Complex Scheduling Models and Analyses for Property-Based Real-Time Embedded Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Niklas Ueter

Dortmund

June 2023

Tag der mündlichen Prüfung:	13. September 2023
Dekan / Dekanin:	Prof. Dr. Gernot Fink
Gutachter / Gutachterinnen:	Prof. Dr. Jian-Jia Chen
	Prof. Dr. Jing Li (New Jersey Institute of Technology)

ABSTRACT

Real-time systems are most prevalent in cyber-physical systems (CPS), which refer to *cyber subsystems* which are integrated into a physical system such as automotive, avionic, or robotic systems. In these systems, relevant aspects of the physical environment are measured and discretized by sensors, which must subsequently be processed by the *cyber subsystem* during the physical system's state evolution. To assure quality-of-service requirements of the cyber-physical system, temporal constraints – among other reliability requirements – are imposed on the *execution behaviour* of the *cyber subsystem*. Environmental and thus sensory inputs to the system as well as the *execution environment* are categorically non-deterministic, which renders exhaustive testing to validate the temporal behaviour unattainable. As a consequence, a hard real-time system must be provisioned under worst-case assumptions. Modern *cyber subsystems*, i.e., multicore architectures and parallel applications pose a significant challenge to the worst-case centric real-time system verification and design efforts. The involved model and parameter uncertainty contest the fidelity of formal real-time analyses, which are mostly based on exact model assumptions. Motivated by that observation, the abbreviated research hypothesis of this dissertation is that; either the increased parameter and model uncertainty must be accepted and considered in the scheduling algorithm design and schedulability analyses or the predictability must be increased by means of hardware or algorithmic restrictions.

The dissertation is structured into four parts as follows. In an attempt to improve predictability in worst-case centric analyses, the exploration of timing predictable protocols are examined for parallel task scheduling on multiprocessors and network-on-chip arbitration. Roughly speaking, the research approach is to impose additional constraints to the problem, which are theoretically and practically beneficial for worst-case centric analyses, conceding to potentially degraded average case performance. A novel scheduling algorithm, called *stationary rigid gang scheduling*, for gang tasks on multiprocessors is proposed. In so-called stationary rigid gang scheduling, all threads of a task are grouped into a gang, which are co-scheduled simultaneously on a set of predetermined processors. The stationary rigid gang schedulability test problem is formally reduced to the dynamic self-suspension schedulability problem for uniprocessor systems. Numerical evaluations are presented, showing improvements over the state-of-the-art in terms of schedulable task sets compared to the state-of-the-art non-stationary gang scheduling algorithms. Moreover, worst-case performance bounds with respect to an optimal scheduling algorithm are proven.

With regards to fixed-priority wormhole-switched network-on-chips, many documented flaws and counter-examples have been reported, disproving many proposed response-time analyses. The proposed analyses are based on the assumption of an equivalence of the *uniprocessor execution model* and the *pipelined transmission model* in network-on-chips, which was never formally proven. Staring from a discussion and illustration of the mismatch of both execution models, a

more restrictive family of transmission protocols called *simultaneous progression switching protocols* is proposed, which are provably compatible with the uniprocessor execution model. Above that, further predictability enhancing properties such as simpler router design, and arbitrary non-minimal routing capabilities come along with this protocol. An implementation, response-time analysis, and a numerical evaluation of the protocol overheads is provided.

In the second part, hierarchical scheduling for parallel *Directed Acyclic Graph* (DAG) tasks under parameter uncertainty is studied. The hierarchical scheduling approach achieves temporal- and spatial isolation, which allows to prevent uncertainty induced timing violations from cascading into the whole system. We firstly study control flow uncertainty using probabilistic branching decisions, under the probabilistic conditional DAG task model. Under this model, we study k-consecutive deadline miss constraints and bounded tardiness. That is, if a job can not finish within its allocated execution time budget, it is allowed to consume the budget of the subsequent job (up to a bounded threshold), or is aborted otherwise. Furthermore, we improve the resource utilization of the hierarchical DAG scheduling, proposing and analyzing a subtask-level prioritization policy, which allows to substantially improve Graham's makespan bound and the resource utilization. Both approaches are extensively compared against the state of the art.

In the third part, fault-tolerance as a supplementary reliability aspect of real-time systems is examined, in spite of dynamic external causes of fault. To assure that an acceptable quality-of-service (QoS), i.e., fault-tolerance can be achieved, upper-bounds on consecutive erroneous job executions, and guaranteed m error-free executions out of k consecutive job executions are studied. Using various job variants, which trade off increased execution time demand with increased error protection, a state-based policy selection strategy is proposed. The derived policy minimizes the expected system utilization and guarantees that all reachable states comply with the QoS- and hard real-time constraints of the task system. The proposed approaches demonstrate significantly decreased system utilization compared to the state of the art in the evaluations.

In the fourth part, the temporal misalignment of sensor data in sensor fusion applications in cyber-physical systems is examined. Starting from a formal definition of the problem as a DAG, we propose a modular analysis based on minimal properties to obtain an upper-bound for the maximal sensor data time-stamp difference. That is, the maximal time-stamp difference of any two sensor data (of different sensors), which are used in a common sensor fusion task is bounded from above.

PUBLICATIONS

The majority of the ideas and findings presented in this dissertation have been published in the following peer-reviewed articles that appeared in international journals and proceedings of international conferences:

[GUC+21] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, Georg von der Brüggen, Junjie Shi, and Jian-Jia Chen. “End-To-End Processing Chain Analysis.” In: *RTSS Industrial Challenge*. 2021.

[SUC+23] Junjie Shi, Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. “Average Task Execution Time Minimization under (m,k) Soft-Error Constraint.” In: *RTAS*. IEEE, 2023, p. 12.

[UCB+20] Niklas Ueter, Jian-Jia Chen, Georg von der Brüggen, Vanchinathan Venkataramani, and Tulika Mitra. “Simultaneous Progression Switching Protocols for Timing Predictable Real-Time Network-on-Chips.” In: *RTCSA*. IEEE, 2020, pp. 1–10.

[UGB+21] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. “Hard Real-Time Stationary GANG-Scheduling.” In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 10:1–10:19.

[UGB+23] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. “Parallel Path Progression DAG Scheduling.” In: *IEEE Transactions on Computers* (2023), pp. 1–15. DOI: 10.1109/TC.2023.3280137.

[UGC21] Niklas Ueter, Mario Günzel, and Jian-Jia Chen. “Response-Time Analysis and Optimization for Probabilistic Conditional Parallel DAG Tasks.” In: *RTSS*. IEEE, 2021, pp. 380–392.

As part of my research, I also contributed to the following peer-reviewed articles that appeared in international journals and proceedings of international conferences but are not part of this dissertation:

[CBS+18] Jian-Jia Chen, Georg von der Brüggen, Junjie Shi, and Niklas Ueter. “Dependency Graph Approach for Multiprocessor Real-Time Synchronization.” In: *RTSS*. IEEE Computer Society, 2018, pp. 434–446.

[CBU18] Jian-Jia Chen, Georg von der Brüggen, and Niklas Ueter. “Push Forward: Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems.” In: *ECRTS*. Vol. 106. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 8:1–8:24.

[CHB+21] Jian-Jia Chen, Wen-Hung Huang, Georg von der Brüggen, Kuan-Hsun Chen, and Niklas Ueter. “On the Formalism and Properties of Timing Analyses in Real-Time Embedded Systems.” In: *A Journey of Embedded and Cyber-Physical Systems*. Springer, 2021, pp. 37–55.

[CSB+22] Jian-Jia Chen, Junjie Shi, Georg von der Brüggen, and Niklas Ueter. "Scheduling of Real-Time Tasks With Multiple Critical Sections in Multiprocessor Systems." In: *IEEE Trans. Computers* 71.1 (2022), pp. 146–160.

[CUB+19] Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. "Efficient Computation of Deadline-Miss Probability and Potential Pitfalls." In: *DATE*. IEEE, 2019, pp. 896–901.

[EUC22] Robin Edmaier, Niklas Ueter, and Jian-Jia Chen. "Segment-Level FP-Scheduling in FreeRTOS." In: *RTCSA*. IEEE, 2022, pp. 186–194.

[GUC21] Mario Günzel, Niklas Ueter, and Jian-Jia Chen. "Suspension-Aware Fixed-Priority Schedulability Test with Arbitrary Deadlines and Arrival Curves." In: *RTSS*. IEEE, 2021, pp. 418–430.

[GUC+23] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. "Timing Analysis of Cause-Effect Chains with Heterogeneous Communication Mechanisms." In: *RTNS*. ACM, 2023, pp. 224–234.

[LSU+22] Ching-Chi Lin, Junjie Shi, Niklas Ueter, Mario Günzel, Jan Reineke, and Jian-Jia Chen. "Type-aware Federated Scheduling for Typed DAG Tasks on Heterogeneous Multicore Platforms." In: *IEEE Transactions on Computers* (2022), pp. 1–14. DOI: 10.1109/TC.2022.3202748.

[SUB+21] Junjie Shi, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. "Graph-Based Optimizations for Multiprocessor Nested Resource Sharing." In: *RTCSA*. IEEE, 2021, pp. 129–138.

[SUB+19b] Junjie Shi, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. "Partitioned Scheduling for Dependency Graphs in Multiprocessor Real-Time Systems." In: *RTCSA*. IEEE, 2019, pp. 1–12.

[TGU+22] Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. "End-To-End Timing Analysis in ROS2." In: *RTSS*. IEEE, 2022, pp. 53–65.

[UCC20] Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. "Project-Based CPS Education: A Case Study of an Autonomous Driving Student Project." In: *IEEE Des. Test* 37.6 (2020), pp. 39–46.

Dedicated to my family.

ACKNOWLEDGMENTS

Seemingly, acknowledgments are merely a page and part of the traditional formalism to hand in the dissertation, but I am truly indebted and sincerely grateful to the many people that have supported, developed, and challenged me during my time as a PhD student and beyond. I really mean it.

First, I would like to thank my PhD adviser Jian-Jia Chen for encouraging me to pursue research and to have always been interested to discuss ideas and problems. In particular, I thank him for all the provided freedom in my research endeavours as well as the provided guidance, patience, challenge and support when needed. Above that, I am grateful for him supporting my Dagstuhl and conference participations as well as introducing me to people from his academic network that helped my research and personal development one way or another. I also would like to thank my second reviewer Jing Li for her support and advice starting with my first conference preparation and presentation continuing to this dissertation. Furthermore, I want to thank Mario Botsch and Peter Ulbrich for being part of my committee, and Frank Weichert for being my PhD mentor.

I want to thank the members of the real-time systems group, namely Georg von der Brüggen, Ching-Chi Lin, Christian Hakert, Nils Hölscher, Daniel Kuhse, Vahidreza Moghaddas, Junjie Shi, Tristan Seidl, Noura Sleibi, Harun Teper, Zahra Valipour, Mikail Yayla, Claudia Graute, and Lars Dröge for making my time at TU Dortmund both interesting and enjoyable. In particular, I want to thank our secretary Claudia Graute for helping and taking care of all the bureaucracy and chaos revolving around research and Christian Hakert for providing the coffee machine. I would also thank Georg von der Brüggen for providing discussions on numerous occasions, as well as for his *unsolicited writing advice*.

I would like to express my thanks to Marco Dürr for sharing the office, his selfless help at any time, and his special persona resulting in the uncountable many memorable and entertaining times spent together. I would like to express my thanks to Kuan-Hsun Chen and Yen Jen Liu for the many interesting and helpful discussions and enjoyable time spent together. I would like to express my thanks to Junji Shi for sharing many hours on writing papers and programming together no matter how close the deadlines and to always provide enthusiasm and positive energy. I want to express my thanks to Mario Günzel for becoming a friend and the interesting and helpful discussions about mathematics and research and Nils Hölscher for the needed laughter at work.

Additonally, I would like to express my thanks and appreciation to all my external co-authors Jing Li, Kunal Agrawal, Matthias Freier, Tulika Mitra, Jan Reinecke, and Vanchinathan Venkataramani.

I would like to thank my parents as well as my brother and my sister for all the support and being family. I especially want to thank my wife Larissa for enduring the hardships of research, and for being in my life.

My work has been supported by the project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). Multiple of my co-authors have been (partially) funded by Deutsche Forschungsgemeinschaft (DFG) in Sus-Aware (Project No. 398602212) and as part of the Collaborative Research Center SFB876, project A1 and A3, and by the German Federal Ministry of Education and Research (BMBF) in the course of the 6*GEM* research hub.

CONTENTS

1 INTRODUCTION 1
1.1 Cyber-Physical Real-Time Systems 1
1.1.1 Timing Constraints & Model-Building 3
1.1.2 Real-Time Scheduling & Schedulability 5
1.2 Contribution of this Dissertation 6
1.3 Author's Contribution to this Dissertation 9
2 REAL-TIME SYSTEM CONCEPTS 13
2.1 Multicore & Real-Time Systems 14
2.1.1 Memory & Cache 16
2.1.2 Interconnects 19
2.2 Timing Analysis 27
2.2.1 Deterministic Timing Analysis 28
2.2.2 Probabilistic Timing Analysis 31
2.3 Real-Time Scheduling 32
2.3.1 Scheduler in the RTOS 33
2.3.2 Task & System Models 35
2.3.3 Scheduling Algorithms 39
2.3.4 Schedulability Analysis 44
2.3.5 Performance of Scheduling Algorithms 48
3 TIMING PREDICTABLE PROTOCOLS 53
3.1 Motivation 54
3.2 Related Work 56
3.3 Stationary Rigid Gang Scheduling 57
3.3.1 System Model & Problem Description 58
3.3.2 Stationary Rigid Gang Scheduling Algorithm 59
3.3.3 Stationary Rigid Gang Scheduling Analysis 60
3.3.4 Stationary Rigid Gang Assignment Algorithm 67
3.3.5 Evaluation 82
3.4 Simultaneous Progression Switching Protocols 86
3.4.1 System Model & Problem Definition 87
3.4.2 Simultaneous Progression Property 89
3.4.3 Protocol Implementation 92
3.4.4 Arbitration Implementation 93
3.4.5 Response-Time & Schedulability Analysis 99
3.4.6 Evaluation 100
3.5 Conclusion 102
4 HIERARCHICAL PARALLEL DAG SCHEDULING 105
4.1 Motivation 105
4.2 Related Work 108
4.3 Hierarchical DAG Scheduling & DAG Task Model 110
4.4 Probabilistic Conditional-DAG Scheduling 114
4.4.1 Task and Problem Model 114
4.4.2 Probabilistic Deadline Miss Description 118
4.4.3 Objective, Discussion of Guarantees & Limitations 120
4.4.4 Hierarchical Scheduling Analysis 121

4.4.5 Reservation Analysis and Optimization 127
4.4.6 Parameter Space Exploration 131
4.5 Parallel Path Progression Scheduling 136
4.5.1 DAG Vertex Coverage 139
4.5.2 Weighted Maximum Coverage 141
4.5.3 Approximation Algorithm 141
4.5.4 Hierarchical Scheduling Extension 145
4.5.5 Evaluation 152
4.5.6 Reclamation & Suspension 157
4.6 Conclusion 164
5 REGULATOR-BASED ADAPTIVITY 165
5.1 Motivation 165
5.2 Related Work 168
5.3 System Model 169
5.3.1 Task Model 169
5.3.2 Fault and Error Model 170
5.3.3 Schedulability and Scheduling 171
5.4 Automata-based Regulator 172
5.4.1 Consecutive-Error Constraints 174
5.4.2 (m,k)-Constraints 178
5.4.3 States Reduction and Minimal Automata 179
5.4.4 Minimization of Expected Execution Time 181
5.5 Reinforcement Learning Based Approach 185
5.6 Evaluation 190
5.7 Conclusion 194
6 MAXIMAL SENSOR DATA TIME-STAMP DIFFERENCE 195
6.1 Motivation 195
6.2 Related Work 197
6.3 Application Model 198
6.4 Maximal Sensor Data Time-Stamp Difference Analysis 201
6.4.1 Minimal Sensor Data Propagation Latency 203
6.4.2 Maximal Sensor Data Propagation Latency 204
6.4.3 Maximal Sensor Data Time-Stamp Difference 208
6.5 Discussion & Extensions 210
6.6 Conclusion 211
7 CONCLUSIONS AND OUTLOOK 213
7.1 Summary of the Contributions 213
7.1.1 Timing Predictable Protocols 213
7.1.2 Hierarchical Parallel DAG Scheduling 214
7.1.3 Regulator-Based Adaptivity 215
7.1.4 Maximal Sensor Data Time-Stamp Difference 215
7.2 Examination of the Dissertation Hypothesis 216
7.3 Future Work 217
7.4 Final Remarks and Outlook 218

BIBLIOGRAPHY 219
INDEX 246
LIST OF FIGURES 251
LIST OF TABLES 257

1 INTRODUCTION

In generic terms, real-time systems are defined as computing systems, in which the functional correctness does not only depend on the correctness of the computed results, but also on the time at which the results are produced [BLA+05]. This means, that a late result may be useless or even cause catastrophic system malfunction. Based on the severity of the consequence, caused by the late results, real-time systems are broadly classified into hard, firm, and soft real-time systems [But11]. A *hard real-time system* is one, such that any timing violation causes catastrophic consequences for the system. In *firm real-time systems*, timing constraints may be violated, resulting in useless results but non-hazardous consequences for the system and environment. Lastly, in *soft real-time systems*, results which are produced late with respect to the timing constraints, still provide some utility for the system at the cost of degraded performance. In this dissertation, all three classes of real-time systems are subject to study.

hard real-time system

firm real-time systems

soft real-time systems

The remainder of this introductory chapter is organized into the following sections. An overview and motivation for the relevance and application of real-time systems theory, is given in Section 1.1 *Cyber-Physical Real-Time Systems*. Subsequently, in Section 1.2 *Contribution of this Dissertation*, the research hypothesis and the scientific contributions of this dissertation are stated and elaborated. Lastly, the author's contributions to the presented research in this dissertation, are listed in Section 1.3 *Author's Contribution to this Dissertation*.

1.1 CYBER-PHYSICAL REAL-TIME SYSTEMS

Real-time systems are most commonly found in the context of cyber-physical systems (CPS), which refer to cyber subsystems that are integrated into a physical system

Real-time systems are most commonly found in the context of cyber-physical systems (CPS), which refer to cyber subsystems that are integrated into a physical system, such as automotive systems, avionic systems, chemical plants, medical systems, robotic systems and many others. The overall cyber-physical system is modeled with a specific model of time in mind, e.g., the newtonian continuous time domain may be assumed when the physical system's behaviour and dynamics are described and designed with differential equations; or the uniform discrete-time domain is assumed when using difference equations to describe a *discretized* system.

At the interface of the physical system and the cyber subsystem, relevant aspects of the physical environment are measured and discretized by sensors. The resulting data stream is relayed to the cyber subsystem for processing. After the processing in the computing subsystem has finished, the produced results are relayed to the physical subsystems, such as for instance actuators. In consequence, the response of the cyber subsystem to external sensor data must occur during the physical system's state evolution. Therefore, the response-time of the cyber

the response of the cyber subsystem to external sensor data must occur during the physical system's state evolution

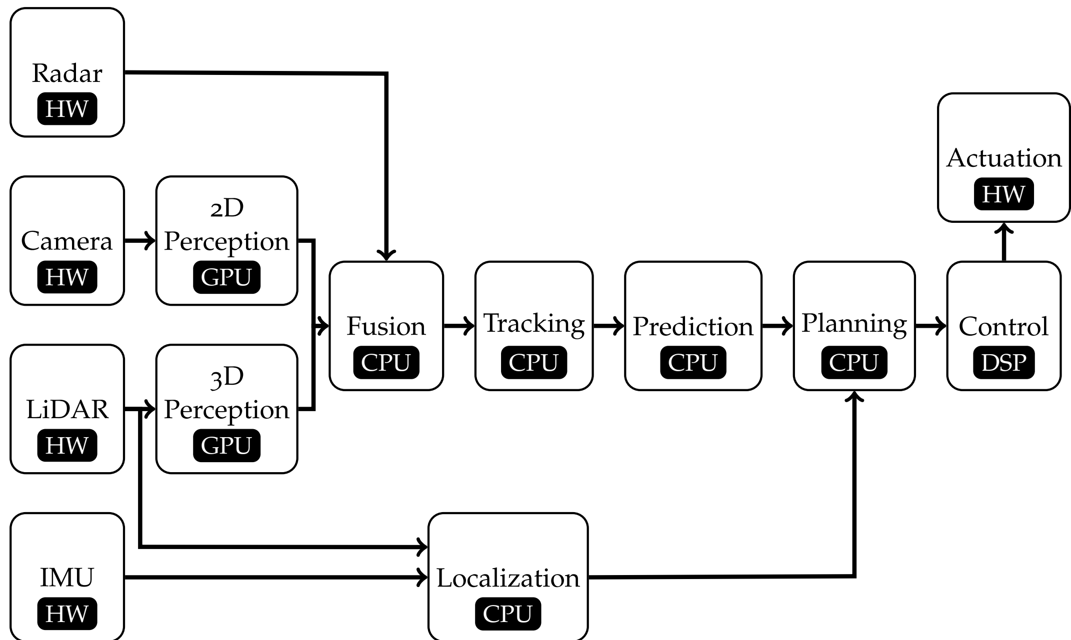

Figure 1.1: An exemplary realistic real-time system architecture, which is proposed by the company *Perceptin* in the RTSS 2021 Industry challenge, to implement an autonomous driving system. The vertices in the processing graph denote functional modules such as sensor data preprocessing, perception, tracking, trajectory planning, and control. The directed edges in the processing graph denote the data dependencies between the modules, e.g., the data produced by the *localization* module is used by the *planning* module in its computation.

subsystem, from sensing to actuation, must firstly be matched with the *dynamics* of the underlying physical system. Secondly, the timing of sensing, computation, and actuation must be matched with the timing model, which is used for the design and verification of the physical system's behaviour.

single-input single-output (SISO) system

quasi-continuous

system modes

multiple-input multiple-output (MIMO) systems

For instance, in the most simplistic setting, a quasi-continuous single-input single-output (SISO) system computes an output based on a sensor sample, arriving at time a, and writing the computed output at time f. The *quasi-continuous* assumption, which was used to specify the algorithms to eventually actuate the physical system, requires that the response-time $f - a$ is a lot smaller than the fastest *system mode*. In that timing model, the timing jitter of sensing and actuation is negligible for the fidelity of the presumed continuous timing model, which is presumed in the control- or filter algorithm design. More realistic multiple-input multiple-output (MIMO) systems, consist of multiple sensors and multiple actuators, which operate at different rates, and are composed of multiple precedence constrained and communicating tasks. Those tasks, are to be mapped and executed on heterogeneous architectures as illustrated in Figure 1.1. To add to that complexity, multiple timing constraints are imposed by the overall system design. For instance, the maximal time-stamp difference of the *Camera* and *LiDAR* sensor samples – that are propagated through the *2D-perception*, *3D-perception* vertex, and joined at the *fusion vertex* – are constrained by the system design specifications to guarantee sufficient quality-of-service (QoS). Other timing constraints are end-to-end response-time bounds, e.g., the maximal time taken to compute a trajectory in the *planning* vertex, once a pedestrian was detected

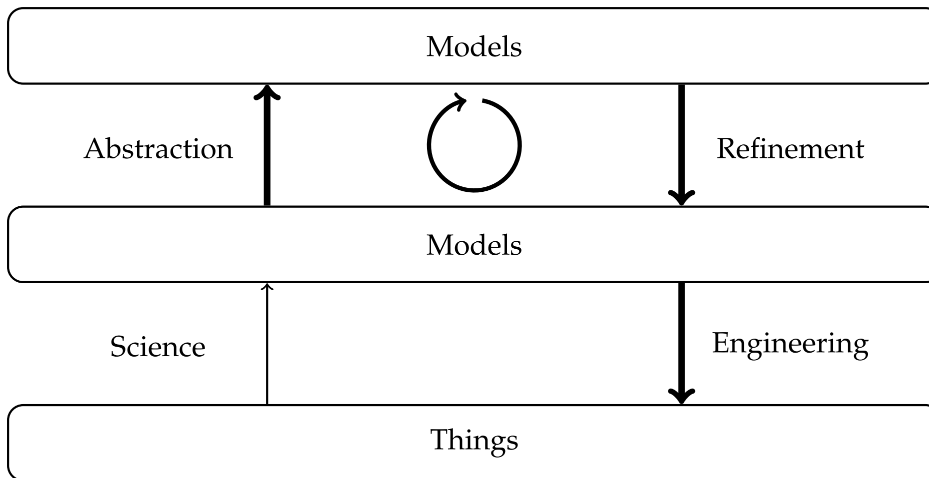

Figure 1.2: Model hierarchy redrawn from Edward Lee: Plato and the Nerd [Lee17]. The emphasized arrows denote the focus of the modeling used in this dissertation.

in the the *perception* vertex, must be upper-bounded according to the system specifications.

It is not possible to evaluate all imposed timing constraints experimentally for achieving predictability in hard real-time systems. This is due to the non-deterministic sensory and environmental input to the system, as well as the non-deterministic execution environment, which can not all be tested exhaustively. Consequently, a hard real-time system must be provisioned under worst-case assumptions, which requires design methodologies, analyses, and operating system mechanisms, to allow for safe and efficient real-time system design and operation. In firm or soft real-time systems however, experimental evaluations and measurements of, e.g., computation times and estimation of timing characteristics of the hardware architecture are possible and commonly used in industry [ANN+22; DC19]. In that case, rare events must be taken into consideration and handled by additional monitoring and provisioning efforts to limit system-wide malfunction.

1.1.1 TIMING CONSTRAINTS & MODEL-BUILDING

> *"A model is any description of a system that is not the thing-in-itself"*
> *Edward Lee on the basis of Immanuel Kant*

In order to formally describe and verify the timing behaviour of computing cyber subsystems as described in the previous Section 1.1, a formal model which is amenable to rigorous analysis must be devised. The problem of model building and interpretation in the context of cyber-physical systems is extensively studied by Lee, such as in [Lee19b; Lee19a; Lee18; LS18]. An illustration of his proposed model hierarchy is shown in Figure 1.2. Lee emphasizes a distinction between different notions of models used in science and engineering. That is, a model in science primarily aims at explaining and analyzing observations made of *the thing* as precisely as possible. In engineering, the objective is the synthesis of systems and thus an engineering model also specifies the intended behaviour of *the thing* to be synthesized. This necessitates the system to conform to the model

scientific model

engineering model

used for the synthesis, as otherwise, the specified and intended behaviour of the system can not be presumed and ascertained.

In this dissertation, we emphasize the engineering model and the related abstraction and refinement cycles, illustrated by the three emphasized arrows in Figure 1.2. Hence, the focus is firstly on the *abstraction* and *refinement* cycle, to find models, which are amenable to rigorous analysis, and secondly can be monitored and enforced in the system to ensure model fidelity. The model refinement effort should result in an abstraction, which is *specific* enough to precisely describe a system, such that an analysis is not over-pessimistic. On the other hand, a certain genericness of the model allows to model and analyze systems with similar characteristics, and thus allows modularity.

