103, 129, 130], the introduced method is designed for an automatic addition of early exits to neural networks in combination with the time-triggered scheduling approach presented in Chapter 4. A detailed comparison of this work to existing works can be found in the related work section, this chapter uses the notation presented in Section 5.1.

Adding an early exit (EE) to an existing neural network is implemented using a multistep process. First, suitable candidate layers in the original model for the branching of the dataflow are identified. This step is presented in Section 5.2.1. Second, the added early exits are trained to achieve a high accuracy, which is introduced in Section 5.2.2. Lastly, testing methods are presented in Section 5.2.3.

The given process allows for the independent optimization of each step and adds interchangeability to each component. Determining the position of the EE could thus be performed using tools like CNNParted [131, 132], while the training of the added early exits can be independently performed using different training algorithms like AdaEE [103] or ICE TEA [Sta24b]. The addition of anomaly detection components can be facilitated using tools like EFFECT [Sta23a]. In contrast to a combined design space exploration like EDANAS [103], this approach allows for a more modular method and a dedicated component evaluation, as presented in Sections 7.4.2, 7.4.3 and 7.5.

5.2.1 Positioning of Early Exits Start Layers

Positioning early exits in an existing neural network requires the identification of suitable candidate layers to attach the early exit and to branch of from the original network structure. A trade-off exists between the accuracy of the

early exit and the execution time reduction achieved by branching earlier in the network. Additionally, the overhead added by a decision algorithm, as presented in Section 5.5, needs to be considered when positioning early exits. Adding a high number of early exits might not leave enough time between two early exits to execute the decision algorithm, thus reducing the effectiveness of the added early exits. It is furthermore possible, that outside requirements (like the ones mentioned in Section 6.1) influence the positioning of the early exits.

The total number of early exits to add is defined by N_e and a time window $t_w = (t_{min}, t_{max})$ between two early exits to ensure enough time for the decision algorithm to execute is defined. This window encompasses the minimum time t_{min} required for the decision algorithm to execute and the maximum time t_{max} can be given by external factors like runtime constraints given by the time-triggered scheduling approach presented in Chapter 4.

Depending on the structure of the original network $\mathcal{N}$, a set of candidate layers $\mathcal{L}_c$ is defined by the user, e.g., only allowing to add early exits after certain layer types like convolutional layers or after block-like structures like e.g. residual blocks in ResNet-based models [45].

Related works like CNNParted [131, 132] can be used to determine the optimal positioning of partitioning points. These partitioning points are selected from a set of candidate layers and used in these original publications to bind partitions of the original network to execution units. The partitioning points could however also be used to determine suitable positions for early exits, as the partitioning points can be determined by the execution time of the partitions. Treating these partitions as sets of layers between early exits would allow for the use of the partitioning points as possible entry points for early exits. CNNParted does not inherently provide a method to also train and evaluate the early exits as well as the anomaly detection components by not making layer traces available [132]. This leads to the need for a faster and more modular method to determine the position of early exits, which is presented in this chapter.

Given an existing neural network $\mathcal{L}_i \in \mathcal{N}$, a total number N_e of early exits to add, a minimum and maximum time window between the early exits t_w, and a given set of possible candidate layers $\mathcal{L}_c \subset \mathcal{L}_i$, the positioning of early exits is defined as finding a set of layers $\mathcal{L}_e = \{l_{e_0}, l_{e_1}, \dots, l_{e_{N_e-1}}\}$, where $l_{e_i} \in \mathcal{L}_c$. The set $\mathcal{L}_e$ defines the layers in the original neural network $\mathcal{L}_i$, where early exits might be added. The positioning of the early exits is defined as an optimization problem with the objective to maximize classification accuracy while adhering to constraints given by the time window t_w between the added early exits.

While a holistic approach of solving a similar optimization is given in [131, 132], this work uses a heuristic approach to first determine suitable positions for the given early exits and then later train the early exits on these positions afterwards.

Example 19: Tracing Component Addition for ResNet-18

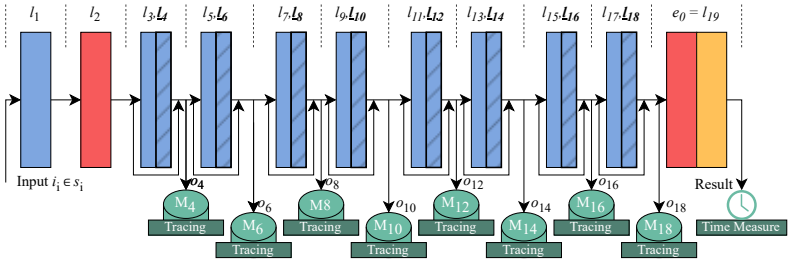


Figure 5.5: Tracing components added to the layers of a ResNet-18 [45] model. Tracing components are marked in green. The output tensor of the last result is not saved. Only the execution time is measured.

The addition of tracing components to the layers of a ResNet-18 model is shown in Figure 5.5. The tracing components are added to every layer in the candidate set $\mathcal{L}_c$. Here, these are the layers between each residual block, marked in bold. The final output is not saved, but the runtime is kept.

The positioning starts by placing tracing components at every layer in the candidate set $\mathcal{L}_c$, these tracing components measure the execution time of the layers during the forward passes and collect the output activations of the layers. The activations are collected in the set $\mathcal{S}_j$, where one element $s_{i,j} \in \mathcal{S}_j$ specifies the sample of the ith input collected at layer l_j. The original ith input to the network is given by $s_{i,-1}$. Collecting these activation samples allows for the training of early exits at the candidate layers. The execution time measurements allow for the estimation of the execution time of each layer under real conditions, populating Equation (5.2).

Tracing layer activations and measuring the execution time of the layers is performed using inferences of the original network with the added tracing components. This uses the training dataset as an input $\mathcal{S}_{\text{train}} \subset \mathcal{S}$ and allows the traced samples to be used for training the early exits in the subsequent steps. It also allows for the measurement of the execution time under realistic conditions. Because the accuracy of the added early exits is dependent on the

exit structure and the given exit structure is not yet determined, an abstract representation needs to be chosen to gauge the performance of a potential early exits at this stage. This leads to the necessity to trace the output samples as well as the execution time. To reduce the complexity of the classification task at hand, the complexity of the necessary output classes could be reduced, by grouping classes into super-classes, like using hypernyms of WordNet [133] for ImageNet [134] classification. The tracing of the intermediate activations and the runtime measurements is performed once for every sample in the training dataset, resulting in the collection of one element $s_{i,j} \in \mathcal{S}_j$ per candidate layer l_j and input sample τ_i.

Contribution 13: Layer Tracing for Early Exit Positioning

A layer trace is defined as the tensor produced by a layer $l_j \in \mathcal{L}_i$ when processing an input sample $s_j \in \mathcal{S}_{\text{train}}$. The collection of all layer traces at layer l_j is facilitated via the presented method and allows for generating training data for early exits, which start at the given layer.

The collected activation samples are then used to calculate the best position to place early exits using a relative distance measurement. This measurement represents the distance of samples of one class of one layer to the samples of the other classes of the same layer. Trivial reasoning would suggest that a chosen distance measurement must be small between two samples of one class and large between two samples of different classes, reflecting the possibility of a high classification accuracy and a high chance for anomaly detection. A minimization of the distance between samples of the same class and a maximization of the distance between samples of different classes is thus desired. This cost function $F_{\text{cost,ee}}(l_k)$ for a candidate layer l_k is defined to find an optimized solution for the positioning of early exits as:

$$F_{\text{cost,ee}}(l_k) = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{s_{i,j} \in \mathcal{S}_j} \frac{1}{|\{s_{\lambda,j} \in \mathcal{S}_j | c_\lambda \neq c_i\}|} \cdot \sum_{\substack{s_{\lambda,j} \in \mathcal{S}_j \\ c_\lambda \neq c_i}} \delta(s_{i,j}, s_{\lambda,j}), \tag{5.10}$$

with the cost being defined as the sum of the average distance between samples of one class $s_{i,j}$ to all samples of other classes. The distance measurement $\delta(a,b)$ is defined as a distance metric between two activation samples a and b. The set of all classes in the dataset is defined as C and the set of collected activation samples for class $c \in C$ at layer l_j is defined as $\mathcal{S}_j^c = \{s_{i,j} \in \mathcal{S}_j | c_i = c\}$, where c_i is the ground truth class label for the input sample $s_{i,-1}$. A detailed explanation of the used datasets and class labels is given in Table 5.3.

The distance metric $\delta(a,b)$ can be chosen freely. In this work, an entropy

Table 5.3: Model definition for the used classes in the ANN input dataset.

Component	Symbol	Description
All Classes	$C = \widehat{C^0}$	Set of all class labels in the dataset.
Abstracted Classes	$\widehat{C^p}$	Set of class labels with reduced complexity for early exit e_p.
Ground Truth	$c_i \in C$	Ground truth class label for input sample $s_{i,-1}$.
Guessed Class	$\hat{c}_{i,j}$	Guessed class label for sample $s_{i,-1}$ at layer l_j.
Layer Sample Set	$\mathcal{S}_j$	Set of all collected activation samples at layer l_j.
Class Sample Set	$\mathcal{S}_j^c$	Set of activation samples at l_j for class c.

measurement is used to determine the inter class entropy at each candidate layer. The inter class entropy measures the distribution of predicted class labels at a given layer, with a low inter class entropy indicating that samples are clustered around certain class labels, while a high inter class entropy indicates a more uniform distribution of predicted class labels. Thus, a low inter class entropy is expected to indicate that an early exit placed at this layer could achieve a high classification accuracy. The inter class entropy H_j is calculated for every candidate layer $l_j \in \mathcal{L}_i$ using the collected activation samples $\mathcal{S}_i$.

Given a set of collected activation samples $\mathcal{S}_i$ at layer l_i and a classification function $f_{\text{class}} : s_{i,j} \rightarrow \hat{c}_{i,j}$ that maps an activation sample $s_{i,j} \in \mathcal{S}_i$ to a predicted class label $\hat{c}_{i,j}$, the inter class entropy H_i at layer l_i is defined as

$$H_i = -\sum_{c \in C} p_{i,c} \otimes \log(p_{i,c}), \tag{5.11}$$

where C is the set of all class labels in the dataset and $p_{i,c}$ is the probability of class c occurring in the predicted class labels $\hat{c}_{i,j}$ for all samples $s_{i,j} \in \mathcal{S}_i$. The operator $\otimes$ defines element-wise multiplication.

Using this inter-class entropy measurement, the candidate layers $\mathcal{L}_c$ are ranked based on their inter class entropy H_j, with lower values indicating better suitability for placing an early exit. The final selection of layers $\mathcal{L}_e$ to place the early exits is then performed by iterating through the ranked list of candidate layers and selecting layers that adhere to the time window t_w constraint between two early exits. For this, at least one early exit is placed between two runtime constraints. This results in the list of possible layers for each runtime constraint $\mathcal{L}_c$.

The sum of the estimated execution times $\sum_{l \in \mathcal{L}_c, l<e_i, l>e_{i-1}} \hat{t}_C(l_j, u)$ of the layers between two early exits e_i and e_{i-1} on execution unit u is used to ensure that the maximum runtime constraints are not violated. The selection process iterates through the ranked candidate layers and adds the first layer that adheres to the time window to the set of selected early exit layers $\mathcal{L}_e$. This process continues until the desired number of early exits N_e is placed or no more suitable candidate layers are available.

Given the inter class entropy based ranking of candidate layers and the constraints given by the time window t_w, the positioning of early exits in an existing neural network is performed automatically, allowing for the addition of exits. This algorithm then returns the best expected classification accuracy under given requirements, resulting from t_w.

5.2.2 Training Strategies for Added Early Exits

After the positioning of the early exits, the training of the added early exits is performed. Because the original neural network is already trained and the weights and biases are fixed, no retraining of layers in the original model is done. Only the weights and biases of the added early exits are trained. This allows for a faster training of the added early exits and eliminates interferences between the original model and further early exits. The training of the added early exits is performed using the collected activation samples $\mathcal{S}_j$ at each early exit layer $e_j \in \mathcal{L}_e$. This represents the training dataset. The ground truth class labels for each sample are taken from the original training dataset $\mathcal{S}_{\text{train}}$ used to trace the activations.

The training of the added early exits is performed using standard training algorithms like stochastic gradient descent (SGD) or adaptive moment estimation (Adam) [57], minimizing a classification loss function like the cross-entropy loss [56] between the predicted class labels of the early exit and the ground truth class labels. For each early exit, the training is performed independently, as explained above, allowing for parallel training of the added early exits, since each exit only depends on the fixed traced activations from the original model. The training is performed for a predefined number of epochs or until convergence is reached, based on a validation dataset, if available.

The labels of the training samples can be further modified to improve the training of the added early exits. For example, class labels can be grouped into super-classes, reducing the complexity of the classification task for the earlier

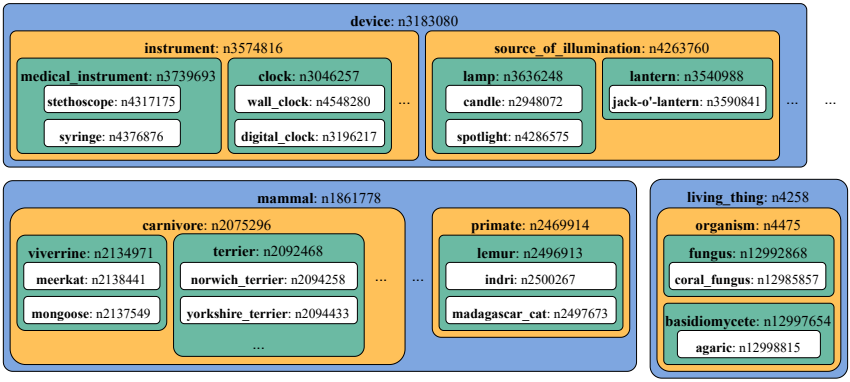


Figure 5.6: Grouping of WordNet [133] synsets into nearest hypernyms. The original synsets used in ImageNet [134] are marked in white, the first abstraction is marked in green, the second abstraction is marked in yellow and the third abstraction is marked in blue.

exits. To accommodate for the expected drop in accuracy due to a reduced amount of computation, the number of classes for each early exit is reduced, leading to $|\widehat{C^{p+1}}| < |\widehat{C^p}|$, where $\widehat{C^p}$ signifies the classes used for the early exit $e_P \in \mathcal{L}_e$, where p represents the position of the early exit in the network with $p = 0$ signifying the original classification layer and $p = N_B - 1$ the earliest used early exit.

An example for this is presented in [Sta24b] and uses hypernyms of the ImageNet [134] class names. Since ImageNet uses WordNet [133] synsets [134] as classes, hypernyms can be used to group classes into super-classes. A possible grouping of ImageNet classes into hypernyms is presented in Figure 5.6. The highest level hypernym used is `entity`, which contains all classes. Shown here are three abstraction groups of synsets with the highest abstraction level presented in blue. The listed examples include `device`, `mammal` and `living_thing` in blue. The next level contains `instrument` and `source_of_illumination` in `device`, `carnivore` and `primate` in `mammal` and `organism` in `living_thing`.

This also highlights one weakness of this approach. Because the grouping of classes into super-classes is based on WordNet, which itself is automatically generated, the grouping includes ambiguities, like `organism` and `living_thing`. Additionally, inconsistencies like `mammal` being a hypernym of `primate` and on one level as `living_thing` exist. While the first one is easily

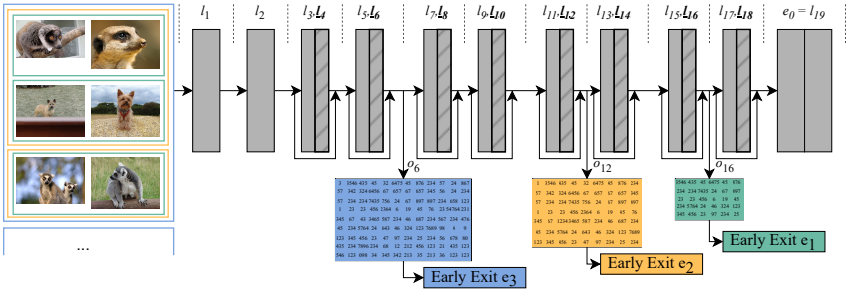


Figure 5.7: Training concept of the added early exits. The original output is not used, since this exit is already trained. The first abstraction level is marked green. The second abstraction level is marked yellow and the third abstraction level is marked blue. The original class labels are taken from the corresponding abstraction level. Inputs examples are taken from the ImageNet [134] validation dataset.

ignored, the second one needs to be studied in more detail. Since WordNet is based on automatically generated semantics, these names are only approximate representations of the underlying concepts. Because of this an abstraction of the synset names from real life examples is justifiable, if the underlying classes are exclusive. This holds true, especially since these inconsistencies are a result of an automatic generation. For later real-world evaluation a handcrafted grouping in a different dataset is used, eliminating this issue. Nevertheless, the same concept of increasing abstraction of classes holds true.

This strategy can be extended to other datasets by defining a hierarchy of classes, allowing for a reduction of the number of classes for early exits placed earlier in the network. In an automotive context, this could be used to group traffic signs into super-classes like *speed limit signs*, *warning signs*, and *prohibitory signs*, allowing for a higher accuracy for a lower complexity task of early exits, which are placed earlier in the network. Depending on the application, different grouping strategies can be employed to optimize the training of the added early exits. Nevertheless, the presented method allows for the automatic addition of early exits to existing neural networks, enabling anytime behavior without changes to or retraining of the original model. Section 7.4 to Section 7.5 shows an evaluation of the presented method.

Figure 5.7 shows the idea for the training process for the given early exits.

The inputs are grouped into the given abstraction levels and the early exits are trained corresponding to that grouping. The graphic shows the concept for inference and layer traces are collected after the given layers once per sample for the candidate layers $\mathcal{L}_c$ (as explained in Section 5.2.1) and during training, these are used for the determined early exit layers. During training the algorithm refers to these traces, since there is no need to perform an inference through the whole model for each training step.

Contribution 14: Abstraction Method for Earlier Inference

The abstraction of class labels into super-classes is a method to increase the accuracy of early exits by reducing the complexity of the classification task. By grouping classes into super-classes, the early exits can achieve a higher accuracy for a reduced number of classes, which is especially beneficial for early exits placed earlier in the network, where less computation is available.

5.2.3 Testing and Evaluation of the Added Early Exits

Generated early exits are tested using a testing dataset $\mathcal{S}_{\text{test}} \subseteq \mathcal{S}$, which was separated from the training dataset used to trace the activations and train the early exits. As done in Section 5.2.2, the input for each early exit is taken from the activations collected at the respective early exit layer $l_{e_i} \in \mathcal{L}_e$ during inferences of the original neural network using the testing dataset as input. The collected activations are then passed through the respective early exit to obtain predicted class labels for each input sample. The original class label is taken from the ground truth class labels of the testing dataset.

Two metrics are used to evaluate the performance of the added early exits: the classification accuracy and the equivalence to the original model. The classification accuracy is calculated as the percentage of correctly classified samples by each early exit, compared to the ground truth class labels and calculated as:

$$\texttt{Acc}(e_i) = \frac{1}{|\mathcal{S}_{\text{test}}|} \cdot \sum_{s_j \in \mathcal{S}_{\text{test}}} \begin{cases} 1, & \text{if } \hat{c}_{i,j} = c_i \\ 0, & \text{otherwise} \end{cases}, \tag{5.12}$$

where $\hat{c}_{i,j}$ is the predicted class label of early exit e_i for sample s_j and c_i is the ground truth class label for sample s_j.

Additionally, the equivalence of the early exits to the original model is evaluated by comparing the predicted class labels of the early exits to those of the original

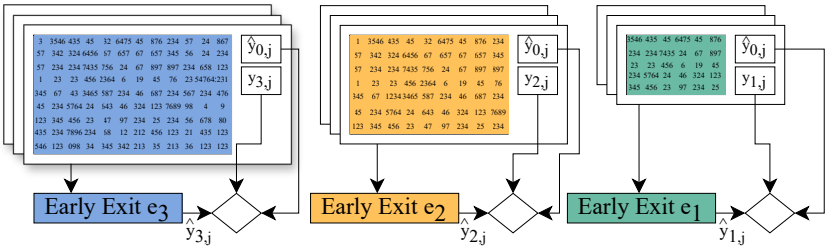


Figure 5.8: Testing process of the added early exits. The original output is not used, since this exit is already trained. The first abstraction level is marked green. The second abstraction level is marked yellow and the third abstraction level is marked blue. The original class labels are taken from the corresponding abstraction level.

model. With the eqivalence metric defined as:

$$\text{Equ}(e_i) = \frac{1}{|\mathcal{S}_{\text{test}}|} \cdot \sum_{s_j \in \mathcal{S}_{\text{test}}} \begin{cases} 1, & \text{if } \hat{c}_{i,j} = \hat{c}_{i,-1} \\ 0, & \text{otherwise} \end{cases}, \tag{5.13}$$

where $\hat{c}_{i,j}$ is the predicted class label of early exit e_i for sample s_j and $\hat{c}_{i,-1}$ is the predicted class label of the original model for sample s_j.

An evaluation of the presented early exit addition method is given in Section 7.4.2. Figure 5.8 shows the concept for the testing process of the added early exits. Because the traces of the original model are collected once for each sample together with the class label, the test of the trained early exit is trivial and does not need a complete forward pass. Only the corresponding traces of the testing dataset are passed through the corresponding early exit to obtain the predicted class labels, which are then compared to the ground truth class labels, resulting in the accuracy. For the equivalence, these are compared to the predicted class labels of the original model.

5.3 Addition of Anomaly Detection to ANNs

International standards necessitate the use of anomaly detection methods and out-of-distribution (OOD) detectors for automotive systems to ensure functional safety and to trigger corresponding safety mechanisms [71]. In this work, these anomaly detection methods are integrated into ANN architectures for the detection part. The anomaly detection methods fulfill these

requirements to detect inputs that are outside the operational design domain (ODD) of the automotive system as mandated by automotive standards like the ISO 8800 [71]. By integrating these methods into the ANN, the system can ensure that only valid and safe inputs are processed further. This aims to fulfill requirements for safety from the aforementioned automotive standards. The reaction to a detected anomaly is facilitated via the early exit part. The components for the detection of anomalies are described in this section.

After positioning, training and testing early exits as described in Section 5.2, these anomaly detection mechanisms are added to the ANN architecture to provide an input to the triggering of the early exit choice. In this work, a method to automatically generate anomaly detection methods is used to place one anomaly detection module before each early exit. This method is based on a tool called EFFECT and takes a set of predefined methods and datasets and returns configured anomaly detection modules [Sta23a].

This method allows for the automatic comparison, selection and configuration of different anomaly detection methods for each early exit position. The selected anomaly detectors are then trained (if necessary) on the features extracted by the ANN up to the respective early exit position. The training is done using different datasets, which depend on the given use case. The used dataset classifies samples into classes which determine an anomaly value. Classes sorting these samples are called anomaly classes. The configured anomaly detection modules can then be integrated into the ANN architecture in front of the associated early exit. During inference, the anomaly detection modules evaluate the extracted features and decide whether to continue processing through the main network, trigger an early exit or abort the inference.

> ***Contribution 15: Training and Configuration of AD Modules***
>
> *The presented method allows for the automated training and configuration of anomaly detection modules for each early exit position. By utilizing the EFFECT tool, different anomaly detection methods are compared, selected and configured based on the specific requirements of the application. This automation streamlines the integration of anomaly detection into ANN architectures, ensuring that the most suitable methods are employed for each early exit position to enhance the safety and reliability of automotive applications.*

Contrary to the early exit addition described in Section 5.2, the positioning and configuration of the anomaly detection methods are undergone in one step.

Because the position of each anomaly detection module is constraint by the position of the early exits, the design space is reduced to using the outputs of the layers before the corresponding early exits.

5.3.1 Layer Tracing for Anomaly Detection Dataset Creation

Similar to the early exit addition described in Section 5.2, the training dataset for the anomaly detection is generated through tracing the activations of training samples through the ANN up to the respective early exit position. From a training dataset $\mathcal{S}\text{train} \subset \mathcal{S}$, the traced activation samples are collected in $\mathcal{S}i = si, j; |; j \in [0, |\mathcal{S}\text{train}|)$ with one element per input sample per candidate layer. This notation mirrors the notation introduced in Section 5.2.1. Contrary to the early exit addition, the class of the input samples might differ from the original class. Here, the sample class indicates whether the input sample is a `normal` input sample, lying inside the ODD, or an `anomalous` input sample, lying outside the ODD. Depending on the use case, this anomalous class might be further divided into classes signaling the resulting action. This includes a change in ODD or a change in class abstraction, which might result in the use of an early exit with a more abstract classification. Additionally, an abortion of the inference as well as the triggering of a safe state can be signaled via distinct anomalous classes. Not every anomaly detection module needs to be trained on all classes. Depending on the use case, some anomaly detection modules might only need to distinguish between `normal` and `anomalous` samples, while others might need to distinguish between multiple anomalous classes.

The generation of the training dataset for anomaly detection then necessitates the extension of the early exit training dataset $\mathcal{S}\text{train}$. This extension includes a second dataset $\mathcal{S}\text{anomaly}$, which contains samples representing the anomalous classes. The generation of this dataset depends on the use case and the desired anomalous classes. An inference is then done once for every input sample through the ANN. For each exit position $l_{e_i} \in \mathcal{L}e$, the traces are collected. These traces then build the training dataset for the anomaly detection module for the respective exit layer le_i as $\mathcal{S}i, \text{anomaly} = si, j; |; j \in [0, |\mathcal{S}_{\text{anomaly}}|)$. The classification output of the original ANN is not considered, because the training label for each trace equals the respective anomaly class.

Figure 5.9 shows the process of tracing the layer activations of the training samples through the ANN for the generation of the anomaly detection dataset. In contrast to Figure 5.7 for the early exit addition, the collection of training data for anomaly detection modules sorts the tracing samples by ODD

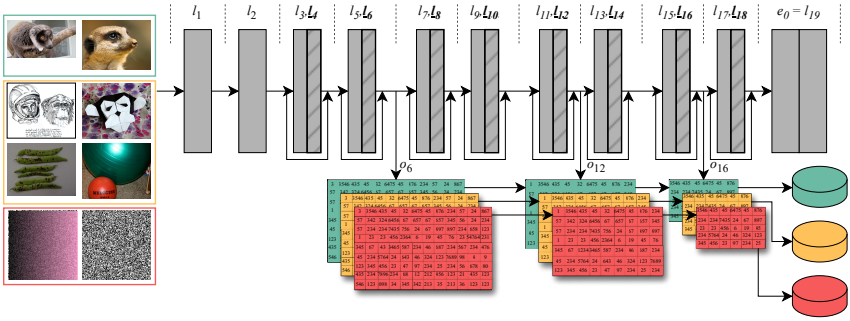


Figure 5.9: Tracing for the generation of the anomaly detection dataset. Green shows normal samples, yellow marked samples are saved into the anomaly dataset and red samples are samples potentially leading into an abortion of the inference. The original class labels are not used, since the training label for each trace equals the respective anomaly class. The input examples are taken from the ImageNet [134] validation dataset and the ImageNet-O [135] and ImageNet-R [136] datasets.

classes and keeps all traces of one input sample together. This allows further anomaly detection modules to potentially take traces of multiple different layers as an input, which aims to increase the detection performance and makes the use of dimension matching strategies necessary. These strategies aim to equalize the tensor dimensions of the traced activations. The example pictures in Figure 5.9 are taken from the validation dataset of ImageNet [134] and the ImageNet-O [135] and ImageNet-R [136] datasets as well as white noise samples.

5.3.2 Configuration of Added Anomaly Detection Modules

Following from ISO 8800, ML-based modules need to indicate the confidence of their classification [71]. As presented in the state of the art, multiple methods for anomaly detection in ML-based systems exist. This score can either be calculated by an integrated ML-based inference concept [72], calculated via a distance measurement between an input sample and a distribution like [74, 137] or given by a classification result of a classification model [73]. This work is based on the EFFECT tool and is able to configure anomaly detection methods based on distance measurements as well as classification models [Sta23a].

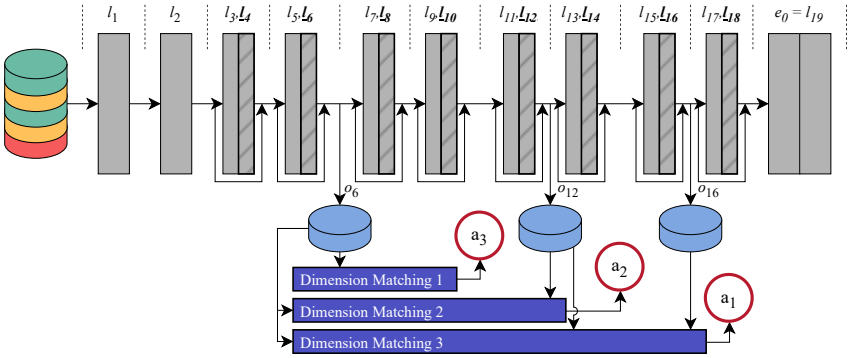


Figure 5.10: Dataflow of the anomaly detection modules. Dimension matching is drawn in blue. Anomaly classification modules are shown with a red border. The inputs can be of either anomaly class, shown with red, green and yellow.

Figure 5.10 shows the dataflow of the anomaly detection modules. Because of the need for a fast decision inside the anomaly detection methods to finish between two exits, the input dimensions into the anomaly detection modules must be selected carefully, and the traced activations need to be properly aligned. Additionally, combining traces from different layers enlarges the input sample dimensions for the anomaly detection methods. Because of this, strategies to combine traces of multiple layers and to potentially reduce the dimensions are part of the configuration method. This results in maximum sample size for an input $s_{i,j}$ into the dimensional reduction of:

$$\max(\texttt{size}(s_{i,j})) = \begin{cases} (\Sigma_{l_k \in \mathcal{L}_i, k \leq i} X_k) \times \max_{l_k \in \mathcal{L}_i, k \leq i}(Y_k) \times \max_{l_k \in \mathcal{L}_i, k \leq i}(Z_k), \\ \qquad \textbf{if } \text{concat along } X \\ \max_{l_k \in \mathcal{L}_i, k \leq i}(X_k) \times (\Sigma_{l_k \in \mathcal{L}_i, k \leq i} Y_m) \times \max_{l_k \in \mathcal{L}_i, k \leq i}(Z_k), \\ \qquad \textbf{if } \text{concat along } Y \\ \max_{l_k \in \mathcal{L}_i, k \leq i}(X_k) \times \max_{l_k \in \mathcal{L}_i, k \leq i}(Y_k) \times (\Sigma_{l_k \in \mathcal{L}_i, k \leq i} Z_k), \\ \qquad \textbf{if } \text{concat along } Z \end{cases} \tag{5.14}$$

where $\mathcal{L}_i$ is the set of layers traced for the anomaly detection module before exit e_i and X_k, Y_k and Z_k are the width, height and depth of the traced ac-

tivation sample at the given layer l_k. As shown, the concatenation of layer traces increases the dimensionality along the given dimension. While this is a maximum value, the actual output size may vary depending on the specific implementation and constraints. For distance based methods, a tensor input shape is required, which necessitates the given shape. In addition to this concatenation, multiple different approaches can be used to reduce the dimensions of the traced activation samples. While more methods to reduce the dimensions, like UMAP [138] exist, this work focuses on the following methods:

Principal component analysis: PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables. These variables are called principal components [50, 139]. By selecting the most significant principal components (i.e. the principal components with the highest variance), the dimensionality of the traced activation samples can be reduced. Simultaneously, the most relevant information is preserved. While the input of the traced activation samples is based on the input samples, PCA allows for a user definition of the output size which is then a list of given values of the specified length.

Concatenation of layer traces: A straightforward method to combine the traced activation samples from multiple layers is to concatenate them along the Z-dimension. Depending on the desired output shape of the resulting tensor, a reshaping operation might be necessary. This resizing is done via padding smaller activation samples with zeros or cropping larger activation samples or a mixture of both. Additionally, resizing via interpolation methods like bilinear or nearest-neighbor interpolation can be used. This leads to an output size as shown in Equation (5.14), where m is the layer with the desired output size.

Summation over channel dimension: Another method to reduce the dimensionality is to sum the values of the traced activation samples over the channel dimension. This results in a single activation sample with reduced depth, where each value represents the sum of the corresponding values across all channels. This leads to an output size of $\sum_k Z_k$.

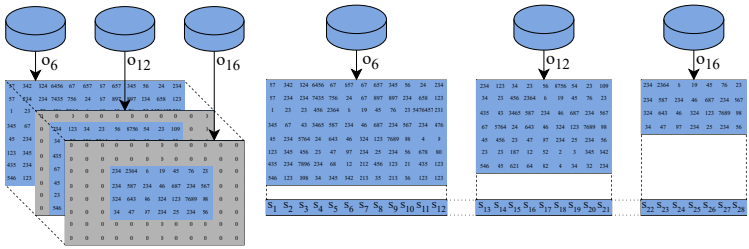
Averaging over channel dimension: Similar to the summation method, averaging the values over the channel dimension can be used to reduce the dimensionality. This results in an activation sample where each value represents the average of the corresponding values across all channels. This also leads to an output size of $\sum_k Z_k$.

Max pooling over channel dimension: Max pooling can be applied over the channel dimension to reduce the dimensionality of the traced activation samples. This method selects the maximum value from each channel for each spatial location, resulting in an activation sample with reduced depth. This also leads to an output size of $\sum_k Z_k$.

Contribution 16: Dimension Matching for AD Modules

Matching the input dimensions of the traced activation samples for the anomaly detection modules is necessary, especially when combining traces from multiple layers. The matching can further be used to reduce the dimensions of the traced activation samples. The presented methods are compared automatically in the presented tool.

Figure 5.11 shows a visualization for the trace concatenation as well as the summation over channel dimensions. In Figure 5.11a, the traces are concatenated along the 3rd dimension. Because the output tensor of later layers have a smaller dimension, these traces are padded with zeros to result in a uniform output tensor. In Figure 5.11b, the values of the same traced activation samples are summed over a channel dimension, resulting in a vector of size $1\times1\times Z_k$, should the summation be in the Z dimension. Afterwards, the resulting vector can be concatenated in the Z dimension, to result in a vector of size $1\times1\times\sum_k Z_k$. The drawn examples reduce the dimensions for clarity, while the actual traced activation samples usually have a tensor shape. The padding is given in gray. Depending on the use case, more complex methods to match the dimensions can be used, like e.g. PCA. In, Section 7.4.3 a detailed evaluation of the different dimension reduction methods is presented.

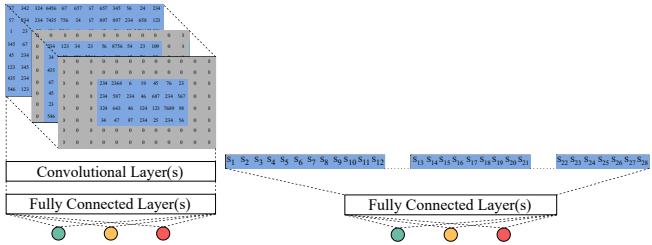


(a) Trace Concatenation (b) Summation over given channel dimension

Figure 5.11: Naive approaches for dimension matching of traced activation samples for anomaly detection. While the traces usually have a tensor shape, the drawn examples reduce this for clarity. Padding is given in gray.

After matching the dimensions to the required inputs, the configuration of these anomaly detection methods is undergone. The configuration of these methods is based on the traced activation samples $\mathcal{S}_i$ and $\mathcal{S}_{i,\text{anomaly}}$ for each exit position l_{e_i}. Because these methods differ between tensor-shaped or single dimensional input data, anomaly detection methods are grouped into methods for tensor-shaped input data, methods for single-dimensional input data or both. The following methods are considered. While there exist more methods to detect anomalies in Anytime neural networks like e.g. VAE [137, 140], the presented methods were chosen for their compatibility with the given framework.

ML-based models: For ML-based models, the traced activation samples are either reshaped into a single-dimensional vector or a tensor shape, depending on the model requirements. The models are then trained on the traced activation samples classifying them into *normal* and *anomalous* classes. If the input shape is tensor-shaped, convolutional neural networks are used. Using convolutional neural networks orients itself on works like [73]. Fully connected networks like [141] are used for vector input shapes.



(a) CNN-based Concept (b) Fully Connected-based Concept

Figure 5.12: ML-based anomaly detection methods. Red signifies anomalous classes, where the execution is terminated. Yellow shows, where execution can continue with a different exit and green signifies classes, where execution is allowed to continue. Input samples are marked in blue.

Figure 5.12 shows the ML-based anomaly detection method. The CNN-based concept employ one or more convolution layers, which are followed by one or more fully connected layers. The fully connected-based concept only uses one or more fully connected layers. Both approaches have 3 output neurons, signifying the classes. Red signifies anomalous classes, where the execution is terminated. Yellow shows, where execution can continue with a different exit and green signifies classes, where execution is allowed to continue with the original exit. Input samples are marked in blue.

Support vector machines: SVM are supervised learning models used for classification and regression analysis. They work by calculating a hyperplane which optimally separates different classes in the feature space [142]. This hyperplane aims to maximize the difference between samples of the given classes and the dividing hyperplane. The samples with the lowest distance to the hyperplane are called support vectors and the number of support vectors can be configured during set-up. For support vector machines, traced activation samples can either be reshaped into single-dimensional vectors or kept in a tensor shape. The SVM is trained on these vectors to classify them into *normal* and *anomalous* classes. Figure 5.13c shows two example planes for a support vector classifier. Again, samples of normal classes are drawn in green, drawn in yellow are anomalies, which lead to a change in exit and red signifies samples from datasets, which lead to a termination of the inference. Additionally, a single class SVM can be used, which only considers the *normal* class and classifies samples as *anomalous* if they are outside a given distance to the normal class. The abortion of the inference can then be signaled via an even bigger distance to the normal class. Both distances must be configured by the user during set-up.

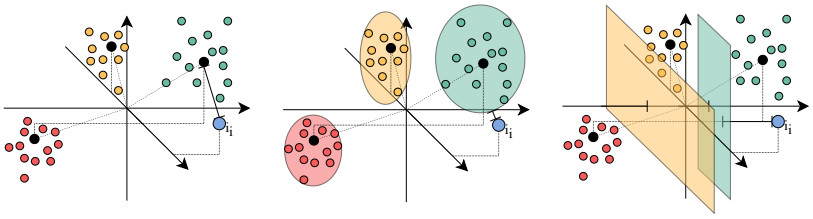
Gaussian naive bayes: GNB is a probabilistic classifier based on applying Bayes' theorem with naive independence assumptions between the features [143]. For GNB, all input dimension size are compatible, and the classifier is trained on the traced activation samples to classify them into *normal* and *anomalous* classes. Should a given sample have a higher probability of being an anomalous class than a normal class, the sample is classified as anomalous, should the sample be more likely to be a normal class, the sample is classified as normal. Lastly, should the sample have the highest probability of being an anomalous class, which leads to a termination of the inference, the inference is aborted.

Euclidean distance-based: Similar to the Mahalanobis distance-based methods, these methods calculate the distance between an input sample and a set of samples taken from the classes. But other than the previous method, the center of the samples from the *normal*, *anomaly* or *abortion* classes is used to calculate the distance, which is given as the mean of the samples. The input sample is then classified based on the distance to this mean of the resulting class. The sample is classified according to the closest class. The input data here can have an arbitrary shape as long as there is a valid distance measurement given. In this context, this classifier can either use a single-dimensional vector or a tensor as the input. Figure 5.13a shows how the input sample is classified according to the distance to the mean of the sample set.

Mahalanobis distance-based: Based on the Mahalanobis distance [112], these methods use the traced activation samples to calculate the distance between an input sample and the distribution of the given classes [74]. In contrast to the Euclidean distance, the Mahalanobis distance considers the distance to a cloud of samples, which is given by the mean and the covariance of the samples. Depending on the distance to the given samples, the input sample is classified as *normal*, *anomalous* or a class to abort the inference, depending on the lowest distance between sample and the given classes. The compatibility of the Mahalanobis distance-based methods depends on the resulting input size after dimension matching. If the resulting input size is equal to $1\times1\times Z$, i.e. the input is a linear vector, this method can be used. Figure 5.12b shows an example of a Mahalanobis distance based classifier, which classifies the given, blue, input sample based on the distance to the sample set.

Entropy-based: Here, a difference between the mean of the classes and the input sample is calculated not by using a distance measurement, but by using the cross-entropy between the input sample and the mean value or the Kullback-Leibler divergence [144] between the two samples. Similarly to the distance-based methods, the input sample is classified based on the lowest cross-entropy or Kullback-Leibler divergence to the mean of the given classes. Should the input sample have the lowest cross-entropy or Kullback-Leibler divergence to the normal class, the sample is classified as *normal*. If the input sample has the lowest cross-entropy or Kullback-Leibler divergence to an anomalous class, the sample is classified as *anomalous*. Lastly, if the input sample has the lowest cross-entropy or Kullback-Leibler divergence to a class signaling the abortion of the inference, the inference is aborted. The input data can have an arbitrary shape as long as there is a valid cross-entropy or Kullback-Leibler divergence measurement given. In this context, this classifier can either use a single-dimensional vector or a tensor as the input.

All anomaly detection methods are compared based on their performance on a validation dataset $\mathcal{S}_{\text{val}}$, which is traced through the ANN using the same methods as for the training dataset. Additionally, the decision time of each method is measured, to ensure that the given method can finish execution before the start of the next exit. A selection of anomaly detection methods can then be made via a multi-objective optimization, considering the detection performance and the decision time. Because the environment considers an embedded environment, used resources are also evaluated. The selected anomaly detection methods are then integrated into the ANN architecture before the respective early exit. Table 5.4 shows the compatibility of the given methods and the resulting choice for ML-based methods. For the



(a) Euclidean Distance (b) Mahalanobis Distance (c) Support Vector Machine

Figure 5.13: Distance-based anomaly detection methods. Red signifies anomalous classes, where the execution is terminated. Yellow shows, where execution can continue with a different exit and green signifies classes, where execution is allowed to continue. Input samples are marked in blue.

Mahalanobis distance, the compatibility depends on the resulting input size after dimension matching. Is the resulting input size a $1 \times 1 \times Z$ sized tensor, i.e. a liner vector, the method is compatible using the dimension matching methods.

Table 5.4: The compatibility of different anomaly detection methods with given dimension matching methods for the ANN architecture.

Anomaly Detection	PCA	Concat	Sum	Average	Max
ML-based	✓^{D}	✓^{C}	✓^{D}	✓^{D}	✓^{D}
SVM-based	✓	✓	✓	✓	✓
GNB-based	✓	✓	✓	✓	✓
Euclidean D.	✓	✓	✓	✓	✓
Mahalanobis D.	✓	✗	✓†	✓†	✓†
Entropy-based	✓	✓	✓	✓	✓

†Only if input is $1 \times 1 \times Z$ sized, ^{D}Fully Connected Network, ^{C}Convolutional Network

Because of the potentially large design space of different anomaly detection methods and dimension matching methods, an automatic configuration and selection is necessary. Figures 5.12 and 5.13 show the concept for the anomaly detection. Using the dimension matching strategies together with the anomaly detection methods, leads to an overall design, which is introduced in Figure 5.10. As Table 5.4 shows, nearly all combinations of dimension matching methods and anomaly detection methods are possible and the selection of the best fitting combination depends on the use case as well as the input data. More importantly, it is necessary to evaluate the performance of using samples

from a combination of layers for the anomaly detection. The activations from one layer might not be sufficient to detect anomalies, while the combination of multiple layers might be more efficient. However, the combination of multiple layers also increases the input size for the anomaly detection methods, which introduces a trade-off between detection performance and decision time and resource usage. A detailed evaluation of different anomaly detection methods and dimension matching methods with different datasets and ANN architectures is given in Section 7.4.3.

Example 20: Exit and Tracing Components added to ResNet-18

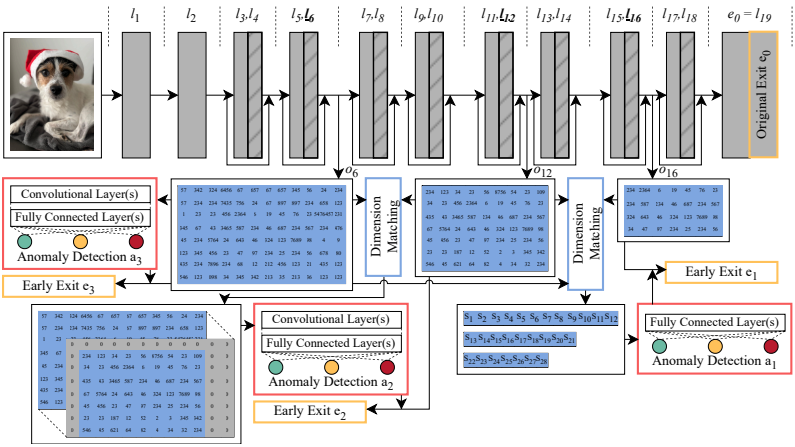


Figure 5.14: Complete ANN adaptation structure for ResNet-18. Original components are marked in gray. Tracing components are marked in blue. Early exits are marked in yellow and anomaly detection components are marked in red.

In Figure 5.14, three exits are added to the ResNet-18 architecture. The earliest exit (index e_3) can be executed after layer l_6 of the original architecture. The traces are fed into the anomaly detection component. Should a normal *sample be detected, the execution continues. If necessary, the execution terminates after executing exit e_3. For e_2 and e_1, a dimension matching strategy is used to also include the traces of previous layers into the decision.*

Example 20 shows the structure of a network with added early exits and anomaly detection modules, where the anomaly detection modules influence the execution of the early exits. Here, one weakness becomes obvious.

During the decision of the anomaly detection module, the next exit is not yet executed, because the result is not clear yet. Furthermore, a runtime anomaly might also influence the system. To handle these problems and to extend the structure with a runtime management component, a scheduler is added. This addition to the system architecture is described in the next chapters.

Contribution 17: Automated AD Selection and Configuration

By automatically comparing and selecting different anomaly detection methods and dimension matching methods, the best fitting combination can be selected for each early exit position. This selection is based on the performance of the methods on a validation dataset as well as the decision time and resource usage.

5.4 Runtime Estimation Strategies

The runtime estimation for the execution time between two exits varies between inferences. This is due to available memory bandwidth as well as differences in input data and potential execution optimizations like quantization [145]. Nevertheless, it is important to provide an estimate for the execution time of a layer to decide on the chosen exit. This chapter includes two parts. First, the a priori estimation of the runtime is presented. This step is undergone during the configuration phase in Section 5.3.1 and Section 5.3, to provide a baseline estimation for the placement of the early exits. Second, this estimation is used during the execution to schedule the execution of early exits of the anytime neural network (ANN). This estimation is updated during the runtime using measurements to reflect the actual execution time of the ANN. The a priori estimation is based on [Sta25b] and the runtime estimation is based on the estimation from [Sta27, Sta25b, Sta23c].

5.4.1 Design-Time Runtime Estimation Strategies

The a priori runtime estimation is performed during the configuration phase of the ANN and makes use of the added tracing components as described in Section 5.3.1 and Section 5.3. The result is the expected runtime for the ANN layers. More importantly it is used to estimate the time between two exits or between an anomaly detector and an exit or an exit and an anomaly detector. Here, the estimation calculates the starting value for the runtime estimation $\hat{t}_C(l_i, u)$ (see Section 5.1) for each layer. A balance needs to be struck between

the worst case estimation and the average case. An overly pessimistic estimation will lead to suboptimal exit selection, because the system will tend to overestimate the required time budget. An overly optimistic estimation will lead to deadline misses, as the actual execution time will exceed the estimated time. Because of this, the estimation must select an appropriate percentile between the worst case and average case to enter as a starting value into $\hat{t}_C$. Since the early exit scheduling follows a catch-up mechanism should one estimation be exceeded, not using the WCET is warranted.

Figure 5.5 also shows an example of the positioning of the tracing components for the a priori runtime estimation. Again, the tracing components are positioned after each original ANN layer which is executed. Additionally, the runtime for every early exit as well as every anomaly detection component is measured during configuration.

Example 21: Early Exit Runtime Measurement for ResNet-18

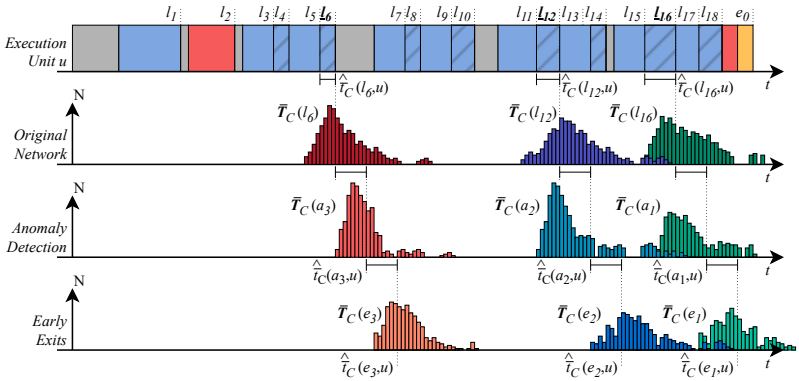


Figure 5.15: Runtime measurements for the layers right before early exits and the corresponding early exit layers and anomaly detection modules.

Figure 5.15 shows a simulation of the runtime measurements for the given ResNet-18 [45] example. The spread in runtime measurements of the measurement set $\overline{T}_C(l_i)$ follows the expected shifted Gaussian distribution for task runtime. In the example, the average of the measurements is used as the a priori estimation $\hat{\bar{t}}_C(l_i)$ for the measurements.

Example 21 shows the measurement sets of the layer finish times $\overline{T}_C(l_i)$ of a network with early exits for the layers l_6, l_{12}, and l_{16}. Additionally, the finish

times are shown for the anomaly detection modules a_3 a_2 and a_1 as well as for the early exits e_1, e_2, and e_3. The anomaly detection modules are using the same intermediate result as the early exit, because they are positioned after the same ANN layer. The measurements from the tracing components are collected in the corresponding sets $\overline{T}_C(l_i)$ for the given execution unit u, and an a priori estimation for the runtime of the given layers is determined. This calculated estimation value is used with the information of the corresponding execution unit u as $\hat{t}_C(l_i,u)$ for the respective task model of the ANN. Additionally, the runtime of the added early exits and anomaly detectors is measured and used to calculate their $\hat{t}_C(l_i,u)$ values respectively. The values of Figure 5.15 are simulated and exaggerated for illustration.

While the anomaly detection modules and early exits are executed in sequence in this example, an improvement is possible, should the anomaly detection module not use the same layer output as the early exit. In this case, the anomaly detection module could be executed in parallel. If only one execution unit is available for both components (early exit and anomaly detection), this would have no direct effect. In all cases this a priori estimation is calculated with:

$$\hat{t}_C(l_i,u) = \frac{\sum_j^{|T_C(l_i,u)|} w_j}{|T_C(l_i,u)|} \sum_{\substack{t_j \in T_C(l_i,u) \\ \phi_i(j)=u}} w_j \cdot t_j, \tag{5.15}$$

where $t_j \in T_C(l_i,u)$ signifies the j^{th} element in the set of all measured execution times for layer l_i and execution unit u. The measured execution times need to be sorted in ascending order before calculating the weighted average, i.e. have an order $\leq$, such that $t_1 \leq t_2 \leq ... \leq t_{|T_C(l_i,u)|}$. Additionally, the chosen execution unit for the given layer is denoted by $\phi_i(j)$. This is included to keep the estimated execution time consistent with the scheduled execution unit. The weights are then selected to reflect the desired type average.

The value at the 80$^{\text{th}}$ percentile is selected, by setting $w_j = 1$ for all t_j smaller than the 80$^{\text{th}}$ percentile and $w_j = 0$ for all t_j greater than the 80$^{\text{th}}$ percentile. Because of the normalization of the weights, each weight can have an arbitrary value and is not bounded to $\sum_{w_j} = 1$.

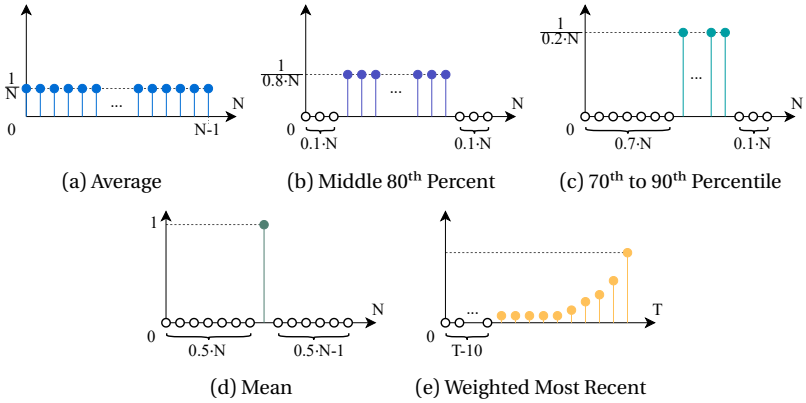
5.4.2 Online Runtime Estimation During Execution

The execution time estimation of a neural network layer is updated after each execution of the layer. This is based on the measured execution time of the layer. The estimation is updated after an input sample has been processed by the layer and before the layer executes the next time. For this, the measured execution time is added to the set of all measured execution times for the given layer l_i, denoted as $T_C(l_i, u)$. After adding the measured execution time to the set, the estimated execution time $\hat{t}_C(l_i, u)$ is updated to reflect the new set of measured execution times. This can be undergone after all layers have been executed for one inference. By being adaptable using different weights, the estimated execution time represents a configurable weighted average of the measured execution times. This is calculated with:

$$\hat{t}_C(l_i, u) = \frac{\sum_j^{|T_C(l_i,u)|} w_j}{|T_C(l_i, u)|} \sum_{\substack{t_j \in T_C(l_i,u) \\ \phi_i(j)=u}} w_j \cdot t_j, \tag{5.16}$$

where w_j signifies the weight of the measured execution time t_j in the set $T_C(l_i, u)$. Here, we define the order of two measurements $<$ as if one measurement is more recent than another, i.e. $t_i < t_j$ if t_i was measured before t_j. While the formula is identical to the a priori measurement, the ordering of $T_C(l_i, u)$ is reconsidered. The weights can be chosen to reflect a desired responsiveness of the estimation. A higher weight for more recent measurements will lead to an estimation which takes recent changes more into account, while a uniform weight will lead to a more stable estimation. The choice of weights can be adapted based on the specific requirements of the application and the characteristics of the neural network being used. As above, the normalization of the weights leads to each weight being able to have an arbitrary value and not being bounded to $\sum_{w_j} = 1$, while still allowing a normalized average.

Figure 5.16 shows the considered weight distributions for the runtime estimations. Subfigure (a) shows a uniform weight distribution, which leads to an average of all measured execution times. For this each weight is set to $w_j = 1/|T_C(l_i, u)|$. Subfigure (b) shows a weight distribution that leads to an average of the middle 80$^{\text{th}}$ percent of the measured execution times. For this, each weight is set to $w_j = 1/(0.8 \cdot |T_C(l_i, u)|)$ for all measurements $t_c(l_i, u)$ between the 10$^{\text{th}}$ and 90$^{\text{th}}$ percentile and $w_j = 0$ for all other measurements.



(a) Average (b) Middle 80th Percent (c) 70th to 90th Percentile

(d) Mean (e) Weighted Most Recent

Figure 5.16: Weights for runtime estimation, where subfigure (a) to (d) expect the samples to be sorted by value from lowest to highest. Subfigure (e) expects the samples to be sorted by recency, with the most recent sample being the last one.

Subfigure (c) shows a weight distribution that leads to an average of the 70th to 90th percentile of the measured execution times. For this, each weight is set to $w_j = 1/(0.2 \cdot |T_C(l_i, u)|)$ for all measurements $t_c(l_i, u)$ between the 70th and 90th percentile and $w_j = 0$ for all other measurements. The last subfigure to determine an initial runtime estimation is given with subfigure (d), where the middle weight is given with $w_j = 1$ and all other weights are set to 0. This leads to the median of the measured execution times. Subfigure (e) shows a weight distribution, which is used to update the runtime estimation during execution.

Here the most recent measurement is given the highest weight and a decrease depending on the recency of the measurement. How this decrease is defined can be adapted to the given use case. The argument for this lies in a potential long term change of execution time, which is used to adapt the estimation. While the figures sum the weighs to 1 in every case, due to Equation (5.15) and Equation (5.16), the weights can be scaled by any factor, because the sum of the weighs will be canceled out by the equations.

A detailed evaluation of these weight distributions and their effect on the error between runtime estimation and actual measured runtime is given in Section 7.1.1 and Section 7.1.2.

5.5 Added Checkpoints for Mid-Term Scheduling of ANNs

After deciding the positioning of the early exits in Section 5.2.1, the training of the early exits in Section 5.2.2, the addition of anomaly detectors in Section 5.3 and the runtime estimation Section 5.4, the final structure of the anytime neural network (ANN) combines these components and adds a checkpoint structure to control the periodic execution of the ANNs with the added early exits.

5.5.1 Component Structure of the Checkpoints

This checkpoint executes the decision-making for the early exits and takes into account schedules from more abstracted components and anomaly detection from the corresponding anomaly detectors. The checkpoint structure is based on works for homogeneous execution units [Sta27] and heterogeneous execution units [Sta25b]. One checkpoint controls the execution from one early exit. It is composed of one component to measure runtime, the anomaly detector and an input from a scheduling component. This input is the given runtime budget and controls the time until which the execution of the next exit has to complete. One checkpoint is positioned at each layer directly before an early exit. The checkpoint structure is executed after the execution of the layer before the early exit.

Figure 5.17 shows the structure of one checkpoint in the ANN. After executing the last original network layer before the early exit, the runtime measurement component uses the measured execution time to update the estimations as described in Section 5.4.2. The scheduling component gives the checkpoint a maximum time until the next exit e_x has to be completed, which is denoted as the timing budget. With this timing budget, the checkpoint component decides whether to use the early exit or continue execution in the original ANN. Additionally, the output of the anomaly detector is taken into account to decide whether to use the early exit or continue execution in the original ANN. If the early exit is chosen, then the output of the current layer is forwarded to the associated early exit and the early exit is executed. The result from this early exit is then considered as the network output. If the continuation of the original ANN is chosen, the output of the current layer is forwarded to the next layer in the original ANN. In both cases, the runtime measurements for early exit or the original ANN are updated.

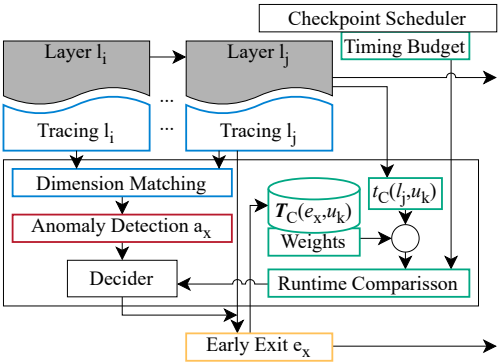


Figure 5.17: Structure of one checkpoint in the ANN. Original network layers are shown in gray. Early Exits are shown in yellow. The runtime measurement components are shown in green and the anomaly detection modules are shown in red.

5.5.2 Mid-Term Scheduling Algorithm for Exit Choice

The decision-making component decides whether to use the current early exit e_x or continue execution in the original ANN. Three cases trigger the choice of the early exit. The easiest case is when the estimated execution time of the remaining layers in the original ANN exceeds the time budget given from the scheduling component. The estimation is based on the past measurements of the exit as well as a safety factor c, used to handle changes in the execution time during the current inference and calculated as:

$$c \geq \frac{\max(\hat{\bar{t}}_C(l_j))}{\max(\bar{t}_C(l_j))} \tag{5.17}$$

Other fixed values are possible and evaluated in Section 7.1.1. The expected layer finish time for the next exit e_{x-1} is calculated as:

$$\hat{\bar{t}}_C(e_{x-1}) = \bar{t}_C(l_{e_x}) + c \cdot \sum_{\substack{l_j \in \mathcal{L}, \\ l_{e_x} < l_j \leq l_{e_{x-1}}}} \bar{t}_C(l_j, u_k) + c \cdot \bar{t}(e_{x-1}, u_k), \tag{5.18}$$

where u_k is the execution unit on which the layers are executed. The estimation itself is based on the past measurements of the layers and the exit multiplied by the expected safety value c. If the remaining estimated execution time exceeds the time budget, i.e., $\hat{\bar{t}}_C(e_{x-1}) > \bar{t}_B(e_{x-1})$, the early exit is chosen. If the given time budget is high enough, the execution continues in the original ANN. In any case, the runtime estimation of the executed layers are updated. Additionally, the runtime estimation for the early exit is updated should it be executed.

The output from the layers, which where selected as an input to the anomaly detector associated with the checkpoint, can be directly forwarded to this anomaly detector, which executes in parallel to the remaining ANN layers. Should the anomaly detector classify the output as inside the original operational design domain (ODD), the execution continues in the original ANN. If the anomaly detector classifies the output as outside the original ODD, an early exit is chosen. An exit is chosen depending on the result from the anomaly detector. This could either be an early exit with the same class abstraction as the original exit or an early exit with a higher class abstraction. Additionally, a completely different set of output classes might be considered as well as the abortion of the execution. In any case, the runtime estimations are updated depending on the executed components.

Table 5.5: Definitions added for the neural network task scheduler.

Component	Formula	Description
Relative Time Budget	$t_B(e_x) \leq D(\tau_i)$	Time budget assigned by the scheduler for each exit e_x.
Absolute Time Budget	$\bar{t}_B(\tau_i, e_x) = t_R(\tau_i) + t_B(e_x)$	Absolute time budget assigned for an exit e_x, equal to the release time plus the relative time budget.

Table 5.5 shows the definition for the given timing budget. The scheduler assigns a relative time budget $t_B(e_x)$ for each exit e_x based on the given deadline of the original task $D(\tau_i)$. The absolute time budget $\bar{t}_B(\tau_i, e_x)$ resembles an absolute deadline for the given exit e_x and is calculated by adding the relative time budget to the release time of the task $t_R(\tau_i)$. The decision-making component in the checkpoint structure uses the absolute time budget to decide whether to use the early exit or continue execution in the original ANN.

5.6 Scheduling Algorithm for ANNs with Checkpoints

Scheduling the execution of anytime neural networks (ANNs) with added early exit and added anomaly detection components requires making use of the checkpoint structure described in Section 5.5. Because the checkpoint structure includes all components to make a decision on the running inference to either split of from the main data path of the ANN and use an early exit or continue execution in the original ANN, only a maximum time budget is given to the checkpoint component. Nevertheless, a scheduling method is required to provide this time budget to the checkpoint structure and to schedule multiple anytime neural networks on the same Heterogeneous System-on-Chip (HESoC) system. This section describes a scheduling method for anytime neural networks based on the checkpoint structure. The checkpoint scheduler can be used as a standalone scheduling method for anytime neural networks on HESoC systems, as well as integrated into the combined scheduler presented in Chapter 4 and Chapter 6. The scheduler can be interpreted as a preemptive earliest deadline first (EDF) scheduler with fixed priorities. The optimization goal of the scheduler is to minimize the choice of early exits, while still meeting task deadlines.

The scheduler works on a set of tasks $\mathcal{T}$ as defined in Section 5.1. These tasks are assumed to be released ready. This includes the input sample $s_{i,-1}$, the assignment of ANN α_i and the given priority σ_i. Since the class of the input data is unknown, the class is not specified. The scheduler works iteratively with two nested loops. The first loop is called the task release loop, is presented in Section 5.6.1 and executes every time a new task is released. The second loop is called the checkpoint loop, is presented in Section 5.6.2 and is executed once per early exit.

5.6.1 Task Release Loop as the Mid-Term Scheduling

Every task is inserted into an input queue upon release. The release time is given by $t_R(\tau_i)$. This ready queue $\mathcal{T}_R \subseteq \mathcal{T}$ is sorted by the absolute deadlines of the inserted tasks $D(\tau_i)$, where an earlier release time is positioned earlier in the queue. The relation, given as $<$ of two elements in the ready queue $\tau_i, \tau_j \in \mathcal{T}_R$ is thus defined as if $D(\tau_i) < D(\tau_j)$, then $\tau_i < \tau_j$. In case two tasks have the same absolute deadline, the task with the higher priority σ is positioned

earlier in the queue. The relation is thus extended as if $D(\tau_i) = D(\tau_j)$ and $\sigma_i > \sigma_j$, then $\tau_i < \tau_j$.

The scheduler also has access to the list of execution units $\mathcal{U}$. As soon as an execution unit becomes idle, this scheduling function is executed. This coincides with the finishing of the exit of the now completed task. Should no task be available for execution, the scheduler will wait until the next task is released. Since the tasks in the input queue are sorted by their absolute deadlines, the first element has the earliest deadline.

The first task in the queue τ_0 is selected for execution. The scheduler then checks all execution units $u_j \in \mathcal{U}$ to find the earliest start time $t_S(\tau_0)$, where the task τ_0 can be completely executed within its deadline $D(\tau_0)$. This takes the estimated execution time of the task into account as $t_S(\tau_0) + \hat{t}_C(e_0, u_j) \leq D(\tau_0)$, where $e_0 = \max(\alpha_0)$. Should there be multiple available execution units, then the available unit resulting in the earliest completion time is selected. The selected execution unit u_{select} and the earliest start time is then calculated as thus calculated as:

$$u_{\text{select}} = \arg\min_{u_j \in \mathcal{U}}(t_{avbl}(u_j) + \hat{t}_C(e_0, u_j)), \tag{5.19}$$

$$t_S(\tau_0) = \min_{u_j \in \mathcal{U}}(t_{avbl}(u_j) + \hat{t}_C(e_0, u_j)), \tag{5.20}$$

where $t_{avbl}(u_j)$ gives the time when the execution unit u_j becomes available and $\hat{t}_C(e_0, u_j)$ gives the total estimated execution time of the task τ_0 on the execution unit u_j until the latest exit $e_0 = \max(\alpha_0)$. The task is then assigned a time budget equal to the estimated execution time until its original exit on the selected execution unit $\bar{t}_B(\tau_i, e_0)$, where $e_0 = \max(\alpha(\tau_0))$ signifies the original output layer of the assigned network for task τ_0. The time budget thus signifies an internal deadline given by the scheduler. While the relative timing budget is defined as relative to the task start time t_S, the absolute time budget $\bar{t}_B(e_0, \tau_0) = t_R(\tau_0) + t_B(e_0)$ orients itself on the absolute deadline $D(\tau_0)$ and signifies the deadline given by the scheduler. See Table 5.5 for an overview of these added definitions for the checkpoint scheduler and Figure 5.18 for a visual representation of the task release loop. Each execution unit then has a given execution queue $\mathcal{T}_{u_j}$ built from the assigned tasks.

The early exits come into play, should no unit be found, where the task finishes execution until its original exit e_0. In this case, the scheduler iteratively reduces the time budget of every scheduled task prior and including to the currently

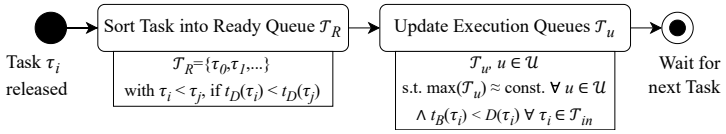


Figure 5.18: Diagram, showing an overview of the task release loop. Once a task is released, the scheduler sorts the task into the input queue and subsequently updates the execution queues for all execution units.

considered task. This results giving a timing budget to earlier exits e_1. Starting with the lowest priority σ_{min}, the timing budget is reduced iteratively by the difference of the two latest used exits e_i, e_{i-1} for every task with this priority. This means, that each task scheduled prior to τ_0 has its scheduled exit moved to the next earlier exit e_{i-1}. Should this not suffice, the next higher priority $\sigma_{min}+1$ is considered and the time budgets of all tasks with this priority are reduced as well, until the priority of the currently considered task τ_0 is reached. Then task τ_0 has its time budget reduced as well. Afterward, this process is repeated until the earliest exit is reached for all tasks scheduled prior to τ_0 with priorities up to and including the priority of task τ_0.

This continues for all tasks in the ready queue $\mathcal{T}_R$. Should a task further down the queue necessitate a time budget reduction of prior tasks, the same process is executed again. This ensures that all tasks are scheduled to meet their deadlines, while minimizing the confidence loss due to early exits. In all cases an allowed slack c can be given to each timing budget to allow for some leeway, should the estimations be too optimistic.

5.6.2 Checkpoint Loop as the Short-Term Scheduler

The given timing budget thus sets the constraints for the execution of each set of layers between two exits. This timing budget is then used inside the checkpoint component to try to ensure a timely execution. Every time a choice between executing an early exit or continuing with the original ANN is made, the checkpoint loop is executed. This means that every time a checkpoint for a given early exit e_x is reached during the execution of a task τ_i, the checkpoint structure is executed as described in Section 5.5. Here, the decision-making component uses the available time budgets $t_B(e_x)$ and more importantly $t_B(e_{x-1})$ assigned by the scheduler as a maximum time available for the given early exit. If the next exit e_{x-1} is reachable within the given budget, the execution continues in the original ANN. The current exit is chosen, if

for the current checkpoint with the index x:

$$c \cdot \hat{\bar{t}}_C(e_{n-1}) > \bar{t}_B(\tau_i, e_{n-1}), \tag{5.21}$$

where e_{n-1} signifies the next exit in the ANN, c gives an allowed slack and $\bar{t}_B(\tau_i, e_{n-1})$ gives the timing budget for the next exit since the start of the task execution. Should the estimated execution time until after the next exit exceed the given time budget, then the early exit associated with the current checkpoint e_x is chosen. In both cases, the runtime estimations are updated as described in Section 5.4.2 depending on the executed components.

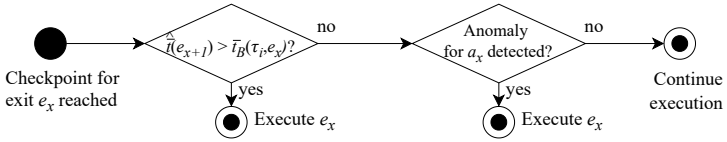


Figure 5.19: Checkpoint loop for the exit scheduler. Two checks are undergone in succession. If the estimated execution time until after the next exit exceeds the given time budget, the early exit is chosen. If the anomaly detector classifies the output as outside the original operational design domain (ODD), an early exit is chosen.

The checkpoint loop also includes the anomaly detection component. This anomaly detection module is denoted with a_x, if it is associated with the checkpoint of exit e_x. Should the output of the anomaly detection module, associated with the given checkpoint, classify the output as an *anomaly*, the early exit is chosen as well. This results in a checkpoint loop which includes two checks, evaluated sequentially. Because an anomaly is expected to be rarer than an execution time overrun, the timing check is executed first. This ensures that the anomaly detection component is only executed when the timing check is passed, aiming to reduce the overhead of executing the anomaly detection modules in case of tight timing budgets. Figure 5.19 shows the checkpoint loop execution, with the two sequential checks. An evaluation of the checkpoint loop is given in Section 7.5.

Contribution 18: Dynamic Scheduler for ANNs

The presented scheduler allows for automated scheduling of anytime neural networks and their associated early exits on the basis of runtime estimations, while minimizing the cost for choosing early exits. This scheduler can be used as a standalone scheduling method for ANNs on HESoC systems. The approach can also be integrated into the combined time-triggered scheduler.

Summary

Chapter 5 introduced the scheduling of anytime neural networks. This includes the addition of early exits and anomaly detection methods as well as the runtime estimation and checkpoint structure. This work is based on the first-author publications [Sta27, Sta25b, Sta23a, Sta24b, Sta23c]. Section 5.1 introduced the model and Section 5.2 introduced a method to add early exits to existing neural networks. Section 5.3 presented an approach to add anomaly detection methods to neural networks. The runtime bounding method is described in Section 5.4, while the checkpoint structure for the execution of anytime neural networks on HESoC is given in Section 5.5. Lastly, Section 5.6 describes the scheduler for the checkpoint structure, representing a standalone scheduling method for ANNs. The content of Chapter 4 and this chapter are developed independently. How a combination is achieved, and the methods introduced in Chapter 5 integrate into the combined scheduler is presented in Chapter 6.

This page intentionally left blank

Chapter 6

System Integration of Time-Triggered and ANN Scheduling

This chapter presents how anytime neural networks (ANNs) are integrated into the scheduling components presented in Chapter 4 and how the schedules are deployed onto the execution platforms.

Section 6.1 shows how an ANN is represented in the TTS model. This includes execution time, precedence constraints, execution units and multiple versions. Section 6.2 shows how the mode change mechanism is adapted to support the choice for different early exits inside scheduled ANNs and how the integration of anomaly detection models for the time-triggered scheduling (TTS) part is facilitated. Lastly, Section 6.3 presents how the combined scheduling approach is deployed on target platforms.

6.1 Model Representation of Anytime Neural Networks

Integrating the anytime neural network (ANN) scheduler and early exit model, which is presented in Section 5.1, with the scheduling model, which is presented in Section 4.1, necessitates adapting the representation of anytime neural networks to fit within the time-triggered scheduling (TTS) system. For this, the given ANN $\mathcal{L}_{\mathcal{N}}$ is turned into a set of tasks with corresponding precedence constraints. These precedence constraints are used provide an ordering, aiming to ultimately lead to the correct execution of the ANN. For this, three types of tasks are created. The first type represents the layers of the original neural network without any exits or anomaly detection modules. This task set is denoted as $\mathcal{T}_l$ and the elements inside are defined with:

$$\mathcal{T}_l = \{\tau_{l.x} \mid x \in [0, N_e)\}, \tag{6.1}$$

where each layer task $\tau_{l.x}$ represents all layers before early exits e_x, with e_0 being the last exit layer, i.e. the original network exit and e_{N_e-1} being the first early exit. Secondly, the anomaly detection modules are modeled with the task set $\mathcal{T}_a$, which is defined with:

$$\mathcal{T}_a = \{\tau_{a.x} \mid x \in [0, N_e)\}, \tag{6.2}$$

where each anomaly detection task $\tau_{a.x}$ represents the anomaly detection module associated with the early exit layer e_x. Lastly, the different exits are defined as a single task with multiple versions, resulting in the task set for the exits only containing one task τ_e defined as:

$$\mathcal{T}_e = \{\tau_e\}, \tag{6.3}$$

where the task τ_e represents all exits and the choice of the exit is defined with different versions. This results in $N_V \geq N_e$ versions. The characteristics of the chosen exit is defined via the version-dependant model components below.

Every task is assigned an execution time C_i^v, equal to the estimated execution time of the corresponding layer, early exit or anomaly detection module. Since each ANN might be executed on different execution units, multiple task versions are supported by the model. The execution time of the layer tasks $\mathcal{T}_l$ are thus calculated as:

$$C_{l.x}^v = \begin{cases} \sum_{l_j \in \mathcal{L}_i, e_{x+1} < l_j < e_x} \bar{t}_C(l_j, u), \textbf{ if } x \neq N_e - 1 \\ \sum_{l_j \in \mathcal{L}_i, l_j < e_x} \hat{t}_C(l_j, u), \textbf{ otherwise}, \end{cases} \tag{6.4}$$

with $u \in U_{l.x}^v$ being the chosen execution unit for this task version, $\bar{t}_C(l_j, u_v)$ is the estimated execution time of layer l_j on the execution unit u and each layer l_j in the set of original layers $\mathcal{L}_i$ lies between the earlier exit e_{x+1} and

the exit afterwards e_x. This is viable for the given task version v. For the exit task τ_e, the execution time depends on the given version and is defined as:

$$C_e^{x.v} = \hat{t}_C(e_x, u), \tag{6.5}$$

where task version 0 represents the original exit and has been assigned the original exit's execution time with $C_e^0 = \hat{t}_C(e_0, u)$ and the exit of exit 1 is given with $C_e^1 = \hat{t}_C(e_1, u)$ and so on. The execution unit u is given for $u \in U_e^{x.v}$. Should an exit be available for execution on multiple execution units, multiple versions of the exit task exist. The corresponding execution times are then defined for each version and the notation is given by $C_e^{x.v}$, where x is the exit index and v is the version index for the given exit. Lastly, the execution time of the anomaly detection tasks $\mathcal{T}_a$ are given by:

$$C_{a.x}^v = \hat{t}_C(a_x, u), \tag{6.6}$$

where a_x is the anomaly detection module associated with exit e_x and $u \in U_{a.x}^v$ is the chosen execution unit for this task version.

In this work, the target platform is a heterogeneous system-on-chip (HESoC) with integrated accelerators for machine learning (ML)-based tasks. Because of this, dedicated load and store operations are required to load data into local scratchpad memory and provide a result after execution, should that exist on the given platform. Additional tasks are required to model this, resulting in the necessity to model the scratchpad as a shared resource $r_{spm} \in \mathcal{R}$, with the corresponding load and store tasks τ_{ld} as well as τ_{str}. Loading data from main memory into the local scratchpad memory is then modeled through setting the access phases accordingly. For the load task τ_{ld}, the access phases towards the scratchpad r_{spm} and main memory r_{mem} are set to:

$$A_{ld}^v(r_{mem}) = \{(t_{ld,ms}, t_{ld,me})\},\ A_{ld}^v(r_{spm}) = \{(t_{ld,sps}, t_{ld,spe})\}, \tag{6.7}$$

where $t_{ld,ms}$ and $t_{ld,me}$ are the start and end times for accessing main memory, while $t_{ld,sps}$ and $t_{ld,spe}$ are the start and end times for accessing the scratchpad memory. Similarly, for the store task τ_{str}, the access phases are set to:

$$A_{str}^v(r_{spm}) = \{(t_{str,sps}, t_{str,spe})\},\ A_{str}^v(r_{mem}) = \{(t_{str,ms}, t_{str,me})\}, \tag{6.8}$$

where $t_{str,sps}$ and $t_{str,spe}$ are the start and end times for accessing the scratchpad memory, while $t_{str,ms}$ and $t_{str,me}$ are the start and end times for accessing main memory. The memory bandwidth value B for these access phases needs to be set to a value, which represents the required memory bandwidth. In this case, a memory bandwidth of 100% is chosen to model full utilization of the memory bus during these operations, resulting in

$$B^{\upsilon}_{ld}(r_{mem}) = \{(1)\}, B^{\upsilon}_{ld}(r_{spm}) = \{(1)\} \tag{6.9}$$

$$B^{\upsilon}_{str}(r_{spm}) = \{(1)\}, B^{\upsilon}_{str}(r_{mem}) = \{(1)\}. \tag{6.10}$$

The execution times for these load and store tasks are estimated based on the data size to be transferred and the bandwidth of the memory bus and are assumed to be know during design time. As for other tasks, these are then set with C^{υ}_{ld} and C^{υ}_{str} for the load and store task respectively. Similar to the load and store tasks, the task sets $\mathcal{T}_l$, $\mathcal{T}_e$ and $\mathcal{T}_a$ also have assigned access phases. These are limited to the scratchpad memory r_{spm}, for the execution on scratchpad-based accelerators, since they are executed entirely in the context of the ML accelerator. The access phases for these tasks are thus set to for the layer tasks $\mathcal{T}_l$ and anomaly detection tasks $\mathcal{T}_a$ to:

$$A^{\upsilon}_{l.x}(r_{spm}) = \{(t_{i,sps}, t_{i,spe})\},\ A^{\upsilon}_{a.x}(r_{spm}) = \{(t_{i,sps}, t_{i,spe})\}, \tag{6.11}$$

where $t_{i,sps}$ and $t_{i,spe}$ are the start and end times for accessing the scratchpad memory during the execution of task $\tau_{l.x} \in \mathcal{T}_l$ and $\tau_{a.x} \in \mathcal{T}_a$. The corresponding bandwidth is again set to full utilization:

$$B^{\upsilon}_{l.x}(r_{spm}) = \{(1)\},\ B^{\upsilon}_{a.x}(r_{spm}) = \{(1)\}. \tag{6.12}$$

For the exit task τ_e, the access phases depend on the given version, since different versions represent different exits, which might have different output data size or use different data sizes for the input, leading to the definition of different access phases and memory bandwidth values as:

$$A^{x.\upsilon}_{e}(r_{spm}) = \{(t^{x.\upsilon}_{e,sps}, t^{x.\upsilon}_{e,spe})\}, \tag{6.13}$$

$$B^{x.\upsilon}_{e}(r_{spm}) = \{(1)\}, \tag{6.14}$$

where $t^{x.\upsilon}_{e,sps}$ and $t^{x.\upsilon}_{e,spe}$ are the start and end times for accessing the scratchpad memory of the exit task τ_e which represents the exit e_x. Again, different task versions are represented by υ.

Lastly, the order of the generated tasks and early exits are defined through precedence constraints. For the definition, the layer tasks are ordered sequentially, with each layer task $\tau_{l,x}$ preceding the next layer task $\tau_{l,x-1}$. Additionally, each layer task $\tau_{l,x}$ is required to precede the exit task τ_e with the corresponding task version $\upsilon = x$. The anomaly detection tasks $\tau_{a,x}$ are also necessitated by the corresponding exit task $\tau_e, \upsilon = x$, depending on the task version. This results in the following precedence constraints.

Starting with the layer tasks, the precedence constraints are simply defined as:

$$P^{\upsilon}_{l.x}(\tau_{l.x+1}) = \{(-\infty, -C^{\upsilon}_{l.x+1})\} \tag{6.15}$$

$$\dots \tag{6.16}$$

$$P^{\upsilon}_{l,max-1}(\tau_{l.max}) = \{(-\infty, -C^{\upsilon}_{l.x+1})\}, \tag{6.17}$$

where x is the index of the exit, which follows after the layer task $\tau_{l,x}$ and max is the index of the very first layer task. Layer task $\tau_{l,x+1}$ needs to be finished before the next layer task can start, which is ensured by requiring it to start at least $-C^{\upsilon}_{l.x+1}$ before the start of the task with index x.

Additionally, the first layer task $\tau_{l.0}$ is required to be preceded by the load task τ_{ld}, which loads the input data into the scratchpad memory, with the following precedence constraint:

$$P^{\upsilon}_{l.0}(\tau_{ld}) = \{(-\infty, -C^{\upsilon}_{ld})\}, \tag{6.18}$$

where the load task τ_{ld} needs to be finished before the first layer task $\tau_{l.0}$ can start, which is ensured by requiring it to start at least $-C^{\upsilon}_{ld}$ before the start of the first layer task $\tau_{l.0}$.

The TTS scheduler uses the task version of the exit task τ_e to determine the used exit of the scheduled ANN. Because of this, the precedence constraints of the exit tasks are defined with respect to the task version, resulting in the following precedence constraints for the exit tasks:

$$P^{x.\upsilon}_{e}(\tau_{l.x}) = \{(-\infty, -C^{\upsilon}_{l,x})\} \tag{6.19}$$

$$\dots \tag{6.20}$$

$$P^{x.\upsilon}_{e}(\tau_{l.max}) = \{(-\infty, -C^{\upsilon}_{l.max})\}, \tag{6.21}$$

where the **task index** e stays constant for all exits. The **task version** is given by the chosen exit e_x and additionally other versions might exist, which are shown with υ.

Additionally, the precedence constraints for the anomaly detection tasks need to be considered to ensure that a result is available before the decision for the corresponding exit and that the scheduler can react to the result of the anomaly detection task. This is done with setting the precedence constraint of the anomaly detection task $\tau_{a.x}$, which influences exit e_X to:

$$P^{\upsilon}_{l.x}(\tau_{a.x}) = \{(t_{l.ad.x.s}, t_{l.ad.x.e})\}, \tag{6.22}$$

where $t_{l.ad.x.s}$ signifies the earliest start of the anomaly detection task and $t_{l.ad.x.e}$ signifies the latest start of the anomaly detection task in respect to a job of the layer task $\tau_{l.x}$. This allows for the modeling of the anomaly detection task to start after some intermediate layers, which might provide an input to said anomaly detection $\tau_{a.x}$ task prior to ending the layer task $\tau_{l.x}$.

The anomaly detection task $\tau_{a.x}$ also needs to access the scratchpad memory to load the given data and to write back a result, which is used for the decision of the exit task τ_e with the corresponding version. This is done by setting the access window accordingly as:

$$A^{\upsilon}_{a.x}(r_{spm}) = \{(t_{a.spe}, t_{a.spe})\}, B^{\upsilon}_{a.x}(r_{spm}) = \{(1)\}, \tag{6.23}$$

where $t_{a.spe}$ and $t_{a.spe}$ are the start and end time for accessing the scratchpad memory during the execution of the anomaly detection task $\tau_{a.x}$. Depending on the given anomaly detection module, this task needs to access the scratchpad memory r_{spm} or the main memory, resulting in different access phases. Should the anomaly detection task $\tau_{a.x}$ require an additional input of an intermediate result of a layer inside a given layer task, additional access windows in the corresponding layer tasks are required to model this.

Lastly, the result needs to be written back to the main memory. This is modeled through the store task τ_{str}, which is required by a precedence constraint from the exit task τ_e:

$$P^{x.\upsilon}_{e}(\tau_{str}) = \{(C^{x.\upsilon}_{e}, D(\tau_e))\}, \tag{6.24}$$

where $D(\tau_e)$ is the deadline of the exit task τ_e and the store task τ_{str} needs to finish before the exit task's deadline.

Creating TTS schedules which include an ANN executed until a specific early exit thus requires the definition of the desired task version inside the mode

Table 6.1: Translation of Model Components between ANN and TTS parts.

ANN Component	TTS Translation	Description
Layers between e_x and e_{x+1}	$\tau_{l.x} \in \mathcal{T}_l$	Layer tasks representing the layers of the neural network.
Exit e_x	τ_e, with $v = x$	Single exit task with multiple versions representing each possible exit.
Anomaly Detection a_x	$\tau_{a.x} \in \mathcal{T}_a$	Anomaly detection tasks representing each anomaly detection module.
Load/Store Operations	τ_{ld}, τ_{str}	Loading and storing task for the scratch-pad memory.
Execution Time	$C_i^v = \sum \bar{t}_C(l_j, u_v)$	Estimated execution time of task τ_i^v on unit u_v.
Precedence Constraints	$P_e^{x.v}(\tau_{l.v})$, $P_{l.x}^v(\tau_{a.x})$	Constraints ensuring correct execution order between layer tasks, exits, and anomaly detection.

specification. The corresponding layer tasks are then required to be scheduled starting with the last layer task before the desired early exit, because they are required by a given precedence constraint. Because the precedence constraints originate from the task version of the exit task, the scheduling algorithm is able to omit scheduling later layer tasks and anomaly detection tasks. Creating schedules for the given early exit is facilitated by adding the exit index x to the list of viable versions V_μ for the exit task τ_e inside the mode specification (see Section 4.3).

Contribution 19: Model Representation of ANNs

The model representation for anytime neural networks allows for the modeling of the layers, early exits and anomaly detection modules as tasks with precedence constraints. This model representation allows for the integration of anytime neural networks into the TTS scheduling system, enabling the creation of schedules which include the execution of anytime neural networks until different early exits, with each exit being modeled as a separate version of the same task.

As shown in Example 22, the presented model representation allows for the modeling of anytime neural networks until different exits. This includes the modeling of the these exits and anomaly detection components as well as their runtime and precedence constraints. To model the execution of a given ANN until a specific early exit, the corresponding exit is marked with a deadline in the mode specification and the corresponding version is selected accordingly.

Additionally, the load and store operations, as introduced above, also need to be scheduled in the model. This is done in Example 23 for the same network structure.

Example 22: Adapting ResNet-18-based ANN into Scheduling Model

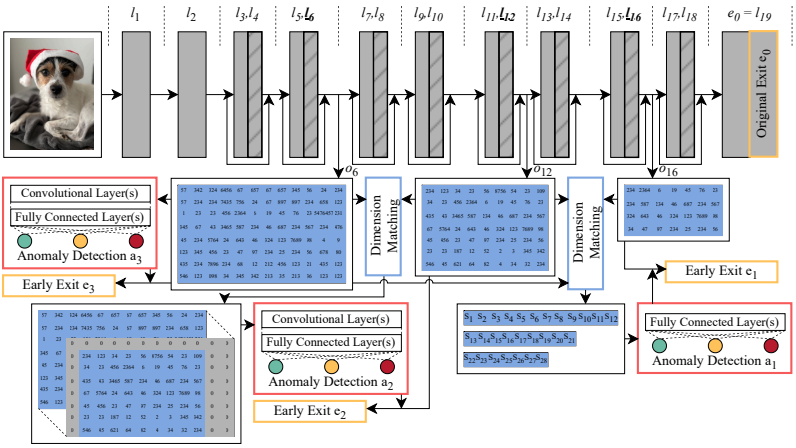


Figure 6.1: ResNet-18 model for schedule creation as in Example 20.
The given net is turned into tasks $\tau_{l.3}$, $\tau_{l.2}$, $\tau_{l.1}$ and $\tau_{l.0}$ to represent the layers of the original network. Tasks $\tau_{a.3}$, $\tau_{a.2}$, $\tau_{a.1}$ and $\tau_{a.0}$ represent the anomaly detection modules for each exit and task τ_e represents the exits with different versions for each exit.

Table 6.2: Translation of ResNet-18 into TTS model.

	Runtime	**Precedence Constraints**
$\tau_{l.3}$	$C^v_{l.3} = \sum_{i=1}^{6} \hat{t}_C(l_i, u)$	$P^v_{l.3}(\tau_{ld}) = \{(-\infty, 0)\}$, $P^v_{l.3}(\tau_{a.3}) = \{(C^v_{l.3}, C^v_{l.3})\}$
$\tau_{a.3}$	$C^v_{a.3} = \hat{t}_C(a_3, u)$	*N.A.*
$\tau_e, 3$	$C^3_e = \hat{t}_C(e_3, u)$	$P^3_e(\tau_{str}) = \{(C^3_e, \hat{t}_D(e_3))\}$, $P^3_e(\tau_{l.3}) = \{-(\infty, C^v_{a.3})\}$
$\tau_{l.2}$	$C^v_{l.2} = \sum_{i=7}^{12} \hat{t}_C(l_i, u)$	$P^v_{l.2}(\tau_{l.3}) = \{-(\infty, C^v_{a.3})\}$, $P^v_{l.2}(\tau_{a.2}) = \{(C^v_{l.2}, C^v_{l.2})\}$
$\tau_{a.2}$	$C^v_{a.2} = \hat{t}_C(a_2, u)$	*N.A.*
$\tau_e, 2$	$C^2_e = \hat{t}_C(e_2, u)$	$P^2_e(\tau_{str}) = \{(C^2_e, \hat{t}_D(e_2))\}$, $P^2_e(\tau_{l.2}) = \{-(\infty, C^v_{a.2})\}$
$\tau_{l.1}$	$C^v_{l.1} = \sum_{i=13}^{16} \hat{t}_C(l_i, u)$	$P^v_{l.1}(\tau_{l.2}) = \{(-\infty, C^v_{l.2})\}$, $P^v_{l.1}(\tau_{a.1}) = \{(C^v_{l.1}, C^v_{l.1})\}$
$\tau_{a.1}$	$C^v_{a.1} = \hat{t}_C(a_1, u)$	*N.A.*
$\tau_e, 1$	$C^1_e = \hat{t}_C(e_1, u)$	$P^1_e(\tau_{str}) = \{(C^1_e, \hat{t}_D(e_1))\}$, $P^1_e(\tau_{l.1}) = \{-(\infty, C^v_{a.1})\}$
$\tau_{l.0}$	$C^v_{l.0} = \sum_{i=17}^{18} \hat{t}_C(l_i, u)$	$P^v_{l.0}(\tau_{l.1}) = \{(-\infty, C^v_{l.1})\}$, $P^v_{l.0}(\tau_{a.0}) = \{(C^v_{l.0}, C^v_{l.0})\}$
$\tau_{a.0}$	$C^v_{a.0} = \hat{t}_C(a_0, u)$	*N.A.*
$\tau_e, 0$	$C^0_e = \hat{t}_C(e_0, u)$	$P^0_e(\tau_{str}) = \{(C^0_e, \hat{t}_D(e_0))\}$, $P^0_e(\tau_{l.0}) = \{-(\infty, C^v_{a.0})\}$

Example 23: ResNet-18 Load and Store Operations in Scheduling Model

The model instance from Example 22 is adapted to include load and store operations and the corresponding access phases and precedence constraints.

Table 6.3: ResNet-18 load and store operations in scheduling model

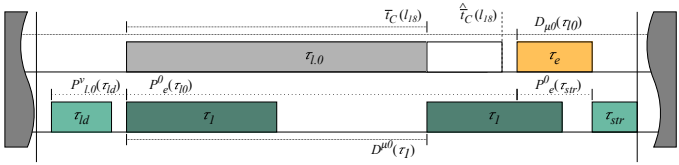
Task	Memory Access
τ_{ld}	$A^{\upsilon}_{ld}(r_{spm}) = \{(t_{ld,sps}, t_{ld,spe})\}$, $B^{\upsilon}_{ld}(r_{spm}) = \{(1)\}$, $A^{\upsilon}_{ld}(r_{mem}) = \{(t_{ld,ms}, t_{ld,me})\}$, $B^{\upsilon}_{ld}(r_{mem}) = \{(1)\}$
τ_{str}	$A^{\upsilon}_{str}(r_{spm}) = \{(t_{str,sps}, t_{str,spe})\}$, $B^{\upsilon}_{str}(r_{spm}) = \{(1)\}$, $A^{\upsilon}_{str}(r_{mem}) = \{(t_{str,ms}, t_{str,me})\}$, $B^{\upsilon}_{str}(r_{mem}) = \{(1)\}$
$\tau_{l.3}$	$A^{\upsilon}_{l.3}(r_{spm}) = \{(t_{l3,sps}, t_{l3,spe})\}$, $B^{\upsilon}_{l.3}(r_{spm}) = \{(1)\}$
$\tau_{a.3}$	$A^{\upsilon}_{a.3}(r_{spm}) = \{(t_{a3,sps}, t_{a3,spe})\}$, $B^{\upsilon}_{a.3}(r_{spm}) = \{(1)\}$
τ_{e}	$A^{3}_{e}(r_{spm}) = \{(t_{e3,sps}, t_{e3,spe})\}$, $B^{3}_{e}(r_{spm}) = \{(1)\}$
$\tau_{l.2}$	$A^{\upsilon}_{l.2}(r_{spm}) = \{(t_{l2,sps}, t_{l2,spe})\}$, $B^{\upsilon}_{l.2}(r_{spm}) = \{(1)\}$
$\tau_{a.2}$	$A^{\upsilon}_{a.2}(r_{spm}) = \{(t_{a2,sps}, t_{a2,spe})\}$, $B^{\upsilon}_{a.2}(r_{spm}) = \{(1)\}$
τ_{e}	$A^{2}_{e}(r_{spm}) = \{(t_{e2,sps}, t_{e2,spe})\}$, $B^{2}_{e}(r_{spm}) = \{(1)\}$
$\tau_{l.1}$	$A^{\upsilon}_{l.1}(r_{spm}) = \{(t_{l1,sps}, t_{l1,spe})\}$, $B^{\upsilon}_{l.1}(r_{spm}) = \{(1)\}$
$\tau_{a.1}$	$A^{\upsilon}_{a.1}(r_{spm}) = \{(t_{a1,sps}, t_{a1,spe})\}$, $B^{\upsilon}_{a.1}(r_{spm}) = \{(1)\}$
τ_{e}	$A^{1}_{e}(r_{spm}) = \{(t_{e1,sps}, t_{e1,spe})\}$, $B^{1}_{e}(r_{spm}) = \{(1)\}$

Tasks for layers and anomaly detections which are presented in Table 6.3 can have multiple different task versions, each denoted with υ. For the exit task, the given exit e_x is denoted with the version x. The original exit e_0 is kept constant.

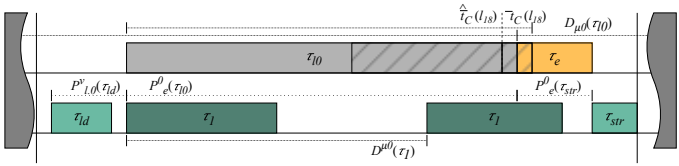
6.2 Scheduling Early Exits as Mode Changes

Because of the different triggers for choosing an early exit, the scheduling of early exits can be modeled as mode changes. This is especially true when integrating the choice of early exits into a time-triggered schedule with an integrated out-of-distribution detection or further anomaly detection methods. The choice of an early exit can then be triggered by the results of the anomaly detection methods, which necessitates further anomaly handling methods to ensure a safe execution. Incorporating the choice for different early exits into a schedule, which was created using the time-triggered scheduling (TTS) approach necessitates an adaptation of the mode change mechanism. Because the introduced mode change mechanism introduces a dynamic aspect to the otherwise static schedule, the mode change mechanism can be used to model the choice for choosing different early exits at runtime.

Because the schedule otherwise remains static, no transitional schedule is necessary, i.e. scheduling function $S_{\Sigma,2}$ (from Section 4.3.2) is not necessary. Instead, all schedule variants need to be interpreted as *adapted schedules* as per Section 4.3.3. This section presents the implementation of this mechanism.



(a) Execution Speed-Up



(b) Execution Delay

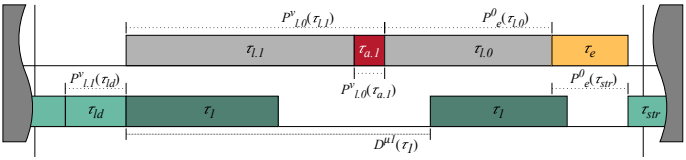
Figure 6.2: Example of execution delay and speed-up inside a given time-triggered schedule. Original network layers are shown in gray, load and store operations are light green and the exit is marked with yellow. Other tasks are marked dark green.