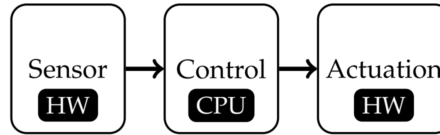

Figure 1.3: An exemplary periodically activated single-input single-output real-time system architecture. The periodically generated sensor sample is processed by the control vertex, which is activated upon new sensor data, and writes the result to the actuator when the execution is finished.

In the beginning of cyber-physical system designs, a recurrent execution model, which was motivated by the simple example shown in Figure 1.3, was assumed. In the example, a sensor value is read repeatedly with a certain delay between two readings. The determined value is processed and the result may lead to an actuation, which changes the system's current state. In pursuit of the corresponding formal response-time analyses, the sequential *periodic task model* and the *sporadic task model*, where each task $\tau_i = (C_i, D_i, T_i)$ is characterized by its *minimum inter-arrival time* (or period, respectively) T_i, its relative deadline D_i, and its worst-case execution time C_i, were proposed in [LL73], and [Mok83], respectively. Each task releases an infinite number of *task instances*, called *jobs*, according to its minimum inter-arrival time constraint. A task is called *periodic*, if two subsequent job releases are always separated by exactly the task's minimum inter-arrival, and *sporadic* if two subsequent job releases are separated by at least the minimum inter-arrival time. A job, which is released at time a_i, must be able to be executed for up to C_i time units before its absolute deadline at time $a_i + D_i$, to meet its deadline. Tasks and therefore the resulting task systems are often distinguished based on the relation between the inter-arrival times and relative deadlines of the tasks. A task is called an *implicit-deadline* task if its relative deadline is equal to its period, and a *constrained-deadline* task if its relative deadline is not larger than its period. Accordingly, a task set is an *implicit-deadline* task set if all tasks have implicit deadlines, a *constrained-deadline* task set if all tasks have constrained deadlines, and an *arbitrary-deadline* task set if tasks are allowed to have a relative deadline larger than their period. Implicit-deadline task sets are a subset of the constrained-deadline task sets, which are in turn a subset of the arbitrary-deadline task sets. Despite the variety of different task models, the activation and deadline constraints are universal to most of them.

periodic task model

sporadic task model

Each task releases an infinite number of task instances, *called* jobs, *according to its minimum inter-arrival time constraint*

Those early task models, invented for uniprocessor systems, are well researched with respect to response-time analyses, sound task parameter estimations, and efficient implementations in real-time operating systems. More recent use cases and applications, as illustrated in Figure 1.1, demand multicore systems, which are less timing predictable. Moreover, tasks may be composed of multiple subtasks, which can execute in parallel and are subjected to precedence- or other constraints. Consequently, more complex modeling, abstraction and engineering efforts are mandatory to provide comprehensive real-time system design and verification methodology; starting from task parameter estimation down to analysis, and real-time operating system implementations. For instance, sound static worst-case execution time estimates are to date unavailable for modern multicore systems [DC19]. Moreover, the applications often use fine-grained parallelism to implement the functionality, and the imposed temporal constraints of the applications are becoming more manifold than just a single relative deadline [ANN+22]. End-to-end latencies, temporal misalignment of sensor data in sensor fusion applications, and data-age constraints, are of particular interest.

More recent use cases and applications, as illustrated in Figure 1.1, demand multicore systems, which are less timing predictable

1.1.2 REAL-TIME SCHEDULING & SCHEDULABILITY

Roughly speaking, the objective of real-time scheduling algorithms is to optimize the system's resource utilization under the imposed timing constraints. A scheduling algorithm describes, which executable entity is executed on which processing unit at each point in time. For instance, a job is scheduled to be executed on processing units, such as processors, digital signal processors (DSPs), or communication links, depending on the system platform under analysis.

Then, for a given formal model of the task set and execution model, a schedulability analysis and schedulability test has to be devised, which formally verifies the imposed temporal constraints for that model. A system of a given task set and execution model is said to be *feasible* with respect to the imposed timing constraints, if there exists at least one scheduling algorithm such that all imposed timing constraints are met. Those timing constraints can be response-time upper-bounds, namely deadlines, end-to-end latencies, or others. Moreover, a scheduling algorithm is said to be *optimal* if all generated schedules meet the imposed timing constraints, in case that the system is feasible. With respect to a specific scheduling algorithm, we say that a scheduling algorithm is *feasible* if any generated schedule by that algorithm meets the imposed timing constraints.

On the basis of a schedulability analysis, a corresponding schedulability test is constructed, which verifies if the underlying scheduling algorithm is feasible with respect to the imposed timing constraints. Schedulability tests are classified into *sufficient*, *necessary*, and *exact* schedulability tests. A sufficient test guarantees that any problem instance, which is deemed feasible by the schedulability test, is guaranteed to be feasible. In contrast, a necessary schedulability test guarantees that any feasible problem instance is verified by the schedulability test. If a schedulability test is both sufficient and necessary, the test is called an *exact* schedulability test. From the perspective of analytic accuracy it is preferential to derive exact schedulability tests; however depending on the timing constraints,

Schedulability tests are classified into sufficient, necessary, *and* exact *schedulability tests*

execution model and task model, this incurs too high computational complexity. This is especially of concern, if decisions of whether or not a task can be accepted into the system without violating the timing constraints must be made online. As a consequence, many research efforts have been spent on deriving sufficient schedulability tests with polynomial-time complexity, which approximate an exact tests as good as possible. In contrast, necessary conditions for schedulability are most commonly used in mathematical arguments to prove approximation quality in terms of speedup factors or capacity augmentation bounds.

1.2 CONTRIBUTION OF THIS DISSERTATION

In safety critical cyber-physical systems, all timing and reliability requirements have to be guaranteed and verified off-line, and the system must be analyzable to predict the consequences of any scheduling decision with regards to those requirements. Philosophically, this dissertation is grounded in the following beliefs, that make up the dissertation hypothesis.

system behaviour is predictable and failures do not immediately lead to system malfunction

Robustness. In particular, if some tasks can not be verified to comply with its imposed timing constraints, the possible violations must be observed and notified in advance to the system, such that compensating or redundant actions can be taken. Thus, the system behaviour is predictable and failures do not immediately lead to system malfunction. To allow for such assurances, the operating system must provide specific kernel mechanisms for time management, monitoring of imposed timing constraints, and for handling tasks with explicit timing constraints.

Parameter uncertainty can be detrimental to model fidelity

Uncertainty. Parameter uncertainty can be detrimental to model fidelity and can lead to grossly over-estimated worst-case parameter bounds. This leads to pessimistic schedulability analyses and schedulability tests, which in turn reduce the system's resource utilization. Recurring back to Figure 1.2, refinement of the abstractions of task, execution model, and the model enforcement engineering, may be beneficial in terms of resource utilization if the refined model improves analysis accuracy, even at the cost of decreased average case performance.

Formal properties, which can be used modularly to compose safe and tight analyses

Modularity. Existing analyses and optimizations for scheduling algorithms and resource management policies in complex cyber-physical real-time systems are usually ad-hoc solutions for a specific studied problem. Formal properties, which can be used modularly to compose safe and tight analyses, as well as optimization for the scheduler design, and schedulability test problems need to be achieved by predictable interplay of computation, communication, and synchronization for soft (weakly hard) and hard real-time systems.

models, which are founded on formal properties and exposed to the system as formal contracts

Property-Based. In particular, this dissertation focuses on property-based decomposition, i.e., the decomposition of systems into models, which are founded on formal properties and exposed to the system as formal contracts. The imposed timing constraints are then verified on the basis of those formal contracts.

Research Overview. The contributions cover many different aspects of real-time systems. Therefore a short very high-level presentation of the studied problems and aspects is given, before summarizing the contributions in detail.

An abbreviated description of the objective is to research formal models and analyses to validate real-time, and other reliability constraints for cyber-physical systems on multicore platforms. In particular, parallel task real-time scheduling algorithms on multiprocessor systems, real-time arbitration protocols for network-on-chips, and the scheduling of cyber-physical system applications, with respect to sensor temporal alignment problems, are studied. Importantly, the formal models are devised such that *compliance* can be monitored and enforced in a real system, and that they are robust against uncertainty of various kinds.

To improve predictability in worst-case centric analyses, the exploration of timing predictable protocols, i.e., task models and scheduling algorithms are examined in Chapter 3 for parallel task multiprocessor scheduling and network-on-chips. Roughly speaking, the research approach is to impose additional constraints to the problem, which are theoretically and practically beneficial for worst-case centric analyses. The contributions of Chapter 4 are hierarchical, i.e., reservation-based parallel DAG task scheduling algorithms, which decompose the scheduling problem into two modular scheduling problems, providing temporal and spatial isolation. The hierarchical scheduling algorithms, allow execution time- and structural uncertainty, and reduce the DAG scheduling problem to a scheduling problem of standard task models. In Chapter 5, fault-tolerance as a supplementary reliability aspect of real- time systems is examined in spite of dynamic and stochastic external causes of fault. An approach is proposed, which allows optimizations to improve average case performance and non-explainable machine learning techniques to be used, while still providing hard QoS guarantees. Lastly, in Chapter 6, the temporal misalignment of sensor data in sensor fusion applications in cyber-physical system is analyzed, assuming a heterogeneous platform with globally asynchronous processing units.

Organization of the Dissertation. This dissertation is structured as follows. In Chapter 2, the general concepts regarding real-time systems architecture, scheduling and analysis, as are relevant to understand the motivation and technical contribution of this dissertation, are introduced. For improved contextual cohesion of the thematically different topics presented in this dissertation, the detailed related work is attributed to the respective chapters, but the more general related work is incorporated into Chapter 2. Afterwards, from Chapter 3 to Chapter 6, the scientific contributions of this dissertation are presented, which are elaborated in more detail hereinafter. At last, Chapter 7 provides a summary of the contributions and results of this dissertation, and discusses the results against the backdrop of the dissertation hypothesis.

Contributions. The contributions and outline of the remainder of this dissertation are summarized as follows:

CHAPTER 3 In the first contribution, a novel rigid gang scheduling algorithm, called *stationary rigid gang scheduling* is proposed, for a task set of sporadic rigid gang real-time tasks with constrained deadlines. Several sufficient schedulability analyses for task-level fixed-priority scheduling algorithms are proposed. A special assignment algorithm, called *consecutive stationary gang assignment*, allows to prove resource augmentation bounds for the provided scheduling algorithm and schedulability analysis. It is shown that *consecutive stationary gang assignment* provides beneficial theoretical properties, which can

be used to upper-bound the worst-case interference suffered by any task according to the ratio of gang sizes of two tasks. The algorithm is compared to the state-of-the-art schedulability analysis for global EDF by Dong and Liu [DL17] using synthetically generated sporadic real-time task systems with implicit deadlines. The evaluation results show that our algorithm outperforms the algorithm by Dong and Liu [DL17]. Furthermore, evaluations for constrained-deadline task systems are conducted, which demonstrate reasonable levels of schedulable task sets. In the second contribution, the fundamental difficulty of worst-case timing analysis, using scheduling theory when flit-based transmissions are handled by switch-based (link-based) scheduling, is formally discussed. Hereinafter, a novel timing predictable architecture and design of a two-dimensional NoC system, which is suitable for real-time multicore systems, is presented. By construction, any non-minimal route is deadlock-free, and therefore, the path diversity can be better utilized to distribute the traffic over the network. An implementation, including router design, and arbitration algorithm is provided and evaluated with synthetically generated data.

CHAPTER 4 In the first contribution, the probabilistic conditional parallel (Directed Acyclic Graph) DAG task model is proposed to express structural uncertainty during execution. A hierarchical scheduling algorithm for the analysis of the probabilistic conditional parallel DAG task model is proposed, and design rules are devised, which provide probabilistic characteristics such as bounded tardiness and probabilistic upper-bounds for k-consecutive deadline misses. In the second contribution, the *parallel path progression* concept is proposed, allowing to consider the parallel execution of multiple paths in the DAG. This property is implemented using a DAG subtask-level fixed-priority policy and a preemptive fixed-priority list-scheduling algorithm. A polynomial-time parametric approximation algorithm is provided for the algorithmic problem to find the optimal makespan, i.e., worst-case response-time. The *parallel path progression* concept is extended to two hierarchical scheduling algorithms, namely a sporadic arbitrary-deadline gang reservation system and a sporadic arbitrary-deadline ordinary reservation system. For both reservation systems, worst-case response time analyses, and algorithms to generate and provision feasible reservation systems are provided. The approach is evaluated using synthetically generated DAG task sets. The evaluations demonstrate that the approach advances the state of the art in high-parallelism scenarios, and show that the performance of the approach is between the start of the art and federated scheduling in more sequential scenarios. Above that, a more strict and related *path-monotonic progression* concept is proposed, which admits to design suspension-aware reservation systems, in order to reduce resource usage.

CHAPTER 5 In Chapter 5, fault-tolerance as a supplementary reliability aspect of real-time systems is examined in spite of dynamic external causes of fault. To assure that an acceptable quality-of-service (QoS), i.e., fault-tolerance, can be achieved, an upper bound on consecutive erroneous job executions, and guaranteed m error- free executions out of any k consecutive job executions, are studied. Using various job variants, which trade off increased execution

time demand with increased error protection, a state-based policy selection strategy is proposed. The policy guarantees that all reachable states comply with the QoS constraints, whilst minimizing the expected system utilization and assuring hard real-time compliance of the task system. The state-based policy further allows for the usage of machine learning techniques, which are then able to provide hard guarantees. The proposed approaches demonstrate significant decreased system utilization, compared to the state of the art in the evaluations. Extensive numerical evaluations showed that the proposed approaches outperform the state of the art in most of the evaluated cases, with respect to the average system utilization. Evaluations of the learning and runtime overheads suggest that the proposed approaches can be reasonably applied in real-time systems.

CHAPTER 6 In Chapter 6, analyses for the maximal *sensor data time-stamp difference* are presented. The results show that in spite of the complex heterogeneous architecture and globally asynchronous processing units, the maximal time difference of any two sensor data samples – that refer to the system state at a specific point in time – when being used for sensor fusion, can be upper bounded in a modular manner. Under the assumption that each task is verified to be schedulable on its respective processing unit according to any readily available task-level fixed-priority worst-case response-time analysis for non-preemptive or preemptive scheduling algorithms, an algorithm is proposed to calculate the maximal *sensor data time-stamp difference*. Based on an abstract *precedence property*, which depends on the task model and scheduling algorithm, the presented analyses can be refined to improve accuracy. Task precedence properties are presented for non-preemptive rigid gang scheduling and preemptive stationary rigid gang scheduling.

1.3 AUTHOR'S CONTRIBUTION TO THIS DISSERTATION

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation must include a list that highlights the author's contribution to research results that were obtained in cooperation with other researchers.

In all the contributions, Jian-Jia Chen and Georg von der Brüggen, provided regular discussion, supervision, and contributions to the writing of the papers. The following overview lists the contribution on the results presented in the individual chapters:

- Chapter 3 focuses on arbitration and scheduling algorithms that improve timing predictability by construction. The first contribution is based on the work published at ECRTS 2021 in [UGB+21]. I was the principal author and provided the idea, theorem and proofs of stationary rigid gang scheduling, the reduction to suspension-aware task-level fixed-priority scheduling, and the consecutive stationary gang assignment with the parametric speed-up factor. Mario Günzel and Jian-Jia Chen contributed suggestions for improvement in Theorem 3.3. I was the principal author of the experimental

evaluation including, experiment design, implementation, and interpretation of the results. Mario Günzel, Jian-Jia Chen, and Georg von der Brüggen provided general discussion, which improved the work.

The second contribution is based on the work [UCB+20] published at RTCSA 2020, where I was the principal author contributing the concepts, theorems and proofs. Tulika Mitra, and Jian-Jia Chen discussed and settled the collaboration, and Tulika Mitra supervised Vanchinathan Venkataramani. Jian-Jia Chen provided the formalization of a *progression* in Definition 3.14. In a close collaboration with Jian-Jia Chen, the concept to use a separate *arbitration net* and *data net* to implement the proposed *simultaneous progression switching protocol* was developed. I was the principal author of the evaluation except for the data provided for the arbitration overhead measurements in Table 3.1 that was provided by Vanchinathan Venkataramani. The contributions are extended beyond the publication by a possible implementation for arbitration and transmission in Section 3.4.4, which I contributed.

- Chapter 4 examines hierarchical scheduling of parallel DAG tasks. The first contribution is based on the work [UGC21] published in RTSS 2021. I was the principal author providing concepts, theorems, and proofs. Mario Günzel improved the presentation of the prior proof of Theorem 4.3 and corresponding Lemmas. An anonymous reviewer of RTSS 2021 provided valuable comments regarding the semantics of the probabilistic conditional DAG task model. I was the principal author of the evaluations, providing implementation, experimental design, and interpretation of the results. The second contribution is based on the work [UGB+23] published in *IEEE transactions on computers* in 2023. The initial idea of *parallel path progression* property was proposed by me to Mario Günzel, which was then developed during a discussion and collaboration together with him. He also provided a more rigorous proof strategy (in comparison to my original version) of Theorem 4.9, and many discussion and feedback that helped with the proofs. I was the principal author of the theorems and proofs concerning the reservation system design, and all theorems and proofs regarding the approximation algorithm and analyses. I was the principal author of the evaluations, providing implementations, experiment design, and interpretation of the results. Beyond the contributions found in the paper [UGB+23], I provided the concept, theorems and proofs for the *path monotonic progression* property and the suspension-aware reservation design.
- Chapter 5 considers fault-tolerance in spite of dynamic external causes of fault in real-time systems are considered an is mostly based on the work published in RTAS 2023 [SUC+23] with additional considerations of k-consecutive error constraints, considerations of the error model, and corresponding analyses in this dissertation and removal of machine learning parts that are irrelevant to understand the evaluation. The regulator-based idea is based on initial research concepts by me, which have been discussed with Junjie Shi and Kuan-Hsun Chen for possible use in fault-tolerance that culminated in the joint published work. I contributed to the concept, and provided the formal definition and apparatus of the problem and terminology, proof of theorems, and text regarding the first part. Kuan-Hsun

Chen provided his expertise on fault-tolerance in the context of real-time systems. The machine learning formulation was proposed, researched, and elaborated by Junjie Shi. The evaluations are designed, conducted, and implemented by Junjie Shi.

- Chapter 6 considers the analysis of *maximal sensor data time-stamp difference* in processing graphs. I was the co-author of the work published in RTSS 2021 in the industry challenge program [GUC+21]. I proposed the general analysis framework to infer best and worst-case sensor propagation latencies and provided the theorems and proofs. Mario Günzel helped to correct a mistake in the drafted statement of Theorem 6.2. I was the author of the *precedence* property and the extensions to gang task scheduling.

2 REAL-TIME SYSTEM CONCEPTS

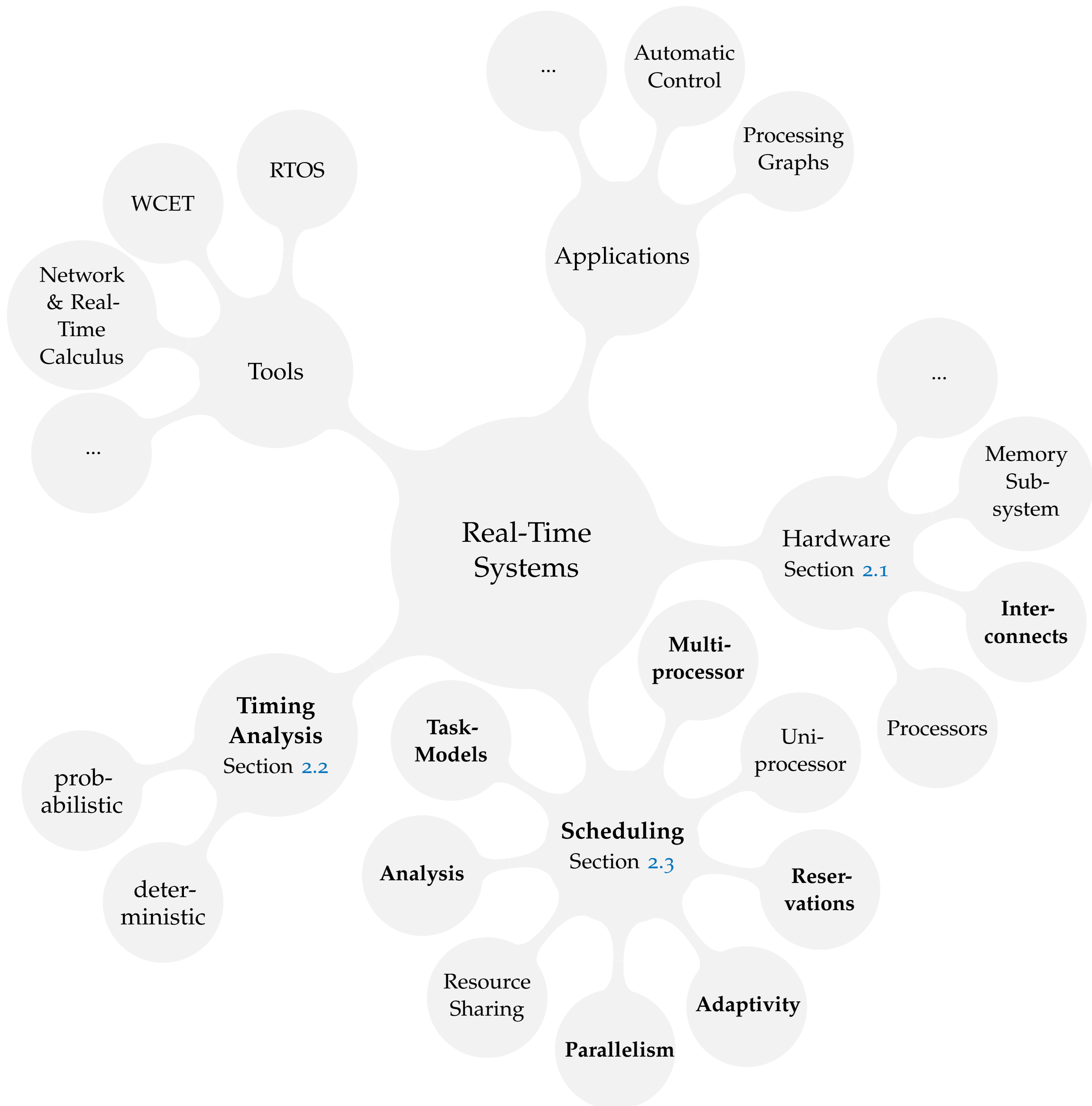

Figure 2.1: A taxonomy of real-time systems as considered relevant to this dissertation, where the bold vertices accent the relevance to the contributions in this dissertation.

A variety of interdependent domains, such as hardware architecture, timing characteristics of the hardware (and in particular timing predictability), real-time scheduling algorithm design and analysis, implementation of schedulers,

consideration of overheads in real-time operating systems, and lastly applications, constitute the domain of real-time systems.

The real-time system design for uniprocessor systems is rather complete in terms of appropriate task models, uniprocessor timing analysis, optimal real-time scheduling algorithms, and existing implementations in real-time operating systems. In contrast, the real-time systems design for multicore systems is still an open problem, which is founded in the decreased predictability of the hardware platform, challenging the assumption that a reliable worst-case execution time estimate can be inferred. Secondly, the complex interference patterns on the shared resources, such as memory and cache, have been shown to have a significant impact on the efficiency and execution time of parallel tasks [ZCC+22; CBN+20; CBN+18a; Yun15; AY19]. Due to the sheer size of the domain of real-time systems, an exhaustive presentation is out of scope in this dissertation.

real-time systems design for multicore systems is still an open problem, which is founded in the decreased predictability of the hardware platform

In this chapter, the most relevant background is provided to elaborate the necessary technical background, context, and jargon, required to understand the motivation and technical contributions in this dissertation. Due to improved contextual cohesion, the related work is placed in the respective chapters later in this dissertation. To bridge this gap, this background chapter includes a depiction of the general related work with regards to the presented topics, whereas the individual chapters contain more specialized related work.

this background chapter includes a depiction of the general related work with regards to the presented topics, whereas the individual chapters contain more specialized related work

The remainder of this chapter is structured according to the illustration in Figure 2.1, with a focus on the most relevant topics, which are emphasized in bold font. In Section 2.1 *Multicore & Real-Time Systems*, an introduction to multicore systems and their classifications as well as the most important components, impacting real-time scheduling are discussed. Following, in Section 2.2 *Timing Analysis*, the problem of inferring the worst-case execution time of a program on a given hardware platform is introduced. In that section, the state of the art in timing analyses are summarized and discussed. In Section 2.3 *Real-Time Scheduling*, the most important background on real-time scheduling theory and implementation considerations are introduced. Starting from fundamental implementation considerations, fundamental task models, real-time scheduling algorithm designs and analyses are categorized and contextualized by algorithmic and practical constraints.