Figure 6.2 again illustrates the problem of needing to choose an early exit at multiple points within a period. In this example, the runtime of a given schedule with a task set including an anytime neural network (ANN) is shown. In Figure 6.2a, the true finish time $\bar{t}_C(l_{18})$ of all layers until right before the exit e_0 is smaller than the estimated finish time $\hat{\bar{t}}_C(l_{18})$, i.e. a speedup of the execution allows for some slack and a successful execution. Contrary to that, Figure 6.2b shows where $\bar{t}_C(l_{18}) > \hat{\bar{t}}_C(l_{18})$, which represents a delay and thus no successful execution. Even though this delay could occur anywhere during the task, the concrete time does not matter, in any case the task exceeds its estimated time.

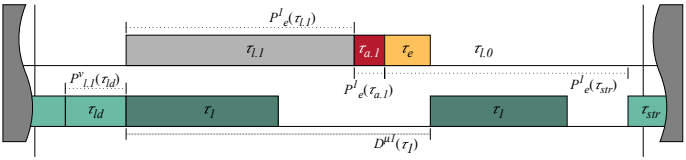
One solution to this would be to use a very conservative estimate for the execution time. This would then allow for a successful execution in all cases, but would also lead to a large overestimation of the average execution time, thus increasing the processor idle time. A balance is struck in Figure 6.3, where the schedule includes one possibility to choose an early exit. Should the ex-

ecution be on time until a given point, the current schedule alternative can be continued. Otherwise, the schedule can be switched to the alternative schedule, which executes the early exit. Both possibilities are shown in Figure 6.3a and Figure 6.3b. This motivation leads to the necessity of the mode change mechanism to allow the modelling of two different possibilities to choose the earlier exits. The scheduling is then performed using variants of the same schedule mode, as described in Section 4.3. Where here, each variant represents a different early exit choice. Thus, for an ANN with N_E early exits, N_E variants are created, starting with the variant executing until the original exit. For this method to work, the following assumptions are made:

It is assumed, that input data is always available for the ANN. This results into the ANN being always ready to execute, and the input queue described in Section 5.6.2 does not run empty. Additionally, it is assumed, that the used task set can be scheduled using the method from Section 4.3. This includes the ANN, which can be scheduled until the last exit. This assumption holds true for the case that multiple jobs for the ANN tasks are scheduled inside one schedule.



(a) Adapted Schedule which Executed Exit e_0



(b) Adapted Schedule which Executed Exit e_1

Figure 6.3: Modeling early exit choices with adapted schedules. Original layers are shown in gray, load and store operations are light green and the exit is marked with yellow. Anomaly detection modules are marked with red. Other tasks are marked dark green.

To create a schedule, the method from Section 4.3.3 is used. First, the schedule variant for the ANN executing until the last exit is created using the simulated annealing-based approach. Here the complete task set is scheduled,

including the ANN tasks, supporting tasks and anomaly detection methods. Only the early exits $e_1, e_2, ...$ are not scheduled, because only e_0 is considered. This is achieved by setting $D^0(\tau_e) \neq \emptyset$ and $V^0(\tau_e) = \{(0)\}$, as described in Section 6.1. This results in an assignment of θ, λ and ϕ for all jobs of all tasks in the task set. After creating the schedule for the ANN including the original exit, the other alternative schedules are created. This is done using the method from Section 4.3.3 and a small adaptation. Starting with the last task which represents ANN layers, an iteration is performed by starting with the removing of the last layer task i.e. $\theta_{l.0}(n) = -1$. Afterwards the next earlier exit, i.e. e_1 is scheduled by setting $V^0(\tau_e) = \{(1)\}$, which in combination with the existing θ values and the precedence constraints of P_e^1 leads to the scheduling of the exit task τ_e and all tasks representing layers until the exit e_1 being kept.

After this, the iteration continues until the earliest exit. Additionally, the anomaly detection tasks are removed, resulting in the values

$$\theta_{l.x}^x(n) = -1,\ \theta_{a.x-1}^{x-1}(n) = -1,\ V^x(\tau_e) = \{(x)\} \tag{6.25}$$

for the choice of early exit $e_x, x \neq 0$. Afterwards, the schedule is completed by scheduling the exits for the corresponding version. This leads to no new layer tasks scheduled and only the exit task being scheduled after the last remaining layer task. Every other task is then expected to be able to be scheduled in the remaining time. This also gives the opportunity to potentially schedule other task versions in the freed up time.

This is implemented inside the simulated annealing scheduler with a modification to the available versions as mentioned above and a fixed $\vec{x}$ vector. After creating the original schedule for either Σ_0 (a standalone schedule) or Σ_1 (a secondary schedule inside a mode change environment), the values for non-layer related jobs are fixed in the starting vector, this includes θ, λ and ϕ. These values are then kept constant during the optimization process. For the exit task τ_e, the version is $\lambda_e(0)$ is increased by one to $x+1$, where x was the last version to represent the earlier exit and $V^\mu(\tau_e) = \{(x+1)\}$. The rest of the cost function and the penalty terms remain unchanged. With the fixed starting points for the ANN tasks, the penalty terms p_{DL}, p_{OL}, p_{prec} and p_{acc} are sufficient to ensure a valid schedule is created. Given the previous assumptions, a valid schedule is expected to be created.

This leads to a flexible adapted schedule (as per Section 4.3.3), where a formal change of schedules can occur at any point during the schedule before the exit. Should an optimized adapted schedule (as per Section 4.3.3) be desired, the

end of all anomaly detection modules must be constant between the schedule alternatives, i.e.

$$\begin{aligned} &\theta^x_{a.x}(n) = \text{const.},\ \phi^x_{a.x}(n) = \text{const.},\ \lambda^x_{a.x}(n) = \text{const.} \\ &\quad \forall x \in [0, N_e), n \in [0, N_\theta(\tau_{a.x})), \end{aligned} \tag{6.26}$$

where x specifies the given exit and thus the given execution mode. With this, the schedule can be optimized by allowing the scheduler to change the starting times of all other jobs as long as all jobs which intersect with the end time of the anomaly detection modules are not changed. This allows for a more optimized schedule, but also requires a more complex mode change mechanism, which needs to ensure that the execution mode can only be switched at the end of the execution time of the anomaly detection modules.

6.3 Runtime Environment and Platform Deployment

The runtime environment (RTE) for the execution of time-triggered scheduling (TTS), which support anytime neural networks (ANNs) needs to be deployed on the target platform. This target platform is assumed to be a heterogeneous system-on-chip (HESoC). In this work, three types of development strategies are presented. First, Section 6.3.1 shows the translation of the schedule into a place-timed Petri net, where the time values at the places represent the maximum execution time for the given phase. Second, two deployment strategies onto HESoCs with dedicated management cores are shown in Section 6.3.2, which target FPGA-based as well as RISC-V-based embedded systems. Third and last, two deployment strategies, which use real-time operating systems (RTOSs) are shown in Section 6.3.3. While many possible RTOSs like PikeOS [38] or hypervisors like Xen [39] exist, this deployment targets FreeRTOS [87].

6.3.1 Abstract Deployment by Petri net Generation

Further analysis of given schedules can be undergone using timed petri-nets, which exist in various forms [146], such as M-timed Petri nets [147],

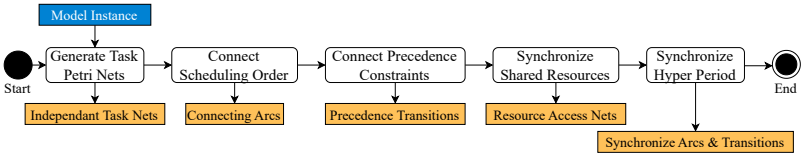


Figure 6.4: Petri net deployment workflow. User provided inputs are marked in blue, generated parts of the Petri net are marked in yellow.

transition-timed Petri nets [148] or delay timed-Petri nets [149]. In this work, an adaptation of M-timed Petri nets [147] is used. Here, each place is assigned two timing values, which represent the minimum and maximum time, after which the given place makes its marks available. The transitions are not assigned any timing values, but are instead used to synchronize the execution.

The used timed Petri nets are a tuple of the form $N = \{P, S, F, W, T\}$, where P defines a set of places, S defines a set of transitions, $F \subset (P \times S) \cup (S \times P)$ defines directed arcs between places and transitions, and $W : F \to \mathbb{Z}$ defines a set of weights assigned to the arcs. In addition to the standard definition of the Petri net, a timing function $T : P \to \mathbb{Q}^2$ is defined, which assigns a pair of timing values to each place. The first value represents the minimum time after which the given place makes its marks available, and the second value represents the maximum time after which the given place makes its marks available. A marking of a Petri net $M : P \to \mathbb{N}$ assigns each position $p \in P$ a number of tokens. The initial marking is defined as M_0. To model a net dynamic, each time step, the given markings are checked [147]. If during a time step (with resolution t_δ) a place has had its token assigned since a time between t_{min} and t_{max}, the tokens are considered. The corresponding transition can then fire with a given probability distribution, resembling the distribution given by the execution time of the corresponding task phase.

Creating the Petri net from the schedule is done in five steps, which are shown in Figure 6.4 and which are explained below.

First, a set of places P is created. Should all access windows have a memory bandwidth of 1, the number of places is bound by

$$|P| \leq 2 + N_{\mathcal{U}} + \sum_{\tau_i \in \mathcal{T}} \Big(\sum_{n \in [0, N_\theta(\tau_i))} \big(3 \cdot \sum_{r_k \in \mathcal{R}} |A_i^{\lambda_i(n)}(r_k)|\big) + 3 \Big), \tag{6.27}$$

where $N_\theta(\tau_i)$ is the number of jobs of task τ_i and $A_i^v(r_k)$ is the set of access windows towards any resource r_k for a given version v of task τ_i. The reasoning for

this upper bound for each job is given in the following. Each access window $A_i^{\lambda_i(n)}(r_k)$ of a task τ_I has one place representing the access window on the execution unit $\phi_i(n)$ and one place representing the access window on the shared resource r_k. Additionally, one place in between access windows might be present, resulting in the upper bound of three places per access window. Should two access windows lie directly after each other, the place in between can be omitted. Additionally, one place is added to represent a possible execution window at the start or end of the given task and a potential second place might be added to represent time between two jobs on the same execution units and lastly, a place might be added to represent possible precedence constraints. Irrespective of the jobs, two places will be added to synchronize the execution to the given hyperperiod and $N_{\mathcal{U}}$ places are added to represent a further synchronization on the given execution units. This value represents an upper bound and not the definite value.

Should the memory bandwidth of the access windows be less than 1, an overlapping of access windows on the same shared resource might occur, which might necessitates multiple places to represent the access towards the shared resource, because multiple access windows might be active at the same time. In Figure 6.5 of Example 24, access towards the r_{mem} resource has overlapping access windows, resulting in the splitting of the place representing access from τ_3 (dark blue) and τ_9 (yellow). This extends the upper bound to

$$
\begin{aligned}
|P| \leq 2 + N_{\mathcal{U}} + \sum_{\tau_i \in \mathcal{T}} \Bigg(\sum_{n \in [0, N_\theta(\tau_i))} \Big(3 \cdot \sum_{r_k \in \mathcal{R}} |A_i^{\lambda_i(n)}(r_k)| \Big) + 3 \Bigg) \\
+ \sum_{r_k \in \mathcal{R}} \sum_{\substack{\tau_i, \tau_j \in \mathcal{T} \\ i \neq j}} \sum_{n \in [0, N_\theta(\tau_i))} \sum_{\substack{m \in [0, N_\theta(\tau_j)) \\ m \neq n}} \sum_{a_i \in A_i^{\lambda_i(n)}(r_k)} \sum_{\substack{a_j \in A_j^{\lambda_j(m)}(r_k) \\ a_i \neq a_j}} \begin{cases} 1 \textbf{ if } \texttt{overlap}(a_i, a_j) > 0 \\ 0 \textbf{ otherwise} \end{cases}
\end{aligned}
\tag{6.28}
$$

where $\texttt{overlap}(a_1, a_2)$ returns the overlapping period of the two access windows a_1 and a_2, which also includes the periodicity of the schedule.

These places are assigned to each job of each task by a series of places for each execution phase and access window, resembling the execution on the execution units. Additionally, each access window has one place assigned. With $\rho : \mathcal{T} \times \mathbb{N} \to \mathcal{P}(P)$ each job is mapped to the corresponding places and $\sigma : \mathcal{T} \times \mathbb{N} \to \mathcal{P}(S)$ maps each task to the corresponding transitions. Each place has one corresponding transition assigned. In the context of the net,

this transition will be placed after the place and before the next execution phase or access window on the same execution unit. The partial ordering of P, $p_1 < p_2$, signifies if the execution phase represented by a place $p_1 \in P$ is executed before another place $p_2 \in P$. The first step connects these places with transitions via arcs, so that each job has one series of places and transitions for the execution on the unit and one place per access window on the shared resources.

Second, the first place of the mth job of task τ_j and the last place of the nth job of task τ_i are connected, if these are executed on the same execution unit $\phi_i(n) = \phi_j(m)$ and no further task τ_x is executed in between, i.e.

$$\begin{aligned} &\nexists\, \tau_x \in \mathcal{T}, k \in [0, N_\theta(\tau_x)) \\ &\quad \text{s.t. } \theta_i(n) < \theta_x(k) < \theta_j(m) \text{ for which } \phi_i(t_1) = \phi_j(t_x) = \phi_k(t_2). \end{aligned} \tag{6.29}$$

Here, the periodicity of the schedule is not taken into account. The last places of each last executed job on each execution unit will be connected to the synchronization transition at a later step.

Third, precedence constraints between tasks are considered, and one synchronization place is added for each precedence constraint. For each precedence constraint $P_i^{\lambda_i(n)}(\tau_j)$ between the nth job of task τ_i and the mth job of task τ_j, a place $p_{i,n,j,m}$ is added, which is connected to the transition before the first place of the earlier job and leads to the transition before the first place of the later task. The timing of the place is then given with $T(p_{i,n,j,m}) = P_i^{\lambda_i(n)}(\tau_j)$. This ensures that the precedence constraint is met, as the transition before the first place of the later job can only fire after the tokens from the place representing the precedence constraint are available, which happens after the timing of the place has passed since the firing of the transition before the first place of the earlier job.

Fifth, the access towards shared resources is synchronized. For each access window $A_i^{\lambda_i(n)}(r_k)$, the place representing the access window on the execution unit and the place representing the access window on the shared resource are connected to a transition, which can only fire if both places have their tokens available. The timing of both places is given by the timing of the corresponding access window. Should two or more access windows overlap on the same shared resource, each time frame of an overlapping is represented by a separate place. All places on the same shared resource r_k are connected via transitions

representing the execution phases of the corresponding tasks Since these are ordered by $\theta_i(n) + A_i^{\lambda_i(n)}(r_k)$, the transition between them can only fire if the previous place has its token available, which ensures that the access towards the shared resource is synchronized. Additionally, the corresponding transition leads into both, the place representing the access window on the execution unit and the place representing the access window. This also ensures the synchronization of parallel access windows.

Lastly, the execution of tasks is synchronized and formed to a loop by connecting the last place on all units with the first transition of the first jobs on all units. For this, arcs are introduced, from the last place of the last task on every execution unit to the synchronization transition. The places at the end of the execution on each execution unit have no minimum and no maximum timing assigned. They can fire each time step. Additonally, arcs are created from the synchronization transition to the first place of the first executed job on each execution unit. Lastly, two places with a timing of $(t_P/2, t_P/2)$ are created to synchronize the execution of the tasks to the given hyperperiod. These places are connected to the synchronization transition and to a second transition as given in the top right in Figure 6.5.

Once a place-timed Petri net is created, this Petri net can be used to model execution and control deployment. The place-timed Petri net representation is schedulable with the hyperperiod t_p, because it is created out of a valid schedule with the same hyperperiod. When the underlying schedule is valid, the Petri net is valid as well.

Example 24 shows an example of a task set, which is turned into a place-timed Petri net of the given task set. The places are shown as circles, the transitions as rectangles and the arcs as directed arrows. The timing of the places is set to the execution time of the corresponding phase. For the time frame, where multiple access windows with $B < 1$ exist, multiple places are generated for the access to the shared resource. A synchronization mechanism is employed via a transition at the start of the hyperperiod. As the diagram has all places marked right before this synchronization point, the diagram shows the beginning of the execution. This Petri net can then be used for further analysis and deployment. Each line represents a shared resource or execution unit. For easy orientation, the shared resources are indicated.

Example 24: Place-Timed Petri net from Example Task Set

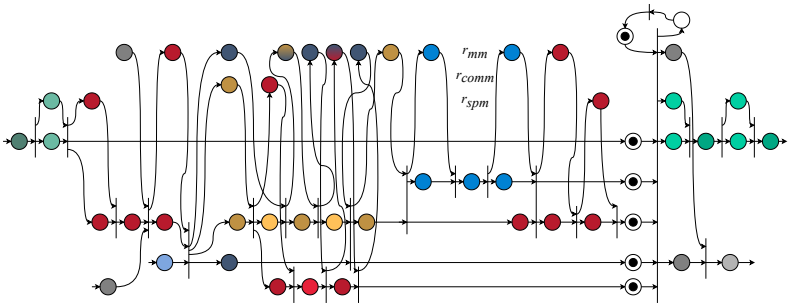


Figure 6.5: Excerpt of the timed Petri net deployment of the example task set. The color is selected to mimic the chosen color of Example 8.

With the given model instance, a place-timed Petri net can be generated using the steps given above. Figure 6.5 shows an excerpt of the timed Petri net from the schedule of Example 8. The timing of the places equals the execution time of the corresponding phase.

A mode change is made possible by changing the Petri net at the synchronization point. If a mode change is triggered at the synchronization point a new set of places, transitions, arcs and timings is loaded to represent the new mode. Should a mode change need to be possible at another point, a second synchronization transition is added and places which are active during this mode change point need to be split. This synchronization transition then allows for a further point to change schedules.

Contribution 20: Deployment via Place-Timed Petri nets

The abstract deployment of a schedule as a place-timed Petri net allows for a platform-agnostic representation of the schedule, which can be used for further analysis and deployment.

6.3.2 Dedicated Management Cores

The first platform deployment strategy involves dedicated management cores, as presented in [150] or used in [Sta26]. The deployment method is demonstrated for two different platforms, both using RISC-V-based cores to showcase the possibility of the abstracted schedule to be deployed onto multiple different platforms. The first platform is a multi-processor system-on-chip (MPSoC)

based on Chipyard [151] RocketCores [152] and was presented in [150]. The second platform uses a set of Xilinx MicroBlaze-V softcore processor [126] and is developed specifically for this purpose. In both systems, one core is designated as the management core, responsible for scheduling and managing the execution of tasks on every other core, allowing for a clear separation between execution and scheduling, but at the cost of a core. Figure 6.7 and Figure 6.8 show the of the deployment of these systems.

Deployment on Chipyard-V-based System

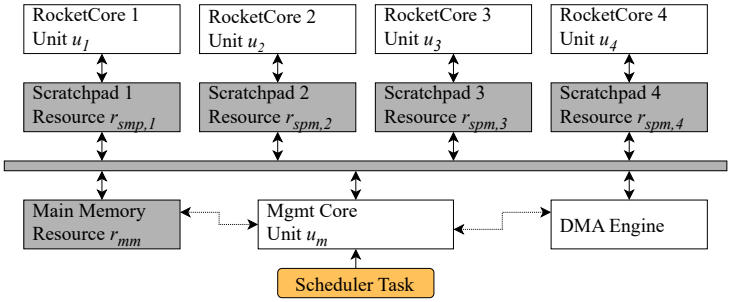


Figure 6.6: Overview of the platform from [150], used to deploy scheduled task sets. Shared resources are marked in gray, execution units are marked in white. All execution cores have access to their local scratchpad memory and the management core (or the DMS engine) is used to transfer data between the main memory and the scratchpad memory.

In Figure 6.6, a system comprised of four $N_{\mathcal{U}}$ RocketCores is shown, where one of the cores is used as a dedicated management core. Each execution core has a local scratchpad memory associated with it. The cores are connected via a crossbar. Additionally, the main memory interface and a direct memory access (DMA) controller are connected to the crossbar, allowing for direct data transfer between the cores and the main memory. The transfer of data between main memory and local scratchpad memory is done via DMA transfers. These transfers are executed via the dedicated management core [150].

This deployment platform is adapted from [150] and an adaptation of the task model needs to be included in the deployment steps. The original model used in [150] includes a *task input load phase*, which loads the tasks inputs into the local scratchpad memory. These inputs are defined as input port

data and the timing window is mapped to the front of a given task. Second, the model includes a *task load phase*, loading all task instructions, and ports into the local scratchpad memory. This phase directly follows the *task input load phase*. Third, the *execution phase* of the task commences without access towards the shared memory and only accesses the local scratchpad. Forth and after the execution, a *task unload phase* is executed unloading the task ports to buffers in the main memory and lastly, the *task output publish phase* publishes the task outputs from the buffers to the output ports [150].

Table 6.4 shows the translation of the values from the deployment method into tasks in the task model. The management core is modeled as $u_m \in \mathcal{U}$, and the execution cores are modeled as $u_k \in \mathcal{U}$ with $k \in [0, N_{\mathcal{U}})$. The scratchpad memories are modeled as resources $r_{spm,j} \in \mathcal{R}$ with $j \in [0, N_{\mathcal{U}} - 1)$, and the main memory interface is modeled as a resource $r_{mem} \in \mathcal{R}$. Access towards main memory by the original task τ_i is replaced with access towards the scratchpad memory $r_{spm,\upsilon}$ of the given core u_υ. Additionally, precedence constraints are set such that the *task input load* phase needs to finish before the *task load* phase can start, and the *task load* phase needs to finish before the execution phase can start. The execution time of the task itself is kept the same, as are precedence constraints towards other tasks which are not used to model the load and store phases. The available execution unit is fixed to a given core for a given version υ by setting $U_i^\upsilon = u_\upsilon$.

Task input load and *task load* phases are modeled as separate tasks $\tau_{TIL,i.\upsilon}$ and $\tau_{TL,i.\upsilon}$, respectively, with the same execution time for all versions. Both tasks have access to the main memory interface r_{mem} and the scratchpad memory $r_{spm,\upsilon}$ of the given core u_υ. In contrast to τ_i, $\tau_{TIL,i.\upsilon}$ and $\tau_{TL,i.\upsilon}$ are not modeled with different versions, but with different tasks, because precedence constraints are not able to include different versions. The timings of the precedence constraints are set such that the *task input load* phase needs to finish before the *task load* phase can start, and the *task load* phase needs to finish before the execution phase can start. Since both precedence constraints are ultimately dependent on the main task τ_i, the chosen version υ determines the used tasks of the *task input load* and *task load* phases. This also determines the scratchpad memory, which is accessed by both tasks.

Task unload and *task output publish* phases are modeled as separate tasks $\tau_{TUL,i.\upsilon}$ and $\tau_{TOP,i.\upsilon}$, as well. Again the execution time stays constant for all versions. Both tasks have access windows to the main memory interface r_{mem} and the scratchpad memory $r_{spm,\upsilon}$ of the given core u_υ, determined by the

chosen version of τ_i, and with that also the chosen execution unit u_υ. Since the scheduling of $\tau_{TUL,i.\upsilon}$ is determined by the precedence constraints of the chosen version υ of τ_i, this chosen version of τ_i also forces the use of the correct *task unload* and *task output publish* tasks. The precedence constraints are set such that the *task unload* phase needs to start after the execution phase starts, and the *task output publish* phase needs to start after the execution phase finishes.

In the original notation, the time is given for the phases ω^i_{TIL}, ω^i_{TL}, ω^i_{TUL} and ω^i_{TOP} as $t_{TIL,i.\upsilon}$, $t_{TL,i.\upsilon}$, $t_{TUL,i.\upsilon}$ and $t_{TOP,i.\upsilon}$ for the *task input load*, *task load*, *task unload*, and *task output publish* phases, respectively. In the task model, these are represented by the execution times of the corresponding tasks, $C^\upsilon_{TIL,i.\upsilon}$, $C^\upsilon_{TL,i.\upsilon}$, $C^\upsilon_{TUL,i.\upsilon}$ and $C^\upsilon_{TOP,i.\upsilon}$. The available execution unit of the task execution is fixed to a given core for a given version υ by setting $U^\upsilon_i = u_\upsilon$, but the *task input load*, *task load*, *task unload*, and *task output publish* tasks are mapped to the management core. Access towards main memory by the original task τ_i is replaced with access towards the scratchpad memory $r_{spm,\upsilon}$ of the given core u_υ. An overview is given in Table 6.4.

The presented deployment of [150] includes some constraints to the scheduling of the given tasks. These constraints can be omitted in this work, since the TTS scheduling presented in Section 4.2 includes these constraints implicitly. Adapting the *input load phase* and *task load phase* to the TTS model is done by modeling it as a dedicated task, running on the management core.

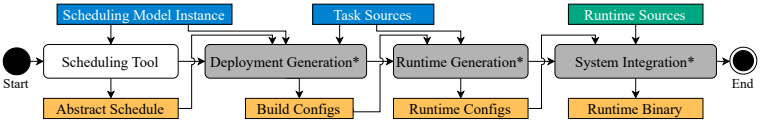


Figure 6.7: Workflow for the deployment of a schedule on the RocketCore-based platform. User provided inputs are marked in blue. Persistent inputs are marked in green. Generated artifacts are marked in yellow. Gray boxes and asterisks signify components adapted from [150]. White boxes signify the addition of this work.

Since the scheduler generates θ, ϕ, and λ, the deployment of the generated schedule on the target platform is straightforward. Each task, including the load and store phases, is executed on the assigned core at the assigned time, with the assigned version. The management core executes the load and store phases, while the execution cores execute the main execution phase of the tasks.

Table 6.4: Mapping of deployment load and unload phases to task model.

Component	Model Values
Management Core	$u_m \in \mathcal{U}$, One core designated as management core.
Scratchpad Memory	$\mathcal{R} = \{r_{spm,j} \mid \forall j \in [1, N_{\mathcal{U}})\} \cup \{r_{mem}\}$, One scratchpad per General Purpose Processor (GPP) core additionally one main memory interface.
Task Input Load ω^i_{TIL} of task τ_i, version υ: $\tau_{TIL,i.\upsilon}$	$C^0_{TIL,i.\upsilon} = t_{TIL,i.\upsilon}$, $U^0_{TIL,i.\upsilon} = u_m$, $A^0_{TIL,i.\upsilon}(r_{spm,\upsilon}) = \{(0, t_{TIL,i.\upsilon})\}$, $A^0_{TIL,i.\upsilon}(r_{mem}) = \{(0, t_{TIL,i.\upsilon})\}$, $B^0_{TIL,i.\upsilon}(r_{spm,\upsilon}) = \{(1)\}$, $B^0_{TIL,i.\upsilon}(r_{mem}) = \{(1)\}$
Task Load ω^i_{TL} of task τ_i, version υ: $\tau_{TL,i.\upsilon}$	$C^0_{TL,i.\upsilon} = t_{TL,i.\upsilon}$, $U^0_{TL,i.\upsilon} = u_m$, $A^0_{TL,i.\upsilon}(r_{spm,\upsilon}) = \{(0, t_{TL,i.\upsilon})\}$, $A^0_{TL,i.\upsilon}(r_{mem}) = \{(0, t_{TL,i.\upsilon})\}$, $B^0_{TL,i.\upsilon}(r_{spm,\upsilon}) = \{(1)\}$, $B^0_{TL,i.\upsilon}(r_{mem}) = \{(1)\}$, $P^0_{TL,i.\upsilon}(\tau_{TIL,i.\upsilon}) = (-D_i, -t_{TIL,i.\upsilon}]$
Original task τ_i, version $\upsilon \in [0, N_{\mathcal{U}} - 1)$	$C^\upsilon_i = C^\upsilon_i$, $U^\upsilon_i = u_\upsilon$, $A^\upsilon_i(r_{spm,\upsilon}) = [0, C^\upsilon_i]$, $A^\upsilon_i(r_{mem}) = \emptyset$, $B^\upsilon_i(r_{spm,\upsilon}) = \{(1)\}$, $P^\upsilon_i(\tau_{TL,i.\upsilon}) = (-D_i, 0]$, $P^\upsilon_i(\tau_{TUL,i.\upsilon}) = [C^\upsilon_i, D_i - t_{TUL,i.\upsilon} - t_{TOP,i.\upsilon})$
Task Unload ω^i_{TUL} of task τ_i, version υ: $\tau_{TUL,i.\upsilon}$	$C^0_{TUL,i.\upsilon} = t_{TUL,i.\upsilon}$, $U^0_{TUL,i.\upsilon} = u_m$, $A^0_{TUL,i.\upsilon}(r_{spm,\upsilon}) = \{(0, t_{TUL,i.\upsilon})\}$, $A^0_{TUL,i.\upsilon}(r_{mem}) = \{(0, t_{TUL,i.\upsilon})\}$, $B^0_{TUL,i.\upsilon}(r_{spm,\upsilon}) = \{(1)\}$, $B^0_{TUL,i.\upsilon}(r_{mem}) = \{(1)\}$, $P^0_{TUL,i.\upsilon}(\tau_{TOP}) = [t_{TUL,i.\upsilon}, D_i - t_{TOP,i.\upsilon})$
Task Output Publish ω^i_{TOP} of task τ_i, version υ: $\tau_{TOP,i.\upsilon}$	$C^0_{TOP,i.\upsilon} = t_{TOP,i.\upsilon}$, $U^0_{TOP,i.\upsilon} = u_m$, $A^0_{TOP,i.\upsilon}(r_{spm,\upsilon}) = \{(0, t_{TOP,i.\upsilon})\}$, $A^0_{TOP,i.\upsilon}(r_{mem}) = \{(0, t_{TOP,i.\upsilon})\}$

Figure 6.7 shows the workflow for the deployment and task generation. In addition to the generation of additional tasks in the model, the task and runtime binaries are generated. Given the task sources, build configs and the binaries are generated to be executed on the target platform. The management core executes the load and store phases, while the execution cores execute the given binaries. The implementation of the load and store phases are provided by the RTE and the implementation of the execution phases need to be provided by the user.

Deployment on MicroBlaze-V-based System

To compare different RISC-V-based platforms, a second platform is introduced. This platform represents two paradigm shifts in comparison to the

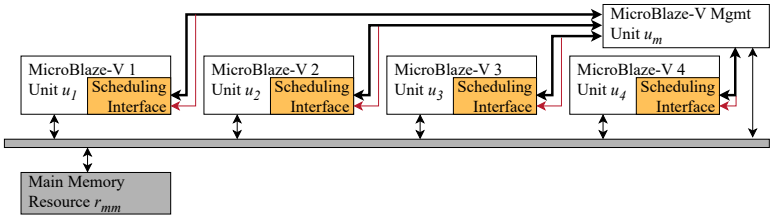


Figure 6.8: MicroBlaze-V-based [126, 153] deployment platform. One MicroBlaze-V core is used as a management core, which directly sends management commands to the execution cores. Scheduling-related components are marked yellow. Shared resource-related components are marked gray. The control interface is marked with red and splits of from the stream interface.

previous method. First, the platform is based on Xilinx MicroBlaze-V softcore processors [126, 153], which represent a mature commercial solution in contrast to the open-source RocketCores [152]. Second, the management core directly controls the execution cores via a custom interface, instead of using a DMA controller. Eliminating collisions on shared resources must be facilitated by scheduling tasks in a way that they do not access the same shared resource at the same time, necessitating a more complex scheduling.

The given platform is described with $N_{\mathcal{U}}$ Xilinx MicroBlaze-V softcore processors [126, 153], where one core is assigned the task of a management core. The structure of the system is shown in Figure 6.8. The management core is connected via a dedicated duplex communication system to each execution core and furthermore controls the execution on these cores. Additionally, a task queue is given via an AXI-stream interface from the management core to the execution cores. In this task queue, the start addresses of the tasks are given. The management core u_m signals to each execution core $u \in \mathcal{U} \setminus \{u_m\}$ the start of the next task via a `command` signal, this signal start or stops the execution of the task binary placed at the address given by the first value in the task queue. It also includes the number of how many arguments the next task requires. It additionally allows the signaling of jumping towards the next task, terminating execution or pausing the execution on the execution core. The execution core is able to signal the status of a task via a `status` signal back towards the management core. This includes a finish signal as well as the occurrence of an anomaly or the choice to change execution modes. How the management core reacts to this, is defined in the generated schedule. The reaction itself is controlled by the management core.

Table 6.5: Custom OP code for execution cores. The management core sends commands to the execution cores via a dedicated interface. The execution cores signal their status back to the management core via a dedicated interface.

Command	Direction	Description
`0000 0000`	$u_m \rightarrow u_i$	Continue execution of the current task.
`001X XXXX`	$u_m \rightarrow u_i$	Start execution of the next task, with the next `X XXXX` values in the task queue as argument addresses.
`0100 0000`	$u_m \rightarrow u_i$	Pause execution of the current task.
`0110 0000`	$u_m \rightarrow u_i$	Terminate execution of the current task.
`1000 0000`	$u_i \rightarrow u_m$	Task is executing.
`1100 0000`	$u_i \rightarrow u_m$	Task has finished execution.
`111X XXXX`	$u_i \rightarrow u_m$	Anomaly with the index `X XXXX` has occurred.
`101X XXXX`	$u_i \rightarrow u_m$	Proposed change to execution mode `X XXXX`.

Table 6.5 shows the control commands for the execution cores. The parallel task queue is filled by the management core according to the generated schedule, with the correct start addresses and argument addresses for each task. The management core then triggers the execution of the tasks on the execution core. The management core reacts to signaled anomalies according to the generated schedule, which could include a change in execution mode or a change towards the next task in the task queue.

The configuration of the management core is done by creating a task queue for every execution core. This queue is saved in a table for each execution core and includes the start addresses and argument addresses for each job of the scheduled tasks. It furthermore includes the time at which the task should be triggered. Since the execution queue can be filled prior to execution, the first period of the schedule can be used to fill the task queues. The task queues will be filled with the tasks from a new schedule, should a change in execution mode be required, either at the request of a task or by design. In both cases, this is triggered by the management core. The management core will then start the execution of the newly scheduled tasks on the execution cores according to the generated schedule. The status signal to trigger a schedule change is then set via an anomaly detection module as described in Section 4.4 or Section 5.3.

Internal to the management core, a schedule is saved as a table of tasks for each execution core, including the start addresses and arguments as well as the time at which the task should be triggered. This comes out of the generated

schedule. These lists are saved using the local cache and thus will not require access towards the shared memory, once loaded. Because of this, a change in execution mode is simple in its execution. The management core will fill the task queues for each execution core with the tasks of the new schedule, and continue to trigger the execution of the new tasks according to the new schedule table.

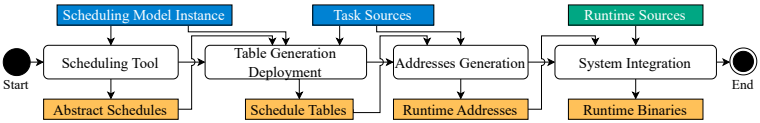


Figure 6.9: Workflow for the deployment of a schedule on the MicroBlaze-V-based platform. User provided inputs are marked in blue. Persistent inputs are marked in green. Generated artifacts are marked in yellow. White boxes signify the addition of this work.

The deployment workflow of the schedules is given by Figure 6.9. The deployment is similar to the previous method shown in Figure 6.7, but adds the generation of the schedule tables as lists of the tasks for each execution core, including the start addresses and arguments as well as the time at which the task should be triggered. Additionally, the runtime addresses are extracted and used to fill the given task queues. These lists and values are then loaded into the management core, which will use it to trigger the execution of the tasks on the execution cores according to the generated schedule.

6.3.3 Real-time Operating System-based Deployment

The second deployment method is developed for the use in commercial off-the-shelf (COTS) systems. These systems run a RTOS, which shows the ability to deploy generated schedules on fixed hardware. The first deployment is implemented for an STM32H747XI microcontroller [154], which features a dual-core HESoC with an ARM Cortex-M7 and an ARM Cortex-M4 execution unit. This is presented in Section 6.3.3. While this system represents a simple HESoC with its two cores, it allows for showcasing the heterogeneous scheduling capabilities of the ANN in a real-world scenario using established Real-time operating systems. The examples from this thesis target a ZCU102 board and the deployment is described in Section 6.3.3.

Deployment on a FreeRTOS-based STM32 Microcontroller

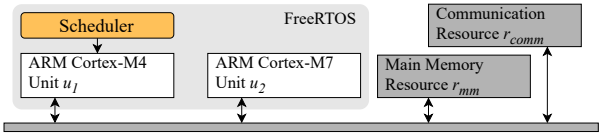


Figure 6.10: ARM-based processing platform in STM32-based system. An FreeRTOS is used as the operating system, with the scheduler task being executed on the Cortex-M4 core. Cores are marked white, shared resources are marked dark gray. Scheduler parts are marked yellow.

The deployment in the STM32H747XI microcontroller [154] uses FreeRTOS [87] as the operating system and on-platform scheduler. For this task, the round-robin mode is selected for the FreeRTOS instance. Additionally, the asymmetric multi-processing (AMP) mode is chosen over the symmetric multi-processing (SMP) mode, since both ARM cores are heterogeneous.

Enabling the deployment of the generated schedule on the STM32H747XI microcontroller is ensured by the implementation of a code generation tool, which takes the source code of the tasks as an input. Together with predefined wrapper functions, the code generation tool generates a complete binary, which includes all necessary components including the scheduled tasks. To integrate the source code of the scheduled tasks, the scheduler needs to know the local file names of the task sources, which is provided by the user together with the job start times and the chosen execution unit. Both are used to fill in the wrapper functions. These wrapper functions use the FreeRTOS API to start and stop the execution of the tasks at the predefined time and are implemented using best practices [93] to ensure correct execution of the tasks. The wrapped task functionality is then integrated as the starting point of the task execution inside the main execution file. The file itself is then turned into the executable using the provided compiler toolchain for the STM32H747XI microcontroller [154]. A switch in the execution mode is implemented by letting the wrapper functions hold a list of different task sources. Depending on the current execution mode, the wrapper function will execute the task source corresponding to the current execution mode. This allows for a simple change in execution mode by changing the index of the task source in the wrapper function. Since the wrapper functions are implemented using the FreeRTOS API, they will ensure the correct execution of the tasks in the time-triggered system, even when changing execution modes.

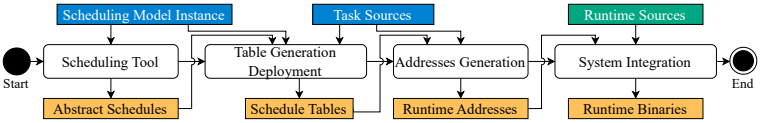


Figure 6.11: Workflow for the deployment of a schedule on the STM32-based platform. User provided inputs are marked in blue. Persistent inputs are marked in green. Generated artifacts are marked in yellow. White boxes signify the addition of this work.

Figure 6.11 shows the workflow for the deployment of the generated schedule on the STM32H747XI microcontroller. The user provides the task sources, job start times and chosen execution units as input. The code generation tool then generates a complete binary, which includes all necessary components including the scheduled tasks. The main difference to the previous approaches is the lack of different predefined binaries and the usage of the source code as the input, which would make this approach unsuitable for systems with external provided tasks. But because of the small size of the system itself, this weakness is considered not impactful.

Deployment on the Example AMD ZCU102 Platform

The last deployment platform is the one shown in the examples throughout this work. Here, the generated schedule is deployed on a Xilinx ZCU102 HESoC [24], which features a quad-core ARM Cortex-A53 processor, which is modeled with $\mathcal{U}_{cpu} = \{u_{cpu,1}, u_{cpu,2}, u_{cpu,3}, u_{cpu,4}\}$ and a dual-core ARM Cortex-R5 processor, which is not considered. Additionally, a MALI GPU is included, which is also not considered in this work. Two accelerators are included, one timing-accurate machine learning accelerator and one profiled encryption module. Both are modeled as execution units $u_{ml,5}$ and $u_{cr,6}$, respectively. The shared resources are modeled as r_{mem} and r_{comm}. For the machine learning accelerator [28], a dedicated scratchpad memory is modeled as r_{spm} is included.

The deployment on the platform follows the similar pattern to the deployment on the STM32H747XI microcontroller, but with a more complex platform and the addition of the GPU and deep learning accelerators as execution units. A RTOS is used as the operating system and on-platform scheduler, with one ARM Cortex-A53 core used as the execution core for the online scheduling task. The other ARM Cortex-A53 cores are then used to execute the scheduled

tasks, while the FPGA is used to host the machine learning accelerator and the encryption module. The deployment is done using a Zephyr implementation similar to [155] as the RTOS. The configuration itself is similar to the deployment on the STM32H747XI microcontroller, in which a code generation tool is used to generate one project, which includes all necessary components, both the execution tasks and the runtime environment. The user provides the task sources, job start times and chosen execution units as input and the tool provides the wrapper functions, which use the Zephyr API to start and stop the execution of the tasks. With the code generation tool, a project is generated, which includes the headers, which link the user provided source code to the runtime environment. The compiler from Xilinx Vitis is then used to generate the executable for the platform [24]. The generated executable is then deployed on the platform, with the management core being used to execute synchronization tasks. This work was first presented in [Fin24]. To facilitate the upper memory bound for the memory bandwidth, for shared resources like the on-chip-memory and the scratchpad memory, a dedicated bandwidth regulator like [84] can be employed in the FPGA.

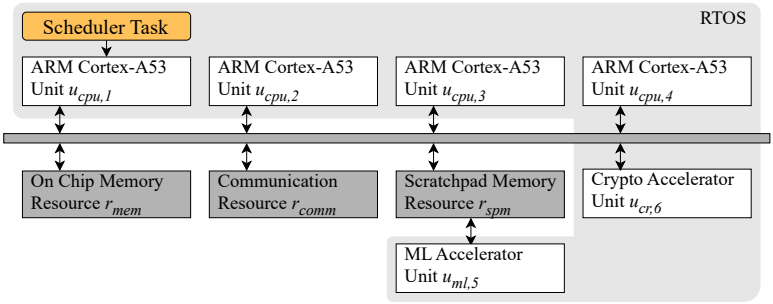


Figure 6.12: Abstracted platform architecture from overarching ZCU102-example. The scheduler task is marked in yellow. Shared resources are marked in gray, execution units are marked in white.

Contribution 21: Platform Deployment Methods

The automated platform deployment methods for bare-metal and RTOS-based platforms shown how an abstract schedule can be used to generate the necessary artifacts to deploy the schedule on a given platform.

While further deployment platforms are possible, the presented deployment methods show the ability to deploy a generated schedule on a variety of platforms, both bare-metal and RTOS-based.

Summary

In this section the combination of the scheduling approaches for TTS and ANNs is presented. With the generation of model instances from ANN instances, as shown in Section 6.1, the scheduling is enabled. A trigger to change the execution mode of the system, as shown in Section 6.2, is then introduced into the system at the checkpoints of the ANNs. Lastly, the anomaly detection capability is introduced as another way to trigger mode changes in Section 4.4. These steps lead to a complete system model which encompasses mode-change capability of central processing unit (CPU)-based tasks and ANN-based tasks as well as integrated out-of-distribution (OOD) detection for both types of tasks. The deployment of this combined scheduling approach using Petri nets, bare metal platforms and RTOSs is then shown in Section 6.3.

This page intentionally left blank

Chapter 7

Evaluation

This chapter gave an evaluation of the introduced methods. Section 7.1 introduced an ablation study for the simulated annealing-based offline scheduler as well as for the runtime estimation of the anytime neural network online scheduler. In Section 7.2, the polyhedron-based offline scheduler was evaluated in terms of its success rate to find valid schedules, which did not need further optimization under different constraints. In the Section 7.3 both time triggered schedulers were evaluated in terms of their algorithm runtime to find valid schedules under different constraints. Section 7.4 evaluated the added exit and anomaly detection components in terms of the accuracy and for the exits also the equivalence to the original network model. The runtime overhead of the online scheduler for neural networks was given in Section 7.5. Section 7.6 evaluated, for the online scheduler, how the deadline success rate was affected by the task release rate and how the early exit choice then led into the accuracy of the classification.

7.1 Ablation Studies

In this ablation study the hyperparameters of both, the simulated annealing-based offline scheduler for time triggered schedules, and the anytime neural network online scheduler were evaluated. For the anytime neural network online scheduler, this included an evaluation of the runtime estimation strategy as well as the given allowed slack in the form of a safety factor c. For the used simulated annealing-based implementation, the hyperparameters included the initial temperature as well as the restart ratio and the ability to do a local search.

To optimize the anytime neural network scheduler, two networks are used as the backbone network for adding early exits to it. A ResNeXt-101 [46] network is used, together with a GoogleNet [44] network. Additionally, a *You*

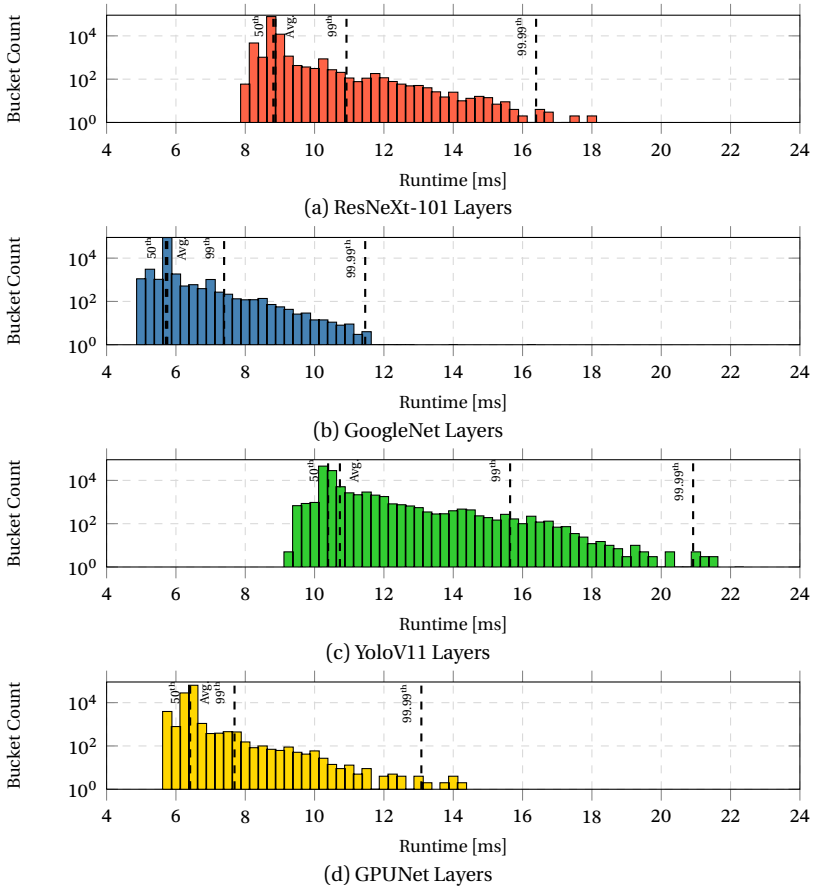
Only Look Once-based [156] architecture in the YoloV11 version [48] is used in further studies to additionally show the ability to integrate single shot object detection into the online scheduler and a GPUNet-based network [47] is included to show the capability for handling simple unaltered neural networks.

7.1.1 Safety Factor for ANN Runtime Estimation

This ablation study evaluated the runtime estimation of the anytime neural network online scheduler. The evaluation aimed to reduce the difference between estimated and actual runtimes while not underestimating the runtime. For this, the online scheduler was executed for 100,000 randomized input samples from the ImageNet validation dataset [134], the ImageNet-R dataset [135] and the ImageNet-O dataset [135]. The bin size for the histograms was set to 0.25 ms, which allowed for an accurate representation of the runtime distribution while still providing a clear visualization.

As shown in Figure 7.1 and Table 7.1, the runtime of the backbone networks follow a Poisson distribution with a long tail. A large number of samples have a runtime close to the mean, while a small number of samples have a higher runtime. The bin size is set to 0.25 ms, allowing for an accurate representation of the runtime distribution, while still providing a clear visualization. For the ResNeXt-101 network, the largest bin is the one at 8.8 ms and the average lies at 8.87 ms. The mean lies at 8.81 ms, showing a skewing of the distribution towards higher runtimes. For the GoogleNet network, the largest bin is the one at 5.75 , where the average also lies. The mean lies at 5.71 ms, again following the skewing of the distribution towards higher runtimes. For the YoloV11 network, the largest bin is the one at 10.75 ms, where the average also lies. The mean lies at 10.39 ms, again following the skewing of the distribution towards higher runtimes. For the GPUNet network, the largest bin is the one at 6.4 ms, where the average also lies. The mean lies at 6.39 ms. Figure 7.1 shows a logarithmic histogram. Here the average, mean and 99^{th} and 99.99^{th} percentile are shown as vertical lines. The logarithmic scale allows for a better visualization of the long tail of the distribution, while still showing the majority of samples close to the mean.

Depending on these runtime distributions, the combination of starting estimation and safety factor was selected. For this Table 7.1 showed the 50^{th}, 60^{th}, 70^{th}, 80^{th}, 90^{th}, 95^{th}, 99^{th} and 99.99^{th} percentile of the runtime distribution for each network. For all networks, the average lay between the 70^{th} and 80^{th} percentile. Based on these percentiles, the starting estimation and safety factor were



(a) ResNeXt-101 Layers

(b) GoogleNet Layers

(c) YoloV11 Layers

(d) GPUNet Layers

Figure 7.1: Logarithmic runtime histogram for the used backbone networks. The used colors correspond to the used networks and will be used for the rest of the evaluation.

Table 7.1: Runtime percentiles for the used backbone networks in [ms].

Network	50th	60th	70th	80th	90th	95th	99th	99.99th	Avg.
R.N.[1]	8.81	8.82	8.84	8.86	8.94	9.09	10.92	16.39	8.87
G.N.[2]	5.71	5.72	5.73	5.74	5.78	5.93	7.39	11.46	5.75
YV11.[3]	10.39	10.44	10.52	10.79	11.71	12.72	15.64	20.92	10.73
GPU.[4]	6.39	6.40	6.41	6.42	6.44	6.51	7.69	13.08	6.41

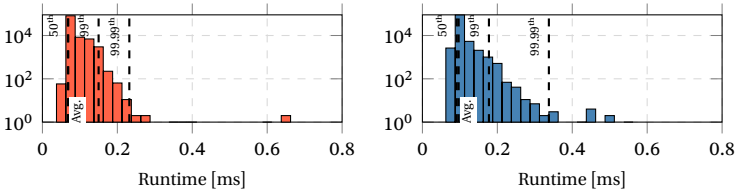
[1]ResNeXt-101 [46], [2]GoogleNet [44], [3]YoloV11 [48], [4]GPUNet [47]

selected to ensure that the runtime estimation was not underestimating the actual runtime while still providing a good estimation for the majority of samples. For all networks except YoloV11, the ration between the average and the percentiles stayed similar. For ResNeXt-101, the ration between the average and the 95$^{\text{th}}$ percentile lay at 8.87 ms/9.09 ms $\approx$ 0.98, for GoogleNet, this ration lay at 0.97, for YoloV11, this ratio lay at 0.84 and for GPUNet, this ratio lay at 0.98. This ration lay at 0.81 for ResNeXt-101, 0.78 for GoogleNet, 0.69 for YoloV11 and 0.83 for GPUNet for the 99$^{\text{th}}$ percentile.

During execution, the runtime estimation was based on a weighted average, which converged against the average, which was measured in this experiment. This average was then used to predict the runtime of the current sample. This prediction needed to strike a balance between being close to the actual runtime for the majority of samples while not underestimating the runtime. Because of this, this weighted average was multiplied with a safety factor, which was chosen based on the desired level of conservatism in the runtime estimation to capture runtimes that lay at the given percentiles. The online scheduler estimated the runtime given the factor f_s. In Figure 7.3, this safety was varied in respect to the ratio of the average and the percentiles.

This diagram shows the average slack, as well as the estimation success rate for different safety factors. The estimation success rate is defined as the percentage of samples for which the runtime estimation is not underestimating the actual runtime. Because of the usual periodicity of periodic tasks, this experiment releases the tasks with a given period, which is set to the average runtime of the backbone network. Additionally, the task is given an absolute deadline, which is equal to the period added to the release time. Here, the deadline success rate is measured. The difference between the estimation success rate lies in the fact, that the estimation success rate only considers the current execution, whereas the deadline success rate also considers any potential backlog from previous executions.

In addition to the runtime distribution of the backbone networks, the runtime distribution of the exit structure as well as the anomaly detection modules need to be evaluated. The exit structure orients itself on the structure of the last feature extraction component together with the fully connected output layers, only the weights and biases are retrained and potentially a layer to adapt the dimensions is added. Because of this, the runtime distribution of the *early* exits is expected to be close to the runtime distribution of the *original* exit of the backbone network and irrespective of the exact positioning. The positioning itself is evaluated later. In this section, only the hyperparameters of the runtime estimation are chosen.



(a) ResNeXt-101 Exit Layers (b) GoogleNet Exit Layers

Figure 7.2: Logarithmic runtime histogram for the used exit structures.

Table 7.2: Runtime percentiles for the used exit structures in [ms].

Network	50th	60th	70th	80th	90th	95th	99th	99.99th	Avg.
R.N.[1], e_0	0.068	0.069	0.072	0.083	0.113	0.129	0.150	0.231	0.069
G.N.[2], e_0	0.092	0.093	0.094	0.099	0.109	0.131	0.178	0.338	0.097

[1]ResNeXt-101 [46], [2]GoogleNet [44]

As in Figure 7.1, Figure 7.2 shows a Poisson-like distribution with a long tail for the runtime of the exit structure. The average, mean and 99th and 99.99th percentile are again shown as vertical lines. These runtime distributions are used to select a safety factor c to multiply the average runtime of the exit structure with to get a runtime estimation, as $\bar{t}_c(l,u) = c \cdot \mathrm{avg}(\overline{\mathrm{T}}_C(l,u))$. This factor is selected to represent the desired percentile of the measured runtime distribution. How the percentiles relate to the safety factor is shown in Table 7.3. These possible safety factors are then used to determine the deadline success rate for different levels of given slack.

Figure 7.3 shows the deadline success rates and the average delay for different safety factors. The solid line represents the deadline success rates and the dotted line represents the average delay. The dashed vertical lines represent the average as well as the 50th and 99th percentile. The 99.99th percentile lies outside the drawn area. The deadline success rate is shown on the left y-axis, while the average delay is shown on the right y-axis. The x-axis shows the safety factor c as a ratio of the runtime estimation and the average runtime in a range of 0.95 to 1.5 representing the range between the 40th and 99th percentile. The deadline success rate increases with the safety factor, as a higher safety factor reserves more time for the task execution, reducing the likelihood of deadline misses. The average delay increases with the safety factor, as a higher safety factor means that more time is reserved for the execution, leading to longer delays if the runtime is shorter than the reserved time.

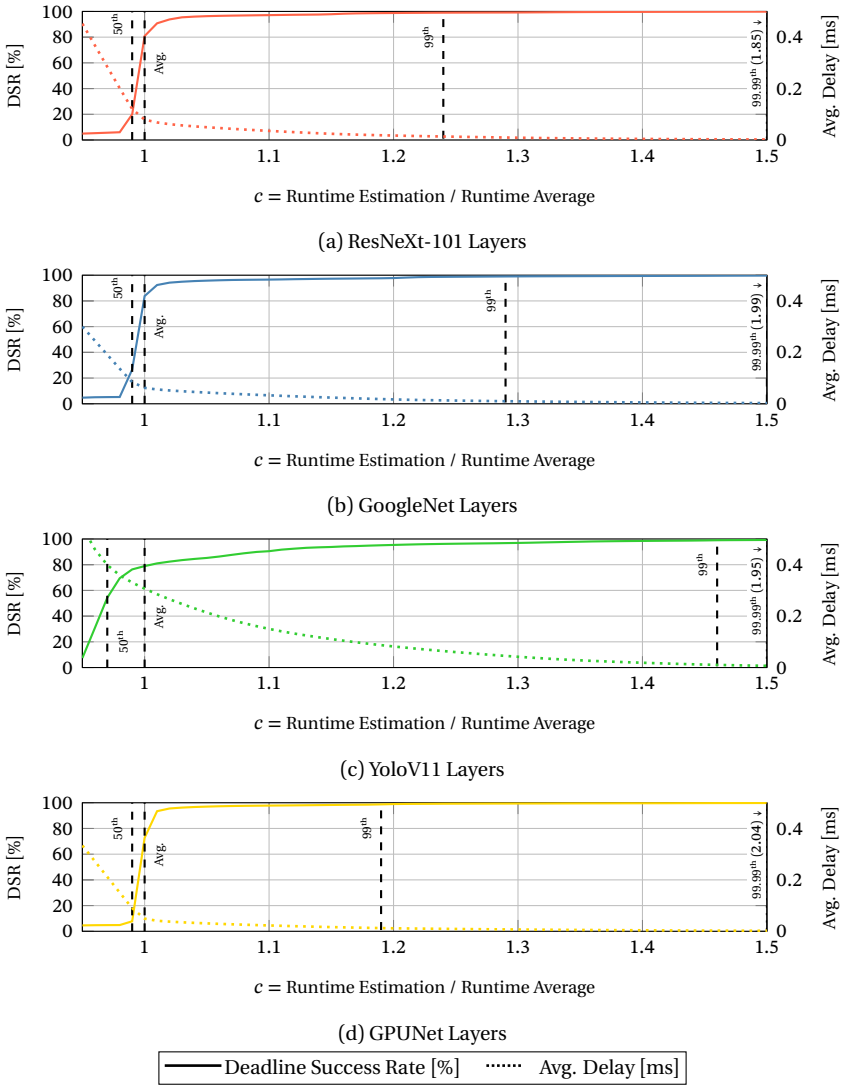


Figure 7.3: Evaluation of the runtime estimation factors for the given networks. The average delay and the processor idle time are shown together with the deadline success rate for different safety factors. The respective percentiles are shown.

Table 7.3: Factor representation of the percentiles of the runtime estimation.

Network	50^{th}	60^{th}	70^{th}	80^{th}	90^{th}	95^{th}	99^{th}	99.99^{th}
R.N.[1]	0.993	0.994	0.997	0.999	1.008	1.025	1.231	1.848
G.N.[2]	0.993	0.995	0.997	0.998	1.005	1.031	1.285	1.993
YV11.[3]	0.968	0.973	0.980	1.006	1.091	1.185	1.458	1.950
GPU.[4]	0.997	0.999	1.000	1.002	1.005	1.016	1.200	2.040

[1]ResNeXt-101 [46], [2]GoogleNet [44], [3]YoloV11 [48], [4]GPUNet [47]

Finding 1: Safety Factor for Online ANN Runtime Estimation

For the further evaluation, a safety factor of $c = 1.2$ is selected, *which corresponds to a deadline success rate of at least 95 % for YoloV11 [48], at least 97 % for GooglNet [44] and at least 98 % for ResNeXt-101 [46] and GPUNet [47]. This represents at least the 99^{th} percentile of the runtime distribution for all networks. This leads to a good balance between the high deadline success rate and the low average delay for both the original model and the exits.*

7.1.2 Weight Distribution for ANN Runtime Estimation

The safety factor takes into account all a priori information about the observed runtime distribution and is used to provide slack for the runtime estimation. It is based on a given average runtime, which was calculated with the complete information, given the runtime distribution. This average might not necessarily be given during runtime and needs to be estimated as well. For this, the online scheduler uses a weighted average of the last $N_W = |W|$ observed runtimes, updating after each execution. This weight distribution also needs to be chosen, and the evaluation is provided in this section. In this work, the weight distribution is evaluated in terms of accurately predicting the next runtime, irrespective of the given safety factor.

Five weight distributions are evaluated for three different weight window sizes N_W. The evaluated weight distributions are shown in Table 7.4 and the weight window size is set to 10, 20 and 50 samples respectively. These weight distributions are used to guess the runtime of the next execution. Four different runtime scenarios are evaluated, which simulate different runtime patterns. The first scenario considers an unaltered runtime pattern, where no interference occurs. The second scenario considers a low frequency runtime shift, where the interference, and with that the runtime, varies close to a sinus dis-

Table 7.4: Weight distributions for the runtime estimation.

Type	Order	Weight Values	Description
Flat Average	N.A.	$w_j = \frac{1}{\lvert N_W \rvert}, \forall\, j$	All measurements weighted equally.
Middle k Avg.	Size	$w_j = \frac{1}{\lvert N_W \rvert - k}$, **if** $j \in [k/2, N_W - k/2]$	Middle k measurements are weighted equally.
Mean	Size	$w_j = 1$, **if** $j = \frac{N_W}{2}$	The mean of the measurement is used.
Linear Decay	Time	$w_j = \frac{N_W - j}{\sum_{k=0}^{N_W-1} N_W - k}$	Linear decay from most recent to least recent.
Exp. Decay	Time	$w_j = \frac{e^{-\lambda j}}{\sum_{k=0}^{N_W-1} e^{-\lambda k}}$	Exponential decay from most recent to least recent.

tribution with a period of 500 samples. The third scenario considers a high frequency runtime shift, where the interference varies with a sinus distribution with a period of 50 samples. Lastly a random interference is considered. Using these sets of weights to determine the next runtime estimation has a resulting error, which is defined as the absolute difference between the estimated runtime and the actual runtime.

Table 7.5 shows the average error in milliseconds between the estimated runtime and the measured runtime for weight distributions dependent on N_W samples, which are ordered by the sample values. The weight distributions using the last N_W samples ordered by time for ResNeXt-101 and multiple scenarios are evaluated.

As shown in Table 7.5, the interference scenario is the main factor influencing the estimation error, with a high frequency shift in interference and a randomized interference characteristic resulting in the highest estimation error. In all scenarios except for the random interference scenario, a smaller size of considered past values yielded a lower estimation error, with the best results being achieved with a weight window size of 10 samples. In the random interference scenario, the estimation error is reduced with a larger weight window size, with the best results being achieved with a weight window size of 50 samples. If no interference occurs or an interference with low frequency shift occurs, taking the mean of the last 10 samples yields the lowest estimation error. For a high frequency shift and a random interference, the average of the middle 60 % of the last 10 samples yields the lowest estimation error.

Table 7.5: Average estimation error for different weight distributions dependent on sample value. Lowest error per scenario is highlighted bold. If a value is the lowest for a given scenario, for all weight windows, it is underlined as well.

		Average			Middle 60 % Avg.			Mean		
Scenario		10	20	50	10	20	50	10	20	50
No Interf.	R.N.	0.148	0.156	0.166	0.129	0.131	0.138	**0.123**	0.128	0.136
	G.N.	0.113	0.119	0.128	0.095	0.098	0.104	**0.093**	0.095	0.102
	YV11	0.611	0.606	0.605	0.478	**0.477**	0.479	0.499	0.486	0.485
	GPU.	0.096	0.101	0.108	0.080	0.082	0.088	**0.078**	0.081	0.086
Low F. Shift	R.N.	0.185	0.244	0.427	0.170	0.230	0.420	**0.166**	0.229	0.388
	G.N.	0.139	0.177	0.295	0.124	0.165	0.289	**0.123**	0.164	0.268
	YV11	0.619	0.627	0.740	0.515	0.544	0.715	**0.498**	0.540	0.678
	GPU.	0.127	0.171	0.307	0.114	0.161	0.301	**0.112**	0.160	0.278
High F. Shift	R.N.	**0.789**	1.267	1.161	**0.789**	1.296	1.161	0.792	1.319	1.386
	G.N.	**0.522**	0.828	0.760	**0.522**	0.846	0.760	0.524	0.860	0.904
	YV11	1.116	1.634	1.506	**1.093**	1.643	1.505	1.098	1.672	1.754
	GPU.	**0.568**	0.913	0.837	**0.568**	0.934	0.837	0.571	0.951	0.997
Random	R.N.	0.520	0.515	0.512	0.525	0.514	**0.509**	0.540	0.521	0.517
	G.N.	0.352	0.348	0.347	0.353	0.346	**0.344**	0.362	0.351	0.349
	YV11	0.853	0.831	0.821	0.827	0.811	**0.804**	0.843	0.820	0.812
	GPU.	0.373	0.368	0.366	0.375	0.368	**0.365**	0.386	0.373	0.370

In addition to using the last N_W samples ordered by sample value, the last N_W samples ordered by time are also evaluated. The results for this evaluation are shown in Table 7.6. In this evaluation, the weight distributions using the last N_W samples ordered by time for ResNeXt-101 and multiple scenarios are evaluated. For the observed error values using time dependent weight distributions, the interference scenario is again the main factor influencing the estimation error, with a high frequency shift in interference and a randomized interference characteristic resulting in the highest estimation error. In these cases, a larger weight window size of 50 samples yielded a lower estimation error, with the best results being achieved with an exponential decay distribution for the high frequency shift scenario and the linear decay distribution for the low frequency shift scenario. For the no interference scenario and the low frequency shift scenario, a smaller weight window size of 10 samples yielded a lower estimation error.

Table 7.6: Average estimation error for different weight distributions dependent on time. Lowest error per scenario is highlighted bold. If a value is the lowest for a given scenario, for all weight windows, it is underlined as well.

		Linear Decay			Exp. Decay		
	Scenario	10	20	50	10	20	50
No Interf.	R.N.	**0.159**	0.777	0.170	0.163	0.181	0.203
	G.N.	**0.121**	0.587	0.131	0.124	0.137	0.155
	YV11	0.646	2.500	**0.609**	0.653	0.654	0.654
	GPU.	**0.103**	0.490	0.111	0.105	0.116	0.132
Low F. Shift	R.N.	**0.212**	1.508	0.464	0.222	0.345	0.725
	G.N.	**0.158**	1.055	0.319	0.165	0.245	0.491
	YV11	**0.660**	2.821	0.775	0.671	0.732	1.069
	GPU.	**0.145**	1.051	0.334	0.153	0.243	0.522
High F. Shift	R.N.	0.987	9.248	1.161	1.063	1.946	**0.485**
	G.N.	0.650	5.998	0.760	0.699	1.265	**0.336**
	YV11	1.340	11.298	1.507	1.428	2.426	**0.836**
	GPU.	0.711	6.667	0.837	0.766	1.403	**0.347**
Random	R.N.	0.532	1.566	**0.514**	0.540	0.545	0.556
	G.N.	0.361	1.087	**0.348**	0.366	0.370	0.380
	YV11	0.885	2.906	**0.825**	0.900	0.902	0.907
	GPU.	0.380	1.088	**0.368**	0.385	0.388	0.397

In comparison to the value dependent weight distributions, the time dependent weight distributions generally yielded a higher estimation error. The exception to this is a scenario, where the runtime varies with a high frequency, where the exponential decay distribution with a weight window size of 50 samples yielded a 20 % to 39 % lower estimation error than all value dependent weight distributions. This can be attributed to the fact that in a high frequency shift scenario, the most recent samples are more indicative of the next runtime than older samples and a sorting of the runtime length does not take this into account. The error value was reduced by approximately 25 % for the no interference and the low frequency shift scenario. For the random interference scenario, the error value is reduced by 1 % to 2.5 %, when using the value dependent weight distributions.

Finding 2: Weight Distribution for Online ANN Runtime Estimation

Because the sorting of the values by size takes extra execution time and the characteristics observed in the execution environment, the exponential decay distribution with a weight window size of 50 samples is selected for the further evaluation, as it provides the lowest estimation error for the high frequency shift scenario, which is expected to be the most common interference scenario in the execution environment.

7.1.3 Parameters of the Simulated Annealing Scheduler

In addition to the online scheduling of the anytime neural networks (ANNs), the offline scheduling for the time triggered scheduling approach also needs its parameters to be selected. The used simulated annealing-based algorithm employs a generalized approach, called dual annealing, which adds a secondary local search [117]. For this work, the implementation inside of the `scipy` library is used [157], which provides four configurable parameters.

The first changeable parameter is the `initial temperature`, which determines the probability of accepting worse solutions at the beginning of the optimization process. A higher `initial temperature` allows for a more extensive exploration of the solution space, while a lower `initial temperature` results in a more focused search. The second parameter is the `restart temperature ratio`. This ratio determines the ratio between the initial temperature and the temperature at which the reannealing process is initiated, which then resets the temperature to the initial temperature. A lower `restart temperature ratio` results in more reannealing, which can help to escape local minima, while a higher `restart temperature ratio` results in less reannealing, which can lead to a focused search.

For the dual annealing specialized parameters, the `visit` parameter is used for the visiting distribution and determines the distance, which is jumped due to the tail created by the visiting distribution. A higher `visit` parameter allows for larger jumps in the solution space, which can help to escape local minima, while a lower `visit` parameter results in smaller jumps, which can lead to a more focused search. Lastly, the `accept` parameter is used for the acceptance distribution and determines the probability of accepting worse solutions. A higher `accept` parameter allows for a higher probability of accepting worse solutions, which can help to escape local minima, while a lower `accept` parameter results in a lower probability of accepting worse solutions, which can lead to a more focused search.

The ablation study aims to find the best combination of these parameters to reduce the optimization time without leading to failure in finding a valid schedule. For this, a scheduling characteristic is defined, which is a subset of the evaluated optimizations of Section 7.3. Here, a model with the parameters listed in Table 7.7 is evaluated.

Table 7.7: Model instance used for the ablation study of the simulated annealing parameters for the offline scheduling of time triggered scheduling.

$N_{\mathcal{T}}$	$N_{\mathcal{U}}$	$N_{\mathcal{R}}$	**Pr. Cons.**	N_W[a]	**Util.**[b]	γ_i	**Mult. Ver.**[c]
6-60	6 (1 T.)	2	$0.1 \cdot N_{\mathcal{T}}$	$N_{\mathcal{T}}/2$	0.6	0.4	0.1

[a]Access Windows Tasks [%], [b]Utilization $(\sum_{\tau_i} C_i^{\upsilon})/(N_{\mathcal{U}} \cdot t_P)$ [%], [c]Multiple Version Tasks [%]

The deadline of the given tasks is set to be equal to the period for all tasks, while the number of tasks $n_{\mathcal{T}}$ is varied between 6 and 60 tasks with a step size of 2. The amount of execution units $N_{\mathcal{U}}$ is set to 6, to be close to real world systems, while the number of shared resources $N_{\mathcal{R}}$ is set to 2. One tenth of all tasks are assigned with precedence constraints and half of the tasks are set to have access windows. The utilization, calculated as $(\sum_i C_i^{\upsilon})/(N_{\mathcal{U}} \cdot t_P)$, is set to 0.6 and the allowed slack per task, γ_i is set to 0.4. Lastly, 10 % of the tasks are set to have multiple versions. For each parameter combination, 540 different valid model instances are generated, out of which 5 are then used to each evaluate a different combination of the hyperparameter combination for the simulated annealing algorithm. The evaluated hyperparameter combinations are shown in Table 7.8.

Table 7.8: Evaluated hyperparameter values for the simulated annealing algorithm.

Initial Temp.	**Re. Temp. Ratio**	**Visit Value**	**Accept**
5230, 10^4, $2 \cdot 10^4$	$2 \cdot 10^{-5}$, 10^{-4}, 10^{-3}	1.1, 2.0, 2.62, 2.9	-10^4, -500, -5.0

Here, the four available hyperparameters are varied inside their maximum values provided by the `scipy` implementation. For the `initial temperature`, the values of 5230, 10^4 and $2 \cdot 10^4$ are evaluated. The `restart temperature ratio` is evaluated with the values of $2 \cdot 10^{-5}$, 10^{-4} and 10^{-3}. For the `visit` parameter, the values of 1.1, 2.0, 2.62 and 2.9 are evaluated, while for the `accept` parameter, the values of -10^4, -500 and -5.0 are evaluated. Every combination of these parameters is evaluated for the model instances given by Table 7.8. Each parameter combination is evaluated with 5 different model instances.

Resulting in a total of 145 evaluations per hyperparameter combination. For each model instance with identical parameters, the average algorithm runtime is considered. A varying task amount is evaluated to test the hyperparameter combination for their characteristic with varying parameters and the task amount is chosen because it is the most varying model parameter in real world application due to the given platform staying constant throughout one development cycle.

The model instances are created in the following way. First, the hyperperiod is calculated as the leas common multiple of all deadlines. It is then divided into $\sum_{i=1}^{N_\mathcal{T}} N_\theta(i)/N_\mathcal{U}$ time frames for each execution unit, to reserve the runtime for the given tasks, since the tasks have to fixed runtime assigned for the test case generation. Should the number of tasks not be divisible by the number of execution units, the remaining tasks are assigned in a round-robin way to the execution units. Out of the task set, one half is given a period of t_P and one fourth is given a period of $0.5 \cdot t_P$, while the remaining one fourth is given a period of $0.25 \cdot t_P$. These time frames are then implied to be the execution time of the chosen version on the chosen execution unit for the given task. Should the given utilization value not equal 1, these times are then multiplied with the given utilization value to increase the processor idle time. Afterwards, the given amount of access windows are assigned to the tasks in a round-robin manner and are ensured to not overlap. Lastly, the specified task percentage are assigned a possible second version. Should this value lie over 1.0, then two or more additional versions are possible for the given task. The execution time of the task τ_i these version $v + x$ is set to $C_i^{v+x} = C_i^v \cdot (1 + 0.1 \cdot x)$, where C_i^v is the execution time of the first version of the task τ_i and x is the number of the additional version, starting with 1.

This is done to increase the complexity of the model. Using this strategy to generate model instances ensures that the given model instance leads to at least one valid solution, while providing a parameterizable test case. This strategy to generate valid but randomized task sets is further used in the following evaluations to provide a consistent test environment.

Since these model instances are generated with a known valid solution, the algorithm runtime is the main factor to evaluate the hyperparameter combinations, since the simulated annealing algorithm is proven to find the valid solution [158].

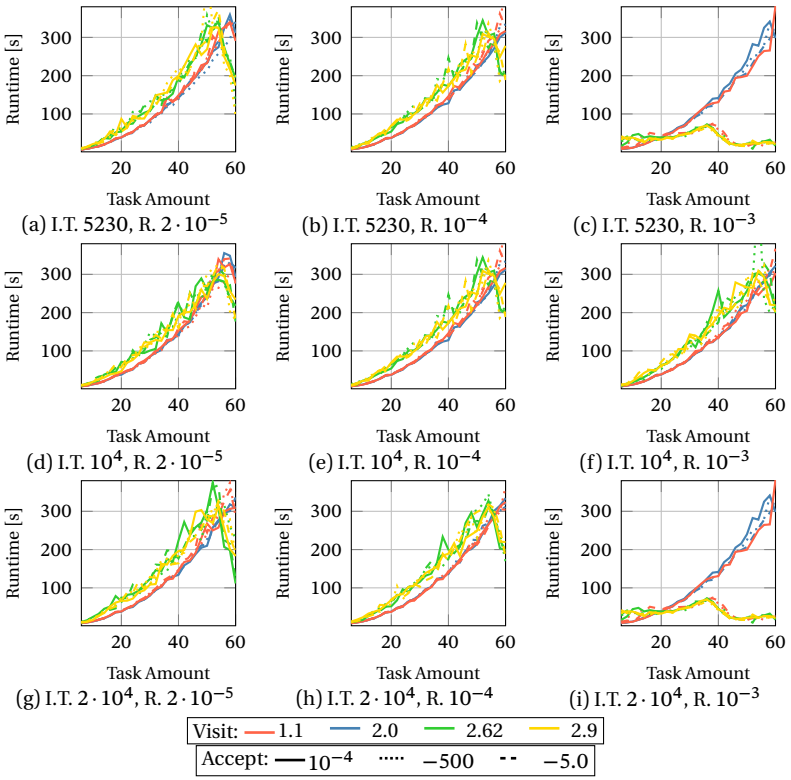


Figure 7.4: Ablation study for the simulated annealing optimization parameters for the offline scheduling of time triggered scheduling. The average algorithm runtime is evaluated for different combinations of the `initial temperature`, `restart temperature ratio`, `visit` and `accept` parameter.

Figure 7.4 shows the average runtime for the different hyperparameter combinations in relation to the number of tasks. The average runtime is shown on the y-axis scale, while the number of tasks is shown on the x-axis. Different lines represent one combination with constant values for the `initial temperature`, `restart temperature ratio`, `visit` and `accept` parameter. Inside one diagram, four graph colors are shown, one for each visit value. The graph styles represent the different `accept` values. In the combined Figure 7.4, 9 sub figures are shown. The diagram rows vary the `initial temperature` and the columns vary the `restart temperature ratio`.

The first noticeable trend considers different impact characteristics of the `accept` parameter. For a `restart temperature ratio` of $2 \cdot 10^{-5}$, the impact of the `accept` parameter is larger than when choosing a `restart temperature ratio` of 10^{-3}, except for the `initial temperature` of 10^4. For the combination of a `restart temperature ratio` of 10^{-3} and an `initial temperature` of $2 \cdot 10^4$ or 5230, the effect of the visit parameter is the significant contribution from a task amount of 40 tasks and more. Here, the strongest effect is observed. The runtime is significantly lower than in other experiments and even decreases with a rise in the number of tasks. All other experiments show an increase in the runtime with the amount of tasks. Another pronounced trend lies in the `visit` value choice, where a visit value of less than 2.62 leads to a steady increase in runtime with the number of tasks and a very uniform runtime across all other hyperparameter combinations. For a visit value of 2.62 and higher, the runtime decreases starting from either 40 tasks or 55 tasks, depending on the specific combination. Lastly, the effect of other hyperparameters increase with a visit value of more than 2.62.

Finding 3: Simulated Annealing Parameters for Offline Scheduling of Time Triggered Scheduling

The best combination of parameters for the given model instances is an initial temperature of $2 \cdot 10^4$*, a restart temperature ratio of* 10^{-3} *and a visit value of* > 2.6*. For the accept value, any value is acceptable for this case, but a value of* -5.0 *is chosen since this value has the least runtime in other experiments.*

These parameters and the methodology to generate test cases is used for the evaluation of the time triggered scheduling approach.

7.2 Scheduling Success Rate of Polyhedron-Based Approach

The simulated annealing-based scheduling algorithm for time-triggered scheduling (TTS) was called should the first polyhedron-based scheduling attempt not provide a valid schedule as the solution. The simulated annealing algorithm found a valid solution in 100 % of all cases, where a valid solution existed, albeit at a considerably runtime increase. The algorithm runtime in respect to different parameter combinations was evaluated in Section 7.3. This chapter evaluated the success rate of the polyhedron-based scheduling approach.

This evaluation of the scheduling success rate used the same strategy to generate model instances as described in Section 7.1.3 with the model values given by Table 7.9. The evaluation was performed on 5 instances for each parameter combination. The results were shown in Figure 7.5.

Table 7.9: Model instances used for the evaluation of the scheduling success rate of the polyhedron-based approach.

$N_{\mathcal{T}}$	$N_{\mathcal{U}}$	$N_{\mathcal{R}}$	**Pr. Cons.**	N_W	**Util.**	γ_i	**Mult. Ver.**
36-60	4 (1 T.)	1	$0.1 \cdot N_{\mathcal{T}}$	$(4 + i \cdot N_{\mathcal{T}}/4)$[a]	0.4-1.0	0.1	0.1

[a] $i \in [0,3]$

To stay consistent with the existing measurements with the polyhedron-based approach, as presented in [Sta24a], the model parameters were chosen to emulate the same model instance. The chosen model instance varied the number of tasks, the number access windows as well as the utilization. The number of tasks varied between 36 and 60, which equated to 9 to 15 tasks per execution unit. The number of access windows varied between 4 and 4.75 per execution unit and the utilization varied between 0.4 and 1.0. Four execution units of one type were modeled together with one shared resource. 10 % of tasks had precedence constraints and the slack in the deadline was set to 10 %. As in Section 7.1.3, half of all tasks had a period equal to t_P, one quarter was set to have a period of $0.5 \cdot t_P$ and the remaining quarter was set to have a period of $0.25 \cdot t_P$.

This model reached similar success rates as were expected if the method presented in the original publication [Sta25a] were used. This also showed the transferability of the model from [Sta24a] into the model, which was also used in the simulated annealing-based approach. The results were shown in Figure 7.5.

The results also lay close to the expected success rates, which were measured in [Sta25a]. The success rate stayed close to 100 % until a utilization of around 70 % was reached. After that, the success rate dropped depending on the number of access windows per task. For one access window per task, the success rate dropped to about 50 % at a utilization of 90 % and to about 20 % at a utilization of 100 %. This was close to and followed the characteristic of the first experiment in [Sta25a], where one access window per task and one task with an access window over all resources was modeled. A gray area signified, where the simulated annealing-based scheduling approach was needed to enhance the solution, since no valid schedule was found, even though

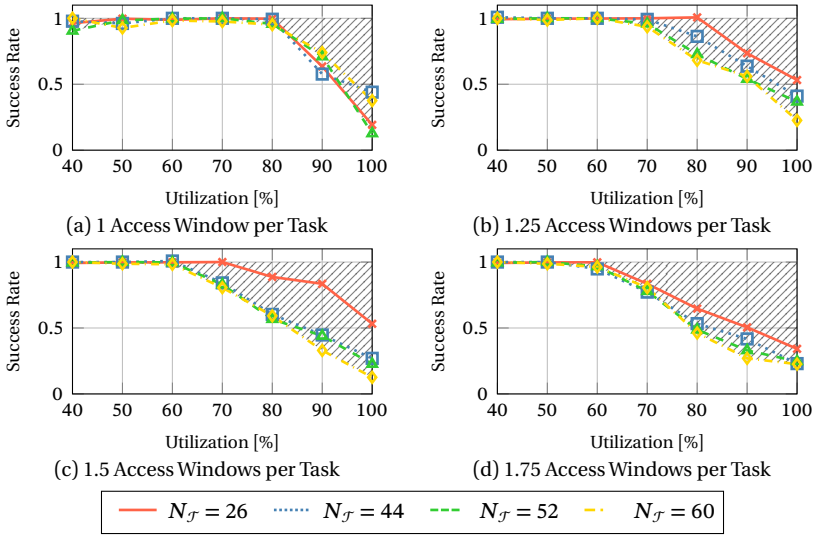


Figure 7.5: Success rate for the polyhedron-based scheduling approach. The success rate is evaluated for different numbers of access windows per execution unit and different utilizations. The success rate is shown for different numbers of tasks per execution unit.

the test case was generated to guarantee a valid solution. With the number of tasks per execution unit, a small decrease in success rate was observed. This was expected, as the number of tasks per execution unit increased, the search space for the scheduling algorithm also increased. For a utilization of 100 %, the success rate dropped about 25 % points between 26 and 60 tasks.

For 1.25 access windows per tasks, the success rate started to drop earlier, from about 70 % utilization, and dropped to about 25 % at a utilization of 100 % for 60 tasks. When 26 tasks are scheduled, the success rate lied at 50 %. For 1.5 access windows per task, the success rate started to drop at about 60 % utilization and dropped as low as 20 % at a utilization of 100 %. Here, a significant difference between 26 and 60 tasks is observed, where the success rate stayed higher significantly longer for 26 tasks.

Lastly, for 1.75 access windows per task, the success rate started to drop at about 60 % and dropped to about 25 % at a utilization of 100 %. Here, the difference between 26 and 60 tasks was less pronounced and the drop went on more linearly.

Finding 4: Success Rate of Polyhedron-based Scheduling

The polyhedron-based scheduling approach for TTS has a high success rate for utilizations up to around 70 %, but the success rate drops significantly for higher utilizations, especially as the number of access windows per task increases. The number of tasks per execution unit also has an impact on the success rate, with a higher number of tasks leading to a lower success rate. It is thus best used to generate schedules for low to medium utilizations or to generate a starting solution for the simulated annealing-based scheduling approach.

Since the polyhedron-based scheduling approach can not find valid schedules for high utilizations, the simulated annealing-based scheduling approach is needed to enhance the solution. The gray area in Figure 7.5 signifies where the simulated annealing-based scheduling approach is needed to enhance the solution, since no valid schedule is found, even though the test case is generated to guarantee a valid solution.

The simulated annealing-based approach is executed until this valid schedule is found. Should no valid schedule be found within the specified time, a restart is performed with a new starting seed and solution. To evaluate the complexity of the simulated annealing-based scheduling approach, the runtime is measured in the next section.

7.3 Runtime of Time-Triggered Scheduling Algorithm

Because the simulated annealing-based scheduling approach is guaranteed to find a valid solution in all valid cases [158], the runtime was needed to evaluate the performance of the scheduling algorithm and the complexity of the scheduling problem.

First, the runtime of the polyhedron-based scheduling approach was evaluated in Section 7.3.1. In a separate evaluation, the runtime of the scheduling algorithm was compared for different starting scenarios. These scenarios used a simple but not valid starting solution, a random initialization or a solution equal to either bounds of the search space. This was shown in Section 7.3.2. Section 7.3.3 shows the runtime evaluation for the simulated annealing-based scheduling and Section 7.3.4 shows an evaluation of the cost function characteristics. This evaluation was performed on an AMD Epyc 9374F 32-Core CPU with a reserved core for the execution.

7.3.1 Runtime of Polyhedron-based Scheduling Approach

The runtime for the polyhedron-based scheduling approach was measured for the same model instances as used for the evaluation of the scheduling success rate in Section 7.2. Again, the model parameters were chosen to emulate the same model instance as in [Sta24a] to stay consistent with the existing measurements with the polyhedron-based approach. Figure 7.6 showed the runtime for the polyhedron-based scheduling approach for different numbers of access windows per task and different utilization levels.

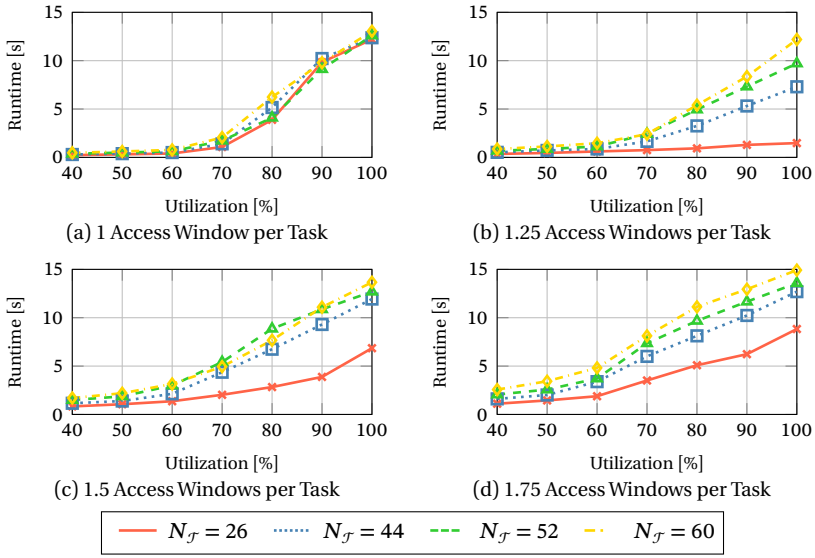


Figure 7.6: Runtime evaluation for polyhedron-based scheduling with varying processor utilization. The runtime is evaluated for different numbers of access windows per execution unit and different utilization levels. The runtime is shown for different numbers of tasks.

It was expected that the runtime of the polyhedron-based scheduling approach would be inverse proportional to the success rate, because the scheduling problem became more complex with a rise in utilization. This expectation was not directly confirmed with the experiment, but a trend was observed, where the runtime increased with a rise in utilization. For one access window per task, no significant runtime increase was observed until the utilization reached 70 %. After that, the runtime increased sharply from a runtime in seconds of single digits to a maximum runtime of 10 s at a utilization of 90 %.

At 100 % utilization, the runtime continued to rise but not as sharply as before, with a maximum runtime of about 12.5 s.

For 1.25 access windows per task, a big difference in runtime is observed between 26 and 60 tasks, starting at 70 % utilization. A sharp rise in the runtime characteristic is observed between 26, 44, 52 and 60 tasks. Starting at 60 % utilization, the runtime increases to 12.5 s for 60 tasks at 100 % utilization, while maxing out at 2.5 s for the experiment with 26 tasks.

1.5 access windows per task shows a similar trend, but with a less pronounced difference between 26 and 60 tasks, where the maximum value lies again at 12.5 s for 60 tasks at 100 % utilization, while maxing out at 7.5 s for the experiment with 26 tasks. While the characteristic of the runtime increase is similar to the one observed for 1 or 1.25 access windows per task, the runtime increase starts and follows a more linear curve. Also the difference between 26 and 60 tasks observes a more pronounced difference between the experiments with a rise in access windows per tasks because the randomized test case generation might lead to a difference in the complexity of the scheduling problem between experiments.

For 1.75 access windows per task, the runtime increase starts even earlier, at 50 % to 60 % utilization, and the runtime maxes out at 15 s for 60 tasks at 100 % utilization, while maxing out at 9.5 s for the experiment with 26 tasks.

When comparing these results to the success rate evaluation in Section 7.2, it can be observed that the runtime increase starts at around 10 % points before the success rate starts to drop. This leads to the interpretation, that the complexity increase in the scheduling problem starts before the scheduling algorithm fails to find a valid solution and that the sweet-spot for the polyhedron-based scheduling approach lies between 50 % and 70 % utilization, depending on the number of access windows per task and the number of tasks per execution unit. This is because of the lower runtime in comparison to the simulated annealing-based approach (as seen in Section 7.3.3), while still providing a valid solution. After that, the runtime increase becomes more significant while the success rate starts to drop and the simulated annealing-based scheduling approach is needed to enhance the solution, since no valid schedule is found, even though the test case is generated to guarantee a valid solution.

Finding 5: Runtime of Polyhedron-based Scheduling

The runtime of the polyhedron-based scheduling approach increases with a rise in utilization, with a significant increase observed after around 70 % utilization. The runtime also increases with the number of access windows per task and the number of tasks per execution unit. The sweet-spot for the polyhedron-based scheduling approach lies between 50 % and 70 % utilization, depending on the number of access windows per task and the number of tasks per execution unit.

7.3.2 Runtime Evaluation for Different Starting Solutions

Before the runtime evaluation of the simulated annealing-based scheduling approach, the total algorithm runtime was evaluated for a subset of all test cases. The test cases are selected to represent a range of utilization levels and problem complexities and are displayed in Table 7.10. This allows for interpreting the effect of the starting solution on the runtime.

Table 7.10: Traced model parameters with values and used markers.

Index	Mark	Fixed parameters
1	○	$N_{\mathcal{T}} = 40$, Ut. = 0.6, $N_{\mathcal{U}} = 4$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
2	×	$N_{\mathcal{T}} = 40$, Ut. = 0.8, $N_{\mathcal{U}} = 4$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
3	◇	$N_{\mathcal{T}} = 40$, Ut. = 0.6, $N_{\mathcal{U}} = 6$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
4	⊗	$N_{\mathcal{T}} = 40$, Ut. = 0.8, $N_{\mathcal{U}} = 6$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
5	⋆	$N_{\mathcal{T}} = 40$, Ut. = 0.6, $N_{\mathcal{U}} = 8$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
6	⬠	$N_{\mathcal{T}} = 40$, Ut. = 0.8, $N_{\mathcal{U}} = 8$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
7	□	$N_{\mathcal{T}} = 100$, Ut. = 0.6, $N_{\mathcal{U}} = 4$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
8	+	$N_{\mathcal{T}} = 100$, Ut. = 0.8, $N_{\mathcal{U}} = 4$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
9	△	$N_{\mathcal{T}} = 100$, Ut. = 0.6, $N_{\mathcal{U}} = 6$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
10	⊕	$N_{\mathcal{T}} = 100$, Ut. = 0.8, $N_{\mathcal{U}} = 6$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
11	∗	$N_{\mathcal{T}} = 100$, Ut. = 0.6, $N_{\mathcal{U}} = 8$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2
12	⅄	$N_{\mathcal{T}} = 100$, Ut. = 0.8, $N_{\mathcal{U}} = 8$, $\gamma_i = 0.2$, MV. = 0.2, PC. = 0.2

MV. = Multi Version, PC. = Precedence Constraint, Ut. = Utilization

These parameters are additionally marked in the runtime evaluation figures to provide an easy visual comparison. For all experiments, 12 values resulting form identical parameter configurations are shown in the different diagrams with constant point marks. This allows for the contextualization of the results,

by showing the effect of the varied parameters in comparison to orthogonal parameters. Table 7.10 shows these traced parameters. Given these parameters and the hyperparameters found in Section 7.1.3, the runtime is measured for the different starting solutions for 100 different problem instances and the average runtime for these parameters is shown in Table 7.11. The starting solution $\vec{x}_0$ is varied between a random initialization, an initialization with the minimum possible values, an initialization with the maximum possible values and an initialization based on the polyhedron-based scheduling approach. The runtime is measured for the time until a valid solution is found, which is the same as in the runtime evaluation of the simulated annealing-based scheduling approach in Section 7.3.3. The results show the marked subset from all experiments. Slight differences are the result of the random experiment values. Table 7.11 shows the runtime evaluation for different starting solutions, the lowest runtime is marked in bold. In the table, the runtime does not differ largely between the different starting solutions, with all starting solutions showing a similar runtime, with the shortest runtime observed for the random initialization in most cases. Only the polyhedron-based scheduling approach might provide some runtime improvement, should the polyhedron-based scheduling approach not find a valid solution. In the given experiments, this is expected to be the case with a deadline success rate of about 30 %.

Table 7.11: Runtime evaluation for different starting vectors in seconds [s].

Mark	○	□	×	+	◊	△	⊗	⊕	⋆	∗	⬠	⅄
Rand.	702	593	581	446	534	724	3399	6211	3997	3269	4835	3625
$\texttt{min}(\vec{x}_0)$	878	729	753	561	698	928	4331	7748	5228	4089	6299	4675
$\texttt{max}(\vec{x}_0)$	878	729	753	561	698	928	4331	7748	5228	4089	6299	4675
Gen.	702	583	603	449	559	743	3465	6198	4183	3271	5039	3740

MV. = Multi Version, PC. = Precedence Constraint, Ut. = Utilization

Finding 6: Runtime of Simulated Annealing-based Scheduling with Different Starting Solutions

The starting vector for the simulated annealing-based scheduling approach does not significantly affect the runtime, with all starting solutions showing a similar runtime. A maximum difference of about 20 % was observed between the different starting solutions, with the pre-generated starting solution based on the polyhedron-based scheduling approach providing a slight runtime improvement in some cases. The random initialization showed a similar but slightly more often a higher runtime.

7.3.3 Runtime of Simulated Annealing-based Scheduling

The runtime evaluation of the simulated annealing-based scheduling approach is performed for a more thorough set of model instances than the polyhedron-based scheduling approach, because of the falling success rate of the polyhedron-based scheduling approach at higher utilizations. Furthermore, the runtime evaluation builds on the ablation study for the simulated annealing-based scheduling approach in Section 7.1.3. While the ablation study was used to find the best hyperparameter combination, this section, extends the evaluation by taking these hyperparameters and evaluating the algorithm runtime to gain a better understanding of the runtime characteristics for different model characteristics.

The evaluation is performed by taking different slices from the same large test set, which is described in Table 7.12. In all cases, the same hyperparameters are used, which are found to be the best performing ones in Section 7.1.3. While the runtime evaluation was performed using a similar structure as the evaluation of the HM2GP scheduler [Sta25b], the evaluation of this extended model also goes beyond the test cases presented in [Sta25b] by evaluating a more diverse set of model instances, which are generated with a wider range of model parameters, as shown in Table 7.12.

Table 7.12: Model instances used for the evaluation of the runtime evaluation of the simulated annealing-based approach.