2.1 MULTICORE & REAL-TIME SYSTEMS

The focus of this dissertation is on cyber-physical systems and real-time system design for multicore systems. Therefore, a brief summary of multicore architectures and its relevance to real-time system design is given. Real-time system designs evolved from unicore, single memory designs to complex multicore system-on-chip architectures as used in, e.g., autonomous driving, robotics, or defense applications to match the computational demands of workloads, durability requirements, tight deadline constraints, and thermal as well as power system design constraints. Multicore chips consist of several smaller cores, running at a lower frequency, which can ideally perform the same amount of work as a single core with a higher frequency without consuming as much energy and

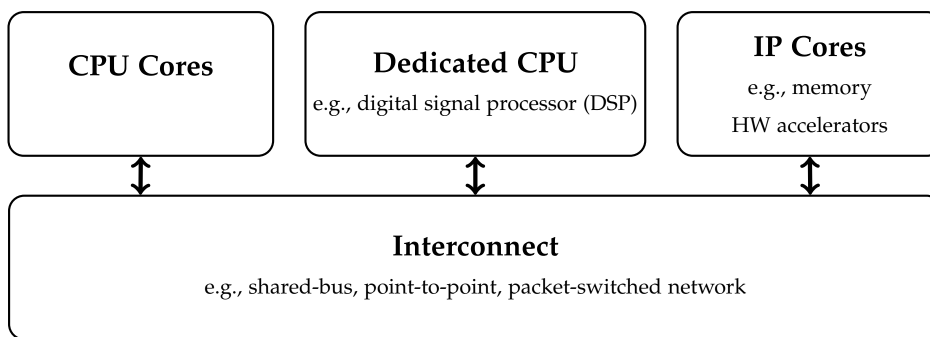

Figure 2.2: Architectural view of an exemplary multicore system-on-chip, which is redrawn from [Abd17].

power [Abd17]. An important distinction in multicore systems is between homogeneous and heterogeneous cores. That is, a set of cores is called *homogeneous (or identical)* if all cores in the set have the same cache hierarchy, cache sizes, core frequencies, and the same functions [Abd17]. Conversely, a set of cores is called *heterogeneous* if some cores have different functions, core frequencies, cache sizes, cache hierarchy, and memory models. Heterogeneous multiprocessor systems are further classified into uniform and unrelated heterogeneous systems. In *uniform heterogeneous multiprocessors*, the processors only differ in the processor frequencies, whereas everything else is identical. In *unrelated heterogeneous multiprocessors*, each processor may have completely different capabilities. As a consequence for real-time scheduling theory of heterogeneous systems, the execution times of each job depends on which processor they are executed on, which complicates the analyses. When we consider heterogeneous multiprocessor systems in the scope of this dissertation, we assume an off-line task to processor mapping and disallow migration such that the varying processing times do not need to be considered during analysis.

homogeneous cores

heterogeneous cores

uniform heterogeneous multiprocessors

unrelated heterogeneous multiprocessors

In this dissertation, most of the presented theory is aimed at homogeneous multiprocessors with uniform memory access, in which the memory and cache contention problem is not explicitly considered such as in memory or cache-aware scheduling approaches, e.g., in [GYY+20; JNS+12; CA08; XCW+20]. Instead, we implicitly consider the efficiency problem and contention problem on all shared resources by recurring back on practically proven efficient scheduling paradigms, such as *federated scheduling* [LCA+14], or *gang scheduling* [Jet97; FR92; AY19] when designing our scheduling algorithms. Similarly, partitioned scheduling is more favorable than global scheduling, due to the lack of process migrations and improved cache affinity.

we implicitly consider the efficiency problem and contention problem on all shared resources by recurring back on practically proven efficient scheduling paradigms

federated scheduling

gang scheduling

On a more general note, there is a trade-off between, theoretic analyzability, real system performance, model robustness and fidelity, and ease of use constraints in the real-time operating system. All of these trade-offs must be purposefully balanced when designing scheduling algorithms and abstractions.

In the remainder of this section, the most determining components in a multiprocessor system – with respect to real-time systems – are explained and important practical considerations for the scheduler design in a real-time operating system are summarized. In Section 2.1.1, the most common cache architectures

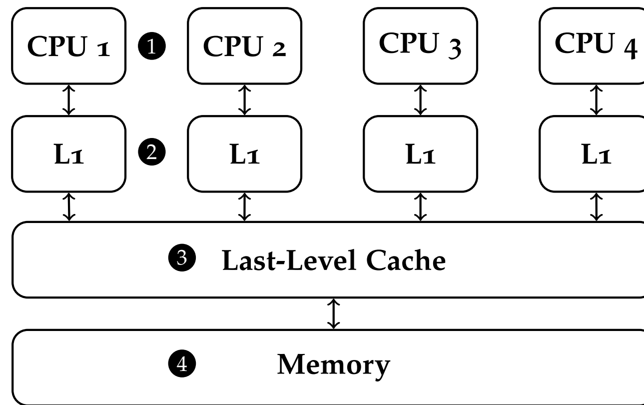

Figure 2.3: Architectural view of an exemplary memory and cache hierarchy.

and modes of operation are explained. Afterwards, in Section 2.1.2, the interconnection system in multiprocessor systems is presented and the problems with regards to real-time systems are discussed.

2.1.1 MEMORY & CACHE

The following section is based on the explanations in [Bra11a] and is enriched with additional information.

uniform memory access

non-uniform memory access

The memory architecture can be classified into *uniform memory access* (UMA) architectures and *non-uniform memory access* (NUMA) architectures, which differ in the maximum memory access latency from a processor to a specific memory word. In UMA architectures, the maximum memory access latency of each memory word is the same for all processors, whereas the maximum access latency differs for each processor in NUMA architectures.

memory architecture impacts the scheduling design considerations and in particular the efficient implementation of scheduling algorithms in real-time operating systems

The memory architecture impacts the scheduling design considerations and in particular the efficient implementation of scheduling algorithms in real-time operating systems. For instance, in a completely centralized shared memory system with uniform memory access, the shared memory bus and shared caches limit the processor utilization, due to waiting cycles to fetch the memory. In contrast, in NUMA systems, each task should be scheduled and executed on a processor, which has a small maximum memory access latency, to the off-chip modules holding the task's data. In consequence, global scheduling, in which each processor can schedule and execute a job of a task, is not efficient on NUMA systems.

Based on the fact, that the maximum access latencies for the main memory are magnitudes larger than the computation times on modern processors, the memory system is constructed as a hierarchy of memory layers. The top layers, which are closer to the processor have smaller memory access latency, but also smaller storage capacity. An exemplary uniform memory architecture is shown in Figure 2.3. The closest memory to the processor is called registers, which is internal to the processors ❶ containing a partial set of variables, and process context such as the *processor status*, *program counter*, and *stack pointer*. Notably,

there is almost no delay in accessing them by the processor in terms of additional processor cycles. In contrast, accessing the main memory can result into hundreds of processor cycles in which the processor is stalled, waiting to fetch the data or instructions from main memory [Bae09]. To that end, caches are used with the intent that most accesses to data and instructions go through the fast cache, instead of, to the slow off-chip memory. Cache replacement policies, prefetching, and the temporal and spatial locality in the accessed instructions and data addresses are facilitated to minimize cache misses. In the presented example, each processor is assumed to have a unified private level-1 cache ❷, but dedicated instruction and data caches are also commonly used. On the next level, a shared last-level cache ❸ is used as a backup cache, if the memory word could not be found in the private level-1 cache. The last level cache ❹ refers to the last level of cache memory, which is accessed by the cores, prior to fetching from main memory.

caches are used with the intent that most accesses to data and instructions go through the fast cache, instead of, to the slow off-chip memory

As a consequence for real-time system design, and in particular from the scheduling point of view, the main difference between shared- and distributed-memory architectures is how process migration is implemented. In addition, cache misses should be avoided to reduce the execution time variability, and execution time penalty and in turn achieve high resource efficiency and improved predictability.

main difference between shared- and distributed-memory architectures is how process migration is implemented

A comprehensive description of caches is given, for instance, in [Bae09; Tan09] and only the most important fundamentals and real-time systems related cache developments are summarized hereinafter. Caches are organized into cache lines, which consist of blocks of consecutive memory addresses, denoting a range of contiguous memory words that are stored in the cache line. Whenever the processor reads or writes to a memory address, the complete cache line – which encompasses the referenced address – is read from or written to. The mapping strategy of a cache is distinguished into direct mapped caches, fully associative caches, and set associative caches, which differ in the restrictiveness of mapping constraints of the cache lines. In a *direct mapped cache*, each cache line may only reside in one specific location in the cache, whereas in a *fully associative cache*, each cache line may reside in any location in the cache. The direct mapped cache has low hardware complexity, however the restrictive policy leads to more cache line evictions and thus higher miss rates. In contrast, fully associative caches have very high hardware complexity, due to the large amount of cache lines which need arbitration logic, but lead to lower miss rates. Set associative caches are a hybrid approach, where each cache line may reside in any location of a set of fixed locations. A set associative cache is said to be an *n-way associative cache* if each cache line may be mapped to any of the n cache locations.

direct mapped cache

fully associative cache

n-way associative cache

When memory is addressed by the processor, which is found in the cache, namely a *cache hit*, then that data is directly read from the cache without any access to main memory. If however the addressed memory is not in the cache, namely a *cache miss*, then the respective cache line is brought into the cache and if necessary another cache line is evicted. Cache line replacement strategies are for instance *random replacement*, *first-in first-out* (FIFO), *least-recently used* (LRU), or *least-frequently used* (LFU), some of which have been analyzed for their performance in, e.g., [MT19]. In the context of real-time systems, the LRU strategy is of

cache hit

cache miss

random replacement

least-recently used

least-frequently used

particular importance, due to the efficiently computable abstract cache semantic [AFM+96], which allows for over- and under-approximations of cache contents; e.g., *may- and must analysis* of cache contents can be derived. An abstract cache semantic is a collection of equivalence classes of cache states at each program point, which any execution may encounter at that point.

may- and must analysis

There are four primary causes for cache misses

There are four primary causes for cache misses, which are shortly listed. *Compulsory cache misses* are unavoidable and occur whenever a cache line is referenced for the first time and must be loaded into the cache. A *capacity cache miss* occurs when the working set size of the process exceeds the cache size, i.e., some cache lines from the working set must be temporarily evicted in favor of other data. In the case of direct mapped and set associative caches, conflict misses occur if – due to the mapping constraints of the caches – some cache line must be evicted. Lastly, *coherency misses* occur when another processor evicted a referenced cache line, due to consistency issues, as managed by some coherency protocol. Cache coherency refers to the problem, that in a multiprocessor, caches on different levels in the hierarchy might become inconsistent if one processor updates a memory location, which is currently cached by other processors. Cache consistency can also become a major source of overhead and timing unpredictability if processors frequently read and write memory locations, which reside in the same cache line. Cache coherent processors employ a cache-consistency protocol to transparently evict outdated cache entries from the caches of other processors to provide a coherent state. A detailed introduction to cache-consistency is provided by, e.g., Hennessy and Patterson [HP12]. A comprehensive survey on the impacts of resource sharing and performance prediction of shared bus, and shared caches with respect to real-time systems is given in [ABD+13].

compulsory cache miss

capacity cache miss

coherency cache miss

Another approach to improve predictability and to reduce interference of co-running tasks is that of cache partitioning

cache partitioning

Another approach to improve predictability and to reduce interference of co-running tasks is that of cache partitioning. In *cache partitioning*, all cache lines are partitioned among the competing tasks. Several cache partitioning strategies have been investigated, e.g., in [Mue95; LHH97; KS90]. In particular, set based partitioning in [ZDS09; SM08] or way-based partitioning techniques in, e.g., [QP06] have been proposed. The authors of [ABD+13] state that the size of the cache partitions have a strong impact on the performance of the co-running tasks and in consequence on the system performance. Even more so, way-based partitioning only allows for coarse-grained allocation of cache space in low-associativity caches. In contrast, set-based partitioning allows for more fine-grained allocation of the available cache memory, since the number of ways is usually larger than the cache's associativity. Another proposed approach to improve predictability on shared resources is predictable memory controllers [PQC+09; RLP+11]. Radojkovic et al. [RGG+12] evaluated the impact of shared resources in multi-threaded commercial-off-the shelf (COTS) processors in time-critical environments. Despite measures to improve predictability, recent results by Yun et al. [Yun15] suggest that on COTS multicore platforms, a process can suffer a slowdown by up to 14×, even when using cache partitioning techniques. In a subsequent work [AY19] Ali and Yun have shown that the suffered slowdown is even larger for highly parallel workloads, when co-run with other memory intensive applications. The authors attribute this findings to not partitionable resources, such as the miss-status holding register, which leads to blocking whenever all available miss-status holdings

registers are used up. Since the extent of cache interference depends on the set of jobs, which execute on the processors and share a cache and memory, interference can also be reduced using cache-aware scheduling approaches such as in [CA08; JNS+12; XCW+20] or memory-aware scheduling as, e.g., in [HCR16].

2.1.2 INTERCONNECTS

By nature of the memory interconnect, multiprocessors can be classified into shared-memory multiprocessors and distributed-memory multiprocessors. In the former, a central memory is accessible to all connected processors by a shared memory bus. In the latter, there are multiple local memories, which are accessible only to a subset of the multiple processors. Processors are still connected to each other in a distributed-memory system, but only via a message bus which does not allow direct access to non-local memory. With increasing cores in a single multicore system-on-chip (MCSoC), the communication latencies are becoming a bottleneck for the overall system's performance and thus more scalable interconnects than a shared bus are required.

communication latencies are becoming a bottleneck for the overall system's performance and thus more scalable interconnects than a shared bus are required

Starting from the initial work of Dally et al. [DT01], a packet-switched on-chip network was proposed in contrast to the prior predominant approach to place dedicated transmission lines between processing elements. The authors identified structural, performance, and modularity advantages compared to *global wires* connecting processing elements individually, which are summarized as follows:

- Due to the regularization in wire length and the routing geometry of the wires in the on-chip network, the electrical properties are optimized; resulting in predictable cross-talk and noise characteristics. Therefore, highly optimized *signaling circuits* can be designed, which reduce power dissipation and decrease propagation delay and thus increase bandwidth.
- By sharing of the network-on-chip with all participants, the transmission lines can be used by any client with transmission requests, thus reducing the idling of resources.
- Lastly, an on-chip interconnection network facilitates modularity by defining a standard interface.

Still many on-chip interconnection networks are implemented using buses in form of bidirectional links, e.g., SPI, I2C, CAN, or AXI, where several masters and slaves can be connected to a shared bus. The bus topology has the benefit of simplicity, i.e., simple addition of new devices, simple broadcasting, low area usage, simple arbitration, and good timing predictability. From a formal real-time analysis perspective, a shared bus – which corresponds to the uniprocessor model in real-time scheduling theory – is a well understood model, for which practically efficient and theoretically optimal scheduling solutions exist. In the bus topology, a centralized arbiter is required to schedule the bus requests. The arbiter periodically examines all accumulated requests from the multiple master interfaces, and grants access to a master, using arbitration mechanisms specified by the bus protocol.

The bus topology has the benefit of simplicity, i.e., simple addition of new devices, simple broadcasting, low area usage, simple arbitration, and good timing predictability

The communication fabric of a multi- or many-core platform must scale with the number of cores, since otherwise the computation capacity of the cores may be

wasted if they are waiting for communication, synchronization, or memory access. In that regard the bus topology does not scale, which is due to electrical issues such as increased reflections, electrical loading, clock skew, propagation delay and the time-multiplexing of the shared bus reducing the bandwidth for each client. A theoretically optimal switching network is the $n \times n$ crossbar, which connects n sources to n sinks in a non-blocking manner. That is, any distinct source-to-sink pair can transmit simultaneously without contention. However, the hardware complexity of a crossbar is infeasible for larger systems. Instead, various multistage switching networks with less hardware complexity and reasonably low blocking properties have been proposed. These multistage switching networks consist of smaller 2×2 crossbars arranged in stages such as the *omega-network* [Law75], *Banyan network*, *Benes network*, or the *Butterfly network* [KBD07].

crossbar

omega network

banyan network

benes network

butterfly network

Another approach to achieve good scalability of communications, is the Network-on-Chip (NoC) architecture, in which a packet-switched network is used to provide the interconnection of the processing elements (including physical cores) on a chip. The NoC architecture allows parallel inter-core communication with moderate hardware costs and allows for asynchronous transfer of information. Consequently, NoCs are the prevalent choice of interconnection, due to their overall good performance and scalability potential as reported by Kavaldjiev et al. [KS03]. To this end, redesigning the interconnection network between cores has been a major focus of chip manufacturers resulting in network-on-chip (NoC) designs, e.g., a ring in the *Intel Xeon Phi 3120A*, a 2-D torus in *MPPA Manycores* by Kalray, and a 2-D mesh in *Tilera TILE-Gx8036*.

NoC architecture allows parallel inter-core communication with moderate hardware costs and allows for asynchronous transfer of information

NoC research and design issues include many trade-offs

NoC research and design issues include many trade-offs such as, e.g., topology, routing, switching, scheduling, flow control, buffer size, network interface, and packet size; many of which have been proposed and evaluated in the literature. Due to the many different approaches to NoC design and performance analyses, we refer to the literature, e.g., [DT04; DPS14] for a more comprehensive presentation. A summary of the technical specifications such as, throughput, area, flit size, and others, of the state-of-the-art NoC architectures can be found in [MCM+04].

The core elements of a network-on-chip, are the network interface, routers, and links, which are briefly presented. The *network interface* decouples the *intellectual property* (IP) cores, e.g., processors, memories or caches, from the communication protocols and message formats within the network. The network interface is most commonly separated into a front-end and a back-end. The front-end interfaces the IP core with socket standard implementations, such as *Open Core Protocol* or *AXI*; the back-end is responsible to packetize and de-packetize the messages and to control the end-to-end flow.

network interface

intellectual property cores

Routers consist of buffers, arbitration logic, and several input and output ports, which can be arbitrated, i.e., switched by the arbitration logic. Routers are connected by links, providing the channels over which the packets are transmitted from the source router to the destination router. The *width* of a link denotes the number of bits, which can be transmitted during one cycle over the link. The width is in the range of 16 to 512 bit in typical network-on-chips [LNP+13]. In general, a link may provide either a *full-duplex*, or a *simplex* connection between an upstream and a downstream router. In the course of this dissertation, we assume that each component in the network is connected by two opposite directed simplex

link width

full-duplex

simplex

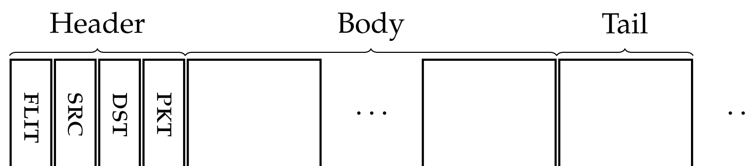

Figure 2.4: Packetization of a message into packets; each of which consist of a head flit, body flits, and a tail flit.

links, which may be separated into multiple *virtual channels*. A virtual channel, denotes a subset of the port buffers, which can be arbitrated independently. The arbitration of links, and the routing algorithms are discussed later in this section.

virtual channel

2.1.2.1 *Messages, Packets and Flits*

As previously mentioned, *packets* are generated by the network interface from the *messages*, which refer to the various heterogeneous data formats injected by the IP cores. These are for instance, memory request in the order of tens of bytes, or reply messages in the order of hundreds of bytes [DPS14]. Packets usually have a static and restricted length and thus a variable length message is divided into a sequence of one or more packets by the *packetizer*. A header is attached to each packet, containing routing, and sequencing information, which are required for channel allocation and de-packetization.

packets
messages
packetizer

The packets themselves – depending on the switching technique – consist of integral units of transmission and arbitration called *flits*, which originates from the term flow-control digits. On the physical layer, flits can be further decomposed into *phits*, which describe the number of bits that can be transmitted during a single cycle over a link. The phit size typically coincides with the flit size.

flits
phits

A message frame is illustrated in Figure 2.4, which consists of a sequence of packets. Each packet itself consists of a unique *header flit*, a sequence of possibly many *body flits*, and a unique *tail flit*. The header flit denotes the beginning of a packet and contains all addressing, sequencing and identification information, required for routing and de-serialization at the destination. The body flits contain the payload of the packet, and the tail flit denotes the ending of a packet. The tail flit is used for signaling in the network, e.g., to de-allocate virtual channels in the router. Throughout this dissertation, two adjacent routers in a path are called *upstream router* and *downstream router*, i.e., the upstream router transmits data to the downstream router.

header flit
body flits
tail flit
upstream router
downstream router

2.1.2.2 *Topology*

Network topology defines the way that routers and links are interconnected and are an important design choice, due to its affect on routing, reliability, throughput, latency, and design complexity. More formally, a topology can be defined as a graph, where the edges represent the network's links and the vertices represent the network's routers. Networks can be differentiated into *direct and indirect networks*. In the former, each router has a terminal vertex and all routers are

direct & indirect networks

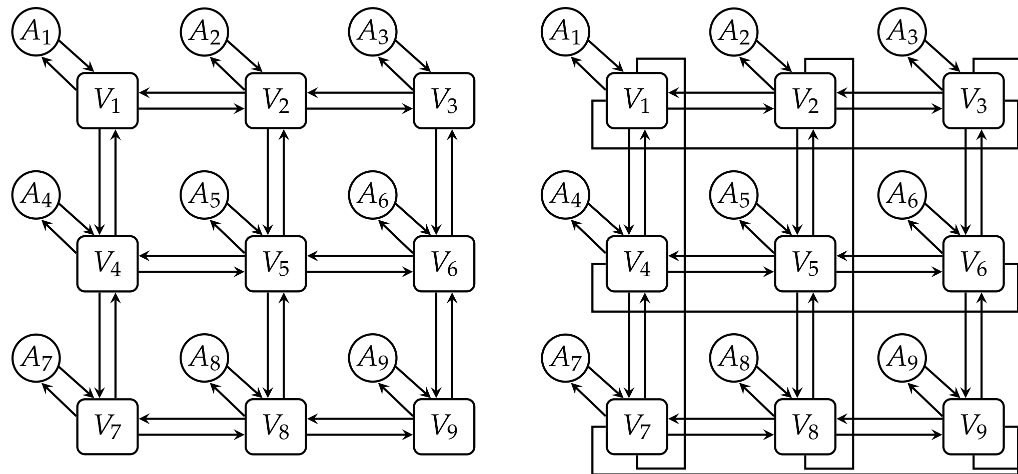

Figure 2.5: Two exemplary direct regular topologies, namely a 2D-mesh on the left, and a 2D-Torus on the right.

sources and destinations of traffic, whereas in the latter routers and terminal vertices are distinct. That is, terminal vertices can be either source of traffic or a sink thereof, whereas the intermediates routers switch the traffic between the terminal vertices. A network topology is associated with metrics such as *degree*, *diameter*, *hop count*, and *bisection bandwidth*, which characterize a specific topology. In the following, these terms are explained with reference to the 2D-Mesh and 2D-Torus topology with network size $N \times N$, shown in Figure 2.5.

Number of links. The number of links is given as a function of the number of routers in the network. A large number of links can increase the throughput at the cost of increased chip complexity and area.

degree

Degree. The degree of a network topology denotes the maximal number of links at any router in the network and is an estimate of the router design cost in terms of required ports per router.

diameter

Diameter. The diameter of a network topology denotes the maximal routing distance within the network in terms of crossed links, which is N in the case of a $2D$-Mesh and $2D$-Torus.

hop count

Hop Count. The hop count denotes the number of hops, which a message takes from source to destination, in terms of links, and is related to the transmission latency.

bisection bandwidth

Bisection Bandwidth. Given a bisection that partitions the network into two (nearly) equally sized halves then the bisection bandwidth of that partition is given by the cumulative link bandwidth connecting both partitions. The bisection bandwidth of a network denotes the minimal bisection bandwidth of any possible bipartition of the network. A higher bisection bandwidth suggests a lower network contention probability. In the $2D$-Mesh, the bisection bandwidth is given by $\sqrt{N}$ and by $\sqrt{N}+2$ for the Torus.

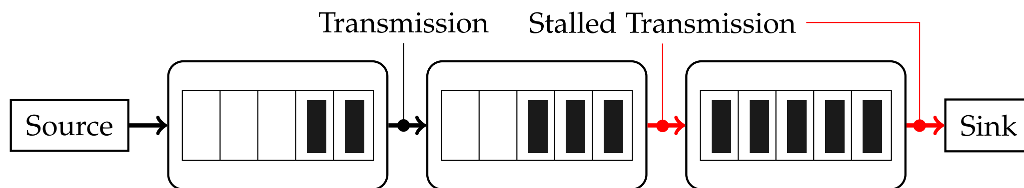

Figure 2.6: Exemplary transmission of flits in a wormhole switched network-on-chip from a source core to a destination core over 3 links, which is redrawn from [Abd17]. The routers are partially represented by the input and output buffers of a single virtual channel. When the buffers are fully exhausted at the downstream router, the upstream router is stalled on transmission.

2.1.2.3 *Switching & Flow Control*

According to Dally [DT04], flow control deals with resource allocation and contention resolution, where the subjected resources are channel bandwidth, buffer slots, and control state. The switching technique determines how network resources are allocated to a flit (or packet), on a route between a source and destination router.

Flow control schemes can be classified into bufferless and non-bufferless schemes

bufferless flow control

Flow control schemes can be classified into bufferless and non-bufferless schemes. In *bufferless flow control*, only the channel bandwidth must be allocated to a flit (or packet). Consequently, waiting, i.e., being queued for an allocation can not be implemented and is usually resolved with flit (or packet) dropping, or deflection routing.

circuit switching

A prominent switching technique for bufferless routing is *circuit switching*, in which a packet (transmission unit) is forwarded by the routers through dedicated links, which are reserved and allocated until the transmission is finished. Therefore, each transmission can only be preempted during the establishing of a route from source to destination. An advantage of this approach is that no buffering is required and therefore any optimized and adaptive routing scheme is deadlock-free. However, the overhead to establish the routes may render this approach infeasible, when small packets are injected frequently.

non-bufferless switching

store-and-forward switching

Prominent *non-bufferless switching* techniques are store-and-forward switching and wormhole switching. In *store-and-forward switching*, switches can only forward a packet once it is completely received and stored, which implies that the switches must provide sufficient buffer capacity to store a complete packet. Fortunately, the arbitration protocol is suitable for real-time analysis, since a packet may compete for at most one link at each point in time. It is, however, not useful in practice because of the large buffer requirement.

In wormhole-switching – which is illustrated in Figure 2.6 – each packet is divided into smaller transmission units, called flits

In wormhole-switching – which is illustrated in Figure 2.6 – each packet is divided into smaller transmission units, called flits, always including a designated header and a designated tail flit, which are used for control and routing. That is, each payload flit follows the output port of the header flit. In fixed-priority wormhole switched NoCs, each router contains virtual channels, i.e., separated buffers which contain flits of a single packet. Once the tail flit is transmitted and removed from the buffer, the virtual channel can be used for flits of another packet. Furthermore, the highest-priority flit is scheduled to transmit over the

link at each router. In this approach, complete packets do not have to be buffered, which allows smaller buffers in the hardware design. On the downside, each packet may be distributed over multiple routers and subsequently compete for multiple links at the same time making the timing analysis complex. Additionally, the limited number of virtual channels and full buffers on the downstream router add additional interference, which complicates the analysis. That is, in buffered switching, an upstream router can only successfully transmit a flit (or packet) if a buffer slot is available at the receiving downstream router, which must hence be coordinated.

credit based flow control

Credit based flow control is the most prominent technique, in which the upstream router maintains the status of available buffer slots at all downstream routers. Whenever the upstream router transmits a flit (or packet), the counter is decremented to account for the buffer slot being occupied by the transmitted flit. Conversely, when the downstream router forwards a flit to the next router, it sends a *credit* back to the upstream router, to signal that the buffer is available again. The most simplistic approach to implement credit based flow control is *on-off* flow control. In this protocol, the downstream router deactivates any incoming transmissions from an upstream router by signaling an *off* signal, if all buffer slots in the downstream router are fully occupied. Conversely, whenever buffer slots are free, the downstream router exerts an *on* signal to the respective upstream routers. Another approach is the *ACK-NACK* flow control, in which the flit is sent from the upstream router to the downstream router without knowledge of the available buffer space. In case that no buffer slots are available, the flit is dropped and a *NACK* is sent to the upstream router, indicating that the flit has been dropped. Conversely, if a slot was available upon reception, an *ACK* is sent to indicate a successful transmission. Irrespective of the specific flow control approach, the flow control needs to be implemented within the router and thus influences the hardware complexity of the router.

on-off flow control

ack-nack flow control

flow control needs to be implemented within the router and thus influences the hardware complexity of the router

Closely connected to flow control is arbitration, i.e., the policy-based allocation of channel bandwidth, virtual channel, and buffer space to flits (or packets). That is, from among several flits or packets, which are concurrently waiting to be transmitted, the arbitration policy decides which flit (or packet) is to be transmitted in each cycle. Several different policies can be used for arbitration, e.g., round-robin or priority-driven policies. In this dissertation, we focus only on fixed-priority arbitration, that is, each flit of a message is attributed with a message-level fixed-priority. In each arbitration round, the flit (or packet) with higher-priority, precedes the flit with lower-priority in transmission.