Test Goal	Main Varied Parameters	Secondary Varied Parameters	Fixed Parameters
Task Amount	$N_{\mathcal{T}} \in \{10, \dots, 100\}$	Ut. $\in \{0.6, 0.8, 1.0\}$, $\gamma_i \in \{0.0, 0.2, 0.4\}$, $N_{\mathcal{U}} \in \{4, 6, 8\}$	MV. $\in \{0.2\}$, PC. $\in \{0.2\}$
Unit Amount	$N_{\mathcal{U}} \in \{4, \dots, 8\}$	$N_{\mathcal{T}} \in \{40, 70, 100\}$, $\gamma_i \in \{0.0, 0.2, 0.4\}$, Ut. $\in \{0.6, 0.8, 1.0\}$	MV. $\in \{0.2\}$, PC. $\in \{0.2\}$
Utilization	$Ut. \in \{0.6, \dots, 1.0\}$	$N_{\mathcal{T}} \in \{40, 70, 100\}$, $\gamma_i \in \{0.0, 0.2, 0.4\}$, $N_{\mathcal{U}} \in \{4, 6, 8\}$	MV. $\in \{0.2\}$, PC. $\in \{0.2\}$
Deadline Slack	$\gamma_i \in \{0.0, \dots, 0.4\}$	$N_{\mathcal{T}} \in \{40, 70, 100\}$, Ut. $\in \{0.6, 0.8, 1.0\}$, $N_{\mathcal{U}} \in \{4, 6, 8\}$	MV. $\in \{0.2\}$, PC. $\in \{0.2\}$
Multi Version	MV. $\in \{0.0, \dots, 0.4\}$	$N_{\mathcal{T}} \in \{40, 70, 100\}$, Ut. $\in \{0.6, 0.8, 1.0\}$, $N_{\mathcal{U}} \in \{4, 6, 8\}$	$\gamma_i = 0.2$, PC. $\in \{0.2\}$
Prec. Constr.	PC. $\in \{0.0, \dots, 0.4\}$	$N_{\mathcal{T}} \in \{40, 70, 100\}$, Ut. $\in \{0.6, 0.8, 1.0\}$, $N_{\mathcal{U}} \in \{4, 6, 8\}$	$\gamma_i = 0.2$, MV. $\in \{0.2\}$

MV. = Multi Version, PC. = Precedence Constraint, Ut. = Utilization

Six different evaluation experiments are performed. One parameter is considered the main varied parameter, while two secondary parameters are selected to provide context. The first evaluation varies the task amount between 10 and 100 and provides the secondary parameters of utilization of 0.6, 0.8 and 1.0 and the deadline slack of 0.0, 0.2 and 0.4. The number of execution units is varied in $N_{\mathcal{U}} \in \{4, 6, 8\}$. The second evaluation varies the unit amount between 4 and 8 and provides the secondary parameters of 40, 70 and 100 for the task amount and the utilization in 0.6, 0.8 and 1.0 as well as the deadline slack variation of 0.0, 0.2 and 0.4. The third evaluation varies the utilization between 0.6 and 1.0 and provides the secondary parameters of task amount as 40, 70 and 100 as well as a deadline slack of 0.0, 0.2 and 0.4 and the number of execution units of $N_{\mathcal{U}} \in \{4, 6, 8\}$. The fourth evaluation varies the deadline slack between 0.0 and 0.4 and provides the secondary parameters of task amount in 40, 70 and 100, a utilization of 0.6, 0.8 and 1.0 as well as the number of execution units of $N_{\mathcal{U}} \in \{4, 6, 8\}$. The fifth evaluation varies the percentage of multi-version tasks between 0.0 and 0.4 and provides the secondary parameters of task amount of 40, 70 and 100, a utilization in 0.6, 0.8 and 1.0 as well as the number of execution units of $N_{\mathcal{U}} \in \{4, 6, 8\}$. The sixth evaluation varies the percentage of tasks with precedence constraints between 0.0 and 0.4 and provides the secondary parameters of task amount of 40, 70 and 100, a utilization in 0.6, 0.8 and 1.0 as well as the number of execution units of $N_{\mathcal{U}} \in \{4, 6, 8\}$. If not varied, the percentage of tasks with multiple versions is fixed to 0.2, the percentage of tasks with precedence constraints is fixed to 0.2 and the deadline slack is fixed to 0.2. Furthermore, two execution unit types are considered in all experiments as well as 3 different shared resources. The amount of access windows is set to 2 per task.

All experiments were repeated 5 times for each model configuration and the average runtime is shown in the corresponding figures. Each model instance is generated to guarantee a valid solution as in other experiments. The model instance is generated in the same way as described in Section 7.1.3. The runtime is then measured for the time until a valid solution is found, which is guaranteed to exist. Since the starting vector is initialized as pregenerated by the polyhedron-based scheduling approach, as described in Section 7.3.2, this pregeneration is included in the runtime measurement. For the evaluation of the runtime for a varying task amount, the runtime was expected to rise with the amount of tasks, because the scheduling problem becomes more complex and the solution vector increases in size. All other parameters are expected to have a smaller effect on the runtime of the algorithm. In all figures, the algorithm runtimes of the traced experiments in Table 7.10 are marked with their corresponding symbols.

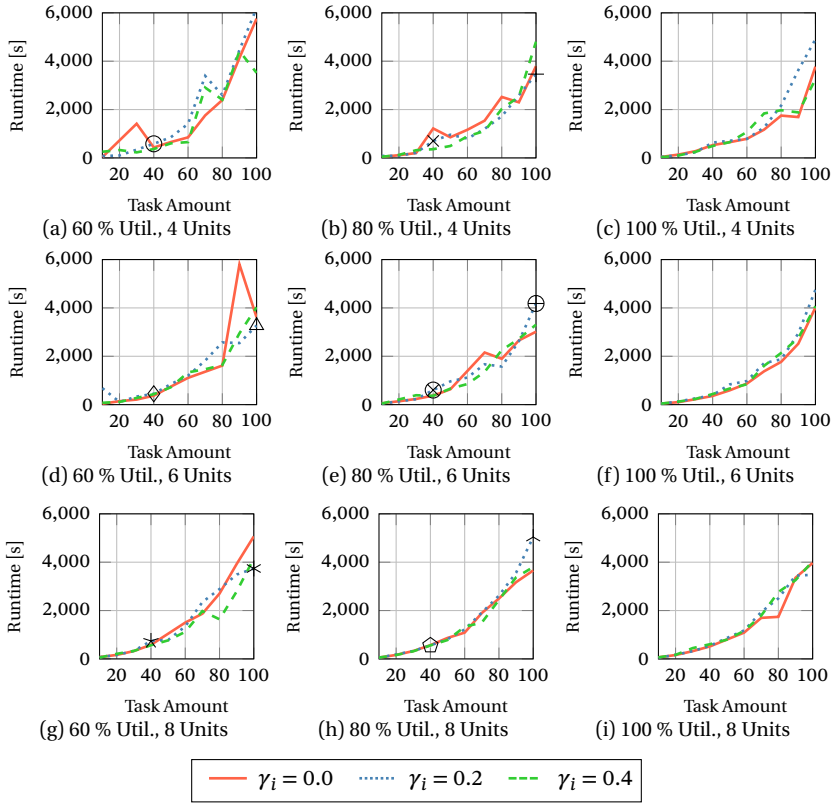


Figure 7.7: Simulated Annealing evaluation for varying task amount.

As seen in Figure 7.7, an increase in runtime was observed with a rise in task amount. As expected, the overall runtime followed a less varying trend for higher utilization values between diagrams. Similarly, inside one diagram, the trend of higher deadline slack values leading to a lower runtime is observed in a small amount. For the amount of execution units, a smaller increase in runtime was expected and observed. The maximum runtime between all experiments was observed for 100 tasks and a slack of 0.0, irrespective of other parameters. Here, the runtime lied between 4,000 s and 6,000 s. A higher amount of execution units was observed to lead to less variety in the runtime between experiments. The minimum runtime was observed for 40 tasks, with a runtime of about 45 seconds.

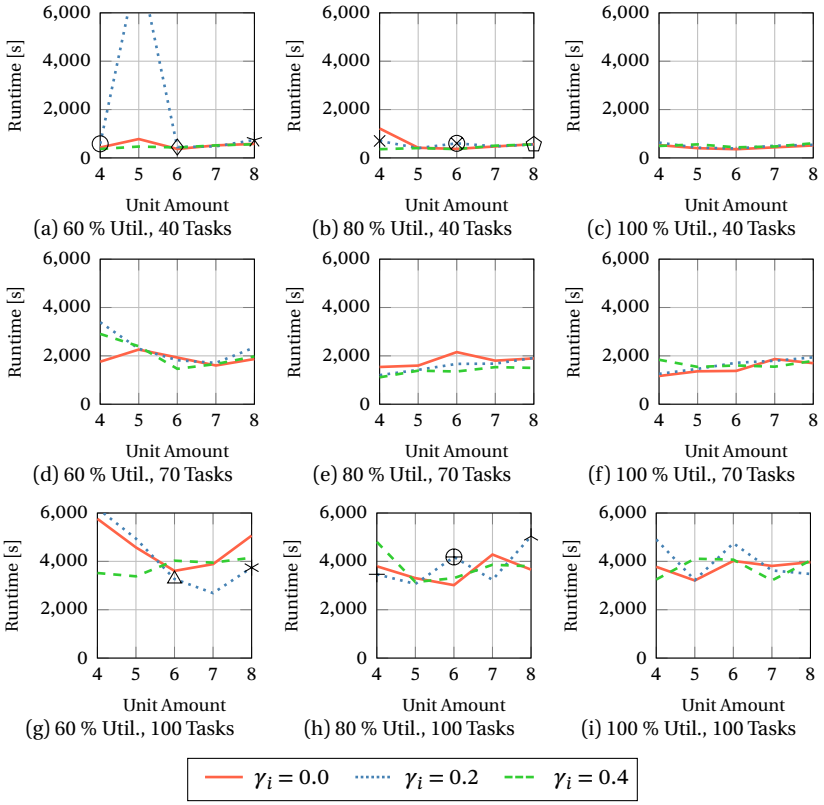


Figure 7.8: Simulated Annealing evaluation for varying unit amount.

Figure 7.8 shows another dimension of the same experiment, by tracing the amount of execution units. Different task amounts are shown in the different rows, the utilization is varied in the columns. As expected, a smaller increase in runtime is observed while increasing the amount of execution units, but still, the task amount was the dominant factor. This is due because an increase in unit amount allows for more possibilities to place tasks, but simultaneously increases the solution vector size, which leads to a more complex scheduling problem. The maximum runtime between all experiments was observed for 100 tasks at 100 % utilization and a deadline slack of 0.0 on 8 execution units, with a runtime of about 6,000 s seconds. The minimum runtime was again observed for 10 tasks at 40 % utilization and a deadline slack of 0.4 on 4 execution units.

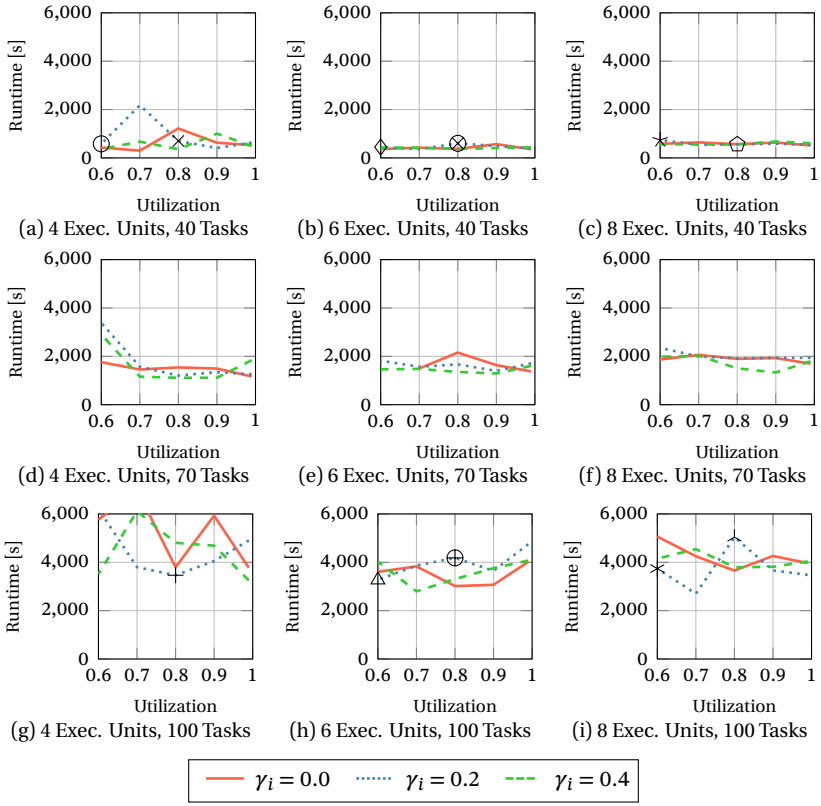


Figure 7.9: Simulated Annealing evaluation for varying utilization.

Figure 7.9, on the other hand, shows the runtime in relation to the utilization. As expected, an increase in runtime is observed with a rise in utilization, but the increase is similarly small as the increase in other parameters except for the task amount. Because the complexity of the scheduling problem increases as well. For 40 tasks, more variety was observed in respect to the utilization. Instead of a strong runtime increase, the strongest effect was that the variance in runtime decreased with the utilization with one figure. This effect is increased with the amount of tasks, since the solution vector increases in size and with that a significant increase in scheduling complexity is observed. A lower utilization lead to a higher runtime in the cases, where more slack was allowed, and less execution units were available.

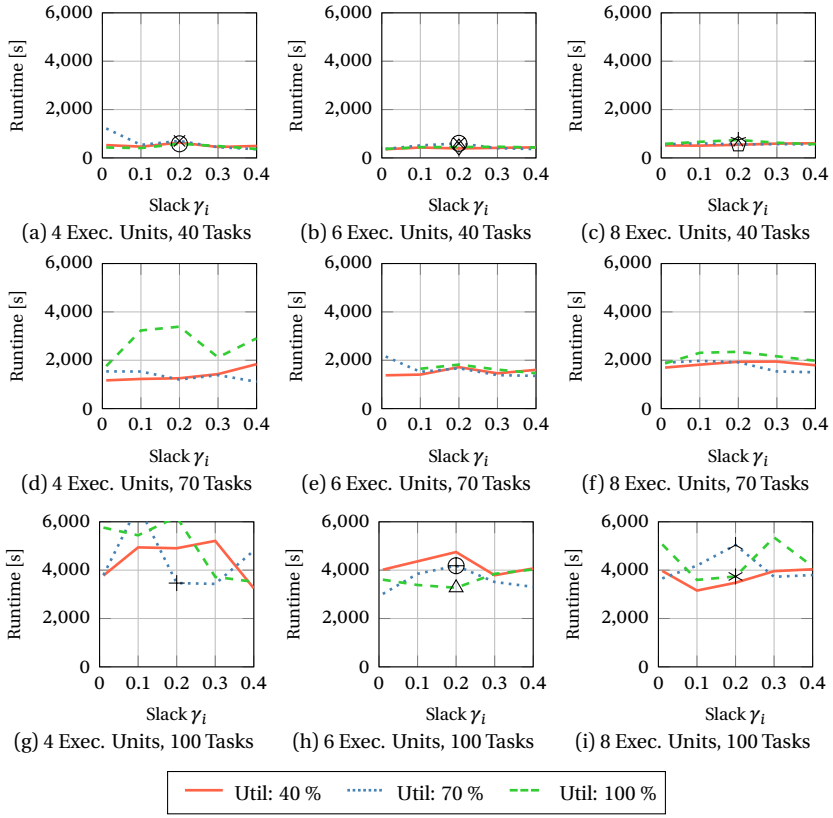


Figure 7.10: Simulated Annealing evaluation for varying deadline slack.

The effect of the available deadline slack, as defined by γ_i, is shown in Figure 7.10. It was expected that a lower deadline slack would lead to a higher runtime, because of the lower flexibility in positioning the tasks. With the utilization being a secondary parameter, the effect of the deadline slack is more pronounced for a higher amount of tasks, because of the lower flexibility in scheduling tasks. The runtime increased with a lower slack slightly in some cases and decreased slightly in others, such that no clear trend was observed. The variance in runtime was less for a lower slack than for a higher amount of slack. The maximum runtime between all experiments was observed for 100 tasks at 100 % utilization and a deadline slack of 0.2 on 4 execution units.

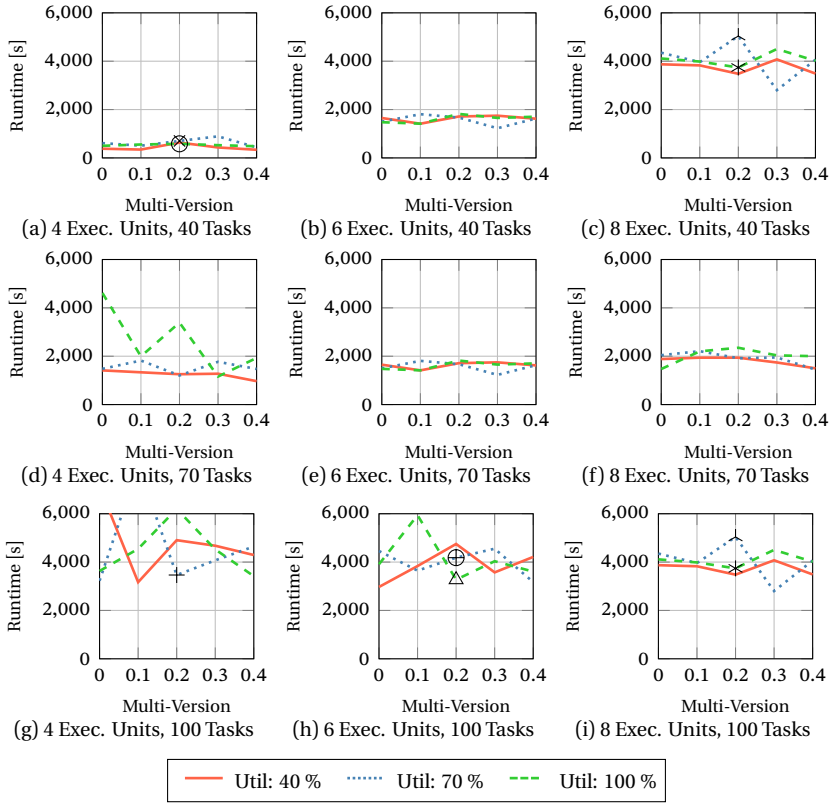


Figure 7.11: Simulated Annealing evaluation for varying multi-version tasks.

Figure 7.11 shows the runtime evaluation for a varying percentage of multi-version tasks. It was expected from previous experiments that an increase in the number of tasks (from 40 to 100) has a greater effect on the runtime. With more multi-version tasks, the constraints of the solution vector increase in complexity, but the size of the vector stays the same. Because of this, the runtime did not increase significantly with the number of multi-version tasks. The percentage of multi-version tasks thus had a similar effect on the runtime than previous experiments. This strengthens the previous conclusion that the size of the solution vector is the primary factor affecting the runtime. For a utilization of 100 %, the runtime varied most for 70 tasks and 4 execution units. The secondary parameters of execution units and task amounts had a more significant effect.

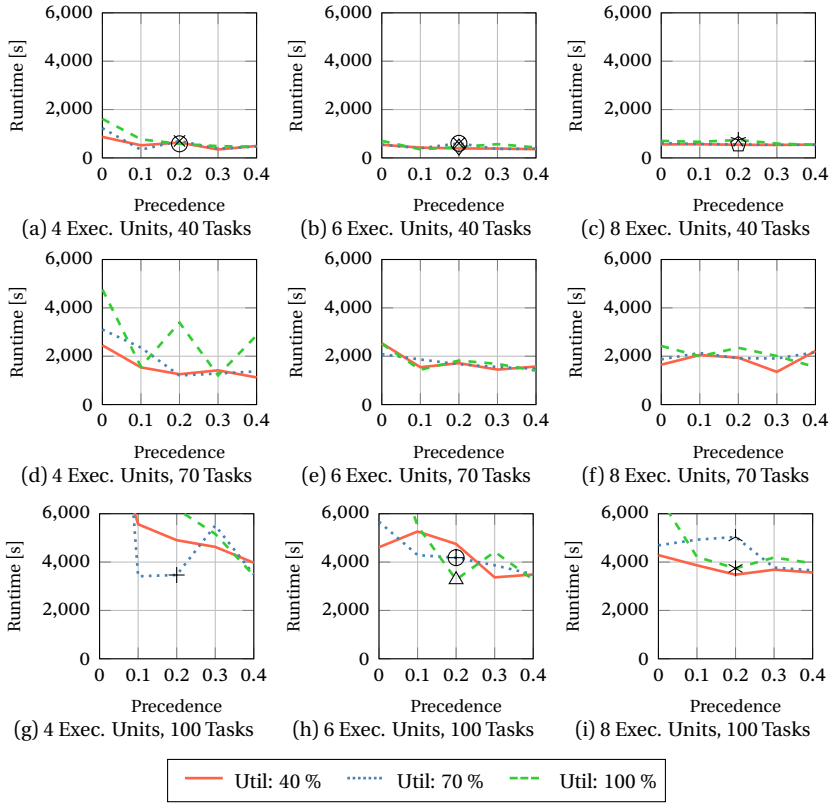


Figure 7.12: Simulated Annealing evaluation for varying precedence constraints.

Lastly, Figure 7.12 shows the runtime evaluation for a varying percentage of tasks with precedence constraints. A small decrease in runtime was observed for a higher percentage of tasks with precedence constraints, which goes against the general trend and the expected behavior. As before, the runtime increase was not significant for a higher percentage of tasks with precedence constraints, but the effect was more pronounced for a higher amount of execution units and tasks. This can be interpreted as a result of the constraining effect on the solution vector, further restricting the search space and thus leading to a faster convergence. The secondary parameters of execution units and task amounts had a more significant effect. Again, for 4 execution units and 70 tasks, a varying runtime between the different precedence percentages was observed.

Finding 7: Runtime of Simulated Annealing-based Scheduling

The runtime of the simulated annealing-based scheduling approach increases the strongest with a rise of the task amount. Other parameters, such as the amount of execution units, the utilization, the deadline slack, the percentage of multi-version tasks and the percentage of tasks with precedence constraints have a smaller effect on the runtime. The maximum runtime was observed for 100 tasks at 100 % utilization and a deadline slack of 0.0 on 8 execution units, with a runtime of about 6,000 s seconds. The minimum runtime was observed for 10 tasks at 40 % utilization and a deadline slack of 0.4 on 4 execution units.

While the runtime evaluation provides insight into the performance of the simulated annealing-based scheduling approach, it is also important to evaluate the behavior of the cost function during the optimization process.

7.3.4 Evaluation of Cost Function Characteristic

Given the parameters presented in Table 7.10, the change in cost function over time during the simulated annealing process was evaluated. This allows for the interpretation of the cost function behavior, by showing the cost function values during the optimization process and the characteristics over the optimization time. As expected, the cost function values start comparable in the order of magnitude for all experiments of one parameter set and decreases roughly logarithmically. Figure 7.13 shows the cost function values during the optimization process for each of the given parameter sets. Each graph shows the cost function values for the five repeated optimization runs. As soon as the cost function value reaches 0, a valid schedule is found and the optimization process can be stopped. The starting cost function is a good indicator for the complexity of the scheduling problem, since it shows how many constraints are violated by the initial solution. For higher task amounts unit amounts, less deadline slack and more precedence constraints, the cost function values start at a higher value and show a more significant decrease during the beginning of the optimization process. For the opposite, a valid schedule is found after a shorter time period and the cost function value started at a lower value.

Additionally, the cost function values at any given time give an indication of the current solution quality, because the cost function value directly translates to the number and severity of violated constraints. With the cost function characteristic being similar for all experiments and the values translating into a

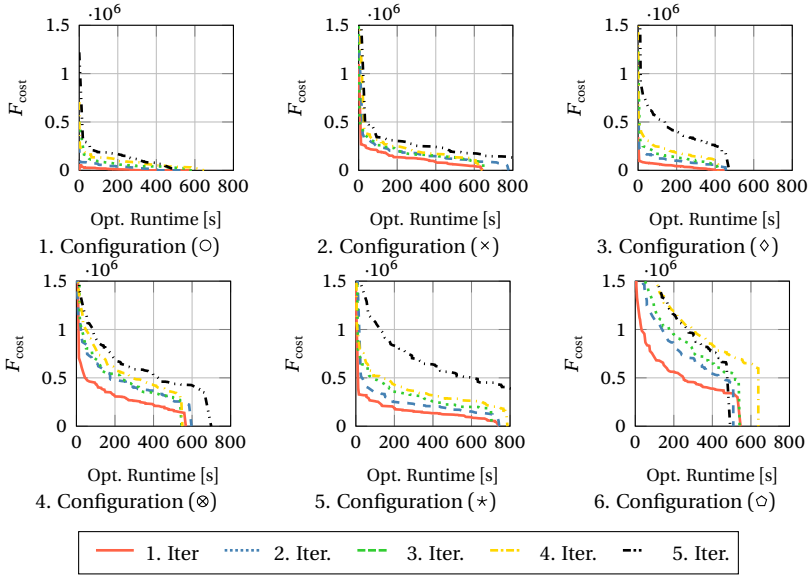


Figure 7.13: Cost function over iteration time for the first 6 configurations.

solution quality, comparing the solution quality between different experiments is possible. As more difficult scheduling problems show a higher starting cost and take longer to reach a cost function value of 0 (a valid schedule), the solution quality at any given time is lower than for easier scheduling problems. For easier scheduling problems, a valid schedule is found after a shorter time period and the cost function value started at a lower value, which means that the solution quality is higher at any given time during the optimization process. Because an invalid schedule is always an invalid schedule and a valid schedule is only found, when the cost function value reaches 0, the cost function value is only an indicator for the remaining violated constraints and an indicator for the remaining execution time. Only when bonus terms are involved which lower the cost function value, should, for example, the period of the tasks be very close to the optimum while simultaneously all constraints are satisfied, the cost function value be an accurate indicator for the solution quality.

This evaluation leads to the finding that the cost function value can be used as an estimation for the remaining scheduling time as well as an indicator for problem complexity. It follows a logarithmic decay as was expected. The Evaluation was undergone for synthetic task set and stopped, when a value

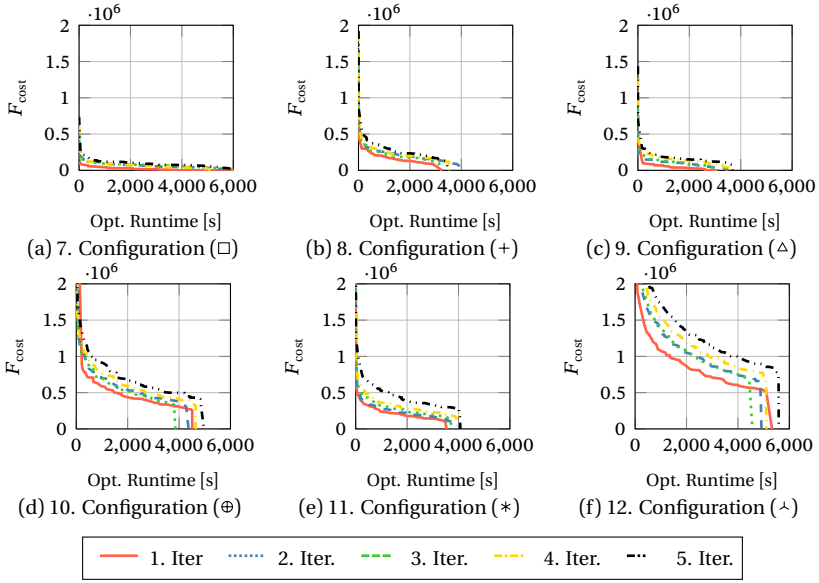


Figure 7.14: Cost function over iteration time for the second 6 configurations.

of 0 was reached. For the 6$^{\text{th}}$ configuration, the cost function dropped significantly after about 80 % of the optimization time. For 100 tasks, a higher cost function value was observed, and a longer optimization time was required to reach a valid schedule, nevertheless a logarithmic decay was observed.

> ***Finding 8: Cost Function Characteristics for Simulated Annealing-based Scheduling***
>
> *The cost function value can be used as an estimation for the remaining scheduling time as well as an indicator for problem complexity for cases where the cost function value is larger than 0. Should a valid schedule be found, and the cost function value is smaller than 0, the cost function value can be used as a direct indicator for the solution quality, since the algorithm guarantees, that a cost function value of 0 or below indicates a valid schedule.*

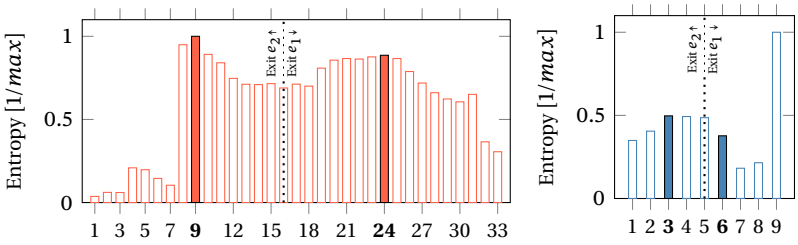
In all cases, the cost function must reach a value of 0 to indicate a valid schedule and the evaluation is always performed with this as the stopping criterion.

7.4 Accuracy, Equivalence and Placements of ANN Components

After the runtime estimation strategy from Section 7.1.1 and Section 7.1.2, the weights and biases of the exit layers needed to be trained and the structure and type of anomaly detection module needed to be determined and configured.

7.4.1 Placement Evaluation of Added Early Exits

To train and test the early exit layers, the positioning of the exit layers needed to be determined first. This was done by using the method presented in Section 5.2.1. Here, two exits were added to two of the networks. These networks were introduced and the respective runtime was measured in Section 7.1. One exit was added in the first half of the network and one exit was added in the second half of the network, which aimed to provide a variety of different exit points for the checkpoint scheduler.



(a) ResNeXt-101 [46] (b) GoogleNet [44]

Figure 7.15: Relative entropy between classes of the traced output layers. Exit layers are marked filled in the bar chart and the corresponding index is marked in bold.

For the first network, a ResNeXt-101 [46] network, the 9th and 18th residual blocks were selected to be the branching layers, which allowed for an early exit. This was done because these layers had the highest relative entropy between classes in their respective possible windows. These windows were determined by splitting the network into two segments, each representing half of the network. One exit layer was placed in each of these segments, which was done to ensure that the exit layers were not placed too close to each

other, which gave the checkpoint structure more slack to run in parallel and allowed for a larger variance in possible runtimes for both layers. The width of these possible placement windows was set to 10 layers for the ResNeXt-101 network.

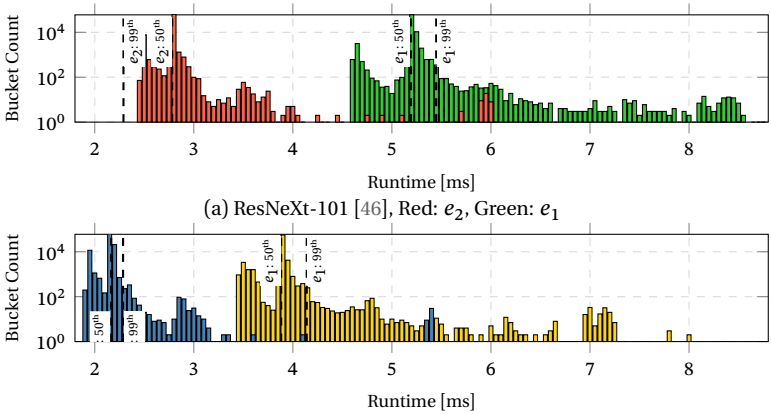
For the second network, a GoogleNet [44] network which had two added exit layers, the 3rd and 6th inception layers were selected to be the branching layer, which allowed for an early exit. These early exits should not be confused with the original early classification layers, which were used during GoogleNet training. These temporary exits were only present and discarded during inference. While the originally added early classification layers were also placed after the 3rd and 6th inception layers (as the ones used during training), the now added layers focused on inference on the already trained model.

Table 7.13: Factor representation of the percentiles of the runtime estimation.

Network	**50th**	**60th**	**70th**	**80th**	**90th**	**95th**	**99th**	**99.99th**	Avg.
R.N.[1], e_2	2.782	2.785	2.789	2.794	2.801	2.808	2.898	6.009	2.764
R.N.[1], e_1	5.193	5.200	5.208	5.218	5.236	5.266	5.447	8.694	5.182
G.N.[2], e_2	2.165	2.168	2.172	2.176	2.182	2.189	2.285	5.417	2.157
G.N.[2], e_1	5.713	5.720	5.729	5.743	5.783	5.930	7.390	11.463	5.699

[1]ResNeXt-101 [46], [2]GoogleNet [44]

The structure was comparable, because both types of exit layers oriented themselves on the original classification layers. This placement was done because these layers had the highest relative entropy between classes in their respective possible windows. The minimum distance between both exit layers was set two 2 layers, i.e. one layer had to lie in between, because this allowed the checkpoint structure to be executed in parallel. While the last inception layer in GoogleNet showed a significantly higher inter class entropy, it was not selected as an exit layer, because it was the last layer of the network and thus an added early exit would be redundant. The relative entropy of both networks is shown in Figure 7.15. These results were in line with the results presented in [Sta24b], where a similar method was used to determine the placement of three exits per network. The runtime until the respective network layers are done is measured for both networks and the results are shown in Figure 7.16 on a logarithmic scale. Additionally, detailed runtime percentile values are shown in Table 7.13.



(a) ResNeXt-101 [46], Red: e_2, Green: e_1

(b) GoogleNet [44], Blue: e_2, Yellow: e_1

Figure 7.16: Runtime until the respective network layers finished execution.

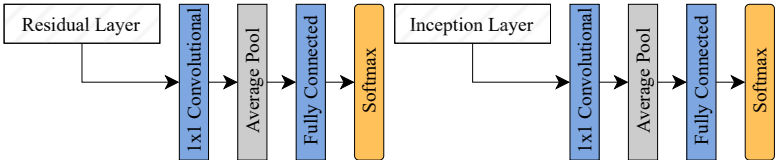
For exit e_2, i.e. the first exit layer, the 50$^{\text{th}}$ percentile of the runtime is at 2.782 ms for the ResNeXt-101 network and 2.165 ms for the GoogleNet network. For exit e_1, i.e. the second exit layer, the 50$^{\text{th}}$ percentile of the runtime is at 5.193 ms for the ResNeXt-101 network and 5.713 ms for the GoogleNet network. The 99$^{\text{th}}$ percentile of the runtime is at 2.898 ms for exit e_2 and 5.447 ms for exit e_1 in the ResNeXt-101 network, while it is at 2.285 ms for exit e_2 and 7.390 ms for exit e_1 in the GoogleNet network. The 99.99$^{\text{th}}$ percentile of the runtime is at 6.009 ms for exit e_2 and 8.694 ms for exit e_1 in the ResNeXt-101 network, while it is at 5.417 ms for exit e_2 and 11.463 ms for exit e_1 in the GoogleNet network.

For the ResNeXt-101 network, the average runtime of 2.764 ms for exit e_2 and 5.182 ms for exit e_1 lie at 31.2 % and 59.0 % of the runtime of the original network, respectively. This divides the original network in three segments of nearly the same execution time and provides a good variety in possible runtimes. For the GoogleNet network, the average runtime of 2.157 ms for exit e_2 and 5.699 ms for exit e_1 lie at 40.0 % and 60.0 % of the runtime of the original network, respectively. This divide is not as even as the divide for the ResNeXt-101 network, but still acceptable in providing a good variety in possible runtimes.

In both cases, this allowed for a variety in runtime with a runtime distribution, which lies between 29 % and 41 % of the total runtime of the unaltered network, allowing for a good variety in possible runtimes.

7.4.2 Accuracy and Equivalence of Added Early Exits

After the placement of the exit layers is determined, the exit layers need to be trained. For this, the training method described in Section 5.2.2 is used and the architecture from the original end layers are used. The exit layers are retrained, with the output dimension either kept at the original size or being reduced to the reduced dataset size, as described in Section 5.2.2. The structure and choice of the exit layers remain unchanged with the goal of keeping the runtime estimations of the original exits consistent with the runtime estimations of the exit layers. Figure 7.17 shows the structure of the added early exits. This also helps in deployment, when the exit is mapped onto a dedicated accelerator, because the accelerator can be optimized for the original exit layer structure and thus can be used for the exit layers without any changes.



(a) ResNeXt-101 (b) GoogleNet

Figure 7.17: Structure of the added early exits. Original layers are shown dashed in gray, exit layers are marked red and retrained layers are marked in blue.

Two metrics are tested. The first test evaluates the accuracy of the exit layers on the original dataset with the reduced number of classes. The second test evaluates the equivalence to the original exit on a dataset with reduced complexity, by merging class labels. For the test with the original model exit, the classification output is then considered correct, if the label of the predicted sample and the true label are both inside the abstracted class hypernym. In this test, ImageNet [134] is used as the original dataset and the classes are merged into more abstracted classes using the WordNet [133] hierarchy as described in Figure 5.6. For the original exit, the 1000 classes of ImageNet are used, for e_1, these classes are merged into 559 classes and are used to train the exits. For e_2, the classes are merged into 179 classes, which are used to train the exit. ImageNet-1k [134] was used for the training, test and evaluation datasets, with the test/training/evaluation split kept from the original dataset. The comparison of the exits is also done using a trained instance with the original dataset abstraction. This is done to evaluate the effect of the reduced complexity of the dataset on the accuracy of the exit.

Table 7.14: Accuracy and Equivalence of the added early exits.

Network	Exit	$\widehat{C^2}$		$\widehat{C^1}$		$\widehat{C^0}$
		Acc. [%]	Eq. [%]	Acc. [%]	Eq. [%]	Acc. [%]
R.N.[a]	e_2, e_1	65.2	74.1	69.0	75.2	/
	e_0	85.2	/	83.6	/	80.2
G.N.[b]	e_2, e_1	62.0	70.9	63.1	68.2	/
	e_0	76.1	/	74.1	/	69.8

[a]ResNeXt-101 [46], [b]GoogleNet [44]

Table 7.14 showed the results of the accuracy and equivalence tests. For the ResNeXt-101 network, the accuracy of exit e_2 was at 65.2 % and the equivalence to the original exit was at 74.1 %, while the accuracy of exit e_1 was at 69.0 % and the equivalence to the original exit was at 75.2 %. For the GoogleNet network, the accuracy of exit e_2 was at 63.1 % and the equivalence to the original exit was at 70.9 %, while the accuracy of exit e_1 was at 62.0 % and the equivalence to the original exit was at 68.2 %. The accuracy of the original exits for both networks was also shown in Table 7.14 for reference, which was at 85.2 % for ResNeXt-101 and 76.1 % for GoogleNet. The results both matched the classification results of the original publications of ResNeXt-101 [46] and GoogleNet [44]. Furthermore, the results were comparable with the already presented work in [Sta24b].

Finding 9: Added Early Exit Positioning and Accuracy

The positioning of the early exits is chosen to split the network into three segments of nearly the same execution time, which provides variety in possible runtimes and allows for the execution of the checkpoint structure in parallel. The accuracy of the exit layers is slightly lower than the accuracy of the original exit, but choosing an early exit allows for a reduction in runtime.

7.4.3 Accuracy and Runtime of Anomaly Detection

In addition to the accuracy of the exit layers, the accuracy of the anomaly detection modules was also evaluated. This was done by using the method described in Section 5.3. The combination between dimension reduction and corresponding anomaly detection modules was evaluated for runtime and accuracy. The runtime was measured for the complete anomaly detection pipeline, which included both dimension reduction and anomaly

detection module. The accuracy was evaluated for the same combination.

Table 7.15: ImageNet-based datasets for training anomaly detection components.

Dataset	Size	Class	Description
ImageNet-1k [134]	50000	`normal`	Original ImageNet dataset with 1000 classes [134]
ImageNet-O [135]	2000	`anom.`	Dataset of samples that are misclassified by a ResNet-50 [45] trained on ImageNet-1k [134] and determined as out of distribution
ImageNet-R [136]	30000	`anom.`	Dataset of samples that are renditions of the original ImageNet-1k [134] samples, e.g. paintings, cartoons, sculptures
ImageNet-C [159]	25000	`anom.`	Dataset of samples that are corrupted versions of the original ImageNet-1k [134] samples, e.g. with noise, blur, weather effects
White Noise	25000	`term.`	Dataset of samples that are generated by white noise sampling
ImageNet-1k with White Noise	25000	`term.`	Dataset of samples that are generated by superimposing white noise on ImageNet-1k samples

As the dataset, a selection of ImageNet versions is chosen to train the anomaly detection modules to discern between `normal` and `anomalous` samples. Additionally, a `termination` class is introduced, which signifies an input, which is not able to be processed by the network, e.g. due to a failure in the sensor or the data transmission. This follows the introduced definition of anomalies in Section 5.3. Table 7.15 shows the datasets used for training and evaluating the anomaly detection modules. The original ImageNet-1k [134] dataset is used as the `normal` class, while the ImageNet-O [135], ImageNet-R [136] and ImageNet-C [159] datasets are used as the `anomalous` class. The ImageNet-O dataset consists of samples that are misclassified by a ResNet-50 [45] trained on ImageNet-1k [134] and determined as out of distribution, while the ImageNet-R dataset consists of samples that are renditions of the original ImageNet-1k [134] samples, e.g. paintings, cartoons, sculptures. The ImageNet-C dataset consists of samples that are corrupted versions of the original ImageNet-1k [134] samples, e.g. with noise, blur, weather effects. For the `termination` class, two generated sets are used. The first consists of samples which were

Table 7.16: Classification accuracy for anomaly detection of ResNeXt-101.

Anomaly Detect.	Exit	PCA	Concat	Sum	Avg.	Max
ML-based	e_2	93.2 %	92.9 %	76.8 %	86.2 %	85.1 %
	e_1	93.1 %	90.2 %	91.0 %	90.0 %	91.0 %
SVM-based	e_2	82.3 %	86.9 %	87.2 %	87.8 %	85.9 %
	e_1	83.0 %	85.7 %	90.1 %	90.1 %	87.0 %
GNB-based	e_2	79.8 %	46.5 %	83.6 %	83.6 %	85.0 %
	e_1	85.0 %	51.0 %	84.4 %	84.4 %	81.0 %
Euclidean D.	e_2	81.9 %	82.7 %	82.1 %	81.5 %	76.8 %
	e_1	78.1 %	79.6 %	80.6 %	80.6 %	48.0 %
Mahalanobis D.	e_2	84.2 %	✗	86.1 %	84.8 %	86.1 %
	e_1	92.4 %	✗	92.6 %	91.2 %	89.0 %
Entropy-based	e_2	61.3 %	79.2 %	81.7 %	53.5 %	53.5 %
	e_1	57.1 %	53.5 %	53.5 %	53.5 %	53.5 %

generated by randomly generating white noise samples, while the second dataset consists of samples that are generated by superimposing white noise onto one half of each ImageNet-1k sample. The classes are balanced with 50000 samples for the `normal` class, 45200 samples for the `anomalous` class and 50000 samples for the `termination` class. For the training of the anomaly detection modules, the dataset is split into a training set with 80 % of the samples and a test set with 20 % of the samples.

The accuracy is evaluated on the test set, while the runtime is measured for the complete pipeline on the training set. The configuration of the anomaly detection modules is dependent on the type of the module. For the support vector machines, the support vectors are selected from the training datasets. The Euclidean distance and the Entropy-based methods each require an average vector for each class, which is calculated from the training dataset. The Mahalanobis distance-based method additionally requires the covariance matrix of the training dataset, which is calculated from the training dataset as well. Lastly, the mean and variance is required by the Gaussian Naive Bayes (GNB)-based method, which are also calculated from the training dataset. The anomaly detection modules are placed at the two layers before the early exits to facilitate the execution of the anomaly detection module in parallel to the original neural network layers and to make a result available, when the checkpoint is reached. This is done to provide a cross-layer input for the anomaly

Table 7.17: Classification accuracy for anomaly detection of GoogleNet.

Anomaly Detect.	Exit	PCA	Concat	Sum	Avg.	Max
ML-based	e_2	92.3 %	90.1 %	87.2 %	80.6 %	81.2 %
	e_1	89.1 %	92.5 %	89.6 %	82.1 %	83.2 %
SVM-based	e_2	79.6 %	81.5 %	84.8 %	85.5 %	87.4 %
	e_1	81.5 %	86.5 %	88.2 %	88.2 %	89.0 %
GNB-based	e_2	79.6 %	46.5 %	83.2 %	83.2 %	79.6 %
	e_1	86.1 %	49.3 %	85.5 %	85.5 %	85.0 %
Euclidean D.	e_2	75.2 %	81.0 %	77.7 %	79.0 %	75.0 %
	e_1	64.6 %	85.5 %	86.5 %	86.5 %	80.6 %
Mahalanobis D.	e_2	85.0 %	✗	93.0 %	93.0 %	94.0 %
	e_1	85.0 %	✗	93.0 %	92.0 %	92.0 %
Entropy-based	e_2	65.1 %	53.5 %	55.0 %	46.5 %	53.5 %
	e_1	53.5 %	46.5 %	46.5 %	46.5 %	53.5 %

detection module with the aim of improving the detection performance. This makes the use of dimensionality reduction necessary. For the dimension reduction, only the principal component analysis (PCA)-based method requires a configuration, which is done by calculating the principal components from the training dataset. Here, 256 principal components are selected. Dimension reduction method based on concatenation, summation, average and maximum of one sample require no configuration. The runtime of the anomaly detection components are measured for all combinations of dimension reduction and anomaly detection methods. The accuracy of the anomaly detection components are shown in Table 7.16 and Table 7.17. The measured runtimes are shown in Tables 7.18 and 7.19.

For the ResNeXt-101 network, the accuracy of the anomaly detection modules reaches an accuracy over 90 % for a subset of combination of dimension reduction and anomaly detection methods, with the ML-based and SVM-based methods showing the highest accuracy at around 99 % for exit e_2 and e_1 and the PCA-based dimension reduction method. The best performing dimension reduction method is the PCA-based method, which shows the highest accuracy for all anomaly detection methods. The worst performing dimension reduction method is the summation-based method, which shows the lowest accuracy for all anomaly detection methods. The best performing anomaly detection method is the ML-based method, which shows

Table 7.18: Max. runtime for anomaly detection modules of ResNeXt-101 in [ms].

Anomaly Detect.	Exit	PCA	Concat	Sum	Avg.	Max
ML-based	e_2	0.624	0.462	0.484	0.472	0.482
	e_1	0.624	0.462	0.484	0.472	0.424
SVM-based	e_2	3.991	101.533	0.717	0.712	0.787
	e_1	6.189	63.124	0.727	0.760	0.731
GNB-based	e_2	2.986	310.051	0.742	0.675	0.724
	e_1	3.567	101.278	0.688	0.656	0.695
Euclidean D.	e_2	0.639	15.774	0.466	0.586	0.540
	e_1	0.655	5.612	0.475	0.459	0.420
Mahalanobis D.	e_2	0.866	✗	1.802	1.836	1.890
	e_1	1.866	✗	1.852	1.870	1.894
Entropy-based	e_2	1.734	18.995	0.602	5.642	0.636
	e_1	7.837	7.906	0.586	0.577	0.578

the highest accuracy for all dimension reduction methods. The worst performing anomaly detection method is the Euclidean distance-based method and the Entropy-based method, which shows the lowest accuracy for all dimension reduction methods. While all methods show promising results, the PCA-based dimension reduction method consistently outperforms the others across different anomaly detection techniques and is chosen to be deployed.

The anomaly detection modules for the GoogleNet network show a similar trend as the anomaly detection modules for the ResNeXt-101 network, with the PCA-based dimension reduction method showing the highest accuracy for all anomaly detection methods and the Entropy-based method showing the lowest accuracy for all anomaly detection methods. The best performing anomaly detection method is again the SVM-based method, which shows the highest accuracy for all dimension reduction methods. The choice for the chosen anomaly detection methods is close enough to be determined by the runtime, which is measured for all combinations of dimension reduction and anomaly detection methods and shown in Table 7.18 and Table 7.19. The runtime of the shows a larger spread than the accuracy with very similar runtimes across the exits. The Mahalanobis distance-based method shows the highest runtime for all dimension reduction methods, while the Euclidean distance-based method shows the lowest runtime for all dimension reduction

Table 7.19: Max. runtime for anomaly detection modules of GoogleNet in [ms].

Anomaly Detect.	Exit	PCA	Concat	Sum	Avg.	Max
ML-based	e_2	0.630	0.471	0.489	0.482	0.488
	e_1	0.634	0.468	0.494	0.478	0.434
SVM-based	e_2	0.406	15.321	0.069	0.071	0.073
	e_1	0.467	5.543	0.069	0.061	0.076
GNB-based	e_2	0.205	22.919	0.067	0.066	0.066
	e_1	0.211	4.335	0.063	0.056	0.066
Euclidean D.	e_2	0.057	1.023	0.047	0.046	0.046
	e_1	0.057	0.293	0.041	0.045	0.045
Mahalanobis D.	e_2	1.072	✗	1.080	1.081	1.080
	e_1	1.087	✗	1.804	1.086	1.084
Entropy-based	e_2	0.125	1.191	0.059	0.059	0.058
	e_1	0.131	0.378	0.057	0.057	0.050

methods. All other methods show a similar and low runtime, which is close enough to be determined by the accuracy. Thus, the choice of the anomaly detection method is determined by the accuracy, which leads to the choice of the ML-based method for deployment.

For the GoogleNet network, the runtime of the anomaly detection modules shows a similar trend as the runtime of the anomaly detection modules for the ResNeXt-101 network, with the Mahalanobis distance-based method showing the highest runtime for all dimension reduction methods and the Euclidean distance-based method showing the lowest runtime for all dimension reduction methods. All other methods show a similar and low runtime, leading to the choice of the ML-based method for both exits.

Overall multiple trends can be observed. In general, each combination of dimension reduction and anomaly detection methods either provide an accuracy of over 75 % or fail to provide an accuracy of over 55 %. The PCA-based dimension reduction method splits the accuracy, with the sum, average and maximum pooling methods showing a similar and slightly higher accuracy overall, while the concatenation method shows a lower accuracy overall. The Mahalanobis distance-based method shows a high accuracy overall, as does the ML-based methods, for both the CNN-based and fully connected networks.