2.1.2.4 *Routing*

Most NoC topologies, such as for instance the 2D-Mesh or 2D-Torus illustrated in Figure 2.5, allow for multiple distinct paths, i.e., an ordered sequence of links, from a source router to a destination router, which is referred to as *path diversity*. In this dissertation, we call a router that is connected to the source *source router* and the router that is connected to the destination core *destination router* analogously.

path diversity

source router

destination router

The purpose of routing, and the routing algorithm, is to determine the ordered sequence of links, which are to be traversed from a source router to a destination router

The purpose of routing, and the routing algorithm, is to determine the ordered sequence of links, which are to be traversed from a source router to a destination router. Routing algorithms determine the NoCs performance, fault-tolerance (due to loss or impairment of certain links), or deadline avoidance by, e.g., turn-based routing as will be explained later in this section.

A routing algorithm is considered *deterministic* if the algorithm always generates the same path, for any given source to destination router pair, and *non-deterministic* otherwise. While deterministic routing algorithms are simple to implement, they can not exploit path diversity, which may be exploited to reduce contention and distribute the injected traffic over all links in the NoC.

deterministic routing

non-deterministic routing

A comprehensive study of routing algorithms can be found, e.g., in [KI21] and only a brief description is given hereinafter. Fundamentally, routing algorithms can be categorized based on their means of implementation, i.e., *source*, *distributed*, and *algorithmic* routing. In *source routing*, the complete path of a packet is stored in the respective packet's header. Consequently, router implementation is simpler, since the routing overhead is offloaded to the packet itself. In turn, the effective bandwidth, which can be used to transmit payload, is decreased accordingly. In the *distributed routing*, all routing related information is stored inside a look-up table at each router. The look-up tables contain routing information, as to which packet – at which input port – is to be forwarded to what output port for transmission. The effective bandwidth of the transmission links is not impaired, however at the cost of increased memory requirements within each router. Lastly, in *algorithmic routing*, the routing information is stored implicitly in the algorithm, which is common for regular NoC topologies. One of the most prominent arithmetic routing algorithms is *dimension order routing*, in which each packet is routed to their destination router along one dimension before being routed along another dimension.

source routing

distributed routing

algorithmic routing

dimension order routing

Another important distinction is between *minimal path* and *non-minimal path* routing algorithms. A path is called minimal, if the number of links in the path equals the minimal hop count from the path's source router to the destination router in the given topology, and non-minimal otherwise. For instance in the illustrated $2D$-Mesh in Figure 2.5, the minimal distance of any source and destination router is given by the accumulated hop difference in x direction and y direction. Non-minimal paths increase the latency, due to additional routers being traversed. However, the path diversity can be exploited and crossing of paths can be circumvented, which in turn can reduce contention and thus transmission times. Similarly, if the algorithm adapts to the state of the network, it is referred to as an *adaptive routing* algorithm, which however necessitates precautions against live-locks to assure that the destination router is reachable.

minimal and non-minimal path routing algorithms

adaptive routing algorithm

In wormhole switched networks, deadlocks may occur if every transmission unit waits for a buffer or a link, which is already occupied by another transmission unit. One solution to avoid circular dependencies is to restrict the turns a transmission unit can take as proposed by Glass et al. [GN92]. Alternatively, Duato [Dua93; Dua94] proposed to use virtual channels to design adaptive deadlock-free routing, which allows to select any flit in the virtual channels to be transmitted. Routing algorithms have also been proposed for fault-tolerance such as, e.g., in [DA93; BC94; CB94].

2.1.2.5 *Analyzability*

Centralized arbitration is often used in predictable protocols such as CAN-Bus or Flexray

Real-time system design is concerned with the construction of systems, which can be formally verified to satisfy timeliness constraints, which requires analyses to verify if each hard real-time message (defined as an instance of a sporadic/periodic flow) can be successfully transmitted from its source to its destination before its deadline. Centralized arbitration is often used in predictable protocols such as CAN-Bus or Flexray, for which response-time analyses have been proposed, e.g., in [DKP+13; DN12; ASE12; TBE+16; TBE+13]. Approaches for real-time communication on a NoC apply one of the following general strategies. One is to utilize time-division-multiplexing (TDM) to ensure that the timing constraints are satisfied by constructing the transmission schedule statically with a repetitive table, e.g., in [GDR05; PK08; SMA+12; KSS+16; Sch; MNT+04; SAA+15; HFB+18]. An approach that is just recently considered for real-time NoC is deflection routing, which is an adaptive bufferless routing algorithm and has been proposed for real-time systems and analyzed by, e.g., [GN20; GNT21; GNT22]. In *deflection based routing*, a flit or packet is routed to non-productive directions at an upstream router and implicitly uses the network links as a buffer. Another approach is to apply a priority-based dynamic scheduling strategy in the routers to arbitrate the flits in the network, e.g., in [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; KP16; XLW+16; NIP16; IBN16; XWL+17; IBN18; NHE19]. The difficulty of the TDM strategy is to construct a feasible TDM schedule and the global clock synchronization, whilst the difficulty of the priority-based scheduling strategy is to validate the schedulability, i.e., whether all messages can meet their deadlines.

deflection based routing

With respect to fixed-priority based scheduling strategies, Table VII in [IBN16] summarizes the recent results for fixed-priority wormhole switched NoCs up to 2017. Eight of the ten results (namely, [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; NIP16; KP16; XLW+16; IBN16]) were already disproved by counter examples. These flaws in the literature *potentially* suggest that the scheduling algorithm and network architecture may be too complex to be correctly analyzed when adopting uniprocessor real-time scheduling theory.

For dynamic scheduling strategies, the wormhole switched fixed-priority NoC with preemptive virtual channels has been considered. The first attempts to tackle the schedulability analysis were in 1994 in [Mut94] and 1997 in [HO97]. Both of them were found to be flawed in 1998 by Kim et al. [KKH+98], whose analysis was later found to be erroneous in 2005 by Lu et al. [LJS05]. The series of erroneous analyses continued in [Mut94; HO97; KKH+98; LJS05]. Shi and Burns [SB08] published an analysis in 2008. Eight years later, Xiong et al. [XLW+16] pointed out that the analyses in [SB08] are unsafe in the sense that they do not consider limited buffer space and virtual channels. The proposed analysis by Xiong et al. [XLW+16] was later disproved by counter examples and fixed by the authors in their journal revision in [XWL+17] in 2017. In addition, Kashif et al. [KGP14] proposed stage-level analysis (SLA) to improve the analysis by Shi and Burns in [SB08]. The SLA in [KGP14] assumes an infinite buffer size. Kashif and Patel [KP16] extended the SLA analysis to cope with limited buffer size, which was disproved by Xiong et al. [XLW+16]. Indrusiak et al. [IBN16; IBN18] presented new analyses, but they "*chose to provide intuitions, insight and experimental evidence on the proposed analysis*

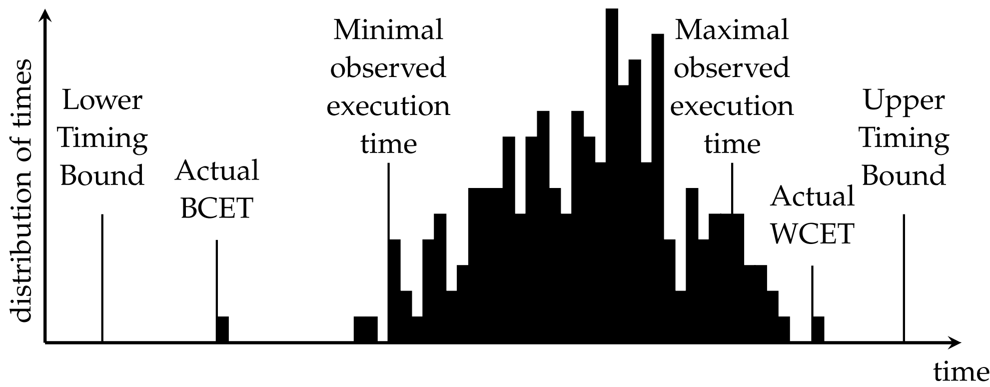

Figure 2.7: Symbolic representation of all execution times in a system and the corresponding terminology with regard to worst-case execution time analysis adapted and redrawn from [WEE+08].

and its improvements, rather than theorems or proofs." They supported their analyses by evaluating concrete cases, i.e., whether there was any observed case which was claimed to meet the deadlines but in fact missed the deadlines. However, such case studies cannot validate the correctness of their analyses, as also stated by Indrusiak et al. [IBN16; IBN18]. Addressing the problem of complex buffering effects at the upstream router with regards to the worst-case transmission latency analysis, Burns et al. [BIS+20] recently proposed a flow control mechanism that avoids such multi-point progressive blocking by construction.

2.2 TIMING ANALYSIS

In this section, the most fundamental concepts of worst-case execution time analysis are introduced. In order to be able to assert timing correctness from formal schedulability analyses in real-time systems, the task models, which all consist of at least the inter-arrival time, worst-case execution time, and deadline, need to be parameterized. The subsequent schedulability analyses must verify if the given parameterized task set can be feasibly scheduled on the given platform [Wil20]. It is evident that the asserted timing correctness depends on the fidelity of the parameters. While robust parameter estimations for inter-arrival times and deadlines can be easily determined by only accounting for clock jitter; determining the amount of execution time of a task remains a very challenging problem. With respect to industry grade systems on modern architectures, Davis et al., even assert that *"comprehensive solutions [for sound timing analyses] are currently tantalizingly out-of-reach"* [DC19].

timing analysis

The term *timing analysis* refers to the problem of characterizing the amount of time that each task (program) can take to execute on a given hardware platform, and is usually obtained as an upper-bound, or estimate, of the actual worst-case execution time, which can occur in the real system. The most relevant terms in timing analysis are illustrated in Figure 2.7, in which all possible execution times of a single task are summarized in a histogram. The actual WCET refers to the worst-case execution time, which can actually occur in the real system, whereas an upper timing bound (WCET estimate) must not refer to a possible

execution time. The same holds for the best-case execution time (BCET) and the lower timing bound (BCET estimate). Additional terms are *minimal- and maximal observed execution times* that refer to a set of (non-exhaustive) measured traces of the real system. A worst-case execution time estimation is called sound if no execution in the real system will ever exceed the estimated worst-case execution time (upper timing bound) and the term precision refers to the ratio of an actual WCET and the estimated worst-case execution time.

minimal- & maximal observed execution times

worst-case execution time estimation is called sound if no execution in the real system will ever exceed the estimated worst-case execution time

Timing analysis is applied to time-critical and safety-critical embedded-system software in problem aware parts of the embedded-systems industry that are subjected by prescription to comply with international safety norms, e.g., DO-178B/C, DO-254, IEC 61508, and ISO 26262. In academia and industry, two approaches are used to estimate the worst-case execution time of a task, namely either as a single *deterministic worst-case execution time* (WCET) value or a probability distribution of the *probabilistic worst-case execution time* (pWCET). Based on the complementary cumulative distribution function (CCDF) of the pWCET distribution, a worst-case execution time estimate – that is exceeded only by probability in the range of e.g., 10^{-9} to 10^{-12} – can be obtained, which is in the same order of magnitude as other dependability estimates. In the context of timing analysis, static analysis techniques, measurement-based or hybrid techniques are used in academia and industry [DC19]. It was reported by Akesson et al. [ANN+22] that automotive (40.57%), avionics (28.3%), industrial automation and manufacturing (13.21%), and defense (13.21%) industry predominantly use measurement-based timing analyses as reported by 76% of the participating respondents. In contrast 38% of the participants use static analysis to obtain a worst-case execution time and 26% of the respondents use both techniques.

deterministic worst-case execution time

probabilistic worst-case execution time

industry predominantly use measurement-based timing analyses

In the subsequent Section 2.2.1, the fundamentals of the state of the art in deterministic timing analysis are presented, and the challenges with respect to modern multicore computer architectures are discussed. Thereafter, in Section 2.2.2, some selected fundamental results in the more recent research in probabilistic timing analysis are presented.

2.2.1 DETERMINISTIC TIMING ANALYSIS

The first sound worst-case execution time estimation for select industry grade systems has been achieved by the static analysis (STA) approach developed by the research group of Wilhelm, e.g., [WLP+12; WW08; RGB+07; FHW04]. Static analyses *"[...] search for a longest path in the state space spanned by the program under analysis and by the architectural platform"* [WPG+21] and consequently relies on timing models obtained from micro-architectural analysis of the platform and analysis of the platform's state space evolution. The research group around Wilhelm developed powerful abstractions of the system states especially for least-recently used (LRU) caches with the introduction of may- and must analyses [AFM+96], allowing to reduce the state space significantly. In the *spanned state space*, tree-based approaches as in [CB02; BB06a], or path based approaches [TFW00; SEE01] have been proposed to calculate the longest path in the state space according to [BCP03]. This approach forms the de-facto standard approach for any static

This approach forms the de-facto standard approach for any static analysis tool to date, which is exemplified in Figure 2.8

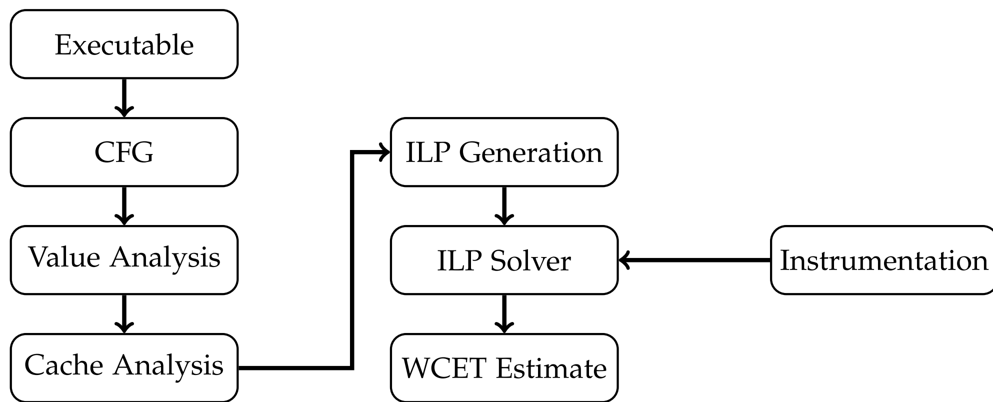

Figure 2.8: Flow chart of the standard approach to static timing analysis as used for instance by the tool AbsInt.

analysis tool to date, which is exemplified in Figure 2.8. According to [RS09], control-flow reconstruction and static analyses for control- and data flow, micro-architectural analysis, which computes upper- and lower-bounds on execution times of basic blocks, and global bound analysis, which computes upper- and lower-bounds for the whole program, are the basics steps in static worst-case execution time analysis.

The stages of an exemplary static timing analysis workflow are shown in Figure 2.8, and each of the stages is briefly elaborated in the following paragraphs. A comprehensive survey of worst-case execution time analysis and in particular static cache analysis can be found in [WEE+08] and [LGR+16] respectively, which also serve as the foundation of content of the following paragraphs – with enrichments as necessary.

1) Control-Flow Graph Reconstruction. In the first stage, an input binary code of the program is analyzed to retrieve a high-level description of the program in form of an inter-procedural control-flow graph. In the input binary code, the memory locations of instructions and program variables are already determined, which is essential for cache analysis. A *control-flow graph* is a directed graph, where each vertex represents a *basic block*, which is a sequence of instructions with no branching instructions along the path, and edges representing the control-flow in between basic blocks [LGR+16]. The control-flow graph is then used in the subsequent stages of value and cache analysis.

control-flow graph
basic block

2) Value Analysis. Value analysis is a static analysis with the objective to infer the contents of a processor's register contents, and the contents of main memory, for every instruction and execution context. The memory contents are not always attributed with exact values, but intervals that represent a set of concrete values. Each instruction in the program is modeled by an appropriate *transfer function*, which maps abstract input states to abstract output states. At control-flow joins, the incoming abstract states are combined into a single outgoing state, using a combination function. Due to loops in the program, and thus in the control-flow graph, respectively, the transfer and combination functions are applied repeatedly until the abstract states converge to a fix-point.

value analysis
transfer function

3) Cache Analysis. Caches are small on-chip memory, which are smaller in size

cache analysis

compared to main memory, but provide significantly faster access latency. Caches contain a small subset of the main memory contents, which are likely to be referenced in a sequence of subsequent executed instructions by exploiting temporal and spatial locality of memory references. If during an instruction, memory is referenced that resides in the cache then this is called a cache hit and the data is fetched from the cache with low latency. On the contrary, if the referenced memory does not reside in the cache then this is called a cache miss and the data must be loaded into the cache from main memory and fetched from the cache thereafter. The latency in case of a cache miss are reported to be magnitudes larger than in case of a cache hit. It is evident, that a worst-case execution time analysis, which has to assume that every memory reference results in a cache miss, is too pessimistic, resulting in low estimation precision. With regards to computational complexity, it is infeasible to collect the set of all possible concrete cache states, i.e., cache contents, at every basic block in the control-flow graph, depending on the visited basic blocks before. To that end, abstract *must and may cache states* have been proposed, which represent a set of concrete cache states, which must (or may) contain certain memory blocks. Abstract must cache states compactly represent the respective sets of concrete cache states, in which specific memory blocks will surely be in. Consequently it is possible to determine, if a request to a memory block will result in a cache hit, given an abstract must cache state at a basic block in the control-flow graph. Analogously, abstract may cache states represent the respective sets of concrete cache states, in which specific memory block may be in, and consequently a cache miss can be asserted if a memory block is not present in a may cache state. At *control-flow joins*, abstract must cache states can be combined by sort of an abstract intersection and abstract may cache states can be combined by sort of an abstract union operator. In the baseline cache analysis, it is assumed that the program executes in isolation without interruptions and thus in the context of preemptive scheduling, *cache-related preemption delay* (CRPD) must be taken into account as reported in e.g., [AM11].

must & may cache states

control-flow joins

cache-related preemption delay

4) Path Analysis. According to [WEE+08] there are three main classes for computing a worst-case execution time estimate, namely *structure-based* (tree-based) [CP00; CB02], *path-based* [HAM+99; SA00; SEE01], and *implicit path enumeration* techniques, which was originally proposed in [LM95] and further refined to account for more complex control flows by others. After the micro-architectural analysis – which determines the worst-case execution times of each basic block – is completed, the program's worst-case execution time estimate is obtained by finding the longest path in the control-flow graph. During value analysis, loop and recursion bounds, are obtained or instrumented by the expert, allowing for a finite state space and thus program termination. In the most prevalent path enumeration technique, flow constraints are formulated, which associate the internal structure of the control-flow graph with the execution count of the basic blocks. That is, each basic block v_i is associated with a counting variable $x_i \in \mathbb{N} \cup \{0\}$, denoting that v_i is executed x_i times, and hence the ILP is given by $WCET \leq \max \sum_{v_i \in V} x_i \cdot c_i$ subject to the loop, flow, and recursion bounds.

implicit path enumeration

While deterministic static analyses are the only known approach to obtain sound estimates, the predictability of modern architectural platforms poses a significant challenge to the precision of the estimated WCET compared to an

actual WCET. This problem is further exacerbated by multiprocessor architectures and advanced processor features.

2.2.2 PROBABILISTIC TIMING ANALYSIS

Modern architectures are designed to improve the average case performance, which is achieved by advanced processor features such as cache hierarchies, branch prediction, out-of-order pipelines, and communication such as network-on-chip. The drawback of these advanced features, with respect to worst-case execution time analysis, are the high execution time variability which are due to the huge possible hardware state space, and the required model complexity for a deterministic static analysis. For instance, in view of timing anomalies, no efficient abstractions for pipeline states could have been found, resulting in the need to maintain large sets of pipeline states to follow all possible cases, whenever several successor states are possible. While this approach could be achieved for single-core processors at an acceptable level, this is not possible anymore for multi-core systems [WR12].

in view of timing anomalies, no efficient abstractions for pipeline states could have been found, resulting in the need to maintain large sets of pipeline states

According to [DC19], probabilistic timing analyses can be broadly classified into static probabilistic timing analyses, which are similar to static analyses of deterministic timing analysis with the difference that parts of the micro-architecture, program or input states are modeled by probability distributions; Measurement-based probabilistic timing analyses, which use representative measurements of program execution times, to estimate a probabilistic worst-case execution time (pWCET) distribution; and hybrid approaches. A comprehensive survey of probabilistic timing analyses is provided by Davis et al. in [DC19] and only focus this section on measurement-based probabilistic timing analyses. *Measurement-based probabilistic timing analyses* (MBPTA) approaches aim to make a statistical estimate of the pWCET distribution of a program based on *execution time* (ET) measurements, obtained by executing the program either on hardware, or a cycle accurate simulator, according to a measurement protocol.

probabilistic worst-case execution time

measurement-based probabilistic timing analyses

Definition 2.1 (Operation Context adapted from [DC19])**.** *An operation context is defined as an infinitely repeating sequence of input states and initial hardware states, characterizing a feasible way, in which recurrent execution of the program may occur.*

For a concrete operation context, a probabilistic execution time distribution can be obtained. Consequently, the pWCET is given by the envelope of all probabilistic execution time distributions for any valid operation context. More commonly, the complementary cumulative distribution is used to bound the probability that the worst-case execution time exceeds a threshold value. Most proposed research uses Extreme Value Theory (EVT) to make a statistical pWCET estimate to calculate tail probabilities e.g., [EB01; BE00; CSH+12; HHM09; SMD+14]. The EVT-based pWCET research is based on the Fisher-Tippett-Gnedenko and Pickands-Balkema-de Haan theorem with the associated block-maxima and peaks-over-threshold statistical estimators. As reported in [DC19], mainly two approaches of EVT analysis to program execution times exist, namely *per-path* and *per-program* analysis. Per-Path analysis is applied at the level of program paths, i.e., all feasible

per-path & per-program analysis

program paths are executed and the observed execution times are associated with the executed path. The EVT analysis is conducted for each program path to estimate a path specific pWCET distribution. Subsequently, the program pWCET distribution is estimated by taking a point-wise upper bound of all path specific pWCET distributions. Secondly, the per-program analysis executes all program paths but estimates the pWCET distribution irrespective of the specific paths.

markov inequality with power-of-k functions to estimate the pWCET based on the k-th moment statistical estimators

Recently, Vilardell et al. [VSM+22] proposed to use the markov inequality with power-of-k functions to estimate the pWCET based on the k-th moment statistical estimators. The main improvement of this new approach is that – unlike EVT – no specific type of tail distribution must be selected and tested for quality [APC+17; ASO+20; RSF20]. In consequence, the markov inequality based approach does not suffer from model uncertainty. However, the approach may require a lot of drawn samples in order to produce safe and precise estimates of the k-th order moments of the underlying distribution for larger values of k [VSM+22].

EVT made strong assumptions that the execution times are independent and identical distributed

As a fact, extreme value theory analysis presumes a continuous distribution, which assumption was shown to be false by Lima et al. [LDB16] and Griffin et al. [GB10b] for some execution time distributions of programs. The authors provided concrete programs, which showed discrete execution time distributions and discussed that the continuity assumption of EVT can not be presumed a priori. Even more so, early results of EVT made strong assumptions that the execution times are independent and identical distributed, which is not the case [GB10b]. Later results showed that EVT can also be used for execution time data, which are stationary and preserve extremal independence [SMD+14]. In pursuit to comply with the assumption of independent and identical data, which is required for trustworthy EVT analysis, Lima et al. [LB17] proposed a randomized sampling method.

The main challenge for either measurement-base methods is the measurement protocol and in particular the representativity

The main challenge for either measurement-base methods is the measurement protocol and in particular the representativity of the evaluated time-limited operation contexts (cf. Definition 2.1) for the operation contexts, which can occur during operation. The problem of representativity can be eased by identifying equivalent input states and hardware sates with respect to execution time characteristics. A general solution for the representativity issue is however still an open problem.

2.3 REAL-TIME SCHEDULING

A scheduling algorithm and the eventual implemented scheduler in a real-time operating system decides, which job is to be scheduled, and executed on the processors, respectively. Whereas real-time scheduling theory is concerned with the design and analysis of scheduling algorithms, which facilitate the formal verification of the temporal behaviour of the system, the scheduler implementation is concerned with overhead considerations.

In this section, the most important aspects of real-time scheduling theory, with respect to this dissertation, are presented. To address practical concerns, the first Section 2.3.1 gives a brief introduction into scheduler implementation and sources of overheads. Other practical concerns and implications of different scheduling paradigms are described alongside the theory presentation. In Section 2.3.2, the

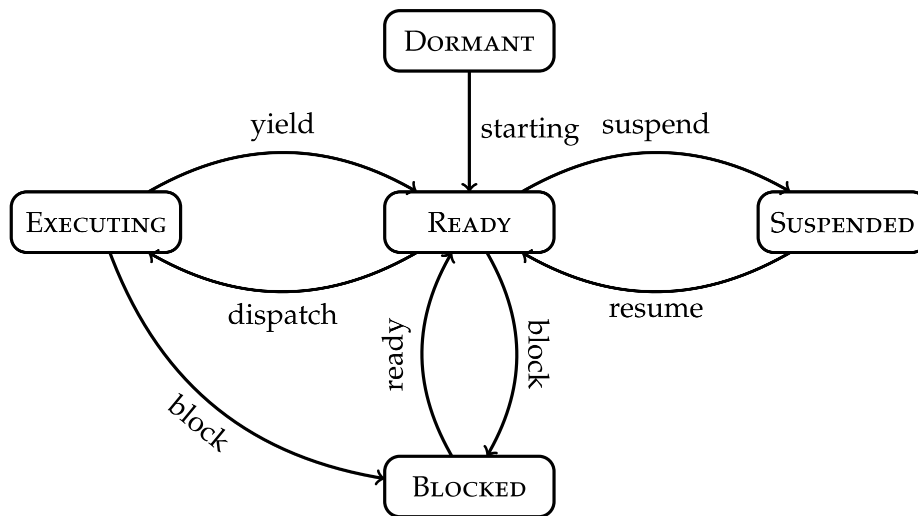

Figure 2.9: Symbolic example task states combined from RTEMS and FreeRTOS specifications.

task models, and system models, as used in the analyses are presented. In Section 2.3.3, scheduling algorithms are explained and classified. Afterwards, in Section 2.3.4, the fundamental techniques and challenges in schedulability analysis are explained. Lastly, the empirical and theoretical performance evaluation of scheduling algorithms and schedulability tests is elaborated in Section 2.3.5.

2.3.1 SCHEDULER IN THE RTOS

From here, a brief introduction of scheduling algorithm implementations and related overheads are discussed, to motivate certain classes of scheduling algorithms for practical use.

Essentially, a scheduler is an operating system level program, which is responsible to *schedule*, that is, to determine which process is to be executed on which processor, and to *dispatch* the currently running process and prepare the system to execute the scheduled process. The dispatcher is responsible to *dispatch* the currently *attached* job (attached because the job is preempted and not executing anymore) and to prepare the next to be scheduled job to commence execution. The dispatcher is preceded by a *context switch*, in which the execution context of the attached job is saved to be resumed at a later point in time; and to initialize the context of the scheduled job. The granularity of a scheduling algorithm is an integral *system tick* (or *tick* for short), which represents the time resolution of the system and is determined by programmable hardware timers in the system, which periodically generate so called interrupts. Interrupts notify the processors of asynchronous events and may occur between (almost) any two instructions [Tan09]. Whenever an interrupt is triggered, the processor halts the normal thread of execution, switches into an architecture dependent privileged mode, and a designated *interrupt service routine* (ISR) is executed. The ISR, responsible to handle the timer interrupt, calls the *sys tick handler*, which invokes the scheduler. According to [Bra11b], some interrupts in multiprocessor systems are local to a specific

dispatcher

context switch

system tick

interrupts

Interrupts notify the processors of asynchronous events and may occur between (almost) any two instructions

interrupt service routine

processor, whereas others may be serviced by multiple or all processors. Specific to multiprocessor systems, software generated *inter-processor interrupts* (IPIs) are used by the real-time operating systems in order to synchronize state changes across processors, e.g., to cause a remote processor to reschedule.

inter-processor interrupts

In scheduling theory, the subjected entity is called tasks and instances thereof are called jobs. In contrast, in the operating system context, a process corresponds with a job, whereas the program corresponds with a task. A process is defined as an *instance of an executing program* [Tan09] and includes the context, e.g., the program counter, register values, access rights, page tables, meta data, such as cpu time consumption, or locked resources, and others.

in the operating system context, a process corresponds with a job

the program corresponds with a task

All software on the computer, is organized into a number of processes, which groups related resources together; the address space consists of a program, text, and data sections [Tan09]. Central to the concept of processes is, that two processes can be switched by the operating system. Particularly, a process context can be stored when the process execution is preempted and resumed at a later point in time, in the prior state. Related to the concept of processes are *threads*, which exist within a process, and thus can be managed more easily. The context of a thread consists of a program counter, registers (which hold the current working variables), and stack (which contains the stack frames of procedure calls). Despite threads being grouped into a process, the threads can be scheduled largely independent of one another, with the benefit that they share the same address space and can share data very efficiently [Tan09].

threads

threads can be scheduled largely independent of one another, with the benefit that they share the same address space and can share data very efficiently

The real-time operating system maintains a table, which has an entry for each process – sometimes called the process control block (PCB). Each entry in the PCB contains information about the state, context, and scheduling information such as, e.g., the *time executed*, *absolute deadline*, or the priority.

With respect to a processes' state in a real-time operating system, Figure 2.9 shows an exemplary symbolic state space, which is combined from the open source real-time operating systems RTEMS and FreeRTOS. If a process is in the *ready* state, then it is queued into the *ready queue* and can be selected to be executed by the scheduler; leading the task into the *executing* state. When a process is blocked or suspended, the process is not yet finished but can not be scheduled temporarily. The difference between both states is; a process is blocked if there is some external reason for it not to be restarted, such as a locked semaphore; whereas suspension means that the operating system has suspended the process, but it could be resumed at any point in time.

ready queue

blocked

suspension

A key term in preemptive scheduling is *preemption*, which refers to the temporarily and non-voluntarily interruption of a not yet finished job execution, initiated by the scheduler. Preemption is one of the primary causes of overhead, due to the context switch-, dispatch-, and access times to the scheduler's data structures. Moreover, in shared memory multiprocessor systems, a process can migrate from one processor to another. The overhead of migration is architecture dependent, namely in shared memory systems only the processor and hardware state such as register contents must be migrated, since the data is still accessible from all processors. In contrast, in distributed memory systems, all data must be migrated along with the hardware state to the memory of the other processor,

preemption

Preemption is one of the primary causes of overhead, due to the context switch-, dispatch-, and access times to the scheduler's data structures

which creates load on the communication bus. As a conclusion, global scheduling algorithms are only feasible for shared memory systems. Another consideration is, that if a thread is able to execute on the same processor for an extended amount of time, then the cache is filled with memory blocks of that thread (called *cache affinity*), leading to shorter execution times and thus lesser overheads. For that reason, real-time operating systems often employ partitioned scheduling to retain cache affinity and achieve reduced overheads.

global scheduling algorithms are only feasible for shared memory systems

cache affinity

2.3.2 Task & System Models

In order to facility the analysis of timing constraints, a formal model must be constructed, which abstracts the relevant system characteristics at a sufficient degree of fidelity.

Starting from the Liu and Layland task model [LL73], many new task models and extensions have been proposed to characterize more flexible job release and execution time patterns. These, for example, include the sporadic real-time task model [Mok83], the self-suspending task model [Raj91], the multiframe model [MC97], the generalized multiframe (GMF) model [BCG+99], the digraph model [SEG+11], elastic task model [BLA98], and the variable rate-dependent behaviour (VRB) model [DFP+14]. With the use of multiprocessor systems, new task models, allowing to express inherent parallelism, such as the bundled task model [WP19], gang task model [GR16], the parallel synchronous task model [LKR10], or the sporadic directed-acyclic graph (DAG) [BBM+12], and conditional DAG task model [BBM15], have been proposed or extended to the particular requirements of real-time systems. The relevant DAG, conditional DAG task, and gang task model are presented in detail in Chapter 3 and Chapter 4. We hence only focus on the commonalities and foundations in the remainder of this section.

new task models and extensions have been proposed to characterize more flexible job release and execution time patterns

2.3.2.1 *General Task Models*

Common to all task models, is a description (or classification) of feasible job activation sequences, the workload they incur, and deadlines. If not stated otherwise, we assume that all task parameters considered in the underlying model are predetermined, i.e., they are known both at design time and at runtime, and that tasks will always respect their parameters.

A task model consists of an activation model and a model of the generated workload. Activation of a task means the times at which instances of a task, so called jobs, are released to the system.

These activations can be either *time-triggered* or *event-triggered*, which refer to either; an activation by a timer in regular time intervals, or an activation by an event, e.g., the arrival of a new sensor sample. Depending on the regularity of either triggers, the activations are called *periodic* $\subseteq$ *sporadic* $\subseteq$ *aperiodic*.

time-triggered

event-triggered

periodic activation

sporadic activation

aperiodic activation

To exemplify, let a_i^ℓ denote the ℓ-th activation of a task τ_i, then if the activation is periodic with period T_i then $a_i^{\ell+1} = a_i^\ell + T_i$, i.e., the next activation is exactly T_i time units apart. In the sporadic activation model, the time interval in between

two activations is lower bounded by T_i, i.e., $a_i^{\ell+1} \geq a_i^\ell + T_i$. This model is suitable for *event-triggered* tasks, for which it is uncertain when events will occur, but the activation frequency is limited. The sporadic model can be enhanced by an upper-bound, i.e., $a_i^\ell + T_i^{max} \geq a_i^{\ell+1} \geq a_i^\ell + T_i^{min}$, which is necessary for end-to-end response-time analyses. If a task has no regularity with respect to its activation, it is called *aperiodic*. A special case of event-triggered activations are precedence constraints, in which jobs have dependencies. That is, a job of a task is released, only if all *preceding* jobs have finished.

implicit deadline
constrained deadline
arbitrary deadline

In general, a real-time task τ is associated with a per-task relative-deadline D_τ such that each job J of that task has to finish execution before the absolute deadline, i.e., $f_J \leq a_J + D_\tau$. Depending on the relation of the relative deadline to the period, the deadline models are classified into *implicit* $\subseteq$ *constrained* $\subseteq$ *arbitrary* deadline tasks. In the implicit model, the relative deadline coincides with the period, specifying that the only constraint is that a job J needs to be finished before the next one is released; whereas in constrained deadlines, the deadline is less than the period $D_\tau \leq T_\tau$. The most general case is arbitrary deadlines, which imposes no restrictions on the relationship of deadline and period of a task. Most notably, in the arbitrary-deadline case there may be multiple unfinished jobs of the same task waiting at the same time without deadline violation.

2.3.2.2 *Self-Suspending Task Models*

Suspension allows to lessen resource contention and wasted cycles

With regards to Figure 2.9, self-suspension refers to the behaviour of an *active* job to *suspend* itself, i.e., to be voluntarily excempted from the scheduling process for the duration of suspension. Self-suspension is motivated by the problem of *active idling* or *polling* as e.g., in offloading scenarios, jobs that perform IO operations incur a lot of waiting. Suspension allows to lessen resource contention and wasted cycles, since the waiting time can be consumed by other jobs to process actual workload.

dynamic self-suspension
segmented self-suspension

In the self-suspension survey literature [CNH+19], two fundamental task models have been identified, namely the *dynamic self-suspension* task model and the *segmented self-suspension* task model. The former dynamic self-suspension model refines the underlying task model of τ_i by an additional parameter $S_i \in \mathbb{R}_{\geq 0}$, which describes the maximal cumulative amount of time that any job of task τ_i can spend in the suspended state, during that job's active interval. This task model is challenging for real-time analyses, in the sense that arbitrary different, suspension and resumption patterns can be generated under the parameter constraint; all of which must be considered in an exact schedulability test. In turn, this suspension model is robust, since no information about suspension patterns is required and any reduction in suspension time can only improve the worst-case response time (as the number of suspension and resumption patterns is only decreased).

The other fundamental model is the *segmented self-suspension model*, in which an underlying task model τ_i is refined by a tuple $(C_{i_1}, S_{i_1}, \ldots, S_{i_n}, C_{i_n})$, where $\sum_{j=1}^n C_{i_j} = C_i$ and $\sum_{j=1}^n S_{i_j} = S_i$. This refinement, constraints the suspension and resumption patterns, which can be generated by any job of that task. That is, after the execution of the first segment, which can execute for at most C_{i_1} time units, the job suspends for at most S_{i_1} amount of time; the procedure is perpetuated until

the last segment finished execution. As has been shown in, e.g., [CHH+19] the more restrictive suspension and resumption pattern can improve the worst-case response time analysis precision. In turn, the model requires more information about the control-flow of the job, which makes it less general. In this dissertation, only the dynamic self-suspension model is considered.

2.3.2.3 *System Model*

In this section, a formal description and the interrelation of all the mentioned models are presented in the unified notation proposed in [Che].

Jobs. A feasible job collection $\mathbb{FJ}$ of a task set $\mathbb{T}$ denotes the universe of all possible job collections, which can be generated under the restrictions of the task model, and the parameters described in the concrete task set. Hence, a job collection $\mathbb{J} \in \mathbb{FJ}$ denotes a concrete set of jobs generated from tasks in $\mathbb{T}$. For a specific task $\tau_i \in \mathbb{T}$, the feasible job collection $\mathbb{FJ}_i$ denotes the universe of all possible job collections that can be generated under the restrictions of the task model and the parameters of τ_i. For instance, if the task set consists only of a single synchronous periodic task with period T then there is only a single job collection $\mathbb{J} \in \mathbb{FJ}_i$, which consists of an infinite sequence of jobs which are released at time $0, T, 2T, \ldots$ with absolute deadlines at time $d, d + T, \ldots$. In contrast, there are infinitely many different job collections for a single sporadic task.

feasible job collection

job collection

Schedule. In multiprocessor platforms, a schedule is a time and processor multiplex to the given jobs, such that each job is executed until completion. More formally, a schedule for a set of jobs $\mathbb{J}$ is a mapping $S : \mathbb{R}, \mathbb{N} \mapsto \mathbb{J} \cup \{\bot\}$

$$S(t, m) = \begin{cases} J & \text{if job } J \in \mathbb{J} \text{ is executed on processor } P_m \text{ at time } t \\ \bot & \text{if processor } P_m \text{ is idle} \end{cases}$$

Together with a schedule S for a job collection $\mathbb{J}$, the jobs $J \in \mathbb{J}$ are refined with their respective *starting time* s_J (which is the first time that this job is executed), finishing time f_J, and execution time C_J, in that concrete schedule. Hence, for a concrete schedule S, the *response-time* of a job J is defined as $f_J - a_J$, the *lateness* at time t is defined as $t - d_J$, and the *tardiness* at time t is defined as $\max\{0, t - d_J\}$.

starting time

response-time

lateness

tardiness

Execution Model. The response-time of a job is determined by the processing and execution of that job on the computing system, which implicitly assumes a model of how the workload is executed. To that end, the *execution model* formally describes how the workload of each job is executed during any interval of length t. To clarify, in the context of processor scheduling, e.g. uniprocessor and homogeneous multiprocessor systems, it is presumed that if a job J is executed in a schedule S during an interval $[t, t + \Delta]$ then the workload is reduced by Δ amount of time. Clearly, if the processor speeds were to be increased or to be decreased, the workload reduction would change accordingly.

execution model

With reference to the formal description, we have that for each processor P_i for $i \in \{1, \ldots, m\}$, with relative processor speed $s \in \mathbb{R}_{>0}$, the following constraint holds for each job collection $\mathbb{J} \in \mathbb{FJ}$

$$\sum_{J \in \mathbb{J}} \left(\int_{v \in [t, t+\Delta]} \mathbb{1}_{S(v,m)=J} \, dv \right) \leq \Delta \cdot s \tag{2.1}$$

where $\mathbb{1}_{S(v,m)=J}$ denotes the indicator function that evaluates to 1 whenever the predicate $S(v, m) = J$ is true for some time $v \in [t, t + \Delta]$.

The specification of an *execution model* is of particular importance in more distributed systems such as for the interleaved and pipelined transmission in network-on-chips. In a network-on-chip, a single message may be distributed over multiple links in the network, each of which can transmit simultaneously. This execution model is orthogonal to the uniprocessor execution model and must be considered in the formal analysis as is considered, e.g., in network calculus [Bou98].

Using the introduced formalism, we can describe the cumulative execution time of a job J in a schedule S during some time interval, that is;

$$exe(S, J, a, b) := \sum_{m=1}^{M} \int_{v \in [a,b]} \mathbb{1}_{S(v,m)=J} \, dv \tag{2.2}$$

a DAG job is formally decomposed into a set of subjobs which are bound to sequential execution

Considering parallel execution of a job, e.g., a DAG job, or gang job; a job can execute on more than one processor simultaneously. However, usually a DAG job is formally decomposed into a set of subjobs which are bound to sequential execution. For instance, in rigid gang scheduling, a job J is composed of subjobs $J_1, \ldots, J_{E_i}$ and each subjob is subjected to the constraint that all of the subjobs have to execute simultaneously, i.e.,

$$\exists m_1 \; S(m_1, t) = J_1 \iff \exists m_2 \; S(m_2, t) = J_2 \ldots \iff \exists m_{E_i} \; S(m_{E_i}, t) = J_{E_i} \tag{2.3}$$

Depending on the scheduling paradigm and the task model, the scheduling function and job collection is imposed with further constraints. Overheads can be integrated with the schedule function, but are neglected in this presentation, since overheads, and context-switch related delays are not considered in the analyses in this dissertation on the formal level.

feasibly schedulable

Schedulability. Based on the presented formalism, the schedulability problem can be stated as follows. The job collection $\mathbb{J}$ is said to be *feasibly schedulable* by a scheduling algorithm $\mathbb{A}$ if the (for $\mathbb{J}$) generated schedule S using scheduling algorithm $\mathbb{A}$ is always feasible under the task model constraints, e.g., that the execution time of a job is never more than the respective task's worst-case execution time or others such as precedence constraints. Consequently, a task set $\mathbb{T}$ is feasibly schedulable by a scheduling algorithm $\mathbb{A}$ if each job collection $\mathbb{J} \in \mathbb{FJ}$ is feasibly schedulable according to the above description.

The feasibility of a schedule depends on the evaluated metric, e.g., recurring back to lateness, and tardiness; a schedule is feasible with respect to deadline constraints if $f_J \leq d_J$ (or the tardiness is 0, equivalently). The meaning of feasibility

is explicitly stated in the respective chapters in this dissertation. Based on the per-job perspective, the following metrics of a generated schedule S for a job collection $\mathbb{J}$ can be stated as *maximum lateness* as $\max_{J \in \mathbb{J}} \{f_J - d_J\}$, and the *makespan* as follows $\max_{J \in \mathbb{J}} f_J - \min_{J \in \mathbb{J}} a_J$.

maximum lateness
makespan

2.3.3 SCHEDULING ALGORITHMS

In this section, we discuss uniprocessor and multiprocessor scheduling algorithms with an emphasis on multiprocessor scheduling, due to the focus of this dissertation.

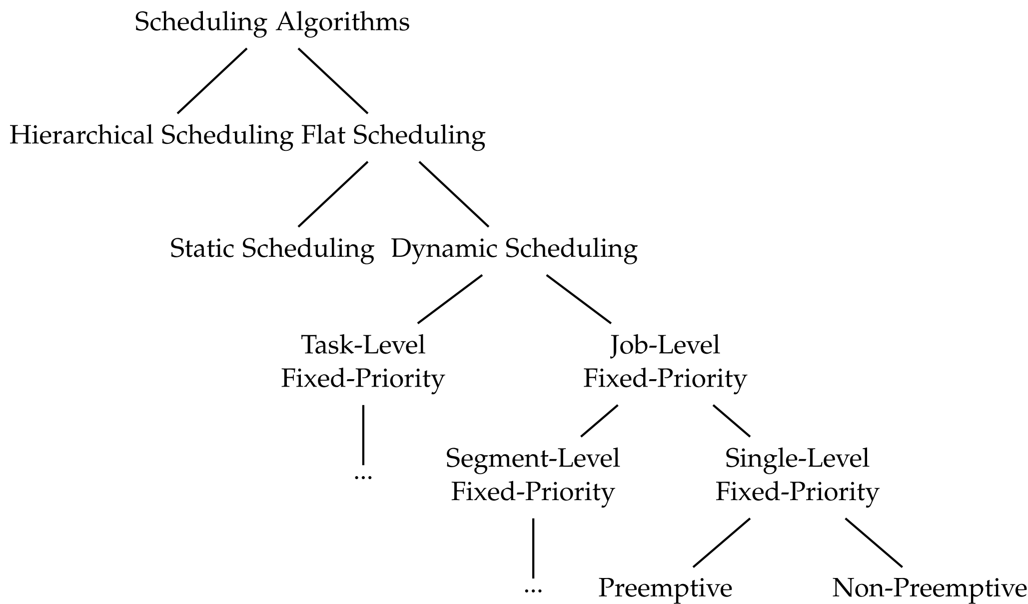

Figure 2.10: Taxonomy of scheduling algorithms as relevant to this dissertation.

2.3.3.1 *General Classification*

Scheduling algorithms can be distinguished on the basis of the taxonomy illustrated in Figure 2.10. Scheduling algorithms can be classified into *hierarchical scheduling algorithms* and *flat scheduling algorithms*.

In *hierarchical scheduling*, the scheduling problem is separated into typically (but not limited to) two different hierarchy levels of scheduling. On the lowest level, a reservation system, which can for instance be implemented as a group of threads, are scheduled on the physical processors by some scheduling algorithm. On the higher level, another scheduling algorithm is used to schedule the workload, which is attached to the reservation system, onto the provided service. Hierarchical scheduling provides temporal and spatial isolation, since the execution behaviour of the attached workload only affects the schedule within the reservation system, but not the overall schedule.

hierarchical scheduling

temporal and spatial isolation

Motivated by this isolation property, a plethora of hierarchical scheduling algorithms have been proposed for sequential tasks, e.g., the *polling server*, *deferrable*

polling server

deferrable server

server [SLS95] [ZGD11; CGJ+22], *total bandwidth server* [WWL00] with extensions to multiprocessor systems in [BL04; KY08], and *constant bandwidth server*, e.g., in [BB06b; CAB+17]. In contrast, the hierarchical scheduling approach for parallel DAG tasks is comparatively understudied [BBW10; UBC+18; UGC21].

flat scheduling algorithm

In contrast, in a *flat scheduling algorithm*, the workload is directly scheduled on the physical processors. Owing to the fact, that hierarchical and flat scheduling are based on similar scheduling algorithms, the presentation in the remainder of this section is only concerned with flat scheduling algorithms.

work-conserving
priority-driven

The first scheduling algorithms in real-time systems have been static table-driven algorithms, in which an offline computed table was used to generate cyclically repeating schedules. In contrast, dynamic scheduling algorithms generate the schedule dynamically on the basis of *work-conserving* and *priority-driven* decision making.

non-work-conserving

The scheduling algorithm can be *work-conserving* or *non-work-conserving*. A schedule is *work-conserving*, if no processor is idle, given the condition that there are at least as many jobs in the ready queue as there are processors. Oppositely, a schedule is *non-work-conserving* if it does not satisfy the work-conserving property. Despite the fact, that work-conserving scheduling algorithms are not always optimal, as is the case, for instance, in non-preemptive scheduling [NF16], most scheduling algorithms found in modern real-time operating systems, are work-conserving. Moreover, work-conserving is a greedy strategy, which is very suitable for online scheduling.

task-level fixed-priority
job-level fixed-priority
segment-level fixed-priority
deadline-monotonic
rate-monotonic

In preemptive priority-driven scheduling algorithms, the available processors are dedicated to the highest-priority active jobs at any time, i.e., a higher-priority job can preempt a lower-priority job. In contrast, in non-preemptive priority driven scheduling, a job runs to completion once it started execution and hence re-scheduling only occurs when some processor idles. Depending on the level on which the priority is determined, the scheduling algorithms can be categorized into *task-level-*, *job-level-*, and *segment-level fixed-priority* policies. In task-level fixed priority, each job which is released by the same task, is assigned the same fixed-priority. Prominent examples are *deadline-monotonic* (DM), or *rate-monotonic* (RM) priority assignment, in which tasks with smaller deadline (period, respectively) have a higher priority. In job-level fixed priority scheduling algorithms, each job is assigned a fixed-priority, which is often related to the state of that job, e.g., the absolute deadline in earliest-deadline first (EDF) or the rank-order first-in first-out (FIFO) scheduling. A further refinement of job-level fixed-priority scheduling is *segment-level fixed-priority scheduling*, in which each job J_i of some task τ_i can be segmented into $J_{i_1}, J_{i_2}, \ldots J_{i_n}$, such that J_{i_2} is released when J_{i_1} is finished. In segment-level fixed-priority, each segment is assigned a fixed-priority, which has application in segmentation of jobs into memory and computation phases [SP19] or segmented self-suspension scheduling [KAN+13]. In the most extreme case, each segment is infinitely small, which hence results in *fluid scheduling algorithms* such as proportionate-fair (P-FAIR) [MR99; AS00].

fluid scheduling algorithm
proportionate-fair scheduling

2.3.3.2 *Uniprocessor Scheduling*

To date, the most prominent real-time scheduling algorithms are uniprocessor task-level fixed-priority scheduling algorithms, which are implemented in most real-time operating systems and standards such as RTEMS, QNX, FreeRTOS, OSEK, AUTOSAR, Linux-RT. From a practical perspective, task-level fixed-priority scheduling provides an intuitive parameterization in the sense that a job is only preempted by higher-priority jobs and thus the response-time of a job (of a task) is only impacted by the interference caused by higher-priority jobs. Moreover, very efficient so called $\mathcal{O}(1)$ scheduler implementations have been developed to select the next to schedule job, even though the claim about efficiency has been challenged by, e.g., [Bra11b]. However, task-level fixed priority scheduling algorithms are shown not to be optimal for preemptive scheduling with respect to hard deadline constraints. In that regard, the job-level fixed-priority scheduling algorithm EDF has been shown to be optimal with respect to hard real-time constraints, that is, each job finishes before its absolute deadline. Above that, EDF provides the property to minimize the maximum tardiness of a task set in the overloaded case, i.e., $U_{\mathbb{T}} > 1$ [Uth04], which provides robustness in the presence of worst-case execution time uncertainties.

task-level fixed priority scheduling algorithms are shown not to be optimal for preemptive scheduling

It was a long assumed that preemptive EDF incurs too much overheads in a real-time operating system implementation, due to more context switches, and more complex data structures to maintain the job-level fixed-priority order, compensating the theoretical benefits with overhead induced capacity loss. Buttazzo in [But05; BG06] proved that EDF causes in fact less preemptions than fixed-priority scheduling. On the other issue, data structures and algorithms, which are used to maintain the job's priority order are more complex than in task-level fixed-priority scheduling, as evident in the predominant black-red tree implementation, e.g., found in RTEMS [1]. However, approaches to lessen the overhead are proposed in e.g., [BG06; Pat16; Sho10].

EDF causes in fact less preemptions than fixed-priority scheduling

Most prominent task-level fixed-priority scheduling algorithms such as deadline-monotonic (DM) or rate-monotonic (RM), and job-level fixed-priority scheduling algorithms such as EDF have extensions to the multiprocessor scheduling problem.

2.3.3.3 *Multiprocessor Scheduling*

In multiprocessor scheduling, the set of jobs that is generated by a taskset $\mathbb{T}$, is to be scheduled on M processors. A multiprocessor scheduling algorithm is hence responsible to decide which job is to be executed on which processor at any time.

In the context of multiprocessor scheduling, there are fundamentally two different approaches to priority-driven scheduling algorithms. That is, either all processors are scheduled using one scheduler with a single shared ready queue, or the set of processors is subdivided into partitions, each of which maintains a per-partition ready queue. If the partition is a single processor then the scheduling is called *partitioned scheduling*; *global scheduling* if the partition is the set of all processors; and *clustered scheduling* otherwise, which are explained in more detail

partitioned scheduling
global scheduling
clustered scheduling

1 https://www.rtems.org/

from here.