In regard to the runtime similar trends are observed. The runtime of the concatenation method is significantly higher than the runtime of the other dimension reduction methods, while the sum, average and maximum pooling methods show a similar and low runtime. The PCA-based method shows a higher runtime than the sum, average and maximum pooling methods, but is still lower than the concatenation method. For the ML-based methods, the runtime is less dependent on the dimension reduction method, while the SVM-based methods and GNB-based methods are more dependent on the dimension reduction method, with the concatenation method showing a significantly higher runtime than the other dimension reduction methods. The Euclidean distance-based methods and the Entropy-based methods show a similar and low runtime for all dimension reduction methods, while the Mahalanobis distance-based methods show a higher runtime for all dimension reduction methods.

> ***Finding 10: Anomaly Detection Accuracy and Runtime***
>
> *A very high accuracy can be achieved for the anomaly detection methods. If these anomaly detection methods are based on the intermediate results from the two residual layers or inception layers before the early exits, the runtime of the anomaly detection modules can be kept low enough to be executed in parallel to the original network layers and to provide a result when the checkpoint is reached.*

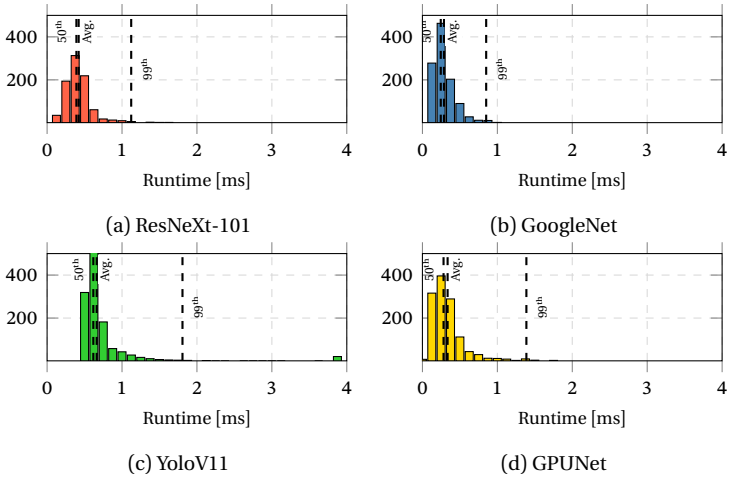
These accuracy and runtime measurements are comparable to related works in [Far24, Abd23], where a subset of these methods were evaluated for anomaly detection for different networks.

7.5 Runtime Overhead of ANN Checkpoints

Before the system can be deployed and evaluated, the runtime overhead of the checkpoint structure needed to be measured. This was done by measuring the runtime of the checkpoint structure, which determined the use of the early exit during inference for both the ResNeXt-101 network and the GoogleNet network as well as the execution of the scheduler choosing the next incoming task. The scheduler choosing the next incoming task was measured in all cases, independent of the chosen neural network structure.

Figure 7.18 showed the incurred overhead of the checkpoint structure for both the ResNeXt-101 network and the GoogleNet network. The total overhead was measured, which included the checkpoint structure as well as the scheduler for the next incoming task and all tasks, which were associated with this. This

included the sorting of the task queue, the estimation of the runtime for the incoming task and the decision of the early exit choice for the incoming task. The overhead was measured for a uniform release of tasks for all neural network models. The release period was equal to the average execution time of the corresponding network. The deadlines of the incoming tasks were randomly set to be between 1.5 and 10 times the average execution time of the network, forcing the scheduler to reorder the task queue.



(a) ResNeXt-101 (b) GoogleNet

(c) YoloV11 (d) GPUNet

Figure 7.18: Runtime overhead of checkpoint scheduler.

Table 7.20: Runtime overhead for the checkpoint scheduler for the four selected networks.

Network	**50th**	**60th**	**70th**	**80th**	**90th**	**95th**	**99th**	**99.99th**	**Avg.**
R.N.[1]	0.391	0.429	0.471	0.514	0.606	0.722	1.223	1.591	0.421
G.N.[2]	0.245	0.281	0.317	0.380	0.477	0.557	0.851	1.950	0.228
YV11.[3]	0.617	0.626	0.636	0.650	0.674	0.701	1.807	3.477	0.622
GPU.[4]	0.282	0.323	0.365	0.433	0.577	0.779	1.388	2.230	0.336

[1]ResNeXt-101 [46], [2]GoogleNet [44], [3]YoloV11 [48], [4]GPUNet [47]

As described in Table 7.20, the average overhead of the checkpoint structure was between 0.228 ms for the GoogleNet network and 0.622 ms for the YoloV11 network. The 99th percentile of the overhead was between 0.851 ms for the

GoogleNet network and 1.807 ms for the YoloV11 network. The added overhead of the checkpoint structure thus added a percentage of 5 % for ResNeXt-101, 4 % for GoogleNet, 6 % for YoloV11 and 5 % for GPUNet to the average execution time of the corresponding network. Additionally, the spread of the measurements was large for all networks, with the 99.99^{th} percentile being between 1.591 ms for ResNeXt-101 and 3.477 ms for YoloV11. The large spread of the measurements was caused by the fact that the scheduler needed to reorder the task queue, should tasks be released with a smaller deadline. Networks like YoloV11 with larger execution times were more likely to have a larger average overhead, as more tasks were released within the execution of the network.

Finding 11: Checkpoint Scheduler Overhead

The overhead of the checkpoint scheduler lies between 0.228 ms for the GoogleNet network and 2.318 ms for the YoloV11 network, which adds a percentage of 4 % to 6 % to the average execution time of the corresponding network. This value is small enough to be executed in parallel to the original network layers and the 99^{th} percentile of the overhead is used for the deployment in the mode-change time-triggered scheduler.

7.6 Anytime Neural Network Behavior with Utilization Increase

With the evaluation of the runtime estimation of the networks, the early exits and the scheduler done and the configuration of the runtime estimator component determined through the ablation study, the next step was to evaluate the deadline success rate of the system for different utilization levels as well as the accuracy drop, which was incurred with a rise in utilization. This dynamic scheduling approach was introduced in Section 5.6 and resembled an additional scheduling approach in contrast to the time-triggered scheduler. The mode-change was implemented in this scheduler via a different tasks release characteristic and a change in utilization. The following chapters evaluated the effect of the change in utilization on the deadline success rate and the accuracy of the given neural network models. The evaluation was based on the works from [Sta25b]. In Section 7.6.1 the chosen exit for the increase in utilization was shown. How this effected the accuracy was presented in Section 7.6.2 and the corresponding deadline success rate was shown in Section 7.6.3.

The experiment is conducted on an NVIDIA Jetson AGX Orin with 64 GB of RAM to simulate embedded HESoCs. Four networks are evaluated. The first is the ResNeXt-101 [46] network with two added early exits after the 9$^{\text{th}}$ and 18$^{\text{th}}$ Residual blocks. The second network is a GoogleNet [44] network with two added early exits after the 3$^{\text{rd}}$ and 6$^{\text{th}}$ Inception layers. The third one is a YoloV11 [48] network without any additional early exits and the last one is a GPUNet [47] with no additional early exits.

The evaluation varies the release time of the tasks to increase the utilization and the release time at 100 % utilization is set to the average runtime of the corresponding network. The release time is varied with a step size of 10 % in the range of 50 % until 250 % of the average runtime. The deadline is set to 1.5 times the average runtime of the corresponding network and the runtime estimation for the task τ_i executing the network is set to

$$\sum_{l \in \alpha_i} \hat{t}_c(l,u) = c \cdot \text{avg}(\sum_{l \in \alpha_i} \widehat{T}(l,u)), \tag{7.1}$$

with the safety factor c being set to $c = 1.2$, as determined in the ablation study. The weight distribution for the runtime estimation was set to the exponential decay in the ablation study.

Three types of experiments are conducted. The first one releases the tasks at a uniform period, the second one releases the tasks with a random triangular distribution between 0.5 and 1.5 times $\hat{t}_c$ and the last one releases the tasks with a batched release characteristic, where 100 tasks are released at the same time and the deadline is set to

$$t_d(\tau_i) = \sum_{\tau_j, j<i} \sum_{l_n \in \alpha_j} \hat{t}_c(l_n,u) + 1.2 \cdot \sum_{l_n \in \alpha_i} \hat{t}_c(l_n,u), \tag{7.2}$$

i.e. the sum of the expected execution time of all previously released tasks plus the expected execution time of the current task multiplied by the safety factor.

7.6.1 Chosen Exit for Utilization Increase

With a rise in utilization, the scheduler chose an earlier exit for networks with early exits. The chosen exit for the ResNeXt-101 and GoogleNet networks was shown in Figure 7.19. Two different instances to choose an early exit were possible. In the first instance, the scheduler chose an early exit before the inference was started by setting the available time budget for the task, which

resulted in the checkpoint scheduler choosing an early exit. This planned early exit was shown in Figure 7.19 in solid colors. The second instance was when the scheduler chose an early exit during the inference, because of a delay in runtime, which in turn led to an expected runtime exceeding the time budget. This exit chosen during runtime was shown in Figure 7.19 with a dashed bar.

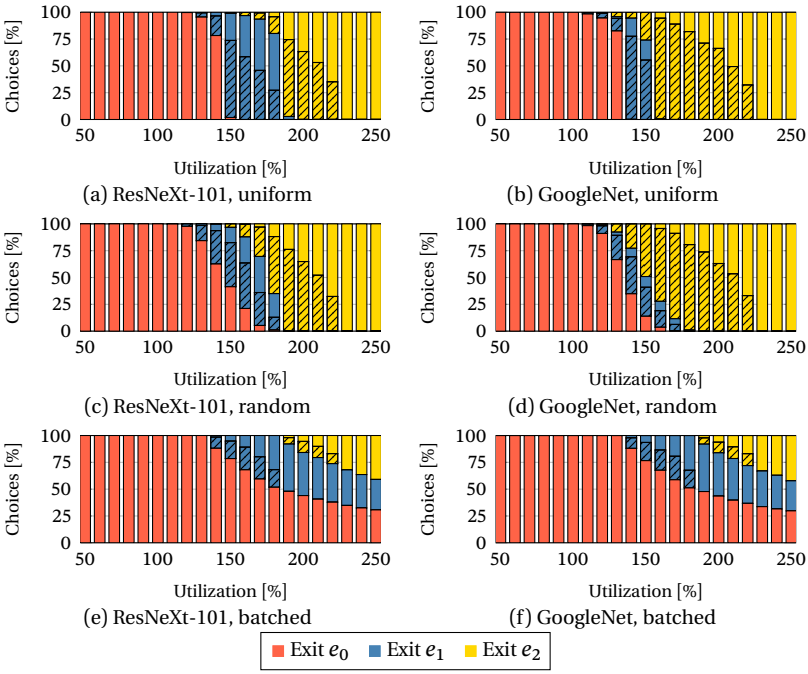


Figure 7.19: Chosen exit for increase in utilization for uniform, random and batched release characteristics for the ResNeXt-101 and GoogleNet networks. Should the planned exit be selected, the exit is shown in solid, should an earlier exit be chosen by the scheduler during runtime, the exit is shown with a pattern. The color gives the executed exit.

As expected, an increase in utilization led to a rise in earlier exits chosen for both networks. This held true for both the planned exit and the exit chosen during runtime. Since both networks were split into approximately three parts, a runtime decrease to around 59 % was expected when exit e_1 was chosen instead of the original exit e_0 and a decrease to around 32 % was expected when exit e_2 was chosen instead of the original exit e_0. If only the ResNeXt-

101 network were present, then the scheduler would choose the exit e_1 for a utilization of around 150 % and the exit e_2 for 300 % utilization. Because the runtime itself was not deterministic, and the scheduler itself added an overhead, the scheduler chose an earlier exit during runtime with a rising percentage as utilization increased. The same held true for the GoogleNet network, where the exit e_1 was expected to be chosen for a utilization of around 150 % and the exit e_2 for 300 % utilization.

Because the evaluation combined the four networks introduced above into one deployment system, a slightly higher percentage of earlier exits was chosen for the ResNeXt-101 and GoogleNet networks, since the scheduler had the option to choose an earlier exit for these networks, while it had to choose the original exit for the YoloV11 and GPUNet networks. This led to a higher percentage of earlier exits for the ResNeXt-101 and GoogleNet networks.

For ResNeXt-101, the percentage of planned early exits, which were also executed, was highest for the batched release characteristic. For the random release characteristic and the uniform release characteristic, the percentage of planned early exits, which were also executed, was lower, since the absolute deadlines gave less time for a predictable execution. The same held true for the GoogleNet network, where the percentage of planned early exits, which were also executed, was highest for the batched release characteristic and lower for the random and uniform release characteristics. In both cases, the checkpoint scheduler executed an earlier exit than planned more often for the random and uniform release characteristics. Since the YoloV11 and GPUNet networks did not have any early exits, the scheduler had to choose the original exit for all utilization levels.

Both experiments followed the same structure as presented in [Sta25b], but the results differed slightly, since the early exits were placed at different locations and the accuracy of the different exits differed. Additionally, the overhead was reduced, and the runtime estimation was improved in this work, which led to a later drop in accuracy than in [Sta25b].

7.6.2 Accuracy Drop for Utilization Increase

With the exits chosen in Section 7.6.1, a drop in accuracy is expected with a rise in utilization. Given the measured accuracy of the different exits from the evaluation presented in Section 7.4.2, a reference value can be defined for every early exit. In Figure 7.20, there reference values are marked with a

dashed, dotted or dash-dotted line respectively. For ResNeXt-101, the reference accuracy lies at 65.2 % for exit e_2, 69.0 % for exit e_1 and 80.2 % for the original exit e_0. For GoogleNet, the reference accuracy lies at 62.0 % for exit e_2, 63.1 % for exit e_1 and 69.8 % for the original exit e_0. The measured accuracy for the experiments is given as the percentage of correct classifications for the tasks released at the corresponding utilization level. It is expected to drop with a rise in utilization, because the scheduler chooses earlier exits with a lower accuracy. The measured accuracy for the different release characteristics and the different networks is shown in Figure 7.20 for the different release characteristics and the different networks.

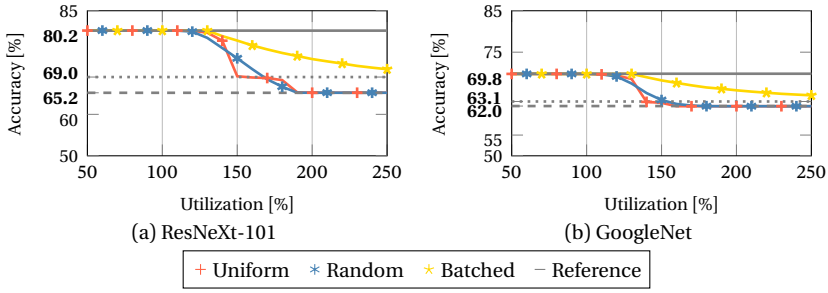


Figure 7.20: Accuracy evaluation for ResNeXt-101 and GoogleNet for an increase in utilization for uniform, random and batched release characteristics.

As expected, the accuracy drops with a rise in utilization for both networks and all release characteristics. The drop in accuracy follows the expected drop, which are given by the reference accuracy of the chosen exits. This characteristic for one model is expected to follow the function of

$$\mathtt{acc}(u_{\%}) = \sum_{e_x} \frac{N(e_x, u_{\%}) \cdot \mathtt{acc}(e_x)}{N_{\text{total}}}, \tag{7.3}$$

where $\mathtt{acc}(u_{\%})$ is the accuracy at utilization level $u_{\%}$, $N(e_x, u_{\%})$ is the number of tasks exiting at exit e_x at utilization level $u_{\%}$, and N_{total} is the total number of inferences executed with the given neural network model. The actual measured accuracy is close to the expected accuracy for both networks. Because the scheduler chooses an earlier exit during runtime more often for the random and uniform release characteristics, the accuracy drop starts earlier for these release characteristics than for the batched release characteristic. The minimum accuracy was observed for both, the ResNeXt-101 and GoogleNet networks at the highest utilization of 250 %, where the scheduler chooses the earliest exit

for most of the tasks. The highest accuracy was achieved the longest for the batched release characteristic, where the scheduler chooses the original exit for most of the tasks, until the utilization of 120 % is reached, where the time between the actual runtime and the estimated runtime exceeds the given buffer of the average runtime and the runtime at the 99^{th} percentile. After this point, the accuracy drops proportionally for all release characteristics. The YoloV11 and GPUNet networks do not have any early exits, so the accuracy does not drop with a rise in utilization for these networks at the cost of potentially not finishing the execution before the deadline.

As above, the same structure as presented in [Sta25b] is followed, but with slightly different results, since the early exits are placed at different locations and the accuracy of the added exits differ. Additionally, the overhead is reduced, and the runtime estimation is improved in this work, which leads to a later drop in accuracy than in [Sta25b].

7.6.3 Deadline Success Rate for Utilization Increase

Lastly, the deadline success rate for the three different networks needs to be evaluated for the different release characteristics and the utilization levels. The deadline success rate is given by the number of tasks, which finish execution before their deadline divided by the total number of tasks released at the corresponding utilization level and is presented for the different release characteristics and the different networks in Figure 7.21. As described above, the deadline is set to 1.5 times the average runtime of the corresponding network. The overhead of the scheduler was not included in setting the deadline to calculate the deadline success rate in comparison to the measured runtime of the original networks.

As expected, the deadline success rate drops later than the average accuracy with a rise in utilization, since the scheduler chooses earlier exits with a lower accuracy, with the aim of keeping the deadline success rate high. The deadline success rate is furthermore highest for the batched release characteristic, since the prediction of the scheduler holds true during the execution for most of the tasks, while allowing the planning of future exits with more available information. It is lower for the random release characteristic, because a varying amount of task can be released during the execution of a task, which leads can lead to both, a reassessment of the chosen exit via the checkpoint structure and a lower deadline success rate. The uniform release characteristic allows for all tasks to be scheduled with a predictable characteristic, leading to a positive effect on the deadline success rate, but also a higher overhead, because the

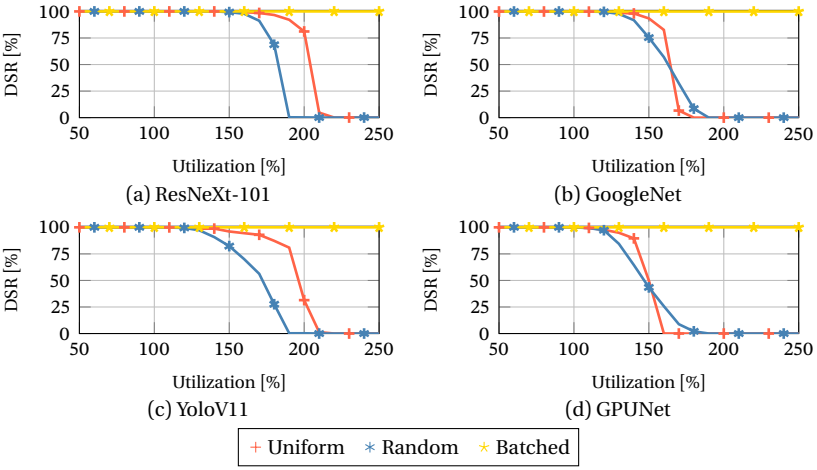


Figure 7.21: Deadline success rate for the ResNeXt-101, GoogleNet, YoloV11 and GPUNet networks for an increase in utilization for uniform, random and batched release characteristics.

scheduler can only schedule the execution of one task at a time. Additionally, it can not predict the next task release time and therefore, future scheduled tasks are likely to incur the overhead introduced by the necessity wait until new tasks are released, leading to a higher overhead and potentially a lower deadline success rate, especially at higher utilization levels. These results follow the same characteristic as presented in [Sta25b], but with a later drop in the deadline success rate, since the overhead is reduced and the runtime estimation is improved in this work.

Finding 12: Early Exit Choice for Utilization Increase

With a rise in utilization, the scheduler chooses an earlier exit for both the ResNeXt-101 and GoogleNet networks. The percentage of tasks, for which an exit is planned, but the checkpoint structure chooses an earlier exit rises with the utilization level. An earlier exit is chosen with the lowest percentage for the batched release characteristic. The expected drop in accuracy is followed by an increase in utilization leads to a drop in accuracy for both networks with early exits. The deadline success rate starts to drop at a higher utilization level, because the early exits are able to provide the necessary reduction in execution time.

Summary

This evaluation showed that the introduced methods were able to generalize for a variety of abstract task sets and utilization characteristics. Both the independent and the combined schedulers were able to find valid schedules and execute dynamic mode changes for a variety of task sets and constraints. Additionally, the checkpoint-based scheduler for early exits was able to handle utilization levels of up to 250 % for the combination of four different neural network models in parallel with a batched release characteristic. For a randomized or uniform release, a utilization of 180 % was achievable without significant effects on the accuracy. The given methods were thus able to handle the challenges of scheduling in dynamic environments and could be used in real world applications of varying characteristics.

This page intentionally left blank

Chapter 8

Case Studies

In addition to the abstract evaluation of Chapter 7, two real world case studies are presented. Section 8.1 presents a condensed overview of the case study, which was used throughout this work to introduce the scheduling and neural network model as well as the schedulers. This case study is based on a flexible and cycle-accurate accelerator, called the FleXNNgine [28], and is motivated by a real world application and uses the motivation from the federal funded project CeCaS [2]. A slice of the case study builds on work presented in [Sta25a].

Additionally, a second case study, presented in Section 8.2, is based on the Roboracer platform, which is a one tenth scale autonomous driving vehicle used in racing competitions [160]. In the context of the European HAL4SDV [1] project, this case study presents a vertical slice of the deployment workflow from task modeling to platform deployment. It is based on the deployment presented in [Sta26] and builds on a collaboration between project partners.

8.1 Deployment of ML-based Tasks on FleXNNgine-based HESoC

This chapter gives a condensed overview of the examples used throughout this thesis and the main feature of the case study is the support for a heterogeneous system-on-chip (HESoC), which includes the FleXNNgine [28] as the main accelerator for neural network inference. Additionally, an accelerator for encrypting data using the DRACO stream cipher [23], developed in the UNCOVER project [Sta24c], is integrated into the system.

8.1.1 Motivation of the FleXNNgine-featuring Case Study

The case study is motivated by a real world adjacent application, which models a machine learning (ML)-based decision system in an automotive vehicle and is designed to showcase a decision pipeline, which includes two neural network models. Based on these two classification results, an automated decision-making task for windshield wipers and headlights is performed. Together with an encryption task and using incoming data from an external network, the result is periodically displayed with a display task. This complete task set was introduced in Example 1.

The fundamental motivation for this case study lies in the trend of compute power centralization [3] and the resulting need for a centralization of the software architecture. This centralization requires the integration and scheduling of the corresponding tasks. The case study is designed to be a condensed version of the real world application and aims to show the deployment capabilities of the proposed scheduler to schedule systems, which include ML-based tasks onto a HESoC, which includes accelerators for encryption [23] and neural network inference [28].

8.1.2 Used Execution Platform and ML-based Task Set

The deployment platform for this case study is an AMD UltraScale+ MPSoC on a Xilinx ZCU102 development board, which features a quad-core ARM Cortex-A53 and a dual-core ARM Cortex-R5 [24]. The programmable logic can be configured to include the FleXNNgine accelerator [28] as well as the DRACO accelerator [23] for encryption. The FleXNNgine is a flexible systolic array, which can be configured into a cycle accurate accelerator for neural network inference [28]. The DRACO accelerator is an accelerator for the stream cipher of the same name and can be used for encrypt streaming data, which is sent over the networking interface of the platform [23]. Because these accelerators are implemented in the programmable logic, a deployment is possible, but not necessary. Should the corresponding tasks be scheduled on a CPU core, the accelerators do not need to be included in the programmable logic and the corresponding resources could be used for other accelerators. In this work, both accelerators and the four A53 cores are available and modeled as execution units and the main memory, a local scratchpad interface as well as the communication interface are modeled as shared resources. A detailed description of the platform was introduced in Example 3.

The task set of this case study includes two ML-based classification tasks, which can be executed on both, the CPU or the FleXNNgine [28], which is placed in the FPGA. These ML-based tasks are used for weather classification and traffic sign classification and are created via the retraining of a GoogleNet [44] and a ResNeXt-101 [46] model, respectively. Here, the weather classification is executed on the retrained GoogleNet model, and two early exits are added after the 3rd and 6th inception layers. For a dataset, the WeatherNet dataset [161] is used. The traffic sign classification is executed on the retrained ResNeXt-101 model, with early exits added after the 9th and 24th residual layers. Here the German Traffic Sign Recognition Benchmark [162] is used as the dataset.

In addition to the ML-based tasks, an encryption task is included, which can be executed on the DRACO accelerator [23] or on the CPU. Lastly, tasks for receiving and sending data, reading data from an external camera and a task for displaying the results are included in the task set and modeled in the presented task model. Additionally, tasks for moving data between the scratchpad memory and the main memory are included. All tasks are presented in Example 1, and a detailed description of the precedence constraints is given in Example 2.

8.1.3 Modeling and Runtime Estimation of Case Study Tasks

These tasks are turned into a model instance of the proposed task model. For this, a runtime estimation as well as the modeling of the execution characteristics of the ML-based tasks is required. The training of the early exits and the resulting accuracy-latency trade-off is described in Section 5.2.2 and the translation of the execution characteristics of the neural networks into a model instance is given in Example 22.

The end-to-end runtime estimation of the ML-based tasks is based on the in Section 5.4 described methods and is evaluated in Section 7.1.1. For the execution on the FleXNNgine, the runtime is estimated based on the cycle-accurate simulation of the accelerator. For the DRACO accelerator, the runtime is estimated based on the number of cycles required for the encryption and the operating frequency of the accelerator. The execution on the CPU is estimated based on measurements of the execution time of the tasks on the corresponding cores. Other task runtimes are estimated conservatively. Model values, including runtime estimations and precedence constraints were presented in Example 5 to Example 7.

Four execution modes are modeled for the task set. The first execution mode misses no tasks and schedules all tasks. The second execution models a turned off display and removes the period specification of the display task, leading into a schedule without jobs from the display task. The third execution mode models a turned off traffic light classification and removes the corresponding task from the list of scheduled tasks. The fourth execution mode models an unavailable network connection and removes the precedence constraints from the receive and send tasks, leading into a schedule without jobs from these tasks. This description is given in Example 11.

Should a bandwidth budget of less than 1 be required, a bandwidth regulator [84] is able to be deployed inside the FPGA to integrate into the scratchpad and on-chip memory interfaces.

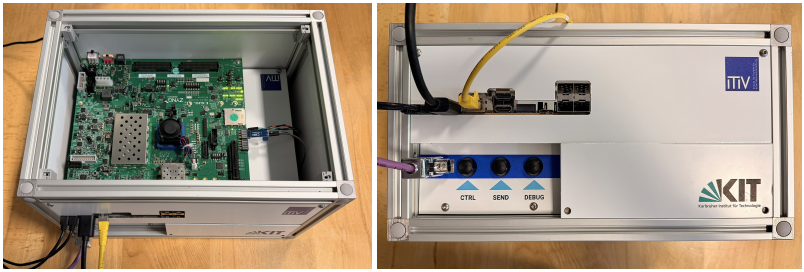
8.1.4 Multi-Mode Scheduling of the Case Study Task Set

This model instance is then turned into a schedule via the Simulated Annealing-based scheduler, described in Contribution 6. An excerpt of the unchanged schedule values is given in Example 8. Using this unchanged schedule, further execution modes are derived from the baseline schedule using the method described in Contribution 8, resulting in the schedule graph which is described in Example 12. For the ML-based tasks, the FleXNNgine is chosen for each job and the schedule is flexible for the execution of these tasks. Because the only communication with the remaining system is via the transfer tasks between main memory and scratchpad memory, only the transfer tasks have a fixed job start time. The weather classification and the traffic sign classification are then executed using the scheduler presented in Contribution 18, which aims to provide a result until a given deadline and supports the use of the early exits, should it be estimated that the deadline will be missed.

8.1.5 Demonstrator Deployment of the Case Study Task Set

All tasks are available as C++ code. This includes the control code for the communication with the FleXNNgine and the DRACO accelerator. Contribution 21 provides a variety of methods for the deployment of the schedule. Since the schedule is generated for the ZCU102 platform, the deployment uses the embedded time-triggered RTOS to schedule jobs on the corresponding execution units. This is done via an automated script, building the sources together with

the RTOS executables. The synchronization and the avoidance of collisions on the shared resources is ensured via the phased execution model approach and the scheduling algorithm. This results in a deployed system, which executes on the ZCU102 platform.



(a) Top View (b) Front View

Figure 8.1: Top view and front view of the used platform of the Case Study, which is based on the Xilinx ZCU102 development board, housed in an ITIV demonstrator box.

Figure 8.1 shows the used ZCU102 demonstrator platform. The configuration and the deployment of the code is done via a JTAG interface (left USB cable) and the data transfer to the platform is done via a dedicated networking interface (right USB cable). The platform is powered via a dedicated power supply (black back cable). For debug purposes, a display port interface is used (right black cable). Communication with other devices is either done via an Ethernet interface (yellow cable) or via a CAN interface (violet cable). Subfigure (a) shows a top view of the platform, the front view with the connected cable as well as control buttons is shown in subfigure (b). The deployed platform executes the scheduled task set and is able to display the output to the display. Should a switch to a different execution mode be triggered, the switches off the corresponding tasks and image transfer is stopped.

8.2 RoboRacer Task Set as a Collaboration in HAL4SDV

The second case study builds on a use case from UNIMORE, which is based on a task set to drive a vehicle from the RoboRacer platform. The RoboRacer platform is a one tenth scale autonomous driving vehicle used in academic

racing competitions [160]. With its vertical slice from task modeling, schedule creation, platform deployment and evaluation, this case study for the RoboRacer competition provides a real-world adjacent use case for the presented scheduling and deployment workflow. The case study is based on a collaboration between project partners in the context of the European HAL4SDV [1] project and was presented in isolation in [Sta26].

8.2.1 Context of the Case Study in the HAL4SDV Project

The case study presented in this section is based on a collaboration between project partners in the context of the European HAL4SDV [1] project. The acronym HAL4SDV stands for Hardware Abstraction Layer 4 Software Defined Vehicles and the project focuses on the development of a service-ariented architecture (SoA) for the safety relevant and non-safety relevant software of software defined vehicles (SDVs). The project aims to provide the following objectives of over-the-air (OTA) updates; hardware abstraction through virtualization, memory management and standardized application programming interface (API); hardware support for heterogeneous components; automated integration of in-vehicle applications; the support of safety and security features; and lastly engineering support software [1]. In this context, this case study and the use of the RoboRacer [160] platform provides a vertical slice approach through the goals of hardware abstraction, hardware support for heterogeneous components and automated integration of in-vehicle applications. This aim of providing a reality-adjacent use case for the presented scheduling and deployment workflow gives the opportunity to evaluate the workflow and the scheduler and deployment approach for a realistic use case with existing hardware and software components. The aim of hardware abstraction is achieved by taking the autonomous driving task set and generating schedules and artifacts for two different execution platforms.

8.2.2 RoboRacer Tasks Description and Execution Platform

The task model instance for this case study models a task stack for the autonomous driving of the RoboRacer vehicle [163] and consists of multiple multi-version tasks. The same task set was used in the collaboration in the context of the HAL4SDV project [Sta26].

The execution platform of the case study is based on an AMD KR260 development board [164], which is a HESoC with a fixed hardware PS part and a

FPGA. The PS consists of a quad-core ARM Cortex-A53 CPU and the FPGA is based on an AMD Xilinx Artix-7. The CPUs are mapped to the set of execution units $\mathcal{U}_{CPU} = \{u_{KR,1}, u_{KR,2}, u_{KR,3}, u_{KR,4}\}$ and the can employ a dedicated accelerator, which is modeled as u_{ml}. Originally, an NVIDIA Jetson Nano development board [165] is used for many RoboRacer platforms [163, 166]. This leads to the modeling of the quad-core ARM Cortex-A57 processor as $\mathcal{U}_{CPU,JS} = \{u_{JS,1}, u_{JS,2}, u_{JS,3}, u_{JS,4}\}$. The GPU of the Jetson Nano development board is modeled as $u_{JS,ml}$. The aim of this case study is to generate schedules for both execution platforms to showcase the hardware abstraction capabilities. The execution time values for the tasks are based on measurements on the respective platforms.

Two main inputs to the navigation are implemented. The first navigation is based on an Light Detection and Ranging (LiDAR) point cloud and the second navigation is based on odometry data, which is sourced from the motor controller. These are model as tasks τ_1 and τ_2. Both navigation inputs are fed to the particle filter localization. The odometry driver produces data at a rate of 40 Hz and the LiDAR driver produces data at a rate of 50 Hz, giving the odometry data a period and with that an implicit deadline of $D(\tau_1) = 25\,\text{ms}$. Since the output of data from the LiDAR driver results in the triggering of the further stages, an incomming precedence constraint is modeled with these further tasks and no deadline is assigned. The odometry driver τ_1 is modeled without precedence constraints and produces data at a constant rate. This data is consumed by the next steps in the pipeline periodically and not event triggered. Since both tasks read the data from external sources, a short execution time is necessary, which is modeled as 4 ms. Both tasks are valid for execution on the CPUs of both platforms, i.e. $\mathcal{U}_{CPU,JS}$ and $\mathcal{U}_{CPU}$, of the platform.

The next task in the execution pipeline is the particle filter localization, which localizes the vehicle on the track and is modeled as τ_3. It is implemented as an iterative algorithm, which accumulates the odometry data and performs a localization correction loop with every new LiDAR point cloud, i.e. it is called each time a new LiDAR input is available, which results in the aformentioned precedence constraint. The execution time of the particle filter localization is measured to be 24 ms on the KR260 platform. The localization loop can be described in five steps. First, the particle set is sampled with a low variance resampling algorithm from the previous loop's pose estimate. Secondly, the particle set is propagated forwards using the available odometry data. Third, for each particle in the set a virtual point cloud is generated by marching the rays from the cars position. Fourth, each particle in the set is compared with the point cloud received from the LiDAR sensor. Fifth and last, a weighted

average of the particle set in each dimension is performed to evaluate the pose estimate. Multiple different versions of this particle filter are available for one or two threads and with different particle set sizes. Nevertheless, each version has the assigned precedence constraints for the LiDAR driver. These versions are valid for execution on the CPUs of the KR260 platform, i.e. $\mathcal{U}_{CPU}$. Additionally, a second version of the particle filter localization is available, which is hardware-accelerated for the step of comparing the virtual point cloud with the point cloud received from the LiDAR sensor. This version is valid for execution on the accelerator u_{ml} of the KR260 platform and has an execution time which is expected to be around one half of the software version. The output of the particle filter localization is then fed into the obstacle detection and tracking task, which is modeled as τ_4. This task uses the point cloud data and with that the position estimate to identify potential opponents and obstacles. The detection loop is triggered by the localization data and thus requires a precedence constraint for the particle filter localization. The execution time of the obstacle detection and tracking is measured to be 13 ms on the KR260 platform. The tasks can be described in five steps. First, the point cloud is transformed from the local frame to the map frame by coordinate transformation using the current position. Secondly, the point cloud is segmented using an adaptive breakpoint detector algorithm, which is a single pass segmenter that returns a set of cluster composed by points. Third, each cluster is aligned to the map and classified using a fixed set of rules as part of the map or as an obstacle. Fourth, each cluster is associated into a set of tracked clusters by similarity and by feeding the center of mass of the cluster into a kalman filter. Fifth and last, the opponents are output as tracked clusters.

Lastly, two different planners for the further driving are implemented. The first planner is a field-based local planner, which is modeled as τ_5. It is tasked with the generation of a locally-optimal overtaking path. It works by modeling each opponent as a repulsive field and compares this to a predefined globally-optimal racing line. This task has a precedence constraint for the obstacle detection. The second planner is a pure pursuit control, which is modeled as τ_6. It is a simple geometric control algorithm that tracks a reference trajectory in time and follows the racing line. Localization data and the reference trajectory of the field planner are used. It has precedence constraints for the field-based local planner and the particle filter localization. It is the second of two possible trajectory planners and is thus given a deadline in the corresponding execution mode, equal to 25 ms.

8.2.3 Deployment onto the Execution Platform

The single version tasks are turned into the model instance presented in Table 8.1 and the multi-version tasks are presented in Table 8.2. The execution times are based on measurements on the respective platforms. The task set is then used to generate schedules for both execution platforms, which are then deployed onto the KR260 platform. For the Jetson platform the same values hold true within a negligible deviation.

Table 8.1: Execution time, units and precedence constraints for single-version tasks.

Task	C_i^v	U_i^v	P_i^v
τ_1, v=1	4 ms	$\mathcal{U}_{CPU}$	$\varnothing$
τ_2, v=1	4 ms	$\mathcal{U}_{CPU}$	$\varnothing$
τ_4, v=1	13 ms	$\mathcal{U}_{CPU}$	$P_4^1(\tau_3) = \{-(C_3^{\lambda_3(n)} + 10, C_3^{\lambda_3(n)} + 5)\}$
τ_5, v=1	24 ms	$\mathcal{U}_{CPU}$	$P_5^1(\tau_4) = \{-(23, 18)\}$
τ_6, v=1	2 ms	$\mathcal{U}_{CPU}$	$P_6^1(\tau_5) = \{-(34, 29)\}$

For task τ_3 different versions for the particle filter localization are available. The first character in the version specification determines the number and type of used cores. A `1` specifies, that one core is used, a `2` specifies the simultaneous parallel execution of the task on two cores. A `a` specifies the use of the hardware-accelerated version. The second character in the version specification determines the particle set size, which is a main factor for the execution time of the particle filter localization, here a `0` specifies a particle set size of 75, a `1` specifies a particle set size of 100, a `2` specifies a particle set size of 200 and so on. The different versions of the particle filter localization allow for a trade-off between the execution time and the result quality. For the navigation, a particle size of 100 is the lowest viable particle size and the sweet spot is determined with 200 particles. Parallel execution on two cores is modeled with a second task, which is required by a precedence constraint from the original task and is assigned the same execution time as the original task. The load and store operations are modeled with a dedicated task $\tau_{ld,st}$, connected via precedence constraints from the respective versions of task τ_3. All other precedence constraints allow for a slack of 5 ms to have the previous task end.

Three execution modes are available. The first execution mode considers single core execution of the particle filter localization and thus allows for the versions τ_3 v=1.0 to v=1.4, defined with $V^1(\tau_3) = \{1.0, 1.1, 1.2, 1.3, 1.4\}$. The

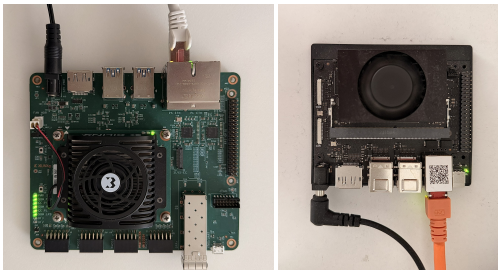
Table 8.2: Execution time, units and precedence constraints for multi-version tasks.

Task	C_i^v	U_i^v	P_i^v
τ_3, v=1.0	33.06	$\mathcal{U}_{CPU}$	$P_3^{1.0}(\tau_1) = \{-(14,9)\}$
τ_3, v=1.1	37.53	$\mathcal{U}_{CPU}$	$P_3^{1.1}(\tau_1) = \{-(14,9)\}$
τ_3, v=1.2	53.9	$\mathcal{U}_{CPU}$	$P_3^{1.2}(\tau_1) = \{-(14,9)\}$
τ_3, v=1.3	71.02	$\mathcal{U}_{CPU}$	$P_3^{1.3}(\tau_1) = \{-(14,9)\}$
τ_3, v=1.4	87.12	$\mathcal{U}_{CPU}$	$P_3^{1.4}(\tau_1) = \{-(14,9)\}$
τ_3, v=2.0	26.45	$\mathcal{U}_{CPU}$	$P_3^{2.0}(\tau_1) = \{-(14,9)\}, P_3^{2.0}(\tau_{3.2.0}) = \{(0,0)\}$
$\tau_{3.2.0}$, v=1	26.45	$\mathcal{U}_{CPU}$	$\varnothing$
τ_3, v=2.1	30.15	$\mathcal{U}_{CPU}$	$P_3^{2.1}(\tau_1) = \{-(14,9)\}, P_3^{2.1}(\tau_{3.2.1}) = \{(0,0)\}$
$\tau_{3.2.1}$, v=1	30.15	$\mathcal{U}_{CPU}$	$\varnothing$
τ_3, v=2.2	37.89	$\mathcal{U}_{CPU}$	$P_3^{2.2}(\tau_1) = \{-(14,9)\}, P_3^{2.2}(\tau_{3.2.2}) = \{(0,0)\}$
$\tau_{3.2.2}$, v=1	37.89	$\mathcal{U}_{CPU}$	$\varnothing$
τ_3, v=2.3	46.87	$\mathcal{U}_{CPU}$	$P_3^{2.3}(\tau_1) = \{-(14,9)\}, P_3^{2.3}(\tau_{3.2.3}) = \{(0,0)\}$
$\tau_{3.2.3}$, v=1	46.87	$\mathcal{U}_{CPU}$	$\varnothing$
τ_3, v=2.4	56.25	$\mathcal{U}_{CPU}$	$P_3^{2.4}(\tau_1) = \{-(14,9)\}, P_3^{2.4}(\tau_{3.2.4}) = \{(0,0)\}$
$\tau_{3.2.4}$, v=1	56.25	$\mathcal{U}_{CPU}$	$\varnothing$
τ_3, v=a.0	22.24	u_{ml}	$P_3^{a.0}(\tau_1) = \{-(14,9)\}, P_3^{a.0}(\tau_{ld,st}) = \{-(1,0.5),(23,24)\}$
τ_3, v=a.1	24.10	u_{ml}	$P_3^{a.1}(\tau_1) = \{-(14,9)\}, P_3^{a.1}(\tau_{ld,st}) = \{-(1,0.5),(24,25)\}$
τ_3, v=a.2	26.96	u_{ml}	$P_3^{a.2}(\tau_1) = \{-(14,9)\}, P_3^{a.2}(\tau_{ld,st}) = \{-(1,0.5),(26,27)\}$
τ_3, v=a.3	32.45	u_{ml}	$P_3^{a.3}(\tau_1) = \{-(14,9)\}, P_3^{a.3}(\tau_{ld,st}) = \{-(1,0.5),(32,33)\}$
τ_3, v=a.4	37.14	u_{ml}	$P_3^{a.4}(\tau_1) = \{-(14,9)\}, P_3^{a.4}(\tau_{ld,st}) = \{-(1,0.5),(37,38)\}$
$\tau_{ld,st}$, v=1	0.5 ms	$\mathcal{U}_{CPU}$	$\varnothing$

second execution mode considers the parallel execution of the particle filter localization on two cores and thus allows for the versions τ_3 v=2.0 to v=2.4, defined with $V^2(\tau_3) = \{2.0, 2.1, 2.2, 2.3, 2.4\}$. The third execution mode considers the use of the hardware-accelerated version of the particle filter localization and thus allows for the versions τ_3 v=a.0 to v=a.4, defined with $V^a(\tau_3) = \{a.0, a.1, a.2, a.3, a.4\}$. In all cases, the deadline of task τ_6 is set to 25 ms, which is the period of the odometry driver τ_1. This is represented in the model as $D^1(\tau_6) = D^2(\tau_6) = D^a(\tau_6) = 25\,\text{ms}$.

The task set is then used to generate schedules for both execution platforms, which are then deployed onto the KR260 platform, by configuring an RTOS simular to the deployment presented in Section 6.3 for the ZCU102. For the

Jetson platform the same is undergone, and the measured values hold true within a negligible deviation. The resulting schedule features multiple simultaneous lines of execution, since the sum of all execution times exceeds the given deadline for the navigation task τ_6.



(a) AMD KR260 Board [164] (b) NVIDIA Jetson Nano [165]

Figure 8.2: Deployment targets for the RoboRacer case study.

> ***Finding 13: Case Studies for the Real-World Applicability of the Presented Workflow***
>
> *Both case studies show the applicability of the presented scheduling and deployment workflow to real-world use cases, which are based on automotive use cases. The scheduling approaches are viable for scheduling the execution of a task set on a wide range of execution platforms, a Jetson Nano as well as a AMD KR260 board in the context of the RoboRacer platform and a ZCU102 development board in the context of the CeCaS project.*

Summary

The presented case studies show the applicability of the presented scheduling approaches to real-world scenarios. In the first case study, the FleXNNgine [28] is used as the main part of the platform and the scheduling approaches are used to schedule the execution of a neural network on this accelerator. The second case study shows the applicability of the scheduling approaches to a real-world use case, which is based on an autonomous driving vehicle. In the second case study, the scheduling approaches are used to schedule the execution of a task set on both, a Jetson Nano and an AMD KR260 board in the context of the RoboRacer platform.

This page intentionally left blank

Chapter 9

Conclusion and Outlook

After the introduction of the problem statement, together with the motivation and the research questions in Chapter 1, the background and related work were discussed in Chapter 2 and Chapter 3. The main part of this thesis, which is Chapter 4 and Chapter 5, presented the proposed time-triggered scheduler for multi-version tasks and the dynamic checkpoint-based scheduler for anytime neural networks. The combination of both methods and the deployment was presented in Chapter 6. Together with the evaluation of the proposed methods in Chapter 7, the case studies in Chapter 8 showed the real world applicability and generalization capabilities of the proposed methods. This chapter now concludes the thesis by summarizing the main contributions and evaluation findings and gives an outlook on future work.

9.1 Summary and Novelty of the Proposed Methods

The first contribution is the time-triggered scheduler for multi-version tasks, which is able to generate periodic schedules for task sets for multiple execution modes in one unified schedule graph. This includes the addition of mode-change schedules to transition between two schedules of different modes. The second contribution is the dynamic checkpoint-based scheduler for anytime neural networks, which is able to automatically generate early exits and anomaly detection modules from given preexisting neural network architectures. It provides a scheduler, which aims to maximize the deadline success rate for the execution of the generated anytime neural networks for phases with release times, which are smaller than the available deadlines, which in turn create over utilization. Simultaneously, the selection of early exits is undergone to maximize the accuracy of the executed early exits in a context of a given model zoo.

9.1.1 Summary of Multi-Version Time-Triggered Scheduling

The time triggered scheduler for multi-version task is the first main contribution of this work. The contribution starts with a task description, which includes execution time, available execution nodes, access windows towards shared resources, memory bandwidth for the access windows as well as precedence constraints towards other tasks. Each of these parameters is associated with a task version. Each scheduled job of a task is assigned the execution version by the scheduler and with that the corresponding parameters. This method describes the scheduling for one schedule. The multi-version task scheduler then extends this method to multiple modes. Each mode is defined by a set of implicit deadlines for scheduled tasks, viable task versions of the scheduled tasks and predecessor and successor modes. These predecessor and successor modes then represent which mode transitions are viable and desired. The scheduler starts with generating the schedule for the most complex mode and then moves through the successor modes until all modes have an assigned schedule. This list then includes mode changes should a successor mode be able to directly be executed after the preceding mode. After all modes have an assigned schedule, the scheduler then generates transitional schedules for all viable mode transitions. To reduce complexity, the scheduler is able to compress two or more execution modes into adapted schedules from a common main schedule, reducing the need for transitional schedules.

The scheduler itself generates the schedules by taking in the task description of the modes and the target mode. Then either a polyhedron-based approach or a simulated annealing-based approach is used to generate the schedule. The polyhedron-based approach generates schedules by first generating polyhedrons for each task and each to be scheduled job. This is repeated for every possible version. Then iteratively a polyhedron is selected and placed on the schedule based on given constraints and the rest of the set of polyhedrons of the same job is removed. This is repeated until all jobs are scheduled, or no polyhedron could be placed. This approach is used for simple scheduling problems and allows for a good expandability of the scheduling problem. Should no schedule be found, or the scheduling problem be too complex, the simulated annealing-based approach has to be used. Here, a starting solution is used and then iteratively adapted by the dual annealing process. To ensure a valid schedule, constraints are turned into penalty terms of the cost function and bonus terms are given for timeliness or using less execution resources. Further bonus terms, rewarding specific task versions could also be added. The cost function is designed to always produce a value larger than 0 for in-

valid schedules, to guarantee a valid schedule at the end of the process. A valid schedule is defined to adhere to given constraints, which are the precedence constraints, limiting the available memory bandwidth at each time step to 100%, ensuring that each execution unit only executes one job per time step and to ensure the meeting of the given deadlines. For the mode change schedules, the scheduler is also constraint to keep jobs executing on the same execution unit as the successor and predecessor schedules and to ensure a valid transition.