2.3.3.4 *Partitioned Scheduling*

In partitioned scheduling, the taskset is partitioned to the set of processors, i.e., each task is assigned to a processor and no two tasks are eligible to execute on more than one processor. From a practical and theoretical point of view, the multiprocessor scheduling problem is reduced into M uniprocessor scheduling problems, i.e., each processor maintains a per-core ready queue and each processor executes a uniprocessor scheduling algorithm. As a result, each scheduler only accesses the per-core ready queue, which improves cache locality [Bra11b], and thus reduces execution time overhead of the scheduler. Additionally, partitioned scheduling forbids task migration, i.e., all task state is local to the core, which further reduces overhead. Alas, optimal partitioning algorithms are NP-hard in the strong sense, as in order to generate an optimal partition of the tasks, the underlying bin-packing problem must be solved, which is shown to be NP-hard in the strong sense. Conclusively, only efficient heuristics and approximation algorithms form the basis of partitioning algorithms [JC07].

optimal partitioning algorithms are NP-hard in the strong sense

Partitioning Algorithms. Several partitioning algorithms have been proposed in the context of partitioned scheduling algorithms, which fundamentally consist of two subsequent stages, namely;

1. The task set, subject to partitioning, is pre-sorted based on some task related parameters, e.g., the periods (RM), deadlines (DM), or utilization (largest-utilization first (LUF)). The choice of the task parameter may be motivated by some intuitive insight for empirically good partitions, or be motivated by induced formal properties allowing for approximation analyses of schedulability tests; using those pre-ordering strategies.
2. Each task is partitioned to the available processors iteratively in the pre-order, which was generated in the first stage. Then the assignment of the task to a processor is evaluated for schedulability – under the condition that all prior tasks are already partitioned – using some schedulability test, such as utilization bounds, demand-bound functions, or time-demand analysis. Eventually, the processor that satisfies the *schedulability test*, first, best, worst, or arbitrary, is chosen in the respective strategies *First-Fit* (FF), *Best-Fit* (BF), *Worst-Fit* (WF), and *Arbitrary Fit* (AF).

partitioning algorithms are often strongly related with the schedulability tests and analyses

The task partitioning algorithms are often strongly related with the schedulability tests and analyses. Based on the deadline-monotonic partitioning strategy it has been shown by Chen [Che16] that such a strategy has a speedup factor of 2.84306 (respectively, 3) against the optimal schedule for ordinary constrained-deadline (respectively, arbitrary-deadline) task systems when the fixed-priority deadline-monotonic scheduling algorithm is used, and a speedup factor of 2.6322 (respectively, 3) against the optimal schedule for ordinary constrained-deadline (respectively, arbitrary-deadline) task systems when the dynamic-priority earliest-deadline-first (EDF) scheduling algorithm is used [CC11; CC13].

A further refinement of partitioned scheduling, is clustered scheduling, in which only a subset of processors, i.e., clusters, are eligible for tasks to be partitioned to. Clustered scheduling can help to reduce the overheads, if for instance, processor clusters with improved connectivity are chosen.

2.3.3.5 *Global Scheduling*

In global scheduling, each job can execute on any of the M processors, i.e., there is a single queue from which the jobs are selected for execution on any of the M processors. In priority-driven scheduling algorithms, the M highest-priority jobs are executed on the M processors at any time. Additionally, in preemptive global scheduling, a job is allowed to migrate from one processor to another. A benefit of work-conserving global scheduling is automatic load balancing, that is, there is never a processor idle while another is overloaded. On the downside, a global scheduler is a complex distributed program, where each processor has its own per-core scheduler, which accesses the shared data structures, which in turn, must be synchronized, introducing significant overhead. In addition, increasing the number of processors, further increase the likelihood of contention.

a global scheduler is a complex distributed program

2.3.3.6 *Semi-Partitioned Scheduling*

Semi-Partitioned scheduling can be interpreted as hybrid between partitioned scheduling and global scheduling. Semi-partitioned scheduling extends partitioned scheduling by allowing a subset of so called *migratory tasks* to migrate, i.e., to be scheduled on any processor, whereas the remaining *fixed tasks* are disallowed to migrate [Bra11b]. Migratory task may be further restricted to, e.g., only migrate in predefined time slots or only depending on specific constraints. The intention of semi-partitioned scheduling is to avoid the algorithmic capacity loss due to non-optimal partitioning and to avoid the implementation induced overheads of global scheduling. In the beginning it was even debated if semi-partitioned scheduling provides any real benefit by, e.g., Bastoni et al. [BBA11], which was later answered positively in, e.g., [BG16].

migratory tasks

fixed tasks

2.3.3.7 *Gang scheduling*

Another scheduling paradigm is *gang scheduling*, which was originally developed for high performance parallel computing. It was observed that most parallel processes consist of frequently communicating and cooperating threads. The thread execution in multiprocessor systems is fundamentally different from the execution in uniprocessor systems, due to the parallel execution potential of threads. Therefore, more complex synchronization efforts are required to implement inter-dependencies and precedence constraints among the related threads. This includes *barriers*, *mutual exclusion*, and implicit synchronization such as precedence constraints due to, e.g., data producer and data consumer relation.

gang scheduling

barrier

mutual exclusion

Subsequent observations supported the benefit of having all cooperating threads to execute simultaneously, such that a message sending and a message

receiving thread can respond to a request almost immediately [Tan09]. Hence, the idea of so called gang scheduling is to have all threads of a process run together. More precisely, a gang is a group of related threads, which are co-scheduled simultaneously on the available processors, and start and finish their execution times simultaneously. An additional benefit of gang scheduling, according to Feitelson et al. [FR92], is that gang scheduling allows the threads to use *busy waiting* instead of blocking without the risk of waiting for a thread that is currently not running. In contrast, without gang scheduling, threads have to block in order to synchronize, thus suffering from blocking and subsequent context-switch costs.

a gang is a group of related threads, which are co-scheduled simultaneously on the available processors

busy waiting

A review and introduction of definitions for recurrent gang scheduling in the context of real-time systems is provided by Goossens et al. [GB10a; GR16]. In *rigid gang scheduling*, the number of processors, i.e., size of the gang, is specified externally to the scheduler and is static throughout the execution. In *moldable gang scheduling*, the gang size of each job is determined by the scheduler and does not change throughout its execution. In *malleable gang scheduling*, the gang size can be changed by the scheduler during the job's execution.

rigid gang scheduling

modable gang scheduling

malleable gang scheduling

The approach by Goossens et al. [GR16], schedules the gang tasks in a fine-grained manner, coming back to proportional fairness policy, which incurs high runtime overhead. More recent results regarding real-time gang scheduling, have been proposed for non-preemptive rigid gang scheduling, e.g., in [LGL22; DL22], malleable gang scheduling [NIN22], and *stationary rigid gang scheduling* in [UGB+21].

stationary rigid gang scheduling

2.3.4 SCHEDULABILITY ANALYSIS

Schedulability analyses and schedulability tests are usually related to a specific scheduling algorithm or a specific class of scheduling algorithms. A schedulability test for a scheduling algorithm, indicates whether a task set $\mathbb{T}$ is feasibly schedulable with respect to a metric of interest, e.g., no deadline misses. Such a test should not only be as precise as possible, but must also be computationally affordable. Unfortunately, it has been shown by Ekberg and Yi that, even for uniprocessor systems, an exact schedulability test for dynamic-priority scheduling of constrained-deadline task sets is strongly coNP-complete and that an exact schedulability test for sporadic tasks under static-priority scheduling is NP-hard. The complexity of uniprocessor scheduling analysis is thoroughly presented in the book by Ekberg and Yi [EY22].

In consequence, exact schedulability tests are not always computationally affordable and thus a lot of research is concerned with the construction of sufficient and computationally affordable schedulability tests and schedulability analyses, which are almost as precise as an exact test. A schedulability test can be either *sufficient*, *necessary*, or *exact* if it is sufficient and necessary. That is, with reference to the formalism of the previous section, the exact schedulability test is given by

$$\mathbb{T} \text{ is schedulable according to test } \mathbb{B} \iff \forall \mathbb{J} \in \mathbb{FJ} : \forall J \in \mathbb{J} : \; f_J \leq d_J \quad (2.4)$$

An important property of a scheduling algorithm or schedulability test is that of *sustainability*, which denotes a monotone behaviour with respect to the *rigor* of

sustainability

the parameters, which confine the respective problems. For instance, if for a task set a scheduling algorithm always derives a feasibly schedule then a sustainable scheduling algorithm is guaranteed to generate a feasible schedule if for instance, the number of tasks in the task set is reduced, the period is increased, or the system is provided with more processors. Sustainability is an important property as non-sustainable scheduling algorithms and tests are sensitive to parameter uncertainty, which may invalidate their correctness in real systems.

Sustainability is an important property as non-sustainable scheduling algorithms and tests are sensitive to parameter uncertainty

The specific analysis techniques in scheduling theory are too vast to be exhaustively presented here, but instead only two fundamental analysis approaches are presented hereinafter; that is, *critical instant* based analyses, and *window of interest* based analyses.

2.3.4.1 *Critical Instant Based Analyses*

For some scheduling algorithms, a *critical instant* can be proved, which is, that for each task $\tau \in \mathbb{T}$ there exists a single job J, for which a single specific job collection $J \in \mathbb{J} \in \mathbb{F}\mathbb{J}$ exists (and can be reconstructed), such that $f_J - a_J$ is maximal (among all jobs of that task). It is evident that if the *critical instant* is verified then Eq. (2.4) is satisfied.

critical instant

For periodic and sporadic tasks with constrained- and implicit deadlines, a critical instant was proposed by Liu et al. [LL73] for task-level fixed-priority scheduling algorithms. Such a job collection that represent the critical instant is defined as follows; All jobs of higher-priority tasks are released synchronously with the job under analysis J, and all jobs release their workload as rapidly as possible. It was proven that J suffers the largest interference and thus has the maximal response time.

Alas, the construction of a critical instant for task sets with other characteristics is not trivial as evident in the incorrect critical instant constructions for self-suspending task systems in several works as explained in Chen et al. [CNH+19]. Moreover, there is no critical instant known for gang scheduling or multiprocessor scheduling. However, some extensions such as the *level-i busy window* concept for sporadic task systems with arbitrary deadlines have been found [Leh90].

the construction of a critical instant for task sets with other characteristics is not trivial

level-i busy window

As a consequence of the existence of a *critical instant*, it is sufficient to simulate the concrete schedule for that specific job collection and to infer the worst-case response time $R_J := f_J - a_j$ of that specific job J with the largest response time. On the basis of a worst-case response time analysis, a schedulability test can be immediately concluded by verification that $R_i \leq D_i$ for all $\tau_i \in \mathbb{T}$. The simulation of the concrete schedule for the critical instant uses *time-demand analysis* (TDA) and computes the maximal time required from the initial idle state at system start to the first time that the considered job J is finished, i.e., the *busy window*. The computation is realized by a fix-point iteration as exemplified as follows for a task set $\mathbb{T} = \{\tau_1, \ldots, \tau_n\}$, which is scheduled by a task-level fixed-priority scheduling algorithm. Then in the worst-case response time analysis, it is verified for each task $\tau_k \in \mathbb{T}$, whether there $\exists 0 < t \leq D_k$ such that

time-demand analysis

busy window

$$C_k + \sum \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \leq t \tag{2.5}$$

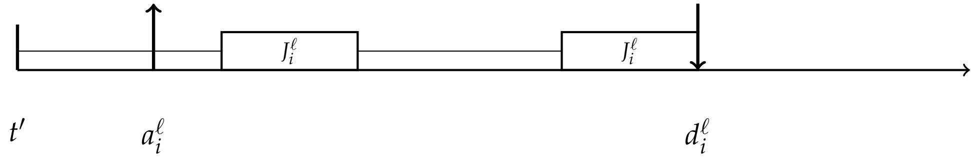

Figure 2.11: An exemplary *window of interest* for a job J_i^ℓ, released at time a_i^ℓ, and missing its deadline at time d_i^ℓ. Time t' denotes the earliest time before d_i^ℓ such that during $[t', d_i^\ell]$ the processor is continuously busy executing jobs with deadline no later than d_i^ℓ.

holds.

polynomial time tests can be derived semi-automatically

For many time-demand analyses based schedulability tests, polynomial time tests can be derived semi-automatically using the *k2U* [CHL15b] or *k2Q* [CHL15a] analysis framework. These frameworks, derive an extreme point solution for an integer-linear program (ILP), which is based on the explicit evaluation of specific k test points, which result in a TDA formulation with integer variables. These extreme points then serve as upper-bounds for the worst-case response time.

2.3.4.2 *Window of Interest Based Analyses*

window of interest

In the case that a critical instant can not be identified for the task model and scheduling algorithm under analysis, a more abstract *window of interest* based analysis must be used. Here, the term abstract refers to the fact that not a single concrete job collection $\mathbb{J}$ can be constructed, but that there are multiple job collections in $\mathbb{J}$ that must be evaluated and that $\mathbb{J}$ is constructed purely based on properties, which will be further explained by example later in this section.

To derive a sufficient schedulability analysis, in the *window of interest* based analyses, the sufficient implication in Eq. (2.4) is proved by contrapositive. That is, if there exists at least one job collection $\mathbb{J}$ and at least one job $J \in \mathbb{J}$ that misses the deadline, i.e., the job has not been executed to completion then the *to be constructed* schedulability test does not hold. This situation is illustrated in Figure 2.11 for sporadic constrained-deadline EDF scheduling on a uniprocessor system.

For the remainder of this section, we assume that the ℓ-th job of task τ_i, namely J_i^ℓ, is the first job to miss the deadline. Then starting from the baseline window of interest $(a_i^\ell, d_i^\ell]$, we know by the assumption of a deadline miss and the *work-conserving* property of preemptive earliest-deadline first scheduling for sequential tasks on uniprocessor systems, the processor must be busy. That is, the processor is continuously executing workload other than J_i^ℓ for at least $d_i^\ell - a_i^\ell - C_i^\ell$ amount of time. Therefore, the following Eq. (2.6) is well-defined

$$exe(S, J_i^\ell, a_i^\ell, d_i^\ell) + exe(S, J \neq J_i^\ell, a_i^\ell, d_i^\ell) = d_i^\ell - a_i^\ell \tag{2.6}$$

Notably this equation, and window respectively, can be extended to

$$exe(S, J_i^\ell, t', d_i^\ell) + exe(S, J \neq J_i^\ell, t', d_i^\ell) = d_i^\ell - t' \tag{2.7}$$

for $t' \leq a_i^\ell$ as long as Eq. (2.6) implies Eq. (2.7). Most proposed *window based* schedulability tests have been concerned with extensions to conclude more preferential job collections $\mathbb{J}$ to be considered in the analysis, e.g., in [BF07; BF08; GYG+08; GUC21; CBU18].

With regards to the example, t' is constructed as the smallest point in time in the interval $t' \in [0, a_i^\ell]$ such that during $[t', d_i^\ell]$ only jobs with absolute deadline no later than d_i^ℓ are executed. By construction of this definition, we know that the processor is continuously executing jobs J that have arrival $a_J \geq t'$ and deadlines no later than $d_J \leq d_i^\ell$, and thus we know that t' is the arrival time of at least one of those jobs.

Hence, starting from Eq. (2.7), we can derive the following

$$\sum_{\tau_j} dbf_j(d - t') \geq C_i + \sum_{\tau_j \neq \tau_i} dbf_j(d - t') \geq C_i^\ell + \sum_{\tau_j \neq \tau_i} dbf_j(d - t') > d - t' \qquad (2.8)$$

Letting $t := d - t'$, the contrapositive of Eq. (2.8) yields a sufficient schedulability test, namely if

$$\forall t > 0 \sum_{\tau_j \in \mathbb{T}} dbf_j(t) \leq t \qquad (2.9)$$

then $\mathbb{T}$ is feasibly schedulable.

Unfortunately, extensions of the window based analysis technique for EDF to multiprocessor systems leads to so-called $1/M$ schedulability tests in G-EDF, which been shown to be dominated by partitioned scheduling [BS18].

window extensions for multiprocessor systems have been proposed

However, window extensions for multiprocessor systems have been proposed by Fisher and Baruah [BF07; BF08]. The authors define $t' < a_i^\ell$ such that at least one processor in the multiprocessor system of M processors is idle right before t'. This construction together with the *work-conserving* property, leads to the conclusion that at most $M - 1$ job could have been eligible for scheduling right before t'. Therefore, at most $M - 1$ jobs can carry-in workload – that was generated before time t' – into the *window of interest*. This approach reduces the number of carry-in jobs to consider from $|\mathbb{T}|$ jobs to only $M - 1$ jobs with the largest demands.

Chen et al. [CBU18] have provided a task-parametric window extension for global task-level fixed-priority scheduling algorithms for sporadic arbitrary-deadline task systems, which has later been further extended to self-suspending task systems by Günzel et al. in [GUC21].

2.3.4.3 *Analyses for Self-Suspending Task Systems*

A comprehensive survey for self-suspension aware analyses is given in [CNH+19] and only a brief motivation to the challenges in self-suspension analysis is given here. Initial works [Raj91], attempted to extend existing analyses, wrongly assuming that the critical instant for sporadic task systems holds in self-suspending task systems. This misconception invalidated many results that built upon the flawed established theory [BAH+15].

In general the analysis techniques for self-suspending task sets in task-level fixed-priority and job-level fixed-priority scheduling algorithms are founded on *window of interest* based analyses, which is further complicated by the fact that every job may have infinitely many suspension and resumption sequences. state-of-the-art schedulability analyses for self-suspending task systems can be found in, e.g., [GUC21; GBC+22; CNH16; ABN22].

2.3.4.4 *Analyses for DAG Tasks*

A detailed discussion of analysis techniques for DAG tasks is omitted, since the relevant techniques are explained in Chapter 4.

inter- and intra task analysis

In general however, DAG response time analysis usually considers the two analyses problems of *inter- and intra task analysis*. The former analysis examines the interference caused by jobs of other tasks than the DAG job under analysis, whereas the latter considers the interference of subjobs – which are subjected to precedence constraints – within the same DAG job. Most analyses concerned with DAG task scheduling, focus on the improvements of intra-task interference and the resulting makespan problem.

2.3.5 PERFORMANCE OF SCHEDULING ALGORITHMS

A review on the evaluation of scheduling algorithms can be found in [Dav16], in which the methods used for schedulability tests for real-time systems are categorized into:

- *Theoretical methods*, which describe a worst-case comparison against a specific competitor, namely *dominance relationships*, *utilisation bounds*, and *resource augmentation bounds*.
- *Empirical methods*, which evaluate the performance of a schedulability test and scheduling algorithms considering *simulation of the scheduling algorithm*, *evaluation based on synthetic task sets*, *case studies*, and *experiments on real hardware*. Empirical methods typically facilitate an average-case comparison against various competitors.

Both methods are relevant to this dissertation, and are elaborated hereinafter.

2.3.5.1 *Theoretical Methods*

dominance relation

Dominance Relation. A *dominance relation* is a theoretical measure to directly relate the performance of two schedulability tests (and thus scheduling algorithms). That is, a schedulability test $\mathbb{A}$ is said to dominate a schedulability test $\mathbb{B}$, if every task set that is schedulable according to $\mathbb{B}$ is guaranteed to be schedulable according to $\mathbb{A}$. That is, more formally, for each task set $\mathbb{T}$ the following implication $\mathbb{B}(\mathbb{T}) \implies \mathbb{A}(\mathbb{T})$ holds true. The dominance relation of scheduling algorithms can be formulated by *exact* schedulability tests for those algorithms, i.e., an equivalent description. Based on the dominance relation, it is possible to define optimality

dominance relation of scheduling algorithms can be formulated by exact *schedulability tests*

among a given class of scheduling algorithms. For instance, if $\mathbb{B}$ is an optimal scheduling algorithm for some class of scheduling algorithms then $\mathbb{A}$ is as well.

speed-up factors

Speed-up Factor. A comprehensive survey and discussion of *speed-up factors* is given in [CBH+17a] and is only briefly presented. Closely related to the dominance relation is the *speed-up factor* [PST+02]. A speed-up factor $s \in \mathbb{R}_{>0}$ is a scalar value, which describes a scaling operation $\gamma : \mathbb{T} \mapsto \mathbb{T}'$ of all task model parameters, which are sensitive to processor speed increases, e.g., the worst-case execution time $\gamma(\tau_i) = (C_i/s, \dots)$. Then in analogy to the dominance relation, a schedulability test $\mathbb{A}$ is said to have a speed-up factor s with respect to a schedulability test $\mathbb{B}$ if the following implication holds $\mathbb{B}(\mathbb{T}) \implies \mathbb{A}(\gamma(\mathbb{T}))$. That is, a task set $\mathbb{T}$, which is verified to be feasibly schedulable according to $\mathbb{B}$, is guaranteed to be feasibly schedulable by $\mathbb{A}$, if the task set is subjected to a speed-up of s. Speed-up factor analysis is most widely used to describe the approximation quality of a schedulability test with respect to an optimal scheduling algorithm of some class. For instance, the sufficient schedulability test for preemptive rate-monotonic scheduling of implicit-deadline tasks by Liu and Layland [LL73]

$$U_{\mathbb{T}} := \sum_{\tau_i \in \mathbb{T}} \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1) \leq ln(2) \tag{2.10}$$

has a speed-up factor of $1/ln(2)$ with respect to preemptive EDF, which is an optimal scheduling algorithm for the problem with the exact schedulability test $U_{\mathbb{T}} \leq 1$. A sufficient schedulability test (usually for implicit-deadline task sets), which determines the schedulability of a task set based on the task set utilization is called a utilization bound. Notably, the optimal scheduling algorithm, used for comparison, may not exist or be known for a class of scheduling problems. Hence, it is often not possible to know whether the ideal scheduler can schedule a given task set on unit-speed processors and thus a speed-up factor bound may not provide a schedulability test.

capacity augmentation bound

Capacity Augmentation Bound. In the context of parallel DAG tasks, Li et al. proposed *capacity augmentation bounds* to theoretically assess the performance of DAG task scheduling algorithms in [LAL+13]. Loosely speaking, capacity augmentation bounds are in the middle of utilization bounds, adapted for the DAG task model , and speed up factors. Most notably, capacity augmentation bounds immediately yield schedulability tests, in contrast to speed-up factors. A scheduling algorithm $\mathbb{A}$ is said to have a capacity augmentation bound $s \in \mathbb{R}_{>0}$ if under the condition stated in

$$\sum_{\tau_i \in \mathbb{T}} \frac{C_i}{T_i} \leq M \tag{2.11}$$

$$\forall \tau_i \in \mathbb{T} \; L_i \leq D_i \tag{2.12}$$

Eq. (2.11) and Eq. (2.12), the DAG task set $\mathbb{T}$ is guaranteed to be feasibly schedulable on M processors with processor speed s. Since the conditions in Eq. (2.11) and Eq. (2.12) are necessary conditions for an optimal scheduling algorithm, the capacity augmentation bound is a (not necessarily tight) speed-up factor.

2.3.5.2 *Empirical methods*

Brandenburg [Bra11b] states that, in a real system it is impossible to provide all hypothetical capacity to the real-time task set, which is due to scheduling overheads, preemptions, and memory access times, and migrations. More precisely, overhead-related capacity loss is due to processor time, which is consumed by hardware inefficiencies, such as cache misses, or computing a scheduling decision, and queue management. Such runtime overheads are unavoidable to some degree, but can differ significantly among schedulers and implementations. Capacity, which is consumed by the overheads, must realistically be accounted for in the schedulability test. On the downside, evaluations based on real hardware requires a real-time operating system implementation and measurements on a real platform.

evaluations based on synthetic task sets are most common in the literature

With regards to empirical methods, evaluations based on synthetic task sets are most common in the literature. Emberson et al. [ESD10] provide a review on task set generation methodology; stating that a task set generation algorithm must be *efficient* to achieve significant large sample sizes; *parameter independent* to be able to vary each parameter of the task set independently, and *unbiased* in the sense that the distribution of task sets generated should be equivalent to selecting task sets uniformly from the set of all possible task sets (under the parameter constraints). To that end, the UUnifast algorithm [BB05] has been proposed for uniprocessor systems. For multiprocessor systems with utilizations up to the number of processors, the UUnifast Discard [BB05], RandFixedSum [Dav16], and more recently the dirichlet-rescale algorithm (DRS) [GBD20] have been proposed to generate task sets with unbiased targeted utilization values. The requirement of *unbiased* data generation is sometimes challenged by researchers arguing that some *parameter vectors* represent unrealistic tasks, resulting in evaluations that do not evaluate the performance in the relevant domain. In that regard, von der Brüggen et al. [BUC+17] have shown that the specialization of the task parameter space to an application specific domain, as described in *Automotive Benchmarks for Free* [KZH15], improves the acceptance ratio.

UUnifast

UUnifast Discard

Rand Fixed Sum

dirichlet-rescale

parameter vectors

In parallel DAG and conditional DAG tasks there is no de-facto standard task generation method

In parallel DAG and conditional DAG tasks there is no de-facto standard task generation method as *parameter independence* is in general not achievable. Hence, most generation methods are biased in the kind of generated tasks. In cases that the schedulability test only depends on parameters, which are derived from the DAG, e.g., the longest path or the total workload, then the DRS algorithm can be used to generate tasks for every utilization level and the longest path can be drawn independently. If however the complete DAG is required to be generated, e.g., when the schedulability test and analyses consider the sub tasks, then *parameter independence* is not achievable. As a consequence there are various commonly used techniques such as the *Erdős–Rényi*, *Layer-by-Layer*, *Fork-Join*, or *domain specific benchmarks*, e.g., [WGS+17; SJX+22; DRW98; RP17].

Erdős–Rényi

Layer-by-Layer

Fork-Join

benchmarks

acceptance ratio

Acceptance Ratio. Among the *empirical methods* in this dissertation, we primarily consider the *acceptance ratio* of synthetic task sets and relative comparisons with respect to resource requirements. Acceptance ratio tests evaluate the percentage of subjected task sets for schedulability testing, which are verified to be feasibly schedulable by the algorithm under evaluation, compared to the total number of

subjected task sets. The subjected task sets are generated with a given utilization value, e.g., task sets with a $U_{\mathbb{T}}$ from 1% to 100% $\cdot M$ in a fixed increment of , e.g., 5%, where M denotes the number of processors. For each schedulability test, the acceptance ratio, i.e., the percentage of accepted task sets, is plotted against the utilization value. It is evident that a schedulability test, which yields a higher acceptance ratio for each utilization value than another schedulability test, is superior in the evaluated experiment. Acceptance ratio tests are the de-facto standard evaluation method for uniprocessor and multiprocessor scheduling algorithms for the sequential task model. This is due to the fact that any generated task set with $U_{\mathbb{T}} \leq 1$ is feasibly schedulable by an optimal scheduling algorithm, and an increase in task set utilization makes the schedulability problem monotonically harder, and hence the acceptance ratio value is easy to interpret. For more complex task models such as the DAG task model, special attention must be paid to the task set generation and interpretation, or a necessary condition for schedulability must be plotted against the utilization to improve interpretability of the acceptance ratios.

Acceptance ratio tests are the de-facto standard evaluation method for uniprocessor and multiprocessor scheduling algorithms for the sequential task model

3 TIMING PREDICTABLE PROTOCOLS

In the position papers concerning the design of predictable systems by Wilhelm et al. [WGR+09] and Axer et al. [AEF+14], the authors state that system properties, which are subject to predictability constraints, should already be considered and guaranteed from the design. Furthermore, the authors note that since the overall timing requirements of the system are propagated down in the system hierarchy, all parts of the system must be designed with respect to predictability. These predictability concerns are exacerbated and relevant for multicore systems and parallel task scheduling, due to complex resource contention, and varying delays in the communication fabrics.

all parts of the system must be designed with respect to predictability

Following the position of predictable system design, the focus in this chapter is on the design of protocols (and scheduling algorithms), which increase predictability, and allow for safe worst-case response time analyses. At first, we propose a novel gang scheduling algorithm called *stationary rigid gang scheduling*, in which gangs are statically assigned to a specific sub set of processors, avoiding task and thread migration. Above that, we examine the communication predictability of network-on-chip (NoC), which are increasingly used in multiprocessor systems. We propose a family of *simultaneous progression switching protocols* for real-time NoC arbitration, which is described by the *all-or-nothing* property, and provides increased predictability at the cost of decreased average case performance. Both approaches share a common underlying analysis concept and framework, which is based on a formal reduction from the *all-or-nothing* property to *suspension-aware uniprocessor scheduling* theory.

simultaneous progression switching protocols

all-or-nothing property

suspension-aware uniprocessor scheduling

The remainder of this chapter is organized into the following sections. In Section 3.1 *Motivation*, the gang scheduling paradigm is motivated as an option for timing predictable system design; the fundamental challenges in distributed and link-based network-on-chip scheduling, and response-time analyses, are examined. Afterwards in Section 3.2 *Related Work*, the related work for real-time network-on-chip arbitration and gang scheduling for real-time systems is presented. We first propose a rigid gang scheduling algorithm for multiprocessor systems in Section 3.3 *Stationary Rigid Gang Scheduling*, and establish the analysis ideas and framework, which is used for the worst-case response time analysis in the the network-on-chip, and the gang scheduling problem. In Section 3.4 *Simultaneous Progression Switching Protocols*, we present the worst-case response time analysis with respect to the peculiarities inherent to NoC arbitration, and propose a protocol implementation. At last, in Section 3.5 *Conclusion*, the results proposed in this chapter are summarized, and concluded.

3.1 MOTIVATION

To increase the performance and predictability of the execution of parallel tasks on multiprocessor platforms, the *global gang scheduling problem* [RGK17; DL17; KI09], in which the set of machines used by a gang task is not fixed, has been proposed. In the gang task model, a set of threads is grouped together into a so called *gang*, with the additional constraint that all threads of a gang must be co-scheduled at the same time on the available processors. It has been demonstrated that gang-based parallel computing can improve the performance in many cases [Jet97; FR92]. Even more, Wasly et al. [WP19] provided experimental evidence of negative effects of non-gang scheduling with respect to the number of context-switches and increased thread execution time, due to blocking when threads are not executed together. Wasly et al. [WP19] argue that by scheduling all threads of a task simultaneously, the communication time can be easily accounted for, given that the inter-processor interconnect provides real-time bounds. The *rigid gang task model* is the simplest and most widely studied gang task model in the real-time systems research literature, which is characterized by a task-level fixed gang size. One particular advantage of the *rigid gang model* is that the interference caused, by shared resource contention, and intra-task parallelism, can potentially be quantified more precisely, due to more constraints, and thus increased certainty about the execution. Consequently, the worst-case execution time of the gang can be reduced. Within a gang, co-scheduling of memory accesses and computation is possible, which can also potentially reduce the worst-case execution time of the gang. Specifically, one strict view of this is the *RT-Gang model* by Ali and Yun [AY19], in which all processors are allocated to a gang at the same time. Unfortunately, even finding an optimal schedule for the rigid gang scheduling problem has been shown NP-hard in the strong sense even when all the tasks have the same period and the same deadline [Kub87]. Moreover, even special cases, like three machines [BDOD+94] or unit execution time per task [HVV94], are also NP-hard in the strong sense. The rigid gang scheduling problem for implicit-deadline periodic real-time task systems (i.e., $D_i = T_i$ for every task τ_i) has been recently studied by Goossens and Richard [GR16] for which the authors presented one algorithm based on linear programming and another algorithm based on a heuristic.

all threads of a gang must be co-scheduled at the same time on the available processors

rigid gang task model

rt-gang model

finding an optimal schedule for the rigid gang scheduling problem has been shown NP-hard in the strong sense

To address the issue in the predictability of inter-processor communication in multiprocessor scheduling of parallel task scheduling, timing predictable network-on-chip interconnects for which safe worst-case response time analyses can be devised are required. Common approaches for real-time communication on a NoC apply one of two general strategies. One is to utilize time-division-multiplexing (TDM) to ensure that the timing constraints are satisfied, i.e., by constructing the transmission schedule statically with a repetitive table, e.g., in [GDR05; PK08; SMA+12; KSS+16; Sch; MNT+04; SAA+15; HFB+18]. Another common approach is to apply a priority-based dynamic scheduling strategy in the routers to arbitrate the flits in the network, e.g., in [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; KP16; XLW+16; NIP16; IBN16; XWL+17; IBN18; NHE19]. The difficulty of the TDM strategy is to construct a feasible TDM schedule and the global clock synchronization, whilst the difficulty of the priority-based

scheduling strategy is to validate the schedulability, i.e., whether all messages can meet their deadlines. With respect to fixed-priority based scheduling strategies, Table VII in [IBN16] summarizes the recent results for fixed-priority wormhole switched NoCs up to 2017. Eight of the ten results (namely, [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; NIP16; KP16; XLW+16; IBN16]) were already disproved by counter examples. These series of flaws in the literature suggests that the scheduling algorithm and network architecture may be too complex to be correctly analyzed adopting uniprocessor real-time scheduling theory and its assumptions. Informally speaking, the researchers in [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; NIP16; KP16; XLW+16; IBN16; XWL+17; IBN18; NTI+19] have tried to construct their worst-case response time analyses by linking the problem to a corresponding uniprocessor scheduling problem instance. Most of them were later found to be flawed, e.g., [Mut94; HO97; KKH+98; LJS05; SB08; NIP16; KP16; XLW+16], or without a formal proof, e.g., [IBN16; IBN18; NTI+19; NHE19]. The proofs in [NTI+19] did not consider the equivalence of the worst-case response time analysis adopted in uniprocessor systems and the analysis of a NoC. Instead, they emphasized the quantification of different types of interferences. However, in many places in the proofs, e.g., the building blocks from Lemmas 3, 4, and 6 in [NTI+19], the derivation is based on examples.

series of flaws in the literature suggests that the scheduling algorithm and network architecture may be too complex to be correctly analyzed adopting uniprocessor real-time scheduling theory and its assumptions

Motivated by the two described problems, either the parameter uncertainty and hardware peculiarities must be considered in the scheduling algorithm design and associated formal schedulability analyses, or the predictability of the hardware must be increased. In this chapter, we propose the following contributions to address the motivated problems of predictability to allow for safe worst-case response time analyses.

At first, a *stationary rigid gang scheduling* algorithm for hard real-time systems is proposed, in which each rigid gang task is restricted to execute on an assigned subset of processors. We provide formal proof, how the corresponding schedulability test problem can be reduced to a uniprocessor suspension-aware schedulability test. Furthermore, we propose so called *consecutive stationary rigid gang assigments* for preemptive deadline-monotonic gang scheduling and provide several sufficient schedulability analyses. The proposed scheduling algorithm is shown to admit a *parametric speed-up factor* with respect to an optimal rigid gang scheduling algorithm, and to bound the worst-case performance with regards to the task set's gang size ratio of any two tasks.

stationary rigid gang scheduling

parametric speed-up factor

Secondly, we propose a family of *simultaneous progression switching protocols* for real-time NoC arbitration, which is described by the *all-or-nothing* property, and provides increased predictability at the cost of decreased average case performance. A possible implementation, including router design, and arbitration algorithm is proposed. Notably, any non-minimal route in simultaneous progression switching protocols is deadlock-free, since the *simultaneous progression* property prevents circular waiting at the buffers. Therefore, the *path diversity* can be better utilized in order to distribute the load over the links, such that network contention is reduced.

simultaneous progression switching protocols

path diversity

3.2 RELATED WORK

In parallel task scheduling, inter- and intra-task parallelism has to be considered in the timing analysis, where inter-task parallelism refers to the co-scheduling of different tasks and intra-task parallelism refers to parallel execution of a single task. In the context of task models for parallel computing, fork/join models [LKR10], synchronous parallel task models, and DAG (directed-acyclic graph) based task models [FNN17; BMSS+13; Bar15b; Bar15a; BMS+13; MBB+15; CBN+18b] have been proposed and analyzed with respect to real-time constraints.

To schedule a set of *ordinary* periodic [LL73] or sporadic [Mok83] real-time tasks on a multiprocessor platform, three paradigms have been widely adopted, namely, partitioned, global, and semi-partitioned multiprocessor scheduling. A comprehensive survey can be found in [DB11]. For the rigid gang scheduling problem, the three scheduling paradigms are slightly modified and called *stationary*, *global*, and *semi-stationary* gang scheduling. The *stationary gang* scheduling paradigm statically assigns a gang task to a set of processors, in which the cardinality of the set is equal to the gang size of the task. After this assignment is done, a gang task is only eligible to be executed on stationary processors assigned to it. The *semi-stationary* scheduling paradigm allows a gang task to execute on any subset of processors within a given set of processors that is larger than the gang size itself. That is, it allows a job of the gang task to migrate from one subset of processors to another sub set of the given processors at any time. The *global* rigid gang scheduler allows a gang task to migrate to any available set of processors as long as the gang size constraints are met. Note that when the gang size is 1 for each task (i.e., tasks are not executed in parallel and are ordinary periodic or sporadic tasks), the *stationary*, *global*, and *semi-stationary* gang scheduling paradigms correspond to the *partitioned*, *global*, and *semi-partitioned* multiprocessor scheduling paradigms, respectively.

stationary rigid gang scheduling

semi-stationary rigid gang scheduling

global rigid gang scheduling

The computational complexity of the rigid gang scheduling problem was studied back in 1980s. Specifically, it has been shown that finding the optimal schedule for the rigid gang scheduling problem is *NP*-hard in the strong sense even when all the tasks have the same period and the same deadline [Kub87]. Even simpler cases, like three machines [BDOD+94] or unit execution time per task [HVV94] are also shown to be NP-hard in the strong sense.

In real-time systems, rigid gang scheduling has been mostly studied under global earliest-deadline-first (EDF) scheduling, in which the set of processors used by a gang task is not fixed and can be dynamically relocated at runtime, e.g., [RGK17; DL17; KI09]. Specifically, in [KI09], the authors extended Baruah's [Bar07] multiprocessor global EDF analysis for ordinary sporadic real-time tasks to deal with global EDF gang scheduling, which has been disproved by Richard et al. [RGK17]. The only valid analysis for global EDF gang scheduling is from Dong and Liu [DL17] and restricted to implicit-deadline sporadic real-time rigid gang task systems. They provide two utilization-based analyses, one optimized and one approximated.

Goossens and Richard [GR16] studied fixed-priority scheduling for the rigid gang scheduling problem for implicit-deadline periodic real-time task systems.

They presented two algorithms, one based on linear programming and another based on a heuristic algorithm, providing exact and sufficient schedulability tests. Moreover algorithms based on *deadline partitioning* (DP-Fair) for periodic gang systems have been proposed. However the many preemptions of DP-Fair make this algorithm impractical and the complexity of the proposed algorithms is high especially for a large number of processors. The authors themselves discuss the problems to extend their algorithms to sporadic job arrival sequences due to its non-determinism.

DP-Fair

For classical multiprocessor scheduling, it has been recently shown that global static-priority scheduling [SN18] and global EDF as well as global FIFO scheduling [BS18] are dominated by partitioned scheduling under state-of-the-art efficient sufficient schedulability tests, e.g., [BC07; GSY+09]. The main reason is due to the inherited pessimism in those tests, which all stem from the work by Baker [Bak03]. Hence, they all use carry-in interference to compensate the lack of a critical instant theorem and divide the higher-priority interference by the number of processors, i.e., they have a multiplicative factor of $1/M$ in the corresponding analyses. We note that the factor $1/M$ also appears in the schedulability tests in [DL17].

Recently, the wormhole switched fixed-priority NoC with preemptive virtual channels has been considered for the use in real-time system. The first attempts to tackle the schedulability analysis were in 1994 in [Mut94] and 1997 in [HO97]. Both of them were found to be flawed in 1998 by Kim et al. [KKH+98], whose analysis was later found to be erroneous in 2005 by Lu et al. [LJS05]. The series of erroneous analyses continued in [Mut94; HO97; KKH+98; LJS05]. Shi and Burns [SB08] published an analysis in 2008. Eight years later, Xiong et al. [XLW+16] pointed out that the analyses in [SB08] are unsafe in the sense that they do not consider limited buffer space and virtual channels. The proposed analysis by Xiong et al. [XLW+16] was later disproved by counter examples and fixed by the authors in their journal revision in [XWL+17] in 2017. In addition, Kashif et al. [KGP14] proposed stage-level analysis (SLA) to improve the analysis by Shi and Burns in [SB08]. The SLA in [KGP14] assumes an infinite buffer size. Kashif and Patel [KP16] extended the SLA analysis to cope with limited buffer size, which was disproved by Xiong et al. [XLW+16]. Indrusiak et al. [IBN16; IBN18] presented new analyses, but without formal proof. Nikolíc et al. [NHE19] proposed a slot-based transmission protocol, which applies two parallel domains, one for arbitration and one for data transmission. The worst-case response time of wormhole switched fixed-priority NoCs with preemptive virtual channels can also be determined by applying Network Calculus and Compositional Performance Analysis (CPA), or their extensions, to analyze the transmission on the links in a compositional manner, e.g., [QLD09; BDG+18; AES+16; AJE+15; TE17; RE15]. Giroudot and Mifdaoui [GM18] use network-calculus to derive worst-case end-to-end response times for wormhole switched fixed-priority NoCs with limited virtual channels and limited buffer space.

3.3 STATIONARY RIGID GANG SCHEDULING

In the gang task model, a set of threads is grouped together into a so called *gang* with the additional constraint that all threads of a gang must be co-scheduled at

the same time on the available processors. The scheduling algorithms for parallel gang tasks can be classified into three models: *rigid*, *moldable*, and *malleable* tasks.

rigid *moldable* *malleable*

A parallel gang task is called *rigid* if the number of processors assigned to it is specified externally to the scheduler a priori and does not change throughout its execution; *moldable* if the number of processors assigned to it is determined by the scheduler and does not change throughout its execution; and *malleable* if the number of processors assigned to it can be changed by the scheduler during its execution. Such classifications can be found in the literature of multiprocessor scheduling and real-time systems such as [GR16].

It has been demonstrated that gang-based parallel computing can improve the performance in many cases [Jet97; FR92]. An advantage of the rigid gang model in particular, is that the interference caused by shared resource and intra-task parallelism can potentially be quantified better, thus reducing the worst-case execution time of the gang. Within a gang, co-scheduling of memory accesses and computation is possible, which can also potentially reduce the worst-case execution time of the gang.

In this section, a novel gang scheduling algorithm, called stationary rigid gang scheduling is proposed, and the properties and analyses are examined. The remainder of this section is structured as follows. In Section 3.3.1, the studied problem is formalized and the rigid gang task model and stationary gang system model is explained. Based on the system and scheduling model, a schedulability analysis and corresponding test is proposed. Based on a formal reduction to the uniprocessor self-suspension schedulability problem, the schedulability analysis is described in Section 3.3.3. In Section 3.3.4, a stationary gang assignment algorithm for the rigid gang task set, with provable worst-case performance guarantees, is proposed. Lastly, the developed analyses are evaluated, which is detailed in Section 3.3.5.

3.3.1 SYSTEM MODEL & PROBLEM DESCRIPTION

We consider a task set $\mathbb{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ of sporadic constrained-deadline rigid gang tasks, which is scheduled on a system of M homogeneous processors with symmetric shared memory, denoted as $\mathbb{P} = \{P_0, P_1, \ldots, P_{M-1}\}$, using our proposed stationary gang scheduling algorithm.

Definition 3.1. *A sporadic constrained-deadline rigid gang task τ_i is defined by the tuple (C_i, E_i, D_i, T_i) where C_i denotes the worst-case execution time, $E_i \in \mathbb{N}$ denotes the rigid gang size, i.e., the static number of subtasks, which are to be co-scheduled simultaneously. The relative deadline $D_i \leq T_i$, is no more than the minimal inter-arrival time T_i, i.e., constrained-deadline.*

Each rigid gang task τ_i releases an infinite number of task instances called jobs where we use J_i^ℓ to refer to the ℓ-th job and a_i^ℓ, f_i^ℓ, and d_i^ℓ to denote that job's arrival time, finishing time and absolute deadline respectively. During the interval $[a_i^\ell, f_i^\ell)$ each of the E_i subtask instances called subjobs have to be executed in parallel. That is, each subjob of J_i^ℓ arrives at time a_i^ℓ and finishes at time f_i^ℓ

such that either all subjobs are scheduled simultaneously, or none is. Each subjob executes for the same amount of time, which is no more than the worst-case execution time, i.e., no subjob is allowed to yield the processor before all subjobs have finished. The period T_i denotes the minimal inter-arrival time of any two consecutive jobs of a task τ_i, i.e., $a_i^{\ell+1} \geq a_i^\ell + T_i \geq a_i^\ell + D_i$. Hence, a total workload of at most $E_i \cdot C_i$ has to be executed in the time interval $[a_i^\ell, d_i^\ell)$. Moreover, the utilization of a gang task τ_i is given by $U_i = E_i \cdot C_i / T_i$.

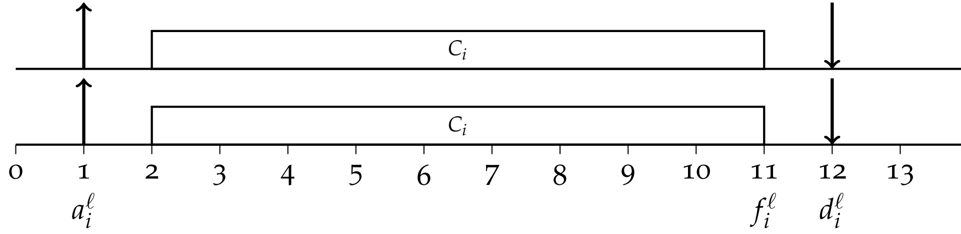

Figure 3.1: A symbolic job J_i^ℓ of a rigid gang task τ_i with gang size $E_i = 2$. Each thread (subjob) in the gang is subject to the co-scheduling constraint.

Similar to non-gang tasks, the response time of a job J_i^ℓ of τ_i is its finishing time minus its arrival time, i.e., $R_i^\ell = f_i^\ell - a_i^\ell$. Hence, the worst-case response time R_i of task τ_i under a given scheduling policy is the maximum response time R_i^ℓ of any job J_i^ℓ for any job arrival sequence possible according to the parameters of tasks in $\mathbb{T}$. A job J_i^ℓ is said to meet its deadline if $f_i^\ell \leq d_i^\ell$ and the corresponding task τ_i is said to meet its deadline if all jobs meet their deadlines, which is guaranteed if the worst-case response time $R_i \leq D_i$, since $R_i \leq D_i = a_i^\ell + R_i \leq a_i^\ell + D_i$ which implies that $a_i^\ell + R_i^\ell \leq d_i^\ell$ or $f_i^\ell \leq d_i^\ell$ equivalently.

3.3.2 STATIONARY RIGID GANG SCHEDULING ALGORITHM

In order to describe and specify the properties of that scheduling algorithm, we first formalize the definition of an arbitrary schedule.

Definition 3.2 (Schedule)**.** *A schedule $S_{P_q} : \mathbb{R} \mapsto \mathbb{T} \cup \{\perp\}$ for a processor P_q with $q \in \{0, \ldots, M-1\}$ is a mapping from the continuous time domain to the task that is executed at time t or to $\perp$ if processor P_q idles, i.e.,*

$$S_{P_q}(t) = \begin{cases} \tau_i & \text{if task } \tau_i \text{ is executed on } P_q \text{ at time } t \\ \perp & \text{if } P_q \text{ is idle at time } t \end{cases} \tag{3.1}$$

Stationary Gang Assignment. Each task is assigned and restricted to a subset of processors to execute on, which do not change in time, and is thus called *stationary gang assignment*. While conceptually related to partitioned scheduling, the term stationary is used to emphasize that the assigned processors of any two rigid gang tasks may have non-empty intersection, i.e., the assignment is not a partition.

stationary gang assignment

Definition 3.3 (Stationary Gang Assignment)**.** *A stationary gang assignment $A_i^* \subseteq \{P_0, P_1, \ldots, P_{M-1}\}$ of a rigid gang task τ_i is a subset of processors of size $|A_i^*| = E_i \leq M$, which are assigned to execute jobs of task τ_i.*

At first we will assume that a stationary gang assignment is given for each task in the task set, and revisit the problem of stationary gang assignment construction with performance guarantees in the later Section 3.3.4. Given the formal definitions of stationary gang assignments and schedules, the properties of a stationary gang schedule can be formalized as follows.

Definition 3.4 (Gang Schedule)**.** *A schedule for a multiprocessor system satisfies the stationary gang property if for each task τ_i and its stationary gang assignment A_i^*, the following property holds:*

$$\bigwedge_{P_q \in A_i^*} [S_{P_q}(t) = \tau_i] \quad \textit{if and only if } \tau_i \textit{ is scheduled at time } t \tag{3.2}$$

where $[\cdot]$ denotes the iverson bracket, which evalutes to true if the inner predicate is met and false otherwise.

Whenever we argue about schedules, which satisfy the stationary gang property, we write $S_{A_i^*}(t) = \tau_i$, if task τ_i is scheduled on all the processors in $P_q \in A_i^*$ at time t.

gang task τ_i is called active *at time t if a job of τ_i is released and not yet finished*

Prioritization. Each task $\tau_i \in \mathbb{T}$ is assigned a task-level fixed-priority and we use the mapping $\Pi : \mathbb{T} \mapsto \mathbb{N}$ to denote the priority of task τ_i and say τ_j has higher priority than τ_i if and only if $\Pi(\tau_j) > \Pi(\tau_i)$. We assume that no two tasks have the same priority, i.e., there are sufficient priority levels and Π is injective. A gang task τ_i is called *active* at time t if a job of τ_i is released and not yet finished. On each processor, the stationary gang scheduler schedules the highest-priority job, subject to the stationary gang assignment and co-scheduling constraint, and preempts a lower-priority job if necessary.

Exemplary Schedule. An exemplary *preemptive task-level fixed-priority stationary rigid gang* schedule is shown in Figure 3.2 with two tasks τ_k and τ_i. The tasks, have the respective stationary gang assignments $A_k^* = \{P_2, P_3\}$ and $A_i^* = \{P_1, P_2, P_3\}$, and priorities $\Pi(\tau_k) < \Pi(\tau_i)$. The job of τ_i, which is released at time 1.5, preempts the job of task τ_k. Whenever the job of τ_i is preempted on A_i^*, the job of τ_k is the highest-priority job among all jobs requesting processors P_2 and P_3. In consequence, the job of τ_k is scheduled by the scheduling algorithm. The preemption of τ_i's job on the processor $P_1 \notin A_k^*$ is *transparent* to τ_k's job, which is similar to self-suspension shown in Figure 3.3.

3.3.3 STATIONARY RIGID GANG SCHEDULING ANALYSIS

This section presents the schedulability analyses for preemptive fixed-priority stationary gang scheduling for a task set $\mathbb{T}$ of sporadic constrained-deadline rigid gang tasks, provided that each task τ_i has a given stationary gang assignment A_i^* and a unique priority.

Our schedulability analyses in this section are inductive in the sense that we validate whether a task τ_k can meet its deadline constraint, provided that all the tasks with higher priorities than τ_k are validated to meet their deadlines

beforehand. Hence, the validation of schedulability iterates from the highest-priority task to the lowest-priority task in $\mathbb{T}$. Towards this, we present methods to analyze the interference exerted by a higher-priority task τ_i onto the task τ_k under analysis in Section 3.3.3.1. Specifically, our result shows that τ_i can be considered as a self-suspending task under certain circumstances. Due to this observation that some higher-priority tasks can be transformed into self-suspending tasks, we employ existing suspension-aware schedulability analysis and present our schedulability test for stationary gang scheduling in Section 3.3.3.3.

our result shows that τ_i can be considered as a self-suspending task under certain circumstances

3.3.3.1 *Interference Analysis*

By construction of the preemptive task-level fixed-priority stationary gang scheduling algorithm; considering the execution of the active job of task $\tau_k \in \mathbb{T}$, the stationary rigid gang scheduling algorithm always schedules the jobs, which have the highest priority (with respect to all other active jobs), assigned to at least one processor in the assignment A_k^*. Hence, to account for all causes of interference, we define an interference domain for any subset of processors as follows, and identify the subset of tasks contributing to the interference.

Definition 3.5. *Let $\Omega \subseteq \{P_0, \ldots, P_{m-1}\}$ then the interference domain $I(\Omega)$ of the processors specified in Ω is given by $I(\Omega) := \{\tau_i \in \mathbb{T} \mid \Omega \cap A_i^* \neq \emptyset\}$.*

Based on this definition, we can prove a necessary condition for a job of task $\tau_i \neq \tau_k$ to be able to preempt a job of task τ_k, which is summarized in the following lemma.

Lemma 3.1. *Let jobs J_i^ℓ and J_k^h of tasks τ_i and τ_k be active at time t and $S_\mathbb{P}$ denote a feasible generated task-level fixed-priority stationary rigid gang schedule, i.e., satisfies Definition 3.2, then*

$$J_i^\ell \text{ preempts } J_k^h \text{ at time } t \text{ in schedule } S_\mathbb{P} \implies \tau_i \in I(A_k^*) \text{ and } \Pi(\tau_i) > \Pi(\tau_k) \quad (3.3)$$

summarized as $\psi_k = \{\tau_i \in I(A_k^*) \mid \Pi(\tau_i) > \Pi(\tau_k)\}$.

Proof. We prove the contrapositive, i.e., if $\tau_i \notin I(A_k^*)$ or $\Pi(\tau_i) \leq \Pi(\tau_j)$ then there exists no job of task τ_i, which preempts a job of τ_k at any time. In the first case, let $\tau_i \notin I(A_k^*)$ then there is no job J_i^ℓ which requests to be scheduled on processors in A_k^* and thus clearly can not preempt any job J_k^h executing on processors A_k^*. In the second case, let $\Pi(\tau_i) \leq \Pi(\tau_k)$ and $\tau_i \in I(A_k^*)$ then no job J_i^ℓ can preempt any job J_k^h, due to the not strictly greater priority. □

As a consequence of Lemma 3.1, only jobs generated by tasks from ψ_k can cause interference to the execution of a job of task τ_k under analysis. Moreover, by the definition of an interference domain, the response-time analysis problem for the task τ_k (under analysis) can be reduced to the response-time analysis of τ_k in an *equivalent* uniprocessor system.