Before evaluating the time-triggered scheduler, an ablation study was undergone to find the best performing hyperparameters for the simulated annealing-based approach. These were selected to be an initial temperature of $2 \cdot 10^4$, a restart temperature ratio of 10^{-3}, a visit value of > 2.6 and an accept value of -5.0. The evaluation of the time-triggered scheduler then focused on the runtime to find valid schedules. An expected trend of a runtime increase with an increase in the number of tasks and versions was confirmed in the evaluation. Additionally, an increase in the number of access windows, precedence constraints and execution units also led to an increased runtime. Since the scheduler used the same concepts for both periodic schedules assigned to one mode as to a transitional schedule assigned to a mode change between two schedules, the evaluation held valid for both.

The generated schedule is then able to be deployed onto a target system. This deployment method is not the main focus of the thesis, but is shown and introduced nonetheless. For the deployment, different methods are introduced, using bare metal applications, dedicated scheduler implementation in hardware or the use of a Real-Time Operating System (RTOS) which implements the generated schedule. Lastly, case study two, which focused on the deployment of an autonomous driving stack in the context of the HAL4SDV project, shows the real world applicability of the proposed scheduler. The first case study furthermore shows the generalization capabilities and the inclusion of the anytime neural network system into the proposed scheduler for multi-version tasks, which is the second main contribution of this thesis.

9.1.2 Novelty of Multi-Version Time-Triggered Scheduling

As presented in Chapter 2, a large amount of works for real-time and mixed-criticality scheduling exist. These works support a subset of the features, which were proposed in this work. Table 9.1 shows a comparison of the proposed

time-triggered scheduler to related works. The comparison considers a selection of twelve works and shows six categories, which are either marked as ✓, signifying the support for the feature, a ✗, signifying that the feature is not supported or †, signifying that a feature might be implicitly supported but is not necessarily mentioned in the publication.

To compare the proposed scheduler to existing works a grouping of the works is helpful. These groups can be defined based on their supported features. One group focusses on generating schedules to minimize or eliminating collisions on a single shared resource. This resource is usually the main memory. This group includes [25, 26, 94, 167]. While this does not mean that other models do not minimize collisions, but rather that the focus of this group of works lies primarily on the modeling of a single shared resource. The second group, which comprising the works of [95, 100], focuses on modeling the impact of multiple shared resources. A third group supports the modeling of tasks on heterogeneous execution units. These units can include units like heterogeneous CPUs or GPUs and includes [97] as well as [146] implicitly. The next type of work focuses on describing task relations via precedence constraints, which also includes the modeling of gang tasks. Models which support gang tasks inherently support precedence constraints because gang tasks can be modeled as having precedence constraints of a simultaneous job start. This group comprises works like [98, 99]. These groups model the task execution either via directed acyclic graphs or via execution phases. A next group is dedicated to the implementation of multi-mode scheduling and multi-version tasks. Here [85, 96, 101] are considered. These works try to minimize a transition and implement a graph based scheduling approach for fixed-priority tasks. Since the priorities or the scheduling of the tasks change, the scheduled tasks are inherently multi-version. Additionally, timed Petri nets support modeling of the presented characteristics, but because a place in a timed Petri net does not correspond to any one characteristic directly, we categorize [146] independently.

The novelty of this thesis lies in the holistic approach to the scheduling of multi-version tasks in a multi-mode system, where multiple shared resources, multiple heterogeneous execution units and precedence constraints are supported. Additionally, the modeling of tasks as having dedicated execution phases, which can be placed at arbitrary time steps, allows for a more flexible scheduling approach, especially, when compared to established models like the Acquisition Execution Restitution (AER) [26] model or Predictable Execution Model (PREM) [25]. With the help of the precedence constraints and the execution phases, the scheduler is able to model gang tasks as well as the execution of ML-based tasks, which are often executed on dedicated accelera-

Table 9.1: Comparison of time-triggered multimode scheduler for multi-version tasks.

Work	Year	Mult. Res.	Heter. Units	Gang Tasks	Multi-V. Tasks	Prec. Constr.	Mult. Mode	Concept
Pellizoni et al. [25]	2011	✗	✗	✗	✗	✗	†	Pred. phases without cache
Durrieu et al. [26]	2014	✗	✗	✗	✗	✗	†	Phase-based with singular mem. acc.
Negrean et al. [85]	2011	✓	✗	†	✗	✓	✓	Bounding mode change transitions
Oliviera et al. [94]	2023	✗	✗	✗	✗	✗	✗	Stalling cores based on CFG
Aceituno et al. [95]	2021	✓	✗	✗	✗	✓	✗	Phase-based TTS minimizing interference
Yip et al. [96]	2014	✗	✗	†	✓	✓	✗	EDF scheduler, distributing slack time equally
Roeder et al. [97]	2021	✗	✓	✗	✓	✗	✗	DAG-based TTS with multiple exec. phases
Ueter et al. [98]	2021	✗	✗	✓	✗	†	✗	Hard real-time scheduling for gang tasks
Bhuiyan et al. [99]	2021	✗	✗	✓	✓	†	✓	GEDF scheduler for MC gang task model
Skalistis et al. [100]	2019	✓	✗	✗	✗	✓	✓	Phase-based TTS with fine-grained res. prot.
Sundar [101]	2022	✗	†	†	✓	✓	✓	Graph-based, fixed-priority and multi-mode
Zuberek [146]	1991	✓	†	✓	✗	✓	†	Timed Petri-Nets to model task execution
This Work	2026	✓	✓	✓	✓	✓	✓	TTS with sched. in directed graph

†Implicitly possible, but not focus

tors, requiring a load of data from the main memory to a scratchpad memory started from a CPU, which also necessitate the modeling of multiple shared resources. In combination with the proposed dynamic checkpoint-based scheduler for anytime neural networks, which is able to generate early exits and anomaly detection modules, this work allows for a holistic scheduling of multiple ML-based tasks with different versions onto a centralized execution platform.

9.1.3 Summary of the Checkpoint-based ANN Scheduler

The second main contribution was the proposed dynamic scheduler for anytime neural networks. This work could either be used in combination with the time-triggered scheduler for multi-version tasks, or be used independently to manage event-triggered incoming ML-based tasks. The contribution had multiple parts and started with the positioning of the early exits. This positioning could either result from an automatic specification from the time-triggered scheduler, an automatic positioning using a distance metric to estimate the best exit placement or be hand-picket by the designer. After the positioning of the exits, the early exits were trained. Since embedded platforms were targeted, where accelerator parameters were predetermined to optimally fit to the given workload, the architecture of the early exits oriented themselves on the original exit to ensure a deployment, which then likely fit onto the configured accelerators.

Automotive guidelines necessitated the use safety strategies in the context of safety critical applications. In this work, this was ensured by adding anomaly detection methods to the generated anytime neural networks. These anomaly detection methods were used to determine, if the inputs lay within the training distribution and operational design domain of the network. Should this not be the case, the scheduler chose an early exit with a more generalized class set or aborted the inference instead of providing a potential wrong inference result. The architecture of the anomaly detection methods was chosen and configured automatically during the design time and used the intermediate result of original network layers as an input. During runtime, the scheduler then chose for a continuation of the inference or an early exit based on a given timing budget, the estimated remaining execution time and the result of the anomaly detection methods.

In this thesis, anomaly detection methods included the use of ML-based methods, Support Vector Machine (SVM)-based methods, Gaussian Naive Bayes

(GNB)-based methods and entropy-based methods as well as distance measurements based on the Mahalanobis distance and the Euclidean distance. Orthogonal to that, the input dimension into the anomaly detection methods was reduced using Principal Component Analysis (PCA)-based methods, the concatenation, the summation or the use of average or max pooling to reduce the input dimension and computational complexity. The contribution also took the estimated runtime of the anomaly detection methods into account to place them in a way, which provided a result before the choice for the exit was undergone.

Choosing the next exit was done in a two-step process. The first step assigned a timing budget to each neural network task, which determined the proposed runtime for the task inference and indirectly chose the early exit. At each possible exit, a checkpoint structure was then used to determine, if the exit was executed or the inference was continued. This checkpoint combined the result of the anomaly detection methods and the estimated remaining execution time. Should one signify the need to choose an early exit, the scheduler chose the early exit of the current checkpoint.

The evaluation of the checkpoint-based scheduler for anytime neural networks again started with the ablation study to determine the closes fitting runtime estimation methods. Here, taking an exponentially decaying weighted average of the last 50 runtime samples multiplied with a correction factor of 1.2 gave enough overhead to meet the 99^{th} percentile of the runtime distribution while providing the best overhead. Additionally, should the inference be finished before the given timing budget, the resulting time could be used by other present tasks, potentially then choosing a later exit than originally planned. The independent evaluation of the scheduler then concerned three different task release characteristics. These characteristics included an uniform release, a randomized release and a batched release. The scheduler was evaluated for a rise in utilization, starting from 50 % utilization (with proposed 50% idle time) and finishing at 250 % utilization. The evaluation showed that the scheduler was able to meet the given deadlines for all three release characteristics up to 150 % utilization. After that, the deadline success rate started to drop, but was still able to meet a deadline success rate of around 90 % at 250 % utilization. An expected accuracy drop was observed with an increase in utilization, which matched the accuracy of the chosen early exits. The scheduler was able to adapt to the different release characteristics, with the best performance for the uniform release and the worst performance for the batched release.

Case study one then showed the real-world applicability of the proposed

scheduler for the inclusion into the time-triggered scheduler. By modelling the exit as one task with multiple versions, the scheduler was then able to choose a fitting version for the exit, which allowed for a valid schedule. The checkpoint structure was then used to determine, if an earlier exit might need to be executed. This then led to a mode change, with the corresponding transition being automatically scheduled.

9.1.4 Novelty of the Checkpoint-based ANN Scheduler

Table 9.2 shows a comparison of the proposed dynamic checkpoint-based scheduler for anytime neural networks to related work. The table shows a comparison of ten recent related works across seven categories. The first of these categories is the automated design of early exits as well as anomaly detection methods for both pretrained neural networks and dedicated structures. Second and third, the inter-inference and intra-inference scheduling concerns the scheduling of early exit choices between and during inferences. Forth, the possibility of multiple different networks to be scheduled in the same system is compared. The backbone network is considered as the fifth comparison and the possibilities to schedule between heterogeneous platforms is compared sixth. Lastly, the optimization goal is listed. Related work can be classified into three categories. Automatic design of anytime neural network (ANN) components, which include the early exits and the anomaly detection modules is the first category. The second category is the scheduling of the inference of existing ANN architectures. Lastly holistic approaches, which include both the scheduling and the automatic design of the ANN components, are the third category.

In the first category, methods like BranchyNet [68], AdaDet [102], XAI-Exit [105] and E^{2}CM [106] are included. These methods all focus on the design of the early exits, where XAI-Exit [105] and E^{2}CM [106] also provide automatized interfaces for adding early exits to arbitrary networks [105, 106]. For AdaDet [102] the design of the early exits focuses on existing networks, still providing the opportunity to automatically add exits to existing models. The second category includes schedulers for neural networks. Here SLEXNet [107], AdaEE [69], S.E. EENNs [104], RT-mDL [108], LaLaRAND [110] and DNN-SAM [111] are included. These methods all provide scheduling for the inference of neural networks, SLEXNet, AdaEE and S.E. EENNs provide interfaces to schedule early exits, RT-mDL [108] and LaLaRAND [110] focus on the layer-wise scheduling of neural networks and DNN-SAM [111] focuses on the scheduling of classification of specific regions of interest based on outside information. Some

Table 9.2: Comparison of the proposed dynamic checkpoint-based scheduler for anytime neural networks to related work.

Work	Year	Auto. EE & AD Design	Inter-Inf. Sched.	Intra-Inf. Sched.	Mult-Net. Sched.	Backbone	Heter. Plat.	Opt. Goal
BranchyNet [68]	2016	✗	✗	✓	✗	Custom	✗	Energy
AdaEE [69]	2023	✗	✗	✓	✓	MobileNetV2 [168]	✗	Confidence
AdaDet [102]	2024	✓	✗	✓	✗	Multiple [169, 170]	✗	Certainty
EDANAS [103]	2023	✓	✗	✓	✗	MobileNetV3 [171]	✓	Energy
S.E. EENNs [104]	2024	✗	✓	✓	✓	Custom & VGG19 [172]	✗	Timeliness
XAI-Exit [105]	2026	✓	✗	✓	✗	Multiple [45, 169, 171]	✗	Transparency
E²CM [106]	2022	✓	✗	✗	✗	Multiple [45, 171, 173]	✗	Cost v. Acc.
SLEXNet [107]	2024	✗	✗	✓	✗	EfficientNet [173]	✗	Power
RT-mDL [108]	2021	✗	✓	✗	✓	Multiple [45, 48, 172]	✓	Timeliness
Pantheon [109]	2024	✓	✓	✓†	✓	Multiple [45, 168, 172]	✗	Timeliness
LaLaRAND [110]	2021	✗	✓	✓	✓	Multiple [44, 168]	✗	Timeliness
DNN-SAM [111]	2022	✗	✓	✗	✗	DarkNet [174]	✗	RoI-Accuracy
This Work	2026	✓	✓	✓	✓	Multiple [44, 46, 175, 176]	✓	Timeliness

†Only for time, not anomalies

works allow for an intra-inference scheduling, readapting the used early exits during execution of one inference [102, 105, 107]. Other works also support a readaptation of schedules between inferences [104, 108, 109, 110] The works from the second group do not support the joint scheduling and design. Lastly, Pantheon [109], EDANAS [103] and this work present a joined scheduling and design approach. Pantheon [109] focuses on the scheduling of multiple networks, which are additionally adapted to include early exits and is closest to this work. A subset of works includes the dedicated support for heterogeneous execution platforms.

This work excels in the joint scheduling and design of the early exits and anomaly detection modules, which is not supported by any other work. Additionally, the proposed scheduler supports the scheduling of multiple networks, which is only supported by a small subset of related work. Lastly, the proposed method is designed to work in conjunction with the time-triggered scheduler for multi-version tasks, which allows for a holistic scheduling of multiple ML-based tasks with different execution modes and the inclusion of the proposed scheduler as one of the modes, which is not supported by any other work.

9.2 Future Work

No research is ever finished, and the work undergone in this thesis is no exception. While providing a holistic system for the scheduling of multi-version tasks and machine learning (ML)-based tasks in a heterogeneous system-on-chip (HESoC) environment, multiple avenues for future research remain open.

The first avenue for future work is the extension of the scheduling framework to support not only time triggered tasks, but also event-triggered tasks. While the current framework is designed to *emulate* event-triggered tasks by using two different schedule modes, one where the corresponding task is scheduled and a second, where the task is not scheduled, a *true support* for event-triggered tasks would allow for a potentially lower response time. This also includes interrupts, modeled as event-triggered tasks. Using this approach would necessitate a schedulability analysis, which necessitates an upper bound on the number of tasks, which can be scheduled in one hyperperiod of the time-triggered schedule. Supporting event-triggered tasks with flexible runtimes would also allow for a more flexible execution of the anytime neural network (ANN)s, which are currently scheduled in a time-triggered manner, but could

be scheduled in an event-triggered manner, potentially allowing for a lower response time and a more efficient use of the available resources by not blocking resources for the allotted execution time.

The second avenue for future work is an extension of the ANN scheduling framework to support transformer networks. While a support for transformers is supported in the model, an implementation in the deployment is not yet available. Because transformer networks have a higher computational demand and a less predictable execution time, a support for transformers would require a substantial change in the scheduling framework and would introduce more overhead in the runtime estimation. However, a support for transformer networks would allow for a more flexible use of ML-based tasks, as transformer networks see a rise in popularity in experimental automotive design.

The final avenue for future work is the extension of the deployment to support further platforms and operating systems. One example for this is the deployment onto Real-time Linux-based platforms, like the TT-Linux operating system of TTTech [89], which would necessitate an extension of the work for multimode scheduling, as the current implementation supports only a single mode of operation. Deploying the schedules onto an NVIDIA AGX Orin [31] platform would also allow using a more powerful platform for the execution of ML-based tasks. The combination of these two points comes as the natural next steps in the development of this heterogeneous scheduling framework.

This page intentionally left blank

Bibliography

[1] Paulweber, Michael; Eckel, Andreas and Azzoni, Paolo: "Shaping the Vehicle of the Future: How FEDERATE and HAL4SDV Are Steering Europe's Software-Defined Vehicle Ecosystem". In: *2025 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Lyon, France: IEEE, Apr. 2025, pp. 1–6. doi: 10.23919/DATE64628.2025.10992971 (cit. on pp. 1, 6–8, 257, 262).

[2] Gutermann, Annina et al.: "Multi-Partner Project: CeCaS Accelerator Design for Efficient Supercomputing in Automotive Systems". In: *2026 Design, Automation & Test in Europe Conference (DATE)*. Verona, Italy: IEEE, Apr. 2026, pp. 1–7. doi: 10.23919/DATE69613.2026.11539165 (cit. on pp. 1, 2, 257).

[3] Burkacky, Ondrej; Deichmann, Johannes and Stein, Jan Paul: Automotive Software and Electronics 2030: Mapping the Sector's Future Landscape Authors, Ondrej Burkacky, Johannes Deichmann, Jan Paul Stein. New York: McKinsey & Company, 2023 (cit. on pp. 1–5, 7, 8, 258).

[4] Ashjaei, Mohammad; Lo Bello, Lucia; Daneshtalab, Masoud; Patti, Gaetano; Saponara, Sergio and Mubeen, Saad: "Time-Sensitive Networking in Automotive Embedded Systems: State of the Art and Research Opportunities". In: *Journal of Systems Architecture* 117 (Aug. 2021), p. 102137. doi: 10.1016/j.sysarc.2021.102137 (cit. on p. 2).

[5] Bandur, Victor; Selim, Gehan; Pantelic, Vera and Lawford, Mark: "Making the Case for Centralized Automotive E/E Architectures". In: *IEEE Transactions on Vehicular Technology* 70.2 (Feb. 2021), pp. 1230–1245. doi: 10.1109/TVT.2021.3054934 (cit. on pp. 2, 3).

[6] Grander, Beate and Badstuebner, Frank: MANNHEIM CeCaS - Central Car Server. 2022 (cit. on p. 2).

[7] Road Vehicles – Functional Safety – Part 5: Product Development at the Hardware Level. Standard ISO 26262-5:2018. Geneva, Switzerland: International Organization for Standardization, 2018 (cit. on pp. 3, 4, 49).

[8] Road Vehicles – Functional Safety – Part 6: Product Development at the Software Level. Standard ISO 26262-6:2018. Geneva, Switzerland: International Organization for Standardization, 2018 (cit. on pp. 3–5, 20, 49).

[9] Moujahid, Abdallah; ElAraki Tantaoui, Mounir; Hina, Manolo Dulva; Soukane, Assia; Ortalda, Andrea; ElKhadimi, Ahmed and Ramdane-Cherif, Amar: "Machine Learning Techniques in ADAS: A Review". In: *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. Paris: IEEE, June 2018, pp. 235–242. doi: 10.1109/ICACCE.2018.8441758 (cit. on p. 3).

[10] Sinaei, Sima; Mohammadi, Mohammadreza; Shrestha, Rakesh; Alibeigi, Mina and Eklund, David: "PRIV-DRIVE: Privacy-Ensured Federated Learning Using Homomorphic Encryption for Driver Fatigue Detection". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. Paris, France: IEEE, Aug. 2024, pp. 427–434. doi: 10.1109/DSD64264.2024.00063 (cit. on p. 3).

[11] De-Las-Heras, Gonzalo; Sánchez-Soriano, Javier and Puertas, Enrique: "Advanced Driver Assistance Systems (ADAS) Based on Machine Learning Techniques for the Detection and Transcription of Variable Message Signs on Roads". In: *Sensors* 21.17 (Aug. 2021), p. 5866. doi: 10.3390/s21175866 (cit. on p. 3).

[12] Jhung, Junekyo; Suk, Ho; Park, Hyungbin and Kim, Shiho: "Hardware Accelerators for Autonomous Vehicles". In: *Artificial Intelligence and Hardware Accelerators*. Ed. by Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin and Kim, Shiho. Cham: Springer International Publishing, 2023, pp. 269–317. doi: 10.1007/978-3-031-22170-5_9 (cit. on p. 3).

[13] Talpes, Emil et al.: "Compute Solution for Tesla's Full Self-Driving Computer". In: *IEEE Micro* 40.2 (Mar. 2020), pp. 25–35. doi: 10.1109/MM.2020.2975764 (cit. on pp. 3, 28).

[14] Bello, Lucia Lo; Mariani, Riccardo; Mubeen, Saad and Saponara, Sergio: "Recent Advances and Trends in On-Board Embedded and Networked Automotive Systems". In: *IEEE Transactions on Industrial Informatics* 15.2 (Feb. 2019), pp. 1038–1051. doi: 10.1109/TII.2018.2879544 (cit. on p. 3).

[15] Road Vehicles – Functional Safety – Part 1: Vocabulary. Standard ISO 26262-1:2018. Geneva, Switzerland: International Organization for Standardization, 2018 (cit. on p. 4).

[16] Jiang, Zhe; Zhao, Shuai; Dong, Pan; Yang, Dawei; Wei, Ran; Guan, Nan and Audsley, Neil: "Re-Thinking Mixed-Criticality Architecture for Automotive Industry". In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. Hartford, CT, USA: IEEE, Oct. 2020, pp. 510–517. doi: 10.1109/ICCD50377.2020.00092 (cit. on p. 4).

[17] Liu, Zongwei; Zhang, Wang and Zhao, Fuquan: "Impact, Challenges and Prospect of Software-Defined Vehicles". In: *Automotive Innovation* 5.2 (Apr. 2022), pp. 180–194. doi: 10.1007/s42154-022-00179-z (cit. on p. 4).

[18] Cinque, Marcello; Cotroneo, Domenico; De Simone, Luigi and Rosiello, Stefano: "Virtualizing Mixed-Criticality Systems: A Survey on Industrial Trends and Issues". In: *Future Generation Computer Systems* 129 (Apr. 2022), pp. 315–330. doi: 10.1016/j.future.2021.12.002 (cit. on pp. 5, 39).

[19] Luna, Ryan and Islam, Sheikh Ariful: "Security and Reliability of Safety-Critical RTOS". In: *SN Computer Science* 2.5 (Sept. 2021), p. 356. doi: 10.1007/s42979-021-00753-y (cit. on p. 5).

[20] Herlt, Anna; Paul, Jana; Kellner, Martin; Küchler, Sebastian and Rochlitz, Henrik: Smartphones on Wheels: New Rules for Automotive Product Development. New York: McKinsey & Company, 2022 (cit. on p. 5).

[21] Michal, Rataj; Chiara, Lodi; Jakub, Zawieska; Marcin, Stepniak; Ilias, Cheimariotis; Monica, Grosso; Francesca, Piazza and Alessandro, Marotta:

New and Emerging Transport Technologies and Trends in European Research and Innovation Projects 2024. Tech. rep. KJ-01-25-103-EN-N (online),KJ-01-25-103-EN-C (print). Luxembourg (Luxembourg): Publications Office of the European Union, 2025 (cit. on p. 6).

[22] Teraglobus: Hardware Abstraction Layer for a European Software Defined Vehicle Approach - HAL4SDV. https://www.hal4sdv.eu/about. 2024. (Visited on 10/29/2025) (cit. on p. 7).

[23] Hamann, Matthias; Moch, Alexander; Krause, Matthias and Mikhalev, Vasily: "The DRACO Stream Cipher: A Power-efficient Small-state Stream Cipher with Full Provable Security against TMDTO Attacks". In: *IACR Transactions on Symmetric Cryptology* (June 2022), pp. 1–42. doi: 10.46586/tosc.v2022.i2.1-42 (cit. on pp. 11, 17, 257–259).

[24] Advanced Micro Devices, Inc.: ZCU102 Evaluation Board User Guide (UG1182). Feb. 2023. (Visited on 02/26/2026) (cit. on pp. 12, 18–20, 199, 200, 258).

[25] Pellizzoni, Rodolfo; Betti, Emiliano; Bak, Stanley; Yao, Gang; Criswell, John; Caccamo, Marco and Kegley, Russell: "A Predictable Execution Model for COTS-Based Embedded Systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Chicago, IL, USA: IEEE, Apr. 2011, pp. 269–279. doi: 10.1109/RTAS.2011.33 (cit. on pp. 13, 18, 37, 41, 43–46, 64, 272, 273, 320).

[26] Durrieu, Guy; Faugère, Madeleine; Girbal, Sylvain; Gracia Pérez, Daniel; Pagetti, Claire and Puffitsch, W.: "Predictable Flight Management System Implementation on a Multicore Processor". In: *Embedded Real-time Software (ERTS'14)*. TOULOUSE, France, Feb. 2014 (cit. on pp. 13, 18, 19, 37, 41, 43, 45, 46, 64, 272, 273).

[27] Hennessy, John L. and Patterson, David A.: Computer Architecture: A Quantitative Approach. Fifth edition. Amsterdam Heidelberg: Elsevier, Morgan Kaufmann, 2012 (cit. on p. 17).

[28] Lesniak, Fabian; Gutermann, Annina; Harbaum, Tanja and Becker, Jürgen: "Enhanced Accelerator Design for Efficient CNN Processing with Improved Row-Stationary Dataflow". In: *Proceedings of the Great Lakes Symposium on VLSI 2024*. Clearwater FL USA: ACM, June 2024, pp. 151–157. doi: 10.1145/3649476.3658737 (cit. on pp. 17, 28, 29, 31, 199, 257–259, 267).

[29] Chen, Yu-Hsin; Yang, Tien-Ju; Emer, Joel S. and Sze, Vivienne: "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (June 2019), pp. 292–308. doi: 10.1109/JETCAS.2019.2910232 (cit. on pp. 17, 28).

[30] Gao, Yuxiang and Zhang, Peng: "A Survey of Homogeneous and Heterogeneous System Architectures in High Performance Computing". In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. New York, NY, USA: IEEE, Nov. 2016, pp. 170–175. doi: 10.1109/SmartCloud.2016.36 (cit. on p. 18).

[31] Leela S. Karumbunathan: NVIDIA Jetson AGX Orin Series - A Giant Leap Forward for Robotics and Edge AI Applications. Technical Brief. Santa Clara, CA, USA, July 2022 (cit. on pp. 18, 279).

[32] Wilhelm, Reinhard et al.: "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53. doi: 10.1145/1347375.1347389 (cit. on pp. 18, 36).

[33] Staschulat, Jan: Instruction and Data Cache Timing Analysis in Fixed-Priority Preemptive Real-Time Systems. 1st ed. Göttingen: Cuvillier Verlag, 2007 (cit. on p. 18).

[34] Bin, Jingyi: "Controlling Execution Time Variability Using COTS for Safety-critical Systems". Doctorat Ès Physique , PhD. Paris 11, July 2014. doi: 10.70675/b306ce6czf998z40bcz9189z7e4e8a292df5 (cit. on pp. 18, 37).

[35] Arm Limited: AMBA AXI Protocol Specification (ARM IHI 0022). Specification Document. Aug. 2025 (cit. on p. 19).

[36] Anantharajaiah, Nidhi; Lesniak, Fabian; Harbaum, Tanja and Becker, Juergen: "Reinforcement Learning Enabled Multi-Layered NoC for Mixed Criticality Systems". In: *2023 IEEE 16th International Symposium on Embedded Multi-core/Many-core Systems-on-Chip (MCSoC)*. Singapore: IEEE, Dec. 2023, pp. 38–44. doi: 10.1109/MCSoC60832.2023.00014 (cit. on p. 19).

[37] Restuccia, Francesco; Pagani, Marco; Biondi, Alessandro; Marinoni, Mauro and Buttazzo, Giorgio: "Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs". In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019), pp. 1–22. doi: 10.1145/3358183 (cit. on p. 19).

[38] Theiling, Henrik: PikeOS and Time-Triggering. Klein-Winternheim, May 2013 (cit. on pp. 20, 38, 39, 185).

[39] Lee, Min; Krishnakumar, A. S.; Krishnan, P.; Singh, Navjot and Yajnik, Shalini: "Supporting Soft Real-Time Tasks in the Xen Hypervisor". In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Pittsburgh Pennsylvania USA: ACM, Mar. 2010, pp. 97–108. doi: 10.1145/1735997.1736012 (cit. on pp. 20, 38, 185).

[40] Dörr, T.: "Pattern-Based Logical Isolation for Safety-Critical Multicore Systems". In: (2024). doi: 10.5445/IR/1000175388 (cit. on pp. 20, 39).

[41] Dörr, Tobias; Schade, Florian and Becker, Jürgen: "Pattern-Based Information Flow Control for Safety-Critical On-Chip Systems". In: *Computer Safety, Reliability, and Security*. Ed. by Guiochet, Jérémie; Tonetta, Stefano and Bitsch, Friedemann. Vol. 14181. Cham: Springer Nature Switzerland, 2023, pp. 181–195. doi: 10.1007/978-3-031-40923-3_14 (cit. on p. 20).

[42] Jordan, M. I. and Mitchell, T. M.: "Machine Learning: Trends, Perspectives, and Prospects". In: *Science* 349.6245 (July 2015), pp. 255–260. doi: 10.1126/science.aaa8415 (cit. on p. 21).

[43] Russell, Stuart J. and Norvig, Peter: Artificial Intelligence: A Modern Approach. 2nd ed. Prentice Hall Series in Artificial Intelligence. Upper Saddle River, N.J: Prentice Hall/Pearson Education, 2003 (cit. on p. 21).

[44] Szegedy, Christian; Wei Liu; Yangqing Jia; Sermanet, Pierre; Reed, Scott; Anguelov, Dragomir; Erhan, Dumitru; Vanhoucke, Vincent and Rabinovich, Andrew: "Going Deeper with Convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA: IEEE, June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594 (cit. on pp. 21, 136, 137, 203, 205, 207, 209, 236–238, 240, 247, 249, 259, 277).

[45] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing and Sun, Jian: "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on pp. 21, 132, 133, 136, 137, 139, 140, 160, 241, 277).

[46] Xie, Saining; Girshick, Ross; Dollar, Piotr; Tu, Zhuowen and He, Kaiming: "Aggregated Residual Transformations for Deep Neural Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, July 2017, pp. 5987–5995. DOI: 10.1109/CVPR.2017.634 (cit. on pp. 21, 136, 137, 203, 205, 207, 209, 236–238, 240, 247, 249, 259, 277).

[47] Wang, Linnan; Yu, Chenhan; Salian, Satish; Kierat, Slawomir; Migacz, Szymon and Florea, Alex Fit: "Searching the Deployable Convolution Neural Networks for GPUs". In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. New Orleans, LA, USA: IEEE, June 2022, pp. 12217–12226. DOI: 10.1109/CVPR52688.2022.01191 (cit. on pp. 21, 204, 205, 209, 247, 249).

[48] Khanam, Rahima and Hussain, Muhammad: YOLOv11: An Overview of the Key Architectural Enhancements. 2024. DOI: 10.48550/ARXIV.2410.17725 (cit. on pp. 21, 204, 205, 209, 247, 249, 277).

[49] Lloyd, S.: "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489 (cit. on p. 21).

[50] Jolliffe, Ian T.: Principal Component Analysis. Springer Series in Statistics. New York: Springer-Verlag, 2002. DOI: 10.1007/b98835 (cit. on pp. 21, 152).

[51] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Łukasz and Polosukhin, Illia: "Attention Is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010 (cit. on p. 22).

[52] Caron, Mathilde; Touvron, Hugo; Misra, Ishan; Jegou, Herve; Mairal, Julien; Bojanowski, Piotr and Joulin, Armand: "Emerging Properties in Self-Supervised Vision Transformers". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Montreal, QC, Canada: IEEE, Oct. 2021, pp. 9630–9640. DOI: 10.1109/ICCV48922.2021.00951 (cit. on p. 22).

[53] Wilcox, Ethan; Qian, Peng; Futrell, Richard; Kohita, Ryosuke; Levy, Roger and Ballesteros, Miguel: "Structural Supervision Improves Few-Shot Learning and Syntactic Generalization in Neural Language Models". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*

(EMNLP). Online: Association for Computational Linguistics, 2020, pp. 4640–4652. DOI: 10.18653/v1/2020.emnlp-main.375 (cit. on p. 22).

[54] OpenAI et al.: GPT-4 Technical Report. 2023. DOI: 10.48550/ARXIV.2303.08774 (cit. on p. 22).

[55] Krishna, Tushar; Kwon, Hyoukjun; Parashar, Angshuman; Pellauer, Michael and Samajdar, Ananda: Data Orchestration in Deep Learning Accelerators. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-031-01767-4 (cit. on pp. 22, 27).

[56] Goodfellow, Ian; Bengio, Yoshua and Courville, Aaron: Deep Learning. MIT Press, 2016 (cit. on pp. 22–25, 143).

[57] Kingma, Diederik P. and Ba, Jimmy: Adam: A Method for Stochastic Optimization. 2014. DOI: 10.48550/ARXIV.1412.6980 (cit. on pp. 24, 143).

[58] Rumelhart, David E.; Hinton, Geoffrey E. and Williams, Ronald J.: "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0 (cit. on p. 24).

[59] LeCun, Yann; Bengio, Yoshua and Hinton, Geoffrey: "Deep Learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. DOI: 10.1038/nature14539 (cit. on pp. 25, 26).

[60] Govindaraju, Naga K.; Lloyd, Brandon; Wang, Wei; Lin, Ming and Manocha, Dinesh: "Fast Computation of Database Operations Using Graphics Processors". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. Paris France: ACM, June 2004, pp. 215–226. DOI: 10.1145/1007568.1007594 (cit. on p. 27).

[61] Krizhevsky, Alex; Sutskever, Ilya and Hinton, Geoffrey E.: "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. DOI: 10.1145/3065386 (cit. on p. 27).

[62] Kung, S.: "VLSI Array Processors". In: *IEEE ASSP Magazine* 2.3 (1985), pp. 4–22. DOI: 10.1109/MASSP.1985.1163741 (cit. on p. 28).

[63] Sze, Vivienne; Chen, Yu-Hsin; Yang, Tien-Ju and Emer, Joel S.: Efficient Processing of Deep Neural Networks. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-031-01766-7 (cit. on pp. 28, 29).

[64] Genc, Hasan et al.: "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Dec. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216 (cit. on p. 28).

[65] Farshchi, Farzad; Huang, Qijing and Yun, Heechul: "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim". In: *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. Washington, DC, USA: IEEE, Feb. 2019, pp. 21–25. DOI: 10.1109/EMC249363.2019.00012 (cit. on p. 28).

[66] Montello, Fabio; Güldenring, Ronja; Scardapane, Simone and Nalpantidis, Lazaros: "A Survey on Dynamic Neural Networks: From Computer Vision

to Multi-Modal Sensor Fusion". In: *Image and Vision Computing* 170 (June 2026), p. 105980. DOI: 10.1016/j.imavis.2026.105980 (cit. on pp. 29–31).

[67] RAHMATH P, Haseena; SRIVASTAVA, Vishal; CHAURASIA, Kuldeep; PACHECO, Roberto G. and COUTO, Rodrigo S.: "Early-Exit Deep Neural Network - A Comprehensive Survey". In: *ACM Computing Surveys* 57.3 (Mar. 2025), pp. 1–37. DOI: 10.1145/3698767 (cit. on pp. 29, 31).

[68] TEERAPITTAYANON, Surat; MCDANEL, Bradley and KUNG, H.T.: "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. Cancun: IEEE, Dec. 2016, pp. 2464–2469. DOI: 10.1109/ICPR.2016.7900006 (cit. on pp. 30, 50, 51, 276, 277).

[69] PACHECO, Roberto G.; SHIFRIN, Mark; COUTO, Rodrigo S.; MENASCHÉ, Daniel S.; HANAWAL, Manjesh K. and CAMPISTA, Miguel Elias M.: "AdaEE: Adaptive Early-Exit DNN Inference Through Multi-Armed Bandits". In: *ICC 2023 - IEEE International Conference on Communications*. Rome, Italy: IEEE, May 2023, pp. 3726–3731. DOI: 10.1109/ICC45041.2023.10279243 (cit. on pp. 30, 53, 138, 276, 277).

[70] EUROPEAN PARLIAMENT AND COUNCIL OF THE EUROPEAN UNION: Regulation (EU) 2024/1689 on Harmonised Rules in the Artificial Intelligence (Artificial Intelligence Act). Official Journal of the European Union, L 2024/1689. June 2024 (cit. on p. 32).

[71] Road Vehicles – Safety and artificial intelligence. Standard ISO 8800:2024. Geneva, Switzerland: International Organization for Standardization, 2024 (cit. on pp. 32, 147, 148, 150).

[72] HOEFER, Julian; LESNIAK, Fabian; GUTERMANN, Annina; WÖRAN, Felix; HARBAUM, Tanja and BECKER, Juergen: "BayNNgine: Hardware-Enabled Bayesian Neural Network Support for Dependable AI Inference". In: *2025 IEEE 38th International System-on-Chip Conference (SOCC)*. Dubai, United Arab Emirates: IEEE, Sept. 2025, pp. 1–6. DOI: 10.1109/SOCC66126.2025.11235419 (cit. on pp. 32, 150).

[73] SCHORN, Christoph and GAUERHOF, Lydia: "FACER: A Universal Framework for Detecting Anomalous Operation of Deep Neural Networks". In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. Rhodes, Greece: IEEE, Sept. 2020, pp. 1–6. DOI: 10.1109/ITSC45102.2020.9294226 (cit. on pp. 32, 56, 150, 154).

[74] HSU, Yen-Chang; SHEN, Yilin; JIN, Hongxia and KIRA, Zsolt: "Generalized ODIN: Detecting Out-of-Distribution Image Without Learning From Out-of-Distribution Data". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, June 2020, pp. 10948–10957. DOI: 10.1109/CVPR42600.2020.01096 (cit. on pp. 32, 56, 57, 150, 156).

[75] GE, Xiaohua; HAN, Qing-Long; ZHANG, Xian-Ming and DING, Derui: "Dynamic Event-triggered Control and Estimation: A Survey". In: *International Journal of Automation and Computing* 18.6 (Dec. 2021), pp. 857–886. DOI: 10.1007/s11633-021-1306-z (cit. on pp. 33, 34).

[76] Harki, Naji; Ahmed, Abdulraheem and Haji, Lailan: "CPU Scheduling Techniques: A Review on Novel Approaches Strategy and Performance Assessment". In: *Journal of Applied Science and Technology Trends* 1.1 (May 2020), pp. 48–55. doi: 10.38094/jastt1215 (cit. on pp. 33–36).

[77] Minaeva, Anna and Hanzálek, Zdeněk: "Survey on Periodic Scheduling for Time-triggered Hard Real-time Systems". In: *ACM Computing Surveys* 54.1 (Jan. 2022), pp. 1–32. doi: 10.1145/3431232 (cit. on pp. 33, 34, 36).

[78] Haupt, R.: "A Survey of Priority Rule-Based Scheduling". In: *OR Spektrum* 11.1 (Mar. 1989), pp. 3–16. doi: 10.1007/BF01721162 (cit. on pp. 34, 35).

[79] Audsley, N. C.: 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 (cit. on p. 35).

[80] Kumar, Niraj; Gao, Chuanchao and Easwaran, Arvind: "Optimal Fixed Priority Scheduling in Multi-Stage Multi-Resource Distributed Real-Time Systems". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Valencia, Spain: IEEE, Mar. 2024, pp. 1–6. doi: 10.23919/DATE58400.2024.10546551 (cit. on p. 35).

[81] Hardy, Damien; Rouxel, Benjamin and Puaut, Isabelle: "The Heptane Static Worst-Case Execution Time Estimation Tool". In: *OASIcs, Volume 57, WCET 2017* 57 (2017). Ed. by Reineke, Jan, 8:1–8:12. doi: 10.4230/OASICS.WCET.2017.8 (cit. on p. 36).

[82] Singh, Nikhilesh; Renganathan, Karthikeyan; Rebeiro, Chester; Jose, Jithin and Mader, Ralph: "Kryptonite: Worst-Case Program Interference Estimation on Multi-Core Embedded Systems". In: *ACM Transactions on Embedded Computing Systems* 22.5s (Oct. 2023), pp. 1–23. doi: 10.1145/3609128 (cit. on pp. 36, 37).

[83] Nagar, Kartik and Srikant, Y. N.: "Fast and Precise Worst-Case Interference Placement for Shared Cache Analysis". In: *ACM Transactions on Embedded Computing Systems* 15.3 (July 2016), pp. 1–26. doi: 10.1145/2854151 (cit. on p. 37).

[84] Brilli, G.; Valente, G.; Capotondi, A.; Burgio, P.; Di Masciov, T.; Valente, P. and Marongiu, A.: "Fine-Grained QoS Control via Tightly-Coupled Bandwidth Monitoring and Regulation for FPGA-based Heterogeneous SoCs". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, July 2023, pp. 1–6. doi: 10.1109/DAC56929.2023.10247840 (cit. on pp. 37, 200, 260).

[85] Negrean, Mircea; Neukirchner, Moritz; Stein, Steffen; Schliecker, Simon and Ernst, Rolf: "Bounding Mode Change Transition Latencies for Multi-Mode Real-Time Distributed Applications". In: *ETFA2011*. Toulouse, France: IEEE, Sept. 2011, pp. 1–10. doi: 10.1109/ETFA.2011.6059009 (cit. on pp. 37, 47, 49, 89, 272, 273).

[86] Crespo, A.; Ripoll, I. and Masmano, M.: "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach". In: *2010 European Dependable*

Computing Conference. Valencia, Spain: IEEE, 2010, pp. 67–72. DOI: 10.1109/EDCC.2010.18 (cit. on p. 38).

[87] AMAZON WEB SERVICES: The FreeRTOS Reference Manual. 2017. (Visited on 02/13/2026) (cit. on pp. 38, 185, 198).

[88] REGHENZANI, Federico; MASSARI, Giuseppe and FORNACIARI, William: "The Real-Time Linux Kernel: A Survey on PREEMPT_RT". In: *ACM Computing Surveys* 52.1 (Jan. 2020), pp. 1–36. DOI: 10.1145/3297714 (cit. on p. 38).

[89] KARACHATZIS, Paraskevas; RUH, Jan and CRACIUNAS, Silviu S.: "An Evaluation of Time-triggered Scheduling in the Linux Kernel". In: *The 31st International Conference on Real-Time Networks and Systems*. Dortmund Germany: ACM, June 2023, pp. 119–131. DOI: 10.1145/3575757.3593660 (cit. on pp. 38, 40, 279).

[90] ZEPHYR PROJECT: The Zephyr Project. 2024. URL: https://www.zephyrproject.org/ (visited on 07/02/2026) (cit. on p. 38).

[91] TANENBAUM, Andrew S.: Modern Operating Systems. Fourth edition. Boston: Pearson, 2015 (cit. on p. 39).

[92] PORTNOY, Matthew: Virtualization Essentials. Second edition. Indianapolis, Indiana: John Wiley & Sons, 2016 (cit. on p. 39).

[93] BRANDENBURG, Björn: Liu and Layland and Linux: A Blueprint for "Proper" Real-Time Task. https://sigbed.org/2020/09/05/liu-and-layland-and-linux-ablueprint-for-proper-real-time-tasks/. 2012. (Visited on 02/23/2026) (cit. on pp. 40, 198).

[94] OLIVEIRA, Daniel; CHEN, Weifan; PINTO, Sandro and MANCUSO, Renato: "Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers". In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. San Antonio TX USA: ACM, May 2023, pp. 221–226. DOI: 10.1145/3576914.3587513 (cit. on pp. 43, 44, 272, 273).

[95] ACEITUNO, José María; GUASQUE, Ana; BALBASTRE, Patricia; SIMÓ, José and CRESPO, Alfons: "Hardware Resources Contention-Aware Scheduling of Hard Real-Time Multiprocessor Systems". In: *Journal of Systems Architecture* 118 (Sept. 2021), p. 102223. DOI: 10.1016/j.sysarc.2021.102223 (cit. on pp. 43, 45, 46, 272, 273).

[96] YIP, Eugene; KUO, Matthew M Y; ROOP, Partha S and BROMAN, David: "Relaxing the Synchronous Approach for Mixed-Criticality Systems". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Berlin, Germany: IEEE, Apr. 2014, pp. 89–100. DOI: 10.1109/RTAS.2014.6925993 (cit. on pp. 43, 46, 272, 273).

[97] ROEDER, Julius; ROUXEL, Benjamin and GRELCK, Clemens: "Scheduling DAGs of Multi-Version Multi-Phase Tasks on Heterogeneous Real-Time Systems". In: *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Singapore, Singapore: IEEE, Dec. 2021, pp. 54–61. DOI: 10.1109/MCSoC51149.2021.00016 (cit. on pp. 43, 46, 47, 272, 273).

[98] UETER, Niklas; GÜNZEL, Mario; VON DER BRÜGGEN, Georg and CHEN, Jian-Jia: "Hard Real-Time Stationary GANG-Scheduling". In: *LIPIcs, Volume 196, ECRTS*

2021 196 (2021), 10:1–10:19. DOI: 10.4230/LIPICS.ECRTS.2021.10 (cit. on pp. 43, 47, 48, 272, 273).

[99] Bhuiyan, Ashikahmed; Yang, Kecheng; Arefin, Samsil; Saifullah, Abusayeed; Guan, Nan and Guo, Zhishan: "Mixed-Criticality Real-Time Scheduling of Gang Task Systems". In: *Real-Time Systems* 57.3 (July 2021), pp. 268–301. DOI: 10.1007/s11241-021-09368-1 (cit. on pp. 47, 48, 272, 273).

[100] Skalistis, Stefanos and Kritikakou, Angeliki: "Timely Fine-Grained Interference-Sensitive Run-Time Adaptation of Time-Triggered Schedules". In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. Hong Kong, Hong Kong: IEEE, Dec. 2019, pp. 233–245. DOI: 10.1109/RTSS46320.2019.00030 (cit. on pp. 47, 48, 272, 273).

[101] Sundar, Vijaya Kumar: "Design and Analysis of Task Models for Mixed-Criticality Systems with Practical Considerations". PhD thesis. Nanyang Technological University, 2022. DOI: 10.32657/10356/155381. HDL: 10356/155381 (cit. on pp. 47, 49, 50, 272, 273).

[102] Yang, Le; Zheng, Ziwei; Wang, Jian; Song, Shiji; Huang, Gao and Li, Fan: "AdaDet: An Adaptive Object Detection System Based on Early-Exit Neural Networks". In: *IEEE Transactions on Cognitive and Developmental Systems* 16.1 (Feb. 2024), pp. 332–345. DOI: 10.1109/TCDS.2023.3274214 (cit. on pp. 50, 276–278).

[103] Gambella, Matteo and Roveri, Manuel: "EDANAS: Adaptive Neural Architecture Search for Early Exit Neural Networks". In: *2023 International Joint Conference on Neural Networks (IJCNN)*. Gold Coast, Australia: IEEE, June 2023, pp. 1–8. DOI: 10.1109/IJCNN54540.2023.10191876 (cit. on pp. 50, 51, 138, 277, 278).

[104] Casale, Giuliano and Roveri, Manuel: "Scheduling Inputs in Early Exit Neural Networks". In: *IEEE Transactions on Computers* 73.2 (Feb. 2024), pp. 451–465. DOI: 10.1109/tc.2023.3333189 (cit. on pp. 50–52, 276–278).

[105] Rahmath P, Haseena; Abraham, Ajith and Chaurasia, Kuldeep: "XAI-Exit: Interpretability-Driven Dynamic Early Exits for Efficient and Transparent DNN Inference". In: *IEEE Transactions on Neural Networks and Learning Systems* (2026), pp. 1–12. DOI: 10.1109/TNNLS.2026.3685408 (cit. on pp. 50, 52, 276–278).

[106] Gormez, Alperen; Dasari, Venkat R. and Koyuncu, Erdem: "E^2 CM: Early Exit via Class Means for Efficient Supervised and Unsupervised Learning". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. Padua, Italy: IEEE, July 2022, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9891952 (cit. on pp. 50, 52, 53, 276, 277).

[107] Kutukcu, Basar; Baidya, Sabur and Dey, Sujit: "SLEXNet: Adaptive Inference Using Slimmable Early Exit Neural Networks". In: *ACM Transactions on Embedded Computing Systems* 23.6 (Nov. 2024), pp. 1–29. DOI: 10.1145/3689632 (cit. on pp. 53, 276–278).

[108] Ling, Neiwen; Wang, Kai; He, Yuze; Xing, Guoliang and Xie, Daqi: "RT-mDL: Supporting Real-Time Mixed Deep Learning Tasks on Edge Platforms". In: *Pro-*

ceedings of the 19th ACM Conference on Embedded Networked Sensor Systems. Coimbra Portugal: ACM, Nov. 2021, pp. 1–14. DOI: 10.1145/3485730.3485938 (cit. on pp. 53, 54, 276–278).

[109] HAN, Lixiang; ZHOU, Zimu and LI, Zhenjiang: "Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs". In: *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*. Minato-ku, Tokyo Japan: ACM, June 2024, pp. 465–478. DOI: 10.1145/3643832.3661878 (cit. on pp. 53, 54, 277, 278).

[110] KANG, Woosung; LEE, Kilho; LEE, Jinkyu; SHIN, Insik and CHWA, Hoon Sung: "LaLaRAND: Flexible Layer-by-Layer CPU/GPU Scheduling for Real-Time DNN Tasks". In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. Dortmund, DE: IEEE, Dec. 2021, pp. 329–341. DOI: 10.1109/RTSS52674.2021.00038. (Visited on 09/30/2026) (cit. on pp. 53, 55, 276–278).

[111] KANG, Woosung; CHUNG, Siwoo; KIM, Jeremy Yuhyun; LEE, Youngmoon; LEE, Kilho; LEE, Jinkyu; SHIN, Kang G. and CHWA, Hoon Sung: "DNN-SAM: Split-and-Merge DNN Execution for Real-Time Object Detection". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Milano, Italy: IEEE, May 2022, pp. 160–172. DOI: 10.1109/RTAS54340.2022.00021. (Visited on 09/30/2026) (cit. on pp. 53, 55, 276, 277).

[112] MAHALANOBIS, Prasanta Chandra: "Reprint of: Mahalanobis, P.C. (1936) "On the Generalised Distance in Statistics."" In: *Sankhya A* 80.S1 (Dec. 2018), pp. 1–7. DOI: 10.1007/s13171-019-00164-5 (cit. on pp. 57, 156).

[113] KEDAD-SIDHOUM, Safia; MONNA, Florence and TRYSTRAM, Denis: "Scheduling Tasks with Precedence Constraints on Hybrid Multi-core Machines". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. Hyderabad, India: IEEE, May 2015, pp. 27–33. DOI: 10.1109/IPDPSW.2015.119 (cit. on p. 64).

[114] GOLDFARB, D. and REID, J. K.: "A Practicable Steepest-Edge Simplex Algorithm". In: *Mathematical Programming* 12.1 (Dec. 1977), pp. 361–371. DOI: 10.1007/BF01593804 (cit. on p. 72).

[115] LAND, A. H. and DOIG, A. G.: "An Automatic Method of Solving Discrete Programming Problems". In: *Econometrica* 28.3 (July 1960), p. 497. DOI: 10.2307/1910129. JSTOR: 1910129 (cit. on p. 72).

[116] RAGONNEAU, Tom M.: Model-Based Derivative-Free Optimization Methods and Software. 2022. DOI: 10.48550/ARXIV.2210.12018 (cit. on p. 72).

[117] XIANG, Y; SUN, D.Y; FAN, W and GONG, X.G: "Generalized Simulated Annealing Algorithm and Its Application to the Thomson Model". In: *Physics Letters A* 233.3 (Aug. 1997), pp. 216–220. DOI: 10.1016/S0375-9601(97)00474-X (cit. on pp. 72, 213).

[118] VAN LAARHOVEN, Peter J. M. and AARTS, Emile H. L.: "Simulated Annealing". In: *Simulated Annealing: Theory and Applications*. Dordrecht: Springer Netherlands, 1987, pp. 7–15. DOI: 10.1007/978-94-015-7744-1_2 (cit. on pp. 72, 321).

[119] Chen, Tianyang and Phan, Linh Thi Xuan: "SafeMC: A System for the Design and Evaluation of Mode-Change Protocols". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Porto: IEEE, Apr. 2018, pp. 105–116. doi: 10.1109/RTAS.2018.00021 (cit. on p. 89).

[120] Phan, Linh T.X. and Lee, Insup: "Towards a Compositional Multi-modal Framework for Adaptive Cyber-physical Systems". In: *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*. Toyama, Japan: IEEE, Aug. 2011, pp. 67–73. doi: 10.1109/RTCSA.2011.82 (cit. on p. 89).

[121] Davis, Robert Ian; Burns, Alan and Bate, Iain: "Compensating Adaptive Mixed Criticality Scheduling". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. Paris France: ACM, June 2022, pp. 81–93. doi: 10.1145/3534879.3534895 (cit. on p. 89).

[122] Zhang, Yi-Wen; Zheng, Hui and Gu, Zonghua: "Energy-Aware Adaptive Mixed-Criticality Scheduling with Semi-Clairvoyance and Graceful Degradation". In: *ACM Transactions on Embedded Computing Systems* 23.1 (Jan. 2024), pp. 1–20. doi: 10.1145/3632749 (cit. on p. 89).

[123] Heilmann, Florian; Syed, Ali and Fohler, Gerhard: "Mode-Changes in COTS Time-Triggered Network Hardware without Online Reconfiguration". In: *ACM SIGBED Review* 13.4 (Nov. 2016), pp. 55–60. doi: 10.1145/3015037.3015046 (cit. on p. 89).

[124] Weiss, Philipp and Steinhorst, Sebastian: "Predictable Timing Behavior of Gracefully Degrading Automotive Systems". In: *Design Automation for Embedded Systems* 27.1-2 (June 2023), pp. 103–138. doi: 10.1007/s10617-023-09271-x (cit. on p. 89).

[125] Advanced Micro Devices, Inc.: MicroBlaze Processor Reference Guide (UG984). Datasheet. Nov. 2025. (Visited on 02/04/2026) (cit. on p. 113).

[126] Advanced Micro Devices, Inc.: MicroBlaze V Processor Reference Guide (UG1629). Datasheet. Nov. 2025. (Visited on 02/04/2026) (cit. on pp. 113, 191, 195).

[127] Advanced Micro Devices, Inc.: Vivado Design Suite: AXI Reference Guide (UG1037). Datasheet. July 2017. (Visited on 02/04/2026) (cit. on p. 114).

[128] Huang, Gao; Liu, Zhuang; Van Der Maaten, Laurens and Weinberger, Kilian Q.: "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, July 2017, pp. 2261–2269. doi: 10.1109/CVPR.2017.243 (cit. on pp. 136, 137).

[129] Scardapane, Simone; Scarpiniti, Michele; Baccarelli, Enzo and Uncini, Aurelio: "Why Should We Add Early Exits to Neural Networks?" In: *Cognitive Computation* 12.5 (Sept. 2020), pp. 954–966. doi: 10.1007/s12559-020-09734-4 (cit. on p. 138).

[130] Bonato, Vanderlei and Bouganis, Christos-Savvas: "Class-Specific Early Exit Design Methodology for Convolutional Neural Networks". In: *Applied Soft Computing* 107 (Aug. 2021), p. 107316. doi: 10.1016/j.asoc.2021.107316 (cit. on p. 138).

[131] Kreß, Fabian; El Annabi, El Mahdi; Hotfilter, Tim; Hoefer, Julian; Harbaum, Tanja and Becker, Juergen: "Automated Deep Neural Network Inference Partitioning for Distributed Embedded Systems". In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Knoxville, TN, USA: IEEE, July 2024, pp. 39–44. doi: 10.1109/ISVLSI61997.2024.00019 (cit. on pp. 138–140).

[132] Kreß, Fabian: Ressourcenbasierte Inferenz-Partitionierung von Deep Neural Networks in eingebetteten Systemen. 2025. doi: 10.5445/IR/1000180033 (cit. on pp. 138–140).

[133] Miller, George A.: "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (Nov. 1995), pp. 39–41. doi: 10.1145/219717.219748 (cit. on pp. 141, 144, 239).

[134] Deng, Jia; Dong, Wei; Socher, Richard; Li, Li-Jia; Kai Li and Li Fei-Fei: "ImageNet: A Large-Scale Hierarchical Image Database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL: IEEE, June 2009, pp. 248–255. doi: 10.1109/CVPR.2009.5206848 (cit. on pp. 141, 144, 145, 150, 204, 239, 241).

[135] Hendrycks, Dan; Zhao, Kevin; Basart, Steven; Steinhardt, Jacob and Song, Dawn: "Natural Adversarial Examples". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Nashville, TN, USA: IEEE, June 2021, pp. 15257–15266. doi: 10.1109/CVPR46437.2021.01501 (cit. on pp. 150, 204, 241).

[136] Hendrycks, Dan et al.: "The Many Faces of Robustness: A Critical Analysis of Out-of-Distribution Generalization". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Montreal, QC, Canada: IEEE, Oct. 2021, pp. 8320–8329. doi: 10.1109/ICCV48922.2021.00823 (cit. on pp. 150, 241).

[137] Ramakrishna, Shreyas; Rahiminasab, Zahra; Easwaran, Arvind and Dubey, Abhishek: "Efficient Multi-Class Out-of-Distribution Reasoning for Perception Based Networks: Work-in-Progress". In: *2020 International Conference on Embedded Software (EMSOFT)*. Shanghai, China: IEEE, Sept. 2020, pp. 40–42. doi: 10.1109/EMSOFT51651.2020.9244027 (cit. on pp. 150, 154).

[138] McInnes, Leland; Healy, John; Saul, Nathaniel and Großberger, Lukas: "UMAP: Uniform Manifold Approximation and Projection". In: *Journal of Open Source Software* 3.29 (Sept. 2018), p. 861. doi: 10.21105/joss.00861 (cit. on p. 152).

[139] Jolliffe, Ian T. and Cadima, Jorge: "Principal Component Analysis: A Review and Recent Developments". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065 (Apr. 2016), p. 20150202. doi: 10.1098/rsta.2015.0202 (cit. on p. 152).

[140] Higgins, Irina; Matthey, Loic; Pal, Arka; Burgess, Christopher; Glorot, Xavier; Botvinick, Matthew; Mohamed, Shakir and Lerchner, Alexander: "Beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework". In: *International Conference on Learning Representations*. 2017. url: https://openreview.net/forum?id=Sy2fzU9gl (cit. on pp. 154, 322).

[141] Schorn, Christoph; Guntoro, Andre and Ascheid, Gerd: "Efficient On-Line Error Detection and Mitigation for Deep Neural Network Accelerators". In: *Developments in Language Theory*. Ed. by Hoshi, Mizuho and Seki, Shinnosuke. Vol. 11088. Cham: Springer International Publishing, 2018, pp. 205–219. doi: 10.1007/978-3-319-99130-6_14 (cit. on p. 154).

[142] Cortes, Corinna and Vapnik, Vladimir: "Support-Vector Networks". In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. doi: 10.1007/BF00994018 (cit. on pp. 155, 321).

[143] McCallum, Andrew; Pal, Chris; Druck, Greg and Wang, Xuerui: "Multi-Conditional Learning: Generative/Discriminative Training for Clustering and Classification". In: *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*. AAAI'06. Boston, Massachusetts: AAAI Press, 2006, pp. 433–439 (cit. on pp. 155, 316).

[144] Csiszar, I.: "Divergence Geometry of Probability Distributions and Minimization Problems". In: *The Annals of Probability* 3.1 (Feb. 1975). doi: 10.1214/aop/1176996454 (cit. on p. 156).

[145] Hotfilter, Tim; KreB, Fabian; Kempf, Fabian; Becker, Jurgen and Baili, Imen: "Data Movement Reduction for DNN Accelerators: Enabling Dynamic Quantization Through an eFPGA". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Nicosia, Cyprus: IEEE, July 2022, pp. 371–372. doi: 10.1109/ISVLSI54635.2022.00082 (cit. on p. 159).

[146] Zuberek, W.M.: "Timed Petri Nets Definitions, Properties, and Applications". In: *Microelectronics Reliability* 31.4 (Jan. 1991), pp. 627–644. doi: 10.1016/0026-2714(91)90007-T (cit. on pp. 185, 272, 273).

[147] Zuberek, W. M.: "M-Timed Petri Nets, Priorities, Preemptions, and Performance Evaluation of Systems". In: *Advances in Petri nets 1985*. Ed. by Rozenberg, G. Vol. 222. Berlin/Heidelberg: Springer-Verlag, 1986, pp. 478–498. doi: 10.1007/BFb0016227 (cit. on pp. 185, 186).

[148] Merlin, P. and Farber, D.: "Recoverability of Communication Protocols–Implications of a Theoretical Study". In: *IEEE Transactions on Communications* 24.9 (Sept. 1976), pp. 1036–1043. doi: 10.1109/TCOM.1976.1093424 (cit. on p. 186).

[149] Juan, E.Y.T.; Tsai, J.J.P.; Murata, T. and Yi Zhou: "Reduction Methods for Real-Time Systems Using Delay Time Petri nets". In: *IEEE Transactions on Software Engineering* 27.5 (May 2001), pp. 422–448. doi: 10.1109/32.922714 (cit. on p. 186).

[150] Dudzik, Konstantin; Kirschner, Maximilian and Becker, Jürgen: "Automated Deployment of Real-Time Tasks for Phased Execution on Scratchpad-Based Multicore Platforms". In: *2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Singapore, Singapore: IEEE, Dec. 2025, pp. 554–561. doi: 10.1109/MCSoC67473.2025.00092 (cit. on pp. 190–193).

[151] Amid, Alon et al.: "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (July 2020), pp. 10–21. doi: 10.1109/MM.2020.2996616 (cit. on p. 191).

[152] Asanović, Krste et al.: The Rocket Chip Generator. Tech. rep. UCB/EECS-2016-17. Apr. 2016 (cit. on pp. 191, 195).

[153] Advanced Micro Devices, Inc.: MicroBlaze Processor Embedded Design User Guide (UG1579). Dec. 2025. (Visited on 02/20/2026) (cit. on p. 195).

[154] STMicroelectronics NV: STM32H747xI/G Datasheet - Production Data (DS12930). Datasheet. Jan. 2026 (cit. on pp. 197, 198).

[155] Serrano-Cases, Alejandro; Mezzetti, Enrico; Abella, Jaume and Cazorla, Francisco J.: "Achieving Flexible Performance Isolation on the AMD Xilinx Zynq UltraScale+". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. Paris, France: IEEE, Aug. 2024, pp. 282–290. doi: 10.1109/DSD64264.2024.00045 (cit. on p. 200).

[156] Redmon, Joseph; Divvala, Santosh; Girshick, Ross and Farhadi, Ali: "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, June 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91 (cit. on p. 204).

[157] Virtanen, Pauli et al.: "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17.3 (Mar. 2020), pp. 261–272. doi: 10.1038/s41592-019-0686-2 (cit. on p. 213).

[158] Granville, V.; Krivanek, M. and Rasson, J.-P.: "Simulated Annealing: A Proof of Convergence". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.6 (June 1994), pp. 652–656. doi: 10.1109/34.295910. (Visited on 11/05/2024) (cit. on pp. 215, 220).

[159] Hendrycks, Dan and Dietterich, Thomas G.: "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations". In: *7th International Conference on Learning Representations, ICLR*. New Orleans, LA, USA: OpenReview.net, May 2019 (cit. on p. 241).

[160] Charles, Israel; Maghsoumi, Hossein and Fallah, Yaser: "Advancing Autonomous Racing: A Comprehensive Survey of the Roboracer (F1TENTH) Platform". In: *2025 6th International Conference on Artificial Intelligence, Robotics and Control (AIRC)*. Savannah, GA, USA: IEEE, May 2025, pp. 207–213. doi: 10.1109/AIRC64931.2025.11077481 (cit. on pp. 257, 262).

[161] Ibrahim, Mohamed; Haworth, James and Cheng, Tao: "WeatherNet: Recognising Weather and Visual Conditions from Street-Level Images Using Deep Residual Learning". In: *ISPRS International Journal of Geo-Information* 8.12 (Nov. 2019), p. 549. doi: 10.3390/ijgi8120549 (cit. on p. 259).

[162] Stallkamp, J.; Schlipsing, M.; Salmen, J. and Igel, C.: "Man vs. Computer: Benchmarking Machine Learning Algorithms for Traffic Sign Recognition". In: *Neural Networks* 32 (Aug. 2012), pp. 323–332. doi: 10.1016/j.neunet.2012.02.016 (cit. on p. 259).

[163] Gavioli, Federico; Moretti, Francesco; Russo, Antonio; Capotondi, Alessandro and Burgio, Paolo: "Thundershot: An Open-Source Autonomous Vehicles

Research Platform for Embedded Heterogeneous MPSoCs". In: *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions*. Cagliari Italy: ACM, May 2025, pp. 26–29. DOI: 10.1145/3706594.3726968 (cit. on pp. 262, 263).

[164] Advanced Micro Devices, Inc.: Kria KR260 Robotics Starter Kit User Guide (UG1092). Apr. 2024. (Visited on 02/04/2026) (cit. on pp. 262, 267).

[165] NVIDIA Corporation: NVIDIA Jetson Nano System-on-Module. Technical Brief. Santa Clara, CA, USA, July 2022 (cit. on pp. 263, 267).

[166] Yuhas, Michael John: "Out-of-Distribution Detection as a Safety Monitor in Cyber-Physical Systems". PhD thesis. Nanyang Technological University, 2025. DOI: 10.32657/10356/199912. HDL: 10356/199912 (cit. on p. 263).

[167] Legout, Vincent; Jan, Mathieu and Pautet, Laurent: "A Scheduling Algorithm to Reduce the Static Energy Consumption of Multiprocessor Real-Time Systems". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. Sophia Antipolis France: ACM, Oct. 2013, pp. 99–108. DOI: 10.1145/2516821.2516839 (cit. on p. 272).

[168] Howard, Andrew G.; Zhu, Menglong; Chen, Bo; Kalenichenko, Dmitry; Wang, Weijun; Weyand, Tobias; Andreetto, Marco and Adam, Hartwig: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. arXiv: 1704.04861 [cs.CV] (cit. on p. 277).

[169] Huang, Gao; Chen, Danlu; Li, Tianhong; Wu, Felix; van der Maaten, Laurens and Weinberger, Kilian Q.: "Multi-Scale Dense Networks for Resource Efficient Image Classification". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018 (cit. on p. 277).

[170] Zhou, Xingyi; Wang, Dequan and Krähenbühl, Philipp: Objects as Points. 2019. arXiv: 1904.07850 [cs.CV] (cit. on p. 277).

[171] Howard, Andrew et al.: Searching for MobileNetV3. 2019. arXiv: 1905.02244 [cs.CV] (cit. on p. 277).

[172] Simonyan, Karen and Zisserman, Andrew: Very Deep Convolutional Networks for Large-Scale Image Recognition. 2015. arXiv: 1409.1556 [cs.CV] (cit. on p. 277).

[173] Tan, Mingxing and Le, Quoc: "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Chaudhuri, Kamalika and Salakhutdinov, Ruslan. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, pp. 6105–6114 (cit. on p. 277).

[174] Redmon, Joseph: Darknet: Open Source Neural Networks in C. https://pjreddie.com/darknet/. 2013–2016 (cit. on p. 277).

[175] Iandola, Forrest N.; Han, Song; Moskewicz, Matthew W.; Ashraf, Khalid; Dally, William J. and Keutzer, Kurt: SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size. Nov. 2016. DOI: 10.48550/arXiv.1602.07360. arXiv: 1602.07360 [cs.CV]. (Visited on 10/08/2026) (cit. on p. 277).

[176] Tan, Mingxing; Chen, Bo; Pang, Ruoming; Vasudevan, Vijay; Sandler, Mark; Howard, Andrew and Le, Quoc V.: MnasNet: Platform-Aware Neural Architecture Search for Mobile. May 2019. doi: 10.48550/arXiv.1807.11626. arXiv: 1807.11626 [cs.CV]. (Visited on 10/08/2026) (cit. on p. 277).

[177] Bucaioni, Alessio and Pelliccione, Patrizio: "Technical Architectures for Automotive Systems". In: *2020 IEEE International Conference on Software Architecture (ICSA)*. Salvador, Brazil: IEEE, Mar. 2020, pp. 46–57. doi: 10.1109/ICSA47634.2020.00013 (cit. on p. 316).

This page intentionally left blank

Publications

This section contains a complete list of own first author publications, followed by the list of co-author publications. The first three letters of the first author are used as the key, followed by the year of publication and an alphabetic identifier, should more than one entry exist for that author and year.

[Sta23a] Stammler, Matthias; Hoefer, Julian; Kraus, David; Schmidt, Patrick; Hotfilter, Tim; Harbaum, Tanja and Becker, Jürgen: "EFFECT: An End-to-End Framework for Evaluating Strategies for Parallel AI Anomaly Detection". In: *Procedia Computer Science* 222 (2023), pp. 499–508. doi: 10.1016/j.procs.2023.08.188 (cit. on pp. 14, 131, 138, 148, 150, 171).

[Sta23b] Stammler, Matthias; Hamann, Matthias and Becker, Jürgen: "Multilevel Security Model for Secure Information Flow Inside Software Components Employing Automated Code Generation". In: *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. Budva, Montenegro: IEEE, June 2023, pp. 1–6. doi: 10.1109/MECO58584.2023.10154914 (cit. on pp. 14, 108, 118, 124).

[Sta23c] Stammler, Matthias; Sidorenko, Vladimir; Kreß, Fabian; Schmidt, Patrick and Becker, Jürgen: "Context-Aware Layer Scheduling for Seamless Neural Network Inference in Cloud-Edge Systems". In: *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Singapore: IEEE, Dec. 2023, pp. 97–104. doi: 10.1109/MCSoC60832.2023.00022 (cit. on pp. 14, 131, 159, 171).

[Sta23d] Stammler, Matthias; Hamann, Matthias; Harbaum, Tanja and Becker, Jürgen: "Mitigating Masking in Automotive Communication Systems: Modeling and Hardware Generation". In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. Golem, Albania: IEEE, Sept. 2023, pp. 561–568. doi: 10.1109/DSD60849.2023.00083 (cit. on pp. 14, 108, 109, 114).

[Sta24a] Stammler, Matthias; Schade, Florian and Becker, Jürgen: "Automated Polyhedron-based TDMA Schedule Design for Predictable Mixed-Criticality MPSoCs". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. Paris, France: IEEE, Aug. 2024, pp. 403–409. doi: 10.1109/DSD64264.2024.00060 (cit. on pp. 14, 72, 74, 218, 221).

[Sta24b] Stammler, Matthias; Hoefer, Julian; Schmidt, Patrick; Harbaum, Tanja and Becker, Jürgen: "ICE TEA: Insertion of Custom Early Exits for Time-, Energy- & Anomaly-Aware Neural Networks". In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Knoxville, TN, USA: IEEE, July 2024, pp. 656–660. doi: 10.1109/ISVLSI61997.2024.00125 (cit. on pp. 14, 31, 131, 138, 144, 171, 237, 240).

[Sta24c] Stammler, Matthias et al.: "UNCOVER: Data-Driven Design Support through Continuous Monitoring of Security Incidents". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Valencia, Spain: IEEE, Mar. 2024, pp. 1–6. doi: 10.23919/DATE58400.2024.10546791 (cit. on p. 257).

[Sta25a] Stammler, Matthias; Lesniak, Fabian; Kumar, Niraj; Easwaran, Arvind and Becker, Jürgen: "HM2GP: A Multi-Version Task Scheduler with Extended Precedence Constraints on COTS SoCs". In: *2025 IEEE 38th International System-on-Chip Conference (SOCC)*. Dubai, United Arab Emirates: IEEE, Sept. 2025, pp. 1–6. doi: 10.1109/SOCC66126.2025.11235395 (cit. on pp. 59, 79, 218, 257).

[Sta27] Stammler, Matthias; Sotiropoulus, Georgios; Hoefer, Julian; Harbaum, Tanja; Guha, Krishnendu; Chakrabarti, Amlan and Becker, Jürgen: "Checkpoint-Based Integration of Early Exits for Automotive ML Tasks in Centralized Platforms". In: *Emerging Centralised Automotive E/E Architectures*. Lecture Notes in Networks and Systems XV. München: Springer International Publishing (Verlag), 2027 (cit. on pp. 10, 14, 131, 159, 164, 171).

[Sta25b] Stammler, Matthias; Sotiropoulos, Georgios; Hoefer, Julian; Harbaum, Tanja; Guha, Krishnendu; Chakrabarti, Amlan and Becker, Jürgen: "CH2AI: Dynamic Checkpoint Scheduling of Anytime Neural Networks on Heterogeneous Units". In: *2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Singapore: IEEE, Dec. 2025, pp. 173–180. doi: 10.1109/MCSoC67473.2025.00036 (cit. on pp. 14, 30, 131, 159, 164, 171, 225, 248, 251, 253, 254).

[Sta26] Stammler, Matthias; Scheidt, Henrik; Harbaum, Tanja; Becker, Jürgen; Dudzik, Konstantin; Pazmino, Victor; Gavioli, Federico; Burgio, Paolo; Easwaran, Arvind and Eckel, Andreas: "Multi-Partner Project: Scheduling-Deployment Workflow for Autonomous RoboRacer Driving Stacks in the HAL4SDV Project". In: *2026 Design, Automation &; Test in Europe Conference (DATE)*. Verona, Italy: IEEE, Apr. 2026, pp. 1–7. doi: 10.23919/DATE69613.2026.11539649 (cit. on pp. 79, 190, 257, 262).

[Hoe24] Hoefer, Julian; Stammler, Matthias; Kreß, Fabian; Hotfilter, Tim; Harbaum, Tanja and Becker, Juergen: "BayWatch: Leveraging Bayesian Neural Networks for Hardware Fault Tolerance and Monitoring". In: *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Didcot, United Kingdom: IEEE, Oct. 2024, pp. 1–6. doi: 10.1109/DFT63277.2024.10753546.

[Sch24a] Schmidt, Patrick et al.: "EMDRIVE Architecture: Embedded Distributed Computing and Diagnostics from Sensor to Edge". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Valencia, Spain: IEEE, Mar. 2024, pp. 1–6. doi: 10.23919/DATE58400.2024.10546796 (cit. on pp. 1, 2).

[Sch24b] Schmidt, Patrick; Pfau, Johannes; Hotfilter, Tim; Stammler, Matthias; Harbaum, Tanja and Becker, Jürgen: "RVVe: A Minimal RISC-V Vector Processor for Embedded AI Acceleration". In: *2024 IEEE 37th International System-on-Chip Conference (SOCC)*. Dresden, Germany: IEEE, Sept. 2024, pp. 1–6. doi: 10.1109/SOCC62300.2024.10737723.

[Sch25] Schmidt, Patrick; Krey, Fabian; Serdyuk, Alexey; Stammler, Matthias; Harbaum, Tanja and Becker, Jürgen: "DSEParted: Co-Optimization of Embedded NPU Architectures and Neural Network Partitioning". In: *2025 28th Euromicro Conference on Digital System Design (DSD)*. Salerno, Italy: IEEE, Sept. 2025, pp. 653–660. doi: 10.1109/DSD67783.2025.00094.

This page intentionally left blank

Student Theses

This section contains a list of student theses which have been supervised by this thesis' author.

[Abd23] Abdollahnejadbarough, Farshid: "Optimierung von ML-Techniken zur Erkennung von Anomalien durch Layer Activation Tracing". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology, 2023 (cit. on p. 246).

[Far24] Farheen, Shaista: "Anomaly Detection Methods During AI Inference Using Conventional Methods to Increase Confidence in Results". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology, 2024 (cit. on p. 246).

[Fin24] Findiku, Viola: "Plattformdeployment zur automatisierten und parallelen Zuordnung von Softwarekomponenten zu Ausführungseinheiten auf Grundlage bestehender TDMA-Schedules". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology, 2024 (cit. on p. 200).

[Rue26] Rueth, Sophia: "Decision Framework for the Systematic Design of Isolation Architecture Configurations in Embedded Mixed-Criticality Systems". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology, 2026.

This page intentionally left blank

Figures

1.1 Automotive Architecture Shift towards a Centralized Car . . . 3
1.2 Hardware Abstraction Layer Envisioned in HAL4SDV 7
1.3 Precedence Constraints of the Example Tasks 11
1.4 System Platform Architecture used in the Examples 12

2.1 Example of Phase-based Execution Model 19
2.2 Types of Machine Learning Workloads and Application Examples 21
2.3 Structure of an Artificial Neuron 22
2.4 Example of One Convolutional Calculation 26
2.5 Structure of a Multiply-Accumulate Unit 27
2.6 Structure of a Systolic Array 28
2.7 Example Structure of an Anytime Neural Network 30
2.8 Event-triggered Scheduling Concept 36

4.1 Creation of the 3D Spaces for Polyhedral-based Scheduling . 73
4.2 Creation of Task Polyhedron Examples 74
4.3 Polyhedron Basket Filling with Generated Polyhedrons 75
4.4 Overview of the Polyhedron-based Scheduling Algorithm . . 76
4.5 Two Examples of Polyhedron Task Sets 77
4.6 Distance Function for Simulated Annealing-based Scheduling 80
4.7 Time Diagram of the Generated Example Schedule 88
4.8 Task Versions for Execution Modes of given Example Tasks . . 93
4.9 Transitional Period and other Timing Definitions 99
4.10 Algorithm Components for Mode Change Scheduling 100
4.11 Example Specification Graph and Schedule Graph 102
4.12 Flexible Multi Mode Schedule Adaptation 104
4.13 Optimized Multi Mode Schedule Adaptation 106
4.14 Specification Graph and Adapted Schedule for Example Tasks 107
4.15 Structure of Attachment and Filter Component 113
4.16 Structure of Monitor Processor Component 114
4.17 Generation Process of Hardware Monitoring Components . . 115
4.18 Monitoring Pipeline Example 117
4.19 Generation Process of Software Anomaly Detection 121
4.20 Model Instance for the Given Code Example 123

4.21 Model Instance Fix for the Given Code Example 127

5.1 ResNet-18 Structure used to Explain the Early Exit Addition . 133
5.2 Timing Example for ResNet-18 135
5.3 Potential Positions of Early Exits and Anomaly Detection . . . 136
5.4 Possible Attachment Layers for ResNet-18 or ResNeXt-101 . . 137
5.5 Tracing Component Addition to ResNet-18 140
5.6 WordNet Grouping into Hypernyms 144
5.7 Training Process Concept for Added Early Exits 145
5.8 Testing Process for Added Early Exits 147
5.9 Tracing Activations for Anomaly Detection Configuration . . 150
5.10 Dataflow of Anomaly Detection Modules 151
5.11 Naive Approaches for Dimension Matching 153
5.12 Examples of ML-based Anomaly Detection Methods 154
5.13 Examples of Distance-based Anomaly Detection Methods . . 157
5.14 Complete ANN Adaptation Structure for ResNet-18 158
5.15 Example ResNet-18 Runtime Measurements with Early Exits . 160
5.16 Weight Distribution for Runtime Estimation 163
5.17 Checkpoint Structure for Deciding Early Exit Execution . . . 165
5.18 Task Release Loop for Checkpoint Scheduler 169
5.19 Checkpoint Loop for Early Exit Scheduler 170

6.1 Adapted ResNet-18 Turned into Time-Triggered Tasks 180
6.2 Example of Execution Delay and Speed-Up 182
6.3 Options to Model Early Exit Choices with Adapted Schedules 183
6.4 Petri net Deployment of Model Instance 186
6.5 Timed Petri net Example of Model Instance 190
6.6 RocketCore-based Deployment System with four Cores . . . 191
6.7 Platform Deployment Workflow for RocketCore-based Platform 193
6.8 MicroBlaze-V-based Platform with Management Core 195
6.9 MicroBlaze-V-based Platform Deployment 197
6.10 ARM-based Processing Platform in STM32-based System . . 198
6.11 STM32-based Platform Deployment 199
6.12 Abstracted Platform Architecture from ZCU102-Example . . 200

7.1 Runtime Histogram for Complete Networks 205
7.2 Runtime Histogram for Exit Layers of Networks 207
7.3 Evaluation of Runtime Factors for Runtime Estimation 208
7.4 Ablation Results for Time-Triggered Scheduling Optimizer . . 216
7.5 Success Rate for Polyhedron-based Scheduling Approach . . 219
7.6 Runtime Evaluation for Polyhedron-based Scheduling 221

7.7 Simulated Annealing Runtime for Var. Task Amount 227
7.8 Simulated Annealing Runtime for Var. Unit Amount 228
7.9 Simulated Annealing Runtime for Var. Utilization 229
7.10 Simulated Annealing Runtime for Var. Deadline Slack 230
7.11 Simulated Annealing Runtime for Var. Multi-Version Tasks . . 231
7.12 Simulated Annealing Runtime for Var. Precedence Constraints 232
7.13 Cost Function Characteristics of the First Six Configurations . 234
7.14 Cost Function Characteristics of the Second Six Configurations 235
7.15 Relative Entropy for GoogleNet and ResNext-101 Early Exits . 236
7.16 Runtime until Exit Layers for Evaluated Networks 238
7.17 Structure of Added Early Exits 239
7.18 Runtime Overhead of Checkpoint Scheduler 247
7.19 Exit Choice for Utilization Increase and Different Characteristics 250
7.20 Accuracy of Networks for Utilization Increase 252
7.21 Deadline Success Rate of Networks for Utilization Increase . 254

8.1 Picture of Used Case Study Platform 261
8.2 Picture of Used RoboRacer Case Study Platform 267

This page intentionally left blank

Tables

1.1 Description of the Used Example Task Set 10

2.1 Comparison of Hypervisors and Real-Time Operating Systems 38

4.1 Description of the Sets in the Task Model 60
4.2 Deadline and Slack Definition for Periodic Tasks 61
4.3 Model Values for the Example Execution Platform 62
4.4 Model Values for the Shared Resources in the Example 62
4.5 Runtime and Deadlines of the Example Task Set 63
4.6 Version Definitions of the Task Model 65
4.7 Examples for Runtime, Execution Unit and Prec. Constraints 66
4.8 Examples for Access Windows and Bandwidth Bounds 67
4.9 Model Definition of Schedule Describing Values 68
4.10 Additional Model Definition Values 69
4.11 Scheduled Values of the Example Task Set 88
4.12 Model Extensions for Execution Modes 90
4.13 Model Extensions for Mode Change Schedules 92
4.14 Example Task Set Values Changed to Introduce Execution Modes 93
4.15 Penalty and Bonus Terms for Scheduling Algorithms 98
4.16 Model Definition of Platform Architecture Components . . . 109
4.17 Model Definition of Functional Implementation 111
4.18 Model Definition of Information Flow Components 111
4.19 Model Definition of Component Interface 119
4.20 Model Definition of Internal Structure Components 121

5.1 Model Definition for Neural Network Variables 132
5.2 Model Definition of Neural Network Runtime Values 135
5.3 Model Definition for ANN Input Classes 142
5.4 Anomaly Detection and Dimension Matching Compatibility . 157
5.5 Definitions added for the Neural Network Task Scheduler . . 166

6.1 Translation of Model Components between ANN and TTS . . 179
6.2 Translation of ResNet-18 into TTS Model 180
6.3 ResNet-18 Load and Store Operations in Scheduling Model . 181

6.4 Creation of Load and Unload Tasks in the Task Model 194
6.5 Custom OP Code for Execution Cores 196

7.1 Runtime Evaluation for Backbone Network 205
7.2 Runtime Evaluation for Exit Layers of Network 207
7.3 Factor Representation of Percentiles of Runtime Estimation . 209
7.4 Evaluated Weight Distributions for Runtime Estimation . . . 210
7.5 Runtime Estimation Error for Size-Ordered Sample Values . . 211
7.6 Runtime Estimation Error for Time-Ordered Sample Values . 212
7.7 Model Instance used for TTS Ablation Study 214
7.8 Evaluated Hyperparameter Values for Simulated Annealing . 214
7.9 Model Values for Polyhedron-based Success Rate Evaluation . 218
7.10 Model Values for Traced Experiments 223
7.11 Runtime Evaluation for Different Starting Vectors 224
7.12 Model Values for Simulated Annealing Runtime Evaluation . 225
7.13 Factor Representation of Percentiles of Runtime Estimation . 237
7.14 Accuracy and Equivalence of Added Early Exits 240
7.15 Datasets for Training Anomaly Detection Evaluation 241
7.16 Classification Accuracy for Anomaly Detection of ResNeXt-101 242
7.17 Classification Accuracy for Anomaly Detection of GoogleNet 243
7.18 Runtime for Anomaly Detection Modules of ResNeXt-101 . . 244
7.19 Runtime for Anomaly Detection Modules of GoogleNet . . . 245
7.20 Runtime Overhead of Checkpoint Scheduler 247

8.1 RoboRacer Case Study Values of Single-Version Tasks 265
8.2 RoboRacer Case Study Values of Multi-Version Tasks 266

9.1 Comparison of Time-Triggered Multi-Mode Schedulers . . . 273
9.2 Comparison of Dynamic ANN Generators and Schedulers . . 277

This page intentionally left blank

This page intentionally left blank

Glossary and Acronyms

AER **Acquisition Execution Restitution**
The AER model describes a phased execution model, where tasks alternate between acquisition, execution and restitution phases to prevent simultaneous memory access from different tasks.

Adam **Adaptive Moment Estimation**
An optimization algorithm that computes adaptive learning rates for each parameter, combining the advantages of the AdaGrad and RMSProp stochastic gradient descent methods.

ADAS **Advanced Driver Assistance Systems**
E/E systems inside automotive vehicles employed to assist the driver in situations like emergency breaking, lane keeping or accelerating.

AXI **Advanced eXtensible Interface**
A high-performance, high-frequency bus protocol used in FPGAs and system-on-chip designs for connecting different components and facilitating data transfer.

AD **Anomaly Detection**
Process of identifying unusual patterns or outliers in data that do not conform to expected behavior.

ANN **Anytime Neural Network**
Neural network model, designed to produce continously better results with increasing computation time, where execution can be terminated at any time and still return a valid (though potentially lower-quality) result.

API **Application Programming Interface**
Interface for developing software applications. Defines function calls and data structures for communication between different software components.

AI **Artificial Intelligence**
Often used as a buzz word and interchangeably with machine learning. Generalized term for algorithms, which are able to perform tasks, which are commonly associated with human intelligence.

AMP **Asymmetric Multi-Processing**
Multiprocessing scheme in embedded systems, where the multicore processors are assumed to be heterogeneous and have different capabilities.

ASIL **Automotive Safety Integrity Level**
Risk classification system introduced in ISO 26262, where ASIL-A describes tasks the lowest risk and ASIL-D defines the highest risks.

BNN **Bayesian Neural Network**
Neural network model that incorporates Bayesian inference to estimate uncertainty in predictions, allowing for probabilistic reasoning and improved robustness.

CECAS **Central Car Server**
High performance HESoC in a centralized automotive E/E architecture used to executed complex ADAS and other automotive tasks.

CPU **Central Processing Unit**
Primary execution unit of a computer system responsible for executing instructions and performing calculations. In contrast to specialized processing units like GPUs, or accelerators, CPUs are designed for general-purpose computing.

Chiplet
Modular semiconductor components, combinable in one package to create a customizable chip system. They allow designers to mix and match specialized functions and enhance flexibility compared to a monolithic chip.

COTS **Commercial Off-The-Shelf**
Existing hardware or software components readily available to purchase and use in various applications without customization.

CAN **Controller Area Network**
A vehicle bus standard designed to allow microcontrollers and devices to communicate with each other without a host computer.

CNN **Convolutional Neural Network**
Type of artificial neural network, which is designed to process data with convolutional operations. Convolutional layers are used to extract features from the input data. Learned parameters include the convolutional filters.

DSR **Deadline Success Rate**
The percentage of tasks that meet their deadlines within a given time frame.

DMS **Deadline-Monotonic Scheduling**
Fixed-priority scheduling algorithm where tasks with shorter deadlines are assigned higher priorities.

DMA **Direct Memory Access**
A feature that allows hardware devices to directly access memory without involving an intermediate interface or virtual adresses.

DAG **Directed Acyclic Graph**
Directed graph with no cycles, used in the scheduling domain to represent dependencies or precedence constraints between tasks.

DCU **Domain Control Units**
Electronic control module inside automotive vehicle controlling all functions inside a vehicular domain.

DDR **Double Data Rate**
Memory technology, which transfers data on both the rising and falling edges of the clock signal. Often used as a synonym for memory.

EDF **Earliest Deadline First**
Scheduler, which executes the task with the earliest deadline first.

EE **Early Exit**
Additional output branches added to neural networks to allow for intermediate results and thus reduced execution time at the cost of reduced accuracy. Used in anytime neural networks to enable anytime behavior.

EENN **Early Exit Neural Network**
Neural network model, which is designed to produce intermediate outputs at various stages of the network, allowing for early termination of computation and providing a trade-off between accuracy and inference time. See also ANN.

Electric / Electronic Architecture
The architectural description of electronics, hardware, communication interfaces and software applications within an automotive vehicle [177].

E/E **Electrical/Electronic**
Electrical/electronic systems. Usually used in the context of automotive E/E architecture.

ECU **Electronic Control Unit**
Electronic control module inside automotive vehicle, controlling one or more electronic functions.

eHPC **Embedded High Performance Computing**
Embedded computation of complex functionality like ADAS and other automotive tasks. Executed on CECAS.

FPGA **Field-Programmable Gate Array**
Integrated circuit which can be programmed after deployment. Used for implementing digital logic functions and custom hardware.

FCFS **First-Come, First-Served**
Scheduling algorithm that executes tasks in the order they arrive, without preemption.

Functional Domain
Functional cluster of multiple ECUs, handling related functions.

GNB **Gaussian Naive Bayes**
Classifier based on applying Bayes' theorem with naive independence assumptions between the features [143].

GPP **General Purpose Processor**
Processor designed for general-purpose use. Able to execute general applications with a broad instruction set.

GEDF **Global Earliest Deadline First**
Scheduler, which prioritizes tasks based on their deadlines across multiple processors, with the earliest deadline executed first.

GPU **Graphics Processing Unit**
Specialized processing unit designed for parallel processing and rendering of graphics. In contrast to CPUs, GPUs are optimized for handling parallel execution of identical simple instructions, making them suitable for tasks like machine learning and scientific computing.

GCD **Greatest Common Divisor**
The greatest common divisor of two or more integers is the largest positive integer that divides each of the integers without leaving a remainder.

HAL **Hardware Abstraction Layer**
Software layer which abstracts the underlying hardware and provides a unified API towards executed software, provides execution guarantees and ensures defined requirements.

HEFT **Heterogeneous Earliest Finish Time**
Scheduling algorithm for heterogeneous systems, which assigns tasks to processors based on the earliest finish time, considering both computation and communication costs.

HESoC **Heterogeneous System-on-Chip**
Multi Processor System on Chip, which incorporates at least two execution units of different architectures.

Hypervisor
Software managing physical hardware by abstracting access to shared resources like memory or CPU.

IMU **Inertial Measurement Unit**
Unit, which measures and provides specific force, angular rate, and similar properties of the body, using accelerometers, gyroscopes, and similar senors.

I/O **Input/Output**
Input and Output communication between a system and the outside world, which can include communication interfaces as well as storage devices.

ISA **Instruction Set Architecture**
Collection of instructions used by a processor to execute programs. Defines the supported operations, data types, registers and memory architecture of a processor. Today, x86, ARM and RISC-V are common ISAs.

ILP **Integer Linear Programming**
Mathematical optimization technique where the objective function and constraints are linear, and the decision variables are restricted to integer values.

LiDAR **Light Detection and Ranging**
Sensing method, which light dots emitted from a pulsed laser to measure distances to objects. Used in the context of autonomous driving for environment perception.

ML **Machine Learning**
Subset of artificial intelligence focused on algorithms, which are trained on a dataset and make predictions based on learned patterns.

MC **Mixed Criticality**
System design approach that considers tasks with varying levels of criticality in the same system.

Multi-Core System
Systems in which tasks with high criticality and low criticality are executed on the same execution unit.

MLP **Multi-Layer Perceptron**
Artificial neural network, consisting of multiple layers of neurons. Each layer is fully connected to the next layer.

MAC **Multiply-Accumulate**
Fundamental operation in digital signal processing and neural network computations, where two numbers are multiplied and then added to an accumulator.

MPSoC **Multi-Processor System-on-Chip**
Type of System-on-Chip design that integrates multiple processing units on a single chip, enabling parallel processing and improved performance.

NAS **Neural Architecture Search**
Automated process of designing neural network architectures, optimizing for performance metrics such as accuracy, latency, or energy efficiency.

NSGA **Non-dominated Sorting Genetic Algorithm**
A multi-objective optimization algorithm used for solving complex optimization problems.

OS **Operating System**
Software that manages computer hardware and software resources and provides common services for computer programs. In some cases also used to provide real-time capabilities as real-time operating systems in the context of embedded systems.

ODD **Operational Design Domain**
The specific conditions under which a given driving automation system or feature is designed to function, including environmental, geographical, and time-of-day constraints. In the context of AI systems, input samples outside the ODD are considered anomalous or out-of-distribution.

OEM **Original Equipment Manufacturer**
Company that designs, develops, and produces components or the final product under its own brand. Car brands are classified as OEMs in the automotive domain.

OOD **Out-Of-Distribution**
Data samples which differ significantly from the training data distribution, used in the context of anomaly detection and robustness evaluation in machine learning models.

OTA **Over-The-Air**
A remote access of software (updates) over wireless channels in contrast to a wired connection.

PREM **Predictable Execution Model**
Execution model, which divides tasks into predictable and non-predictable phases [25]. The predictable phases are executed with timing guarantees, while the non-predictable phases are executed without timing guarantees.

PCA **Principal Component Analysis**
Statistical procedure using an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

PS **Processing System**
In contrast to the PL, the PS describes the fixed hardware components of a system, like the CPU, memory and peripherals.

PL **Programmable Logic**
Digital configurable circuits. Includes devices like FPGAs, which allow for the implementation of custom hardware functionality or coarsely-grained reconfigurable arrays.

RMS **Rate-Monotonic Scheduling**
Fixed-priority scheduling algorithm where tasks with shorter periods are assigned higher priorities.

RTOS **Real-Time Operating System**
An operating system designed to execute tasks within strict timing constraints, ensuring timely responses for real-time software tasks.

ReLU **Rectified Linear Unit**
Activation function used in neural networks, which outputs the input directly if it is positive, otherwise it outputs zero.

RISC **Reduced Instruction Set Computer**
The Reduced Instruction Set Computer is an open-source and royalty-free instruction set architecture to provide a standardized CPU design for research and commercial use.

RoI **Region of Interest**
Specific area within an image, which is of particular interest for analysis or processing, often used in computer vision and image processing tasks.

RTE **Runtime Environment**
Software environment, which provides necessary services and interfaces for executing applications as well as managing resources.

SPI **Serial Peripheral Interface**
Synchronous serial communication protocol used for short-distance communication, primarily in embedded systems.

SoA **Service-Oriented Architecture**
Design approach in which software components are designed as modules or services, which are combined to create a larger application as the combination of those modules or services.

SJN **Shortest Job Next**
Scheduling algorithm that selects the task with the shortest execution time to execute next, aiming to minimize average waiting time.

Simulated Annealing
Simulated Annealing describes an optimization algorithm, which is models the annealing process of cooling crystals. The solution space is explored further by allowing the acceptance of worse solutions and thus the escape from local optima [118].

Softcore Processor
A microprocessor core that is implemented using programmable logic, such as in a FPGA, rather than being fabricated as a fixed hardware component.

SDV **Software Defined Vehicle**
An automotive vehicle, whose characteristics are primarily defined and controlled via its software, rather than its hardware.

SGD **Stochastic Gradient Descent**
An iterative training method in machine learning for optimizing an objective function. The function includes smoothness properties and is used for large-scale training problems.

SVM **Support Vector Machine**
A supervised learning model used for classification and regression analysis by calculating a hyperplane which optimally separates different classes in the feature space [142].

SMP **Symmetric Multi-Processing**
Multiprocessing scheme in embedded systems, where the multicore processors are assumed to be homogeneous and have the same capabilities.

SOC **System-on-Chip**
Integrated circuit, containing all components of a digital system on a single chip. This includes e.g. processing units, memory, communication interfaces, peripherals and other components.

TSN **Time-Sensitive Networking**
IEEE standards, defining time-sensitive communication in Ethernet networks.

TTS **Time-Triggered Scheduling**
A scheduling approach where tasks are executed at predetermined times for predictability and timing guarantees.

UMAP **Uniform Manifold Approximation and Projection**
A dimension reduction technique for visualizing high-dimensional data by preserving the global structure and local relationships within the data.

USP **Unique Selling Point**
The defining feature or characteristic of a product or service that distinguishes that product or service from competitors.

UART **Universal Asynchronous Receiver Transmitter**
A hardware communication protocol used for serial communication between devices, often used for debugging and data transfer in embedded systems.

VAE **Variational Autoencoder**
A generative model that learns to encode input data into a lower-dimensional latent space and decode it back to the original space, used for tasks such as anomaly detection by measuring reconstruction error [140].

WCET **Worst-Case Execution Time**
The maximum time a task or program can take to execute on a specific hardware platform, used for real-time system analysis and scheduling.

XMMU **Xilinx Memory Management Unit**
Hardware component in Xilinx MPSoCs, which provides memory protection and isolation for memory regions.

XMPU **Xilinx Memory Protection Unit**
Hardware component in Xilinx MPSoCs, which provides access protection and isolation for peripherals.