That is, the schedulability of a gang task τ_k can be reduced to the schedulability of a sequential, i.e., non-gang, task with worst-case execution time C_k, which is subjected to the maximum interference by jobs of tasks in ψ_k. In the remainder of

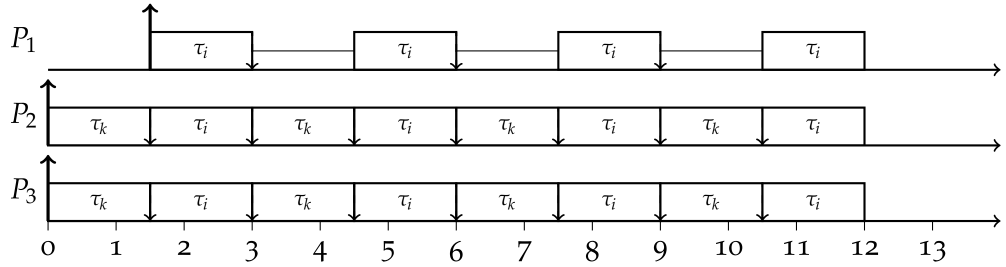

Figure 3.2: A cut-out of an exemplary task-level fixed-priority stationary rigid gang schedule with respect to the processors' local schedules of the three processors P_1, P_2, and P_3. In the shown example, the two tasks τ_k and τ_i are analyzed where τ_i has a higher priority than τ_k. Task τ_k is assigned to processors P_2 and P_3, and τ_i is assigned to P_1, P_2, P_3.

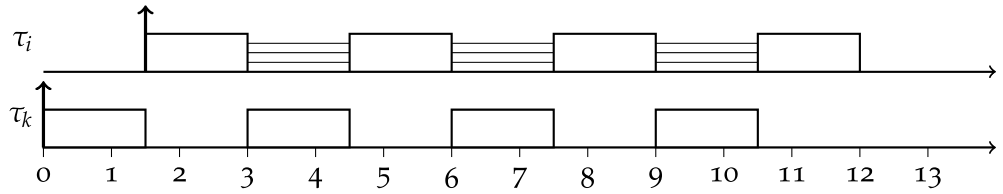

Figure 3.3: An illustration of the suspension induced behavior of task τ_i from the *perspective* of task τ_k under analysis as derived from Figure 3.2. The cause of the interference of the higher-priority task τ_i in the *schedule* is transparent to the *local schedule* of τ_k. This transparent behaviour is modeled as *self-suspension*.

this subsection, we show that the interfering behavior of task τ_i in ψ_k can be over approximated by the interference behaviour of a corresponding sequential task with dynamic self-suspension behavior, where the suspension-time depends on the stationary gang assignments of the interfering tasks.

3.3.3.2 *Induced Self-Suspension Behaviour*

local schedule

The link between preemptive stationary rigid gang schedules and the dynamic self-suspension behavior is illustrated in the schedule shown in Figure 3.2 and a corresponding *local schedule* with induced self-suspension behaviour is shown in Figure 3.3. Let $\mathbb{T}$ denote a set of rigid gang tasks, which is scheduled on the available processors by any *preemptive task-level fixed-priority stationary rigid gang scheduling* algorithm according to Definition 3.4.

job of τ_k is preempted as denoted by the small downwards-pointed arrow

We want to analyze the schedule for a job of task τ_k, i.e., the *local schedule* on τ_k's assigned processors P_2 and P_3. The job is released at time $t = 0$ and finishes at time $t = 10.5$, as shown in Figure 3.3. In this here discussed example, we assume that only tasks τ_i and τ_k use processors P_2, P_3 where τ_i has higher priority than τ_k. The respective stationary gang assignments are given by $A_i^* = \{P_1, P_2, P_3\}$ and $A_k^* = \{P_2, P_3\}$, and the remaining processors can be used by any other task in the set $\mathbb{T}$. Due to the arrival of a job of the higher-priority task τ_i at time $t = 1.5$, the job of τ_k is preempted as denoted by the small downwards-pointed arrow. During time $t = 1.5$ to time $t = 3$, the job of τ_i is the highest-priority job on all processors P_1, P_2, P_3 and is thus scheduled and executed. At time $t = 3$, the executed job of τ_i is preempted due to interference by some higher-priority

jobs (denoted by the line), which are transparent to the local schedules of the processors P_2, P_3. During the times, in which the job of τ_i is preempted on P_1, that job is not eligible to be scheduled on any of the remaining assigned processors P_2, P_3, and thus the job of τ_k is executed. Hence, if we only analyze the execution of τ_k with respect to its assigned processors, then transparent preemption of τ_i is similar to self-suspension behavior, which needs to be accounted for in the response-time analysis of τ_k.

transparent preemption of τ_i is similar to self-suspension behavior

Definition 3.6 (Dynamic Self-Suspension [CBH+17b]). *A task is said to have dynamic self-suspension behavior if an active task can transition from a ready state into a suspended state, in which the task is exempted from the scheduling decisions, and resume into a ready state at any time. The cumulative amount of time that an active task τ_i can spend in a suspended state is upper-bounded by a parameter S_i.* □

In the following, we formalize and explain how these task model substitutions can be safely obtained. Before moving into the formal proof, we present the conditions, which hold for our scheduling policy.

Definition 3.7. *In any preemptive task-level fixed-priority stationary rigid gang schedule, a job J_i^ℓ of a task $\tau_i \in \mathbb{T}$ is executed at time t, if and only if the following conditions are satisfied:*

1. *Job J_i^ℓ is released and not yet finished* (active) *at time t*
2. *There is no active job of a task $\tau_j \in \psi_i$ executed at time t*

Based on Definition 3.7, the *induced self-suspension behaviour* with respect to a specific job under analysis, can be defined as follows.

induced self-suspension behaviour

Definition 3.8. *In any task-level preemptive fixed-priority stationary rigid gang schedule, a job J_i^ℓ of task $\tau_i \in \psi_k$ is in an induced suspension state at time t with respect to a job of task τ_k under analysis if and only if the following conditions are satisfied:*

1. *Job J_i^ℓ is active at time t*
2. *Job J_i^ℓ is not executed at time t*
3. *Job J_i^ℓ has the highest priority among all active jobs on the processors in the assignment A_k^*, i.e., $\Pi(\tau_i) \geq \max\left\{\Pi(\tau_j) \,\middle|\, \text{All jobs } J_i^\ell \text{ of task } \tau_j \in \psi_k \text{ active at } t\right\}$*

In the following, we seek to partition the set of tasks that can generate jobs that interfere with a job of task τ_k under analysis, namely ψ_k, into those that can be in an *induced suspension state* at some time t, and into those that can not. To that end, for a task τ_k under analysis, we identify the subset of suspension inducing tasks in $\mathbb{T}$, which can cause the conditions stated in Definition 3.8 for a task in ψ_k. In consequence, we consider those tasks in ψ_k for which no suspension inducing tasks exist, to be not self-suspending with respect to τ_k.

we identify the subset of suspension inducing tasks in $\mathbb{T}$

Definition 3.9 (Self-Suspension Inducing Tasks). *The set of tasks (that release jobs) which can induce self-suspension behavior of jobs of task τ_i when analyzing task τ_k is denoted by*

$$V_{i,k} = \left\{\tau_j \in \psi_i \mid \tau_j \notin \psi_k\right\} \tag{3.4}$$

We now prove that the set of *self-suspension inducing tasks* according to Definition 3.9 is a subset of $V_{i,k}$ as stated in Eq. (3.4) in the following Lemma 3.2.

Lemma 3.2. *Suppose that task τ_i is in a suspension induced state at time t with respect to a task τ_k under analysis, then at least one job of a task in $V_{i,k}$ is executed at time t.*

Proof. By assumption, a job J_i^ℓ of task τ_i is in a suspension induced state at time t with respect to the job under analysis of task τ_k according to Definition 3.8.

By the properties stated in Item 1 and Item 2 in Definition 3.8, there exists a job J_i^ℓ of task τ_i which is active and is not executing at time t. In conjunction with the properties of any preemptive fixed-priority stationary gang schedule from Definition 3.7, we conclude that since J_i^ℓ is not executing, there exists an active job J_j^h of a task $\tau_j \in \psi_i$ which is executed at time t. Note that since J_j^h is executed it must also be active, i.e., more formally, we have shown that if

$$\exists J_i^\ell \textit{ in suspension induced state at } t \implies \exists \tau_j \in \psi_i \textit{ with active job } J_j^k \quad (3.5)$$
$$\implies \exists \tau_j \in \psi_i \quad (3.6)$$

It remains to show that $\tau_j \notin \psi_k$, which we prove by contradiction. Assume that $\tau_j \in \psi_k$, then from Item 3 in Definition 3.8 it follows that $\Pi(\tau_j) > \Pi(\tau_i)$, which would contradict the assumption that a job J_i^ℓ is in a suspension induced state. In conclusion we have that if

$$\tau_i \textit{ can be in an suspension induced state } \implies \exists \tau_j \in V_{i,k} \quad (3.7)$$

□

Now, we can provide a safe upper bound of the self-suspension time if $V_{i,k}$ is not empty.

Theorem 3.3. *Suppose that $\Pi(\tau_i) > \Pi(\tau_k)$ and $R_i \leq D_i \leq T_i$, where R_i is an upper bound on the worst-case response time of task τ_i, which was already verified beforehand. The amount of time $S_{i,k}$ that any job of an active task τ_i self-suspends with respect to a job of τ_k under analysis is bounded from above by*

$$S_{i,k} \leq \min \left\{ R_i - C_i, \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil \right) \cdot C_j \right\} \quad (3.8)$$

Proof. By Definition 3.8, all times t during the active interval starting from the release time and ending with the finishing at time of a job J_i^ℓ of a task τ_i is well defined. Moreover, by the assumption of schedulability of τ_i it is guaranteed that $f_i^\ell \leq a_i^\ell + R_i \leq a_i^\ell + D_i$. We prove both parts of the minimum in Eq. (3.8) to be upper-bounds individually.

First Case. Suppose that the cumulative amount of time, that a job J_i^ℓ of τ_i is in a suspension induced state during the interval $[a_i^\ell, f_i^\ell]$, is strictly more than $R_i - C_i$ for contradiction. This implies that J_i^ℓ has only completed $f_i^\ell - a_i^\ell - R_i + C_i$ amount of computation. This however violates the assumption that R_i is the worst-case response time of τ_i.

Second Case. By Lemma 3.2, we know that with respect to the analysis of a job of task τ_k, the suspension induced behaviour of jobs of τ_i (which interfere with τ_k) is caused by preemption due to jobs of tasks in the set $V_{i,k}$. Since $R_j \leq T_j$ and $\Pi(\tau_j) > \Pi(\tau_k)$ for every task $\tau_j \in V_{i,k}$, we know that the amount of time that jobs of task τ_j are executed during the active interval of J_i^ℓ is at most $\left(1 + \lceil \Delta_i / T_j \rceil\right) \cdot C_j$ where $\Delta_i := f_i^\ell - a_i^\ell$. This can be proved by showing that the jobs of τ_j, which are executed in the interval $[a_i^\ell, a_i^\ell + \Delta)$ are (i) at most only one job released prior to a_i, and (ii) the amount of jobs that we get by releasing jobs with minimal inter-arrival time. This is typically done with the concept of *carry-in* jobs. Since $R_i \leq T_i$, there is at most one carry-in job of τ_j released before a_i^ℓ. Summing up all tasks in $V_{i,k}$, we have

$$S_{i,k} = \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{\Delta_i}{T_j} \right\rceil\right) \cdot C_j \leq \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil\right) \cdot C_j \tag{3.9}$$

where the inequality is due to the assumption that $\Delta_i \leq R_i$. □

The upper-bound in Theorem 3.3 is not tight as $V_{i,k}$ is a superset of tasks, which induce suspension behaviour of task τ_i with respect to τ_k. For completeness, we state the following corollary as a direct implication of Theorem 3.3.

Corollary 3.4. *If $V_{i,k}$ is empty, then task τ_i does not have any self-suspension behavior, i.e., $S_{i,k} = 0$ when analyzing task τ_k.*

Proof. This is because the right-hand side of Eq. (3.8) is 0 under this condition. □

3.3.3.3 *Schedulability Analysis*

After analyzing the link between the stationary rigid gang scheduling problem and the dynamic self-suspension problem, we now construct a worst-case response time analysis and schedulability analysis for each task $\tau_k \in \mathbb{T}$. More precisely, we provide a response-time bound based on suspension-aware worst-case response-time analyses on uniprocessor systems.

On the basis of Theorem 3.3 and Corollary 3.4, we can safely upper bound the interference of task τ_k. Given the collection ψ_k of higher-priority tasks, which interfere with τ_k, we substitute the task model of $\tau_i \in \psi_k$ with a dynamic self-suspension task model as follows:

Definition 3.10. *Let a sporadic rigid gang task $\tau_i \in \psi_k$ be transformed to the corresponding self-suspending task $\overline{\tau_i} = (C_i, D_i, T_i, S_{i,k})$ with the same C_i, D_i, and T_i as τ_i, where*

$$\begin{cases} S_{i,k} = \min\left\{R_i - C_i, \sum_{\tau_j \in V_{i,k}} \left(1 + \left\lceil \frac{R_i}{T_j} \right\rceil\right) \cdot C_j\right\} & \text{if } V_{i,k} \neq \emptyset \\ S_{i,k} = 0 & \text{otherwise} \end{cases} \tag{3.10}$$

and $V_{i,k}$ is defined as in Definition 3.9. Moreover, let ψ_k^{sus} denote the set of all transformed tasks in ψ_k.

self-suspension aware worst-case response-time analyses are sustainable with respect to reducing the self-suspension time

Please note that self-suspension aware worst-case response-time analyses are sustainable with respect to reducing the self-suspension time. That is, if a job suspends for less amount of time than specified as an upper-bound, the schedulability analysis remains valid. This is explainable by the fact that the collection of schedules, which can be generated for a task with suspension time $S - \epsilon$ for $\epsilon > 0$ is a subset of schedules, which can be generated for a task with suspension time S. The worst-case response-time, which is the supremum of all response-times of any job, in any legally possible schedule, can in consequence only increase for a super set.

Corollary 3.5. *Suppose that all higher-priority tasks $\tau_1, \ldots, \tau_{k-1}$ (with respect to task τ_k under analysis) are already verified to be schedulable given their respective stationary gang assignments $A_1^*, \ldots, A_{k-1}^*$ by a preemptive task-level fixed-priority stationary rigid gang scheduling algorithm. A sporadic constrained-deadline rigid gang task τ_k with stationary gang assignment A_k^* is schedulable by the preemptive task-level fixed-priority stationary rigid gang scheduling algorithm if the worst-case response time R_k of task τ_k is at most $D_k \leq T_k$, given the interference of ψ_k^{sus} on one processor under the same priority assignment.*

We adopt the current sound state-of-the-art task-level fixed-priority self-suspension aware uniprocessor schedulability analyses by Chen et al. [CNH16] for the proposed stationary gang scheduling schedulability analyses. Based on the results of Corollary 3.5, the correctness of the following corollaries follows directly from the related proofs in [CNH16].

Corollary 3.6. *A sporadic constrained-deadline rigid gang task $\tau_k \in \mathbb{T}$ is schedulable by a preemptive task-level fixed-priority stationary rigid gang scheduling algorithm if*

$$\exists 0 < t \leq D_k, \quad C_k + \sum_{\tau_i \in \psi_k^{sus}} \min\{C_i, S_{i,k}\} + \sum_{\tau_i \in \psi_k^{sus}} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \leq t \tag{3.11}$$

under the assumption that $\tau_1, \ldots, \tau_{k-1} \in \mathbb{T}$ are already verified to be schedulable.

Corollary 3.7. *A sporadic constrained-deadline rigid gang task $\tau_k \in \mathbb{T}$ is schedulable by a preemptive task-level fixed-priority stationary rigid gang scheduling algorithm if*

$$\exists 0 < t \leq D_k, \quad C_k + \sum_{\tau_i \in \psi_k^{sus}} \left\lceil \frac{t + R_i - C_i}{T_i} \right\rceil \cdot C_i \leq t \tag{3.12}$$

under the assumption that $\tau_1, \ldots, \tau_{k-1} \in \mathbb{T}$ are already verified to be schedulable.

Corollary 3.8. *Suppose that there are z tasks in ψ_k^{sus} that are indexed from the highest priority to the lowest priority. A sporadic constrained-deadline rigid gang task τ_k is schedulable by a preemptive task-level fixed-priority stationary rigid gang scheduling algorithm if there is a vector $\vec{x} = [x_0, x_1, \ldots, x_{z-1}]$ with $x_i \in \{0, 1\}$ for $i \in \{0, \ldots, z-1\}$ such that*

$$\exists 0 < t \leq D_k, \quad C_k + \sum_{\tau_i \in \psi_k^{sus}} \left\lceil \frac{t + \sum_{j=i-1}^{z-1} S_{j,k} \cdot x_j + (1 - x_{i-1})(R_i - C_i)}{T_i} \right\rceil \leq t \tag{3.13}$$

The provided schedulability analyses in Corollary 3.6, Corollary 3.7, and Corollary 3.8 can be solved via fix-point iteration techniques.

Corollary 3.6, Corollary 3.7, and Corollary 3.8 can be solved via fix-point iteration techniques

More precisely, let $W_k(t)$ denote the left-hand sides of the inequalities in the above corollaries and let $\epsilon > 0$, we then verify all test-points $t_0 = W_k(\epsilon)$, $t_1 = W_k(t_0), \ldots, t_n = W_k(t_{n-1})$ until convergence is reached or $t_n > D_k$.

Due to the fact that the above equations are step-functions and can thus only change at discontinuity points of $W_k(t)$, the amount of test-points is at most $k \cdot D_k / \min_{i<k}\{T_i\}$ resulting in pseudo-polynomial time-complexity. In the remainder of this paper, we only use the landau notation $\mathcal{O}(kD_k)$ for time-complexity, since the scaling of the deadline does not change the asymptotic complexity.

As discussed in [CNH16], neither of the schedulability analyses in Corollary 3.6 and Corollary 3.7 dominate each other analytically and are incomparable. The authors also showed that the test in Corollary 3.8 dominates those in Corollary 3.6 (i.e., Lemma 17 in [CNH16]) and Corollary 3.7 (i.e., Lemma 16 in [CNH16]). To find a vector $\vec{x}$ for the worst-case response time analysis in Corollary 3.8 in a computationally efficient manner, the authors suggest to use three distinct vectors. One is based on a linear approximation, one sets all elements of $\vec{x}$ to 0, and one sets the entry x_{i-1} in $\vec{x}$ to 1 if $S_{i,k} \leq C_i$, and 0 otherwise for $i \in \{1, \ldots, z\}$. Specifically, in the case when the entries in $\vec{x}$ are all 0, Eq. (3.13) is the same as Eq. (3.12). In our evaluations we use the analysis from Corollary 3.8 with the above distinct three vectors and choose the best one. That is, a task is deemed to be schedulable if it is schedulable for at least one of the three vectors.

3.3.4 STATIONARY RIGID GANG ASSIGNMENT ALGORITHM

As described in Section 3.1, finding optimal schedules for the general rigid gang scheduling problem has been shown to be NP-hard in the strong sense – even in the case in which all tasks have the same period and the same deadline. Even the additional enforcement of stationarity, i.e., to assign a gang onto a fixed set of processors does not reduce the computational complexity. Moreover, by the trivial reduction of the special case of a task set with gang tasks of size one, an optimal algorithm for our sporadic stationary rigid gang scheduling algorithm could solve the partitioned scheduling problem of sporadic tasks onto multiple identical processors, which was shown however to be NP-hard in the strong sense by Baruah et al. [BF05]. Due to the issue of computational complexity of an optimal algorithm, our objective is to find approximation algorithms to solve the stationary rigid gang assignment problem for sporadic task sets with provable performance bounds, which is described in the remainder of this section.

enforcement of stationarity, i.e., to assign a gang onto a fixed set of processors does not reduce the computational complexity

3.3.4.1 *Processor Assignment Problem & Complexity*

It is evident that in our proposed preemptive task-level fixed-priority stationary rigid gang scheduling algorithm, both the concrete priorities, and the concrete gang assignments to processors, determine the schedulability of a task set $\mathbb{T}$.

At first glance, the partitioning problem of a set of sporadic rigid gang tasks onto multiple identical processors under fixed-priority scheduling seems similar

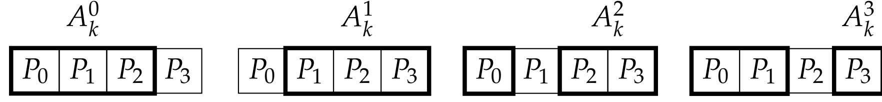

Figure 3.4: Consecutive stationary gang assignments $A_k^0, A_k^1, A_k^2, A_k^3$ of a gang task τ_k with $E_k = 3$ on a system using 4 identical processors P_i for $i \in \{0, \ldots, 3\}$. The four distinct assignments are generated by a sliding window of size 3.

to the problem of partitioning normal sporadic tasks onto multiple identical processors under fixed-priority scheduling. However, there are two distinct differences, namely;

1. Each task $\tau_k \in \mathbb{T}$ under consideration can have M choose E_k many distinct gang assignments in terms of gang to processor mappings, instead of just M partitions in case of normal sporadic tasks;
2. And the complex dependency of gang assignments and the resulting interference behaviour of higher-priority tasks.

To that end, one may try to identify equivalence classes of all possible stationary rigid gang assignments, where two assignments are equivalent if and only if the resulting interference caused by all higher-priority tasks of τ_k is identical. If an equivalence class is found that satisfies the schedulability conditions then any representative of that class can be chosen for the gang assignment. A trivial example for such an equivalence class is the gang assignment of the highest-priority task, in which case all possible gang assignments are equivalent, since there are no interfering tasks. However, finding all equivalence classes results in an exhaustive exploration of all possible solutions, which is computationally expensive especially for larger task sets and thus infeasible. Please recall that our intention is to identify a class of computationally feasible gang assignment algorithms, which allow to formulate *provable worst-case performance guarantees* with respect to any optimal rigid gang scheduling algorithm.

In order to obtain analytical worst-case performance guarantees, it is mandatory to find (preferably small) upper bounds of interference caused by higher-priority tasks. An interesting question in that regard is whether there may exist a class of more specialized gang assignment policies for which such worst-case performance guarantees can be proven.

consecutive stationary gang assigments

We answer this question positively by proposing the class of *consecutive stationary gang assigments*, which is formalized in the following definition and subsequently show the beneficial theoretical properties of this class of assigments for our analyses. We note however that other gang assignment policies can be explored starting from the consecutive stationary gang assignments and thus the approximation properties can be kept whilst improving the schedulability using any heuristic.

Definition 3.11. *A consecutive stationary gang assignment A_k^ℓ for $\ell \in \{0, 1, \ldots, M-1\}$ of a gang task τ_k in a system of M processors is a set of consecutive processor indices*

$$A_k^\ell := \{\ell \bmod M,\ (\ell+1) \bmod M,\ \ldots,\ (\ell+E_k-1) \bmod M\} \tag{3.14}$$

where $|A_k^\ell| = E_k \leq M$. Moreover, we use A_k to refer to the set $\left\{A_k^0, \ldots, A_k^{M-1}\right\}$.

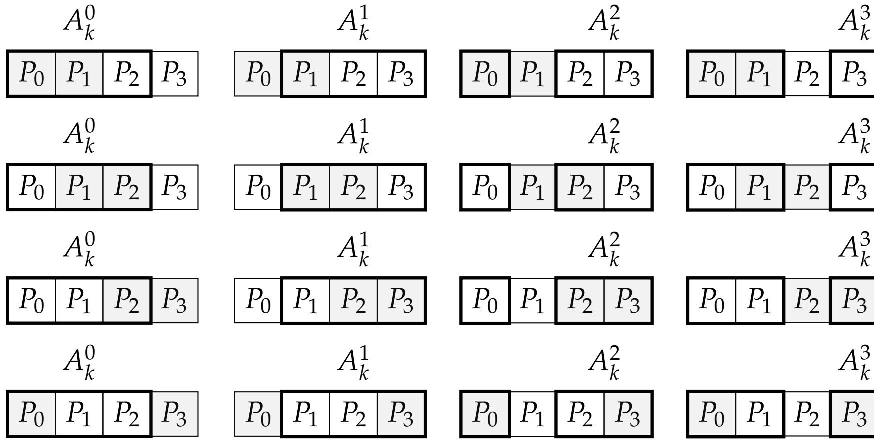

Figure 3.5: Enumeration of all consecutive stationary gang assignments of a task τ_k (black window) under the condition of a given consecutive stationary gang assignment of a higher-priority task (light gray window).

To clarify the above definition, an exemplary consecutive stationary gang assignments of a task τ_k with $E_k = 3$ on a platform of 4 processors is illustrated in Figure 3.4. Intuitively, the consecutive stationary gang assignments are generated by a sliding window of length 3. Using Eq. (3.14) yields that $A_k^0 = \{0, 1, 2\}$, $A_k^1 = \{1, 2, 3\}$, $A_k^2 = \{0, 2, 3\}$, and $A_k^3 = \{0, 1, 3\}$. In consequence $A_k = \{\{0, 1, 2\}, \{1, 2, 3\}, \{0, 2, 3\}, \{0, 1, 3\}\}$. An important implication of this consecutive stationary gang assignment class is that each task τ_k has exactly M distinct assignments, irrespective of its gang size E_k, in contrast to the gang size dependent M choose E_k in the general case, summarized in the following observation.

Observation 3.9. *The number of distinct consecutive stationary gang assignments of a rigid gang task τ_k with gang size $E_k \leq M$ is exactly the number of processors M irrespective of its gang size E_k.*

Another dimension in the consecutive stationary gang assignment algorithm is the order in which the different tasks in $\mathbb{T}$ are assigned to the processors. In our algorithm, we devise gang assignments in priority-order under the premise that all higher-priority tasks have already been verified to be fixed-priority schedulable using the determined consecutive stationary gang assignment. By this restriction, in the assignment iteration of the k-th task in $\mathbb{T}$, we only have to determine the assignment dependent interference behaviour of all higher-priority tasks with respect to the candidate consecutive stationary gang assignment A_k^ℓ for $\ell \in \{0, \ldots, M-1\}$. By virtue of these restrictions, we are able to find upper-bounds for the number of consecutive stationary gang assignments of τ_k, in which higher-priority tasks have self-suspension behaviour and non self-suspension behaviour, respectively. That means, we are able to argue that in at most x out of the M consecutive stationary gang assignments, a higher-priority task τ_i has self-suspension behaviour, irrespective of the actual consecutive assignment of τ_i.

By this restriction, in the assignment iteration of the k-th task in $\mathbb{T}$, we only have to determine the assignment dependent interference behaviour of all higher-priority tasks with respect to the candidate consecutive stationary gang assignment A_k^ℓ

A motivating example is illustrated in Figure 3.5, where each column shows a single (out of M) consecutive stationary gang assignments of a task τ_k that is subject to assignment and schedulability analysis, with respect to a given

consecutive stationary gang assignment of some higher-priority rigid gang task τ_i. Each row shows a different (out of M) consecutive stationary assignment of said higher-priority task τ_i as indicated by the light gray window. According to previous results in Section 3.3.3, task τ_k suffers interference from task τ_i if and only if $A_k^* \cap A_i^* \neq \emptyset$ where $A_k^* \in A_k$ and $A_i^* \in A_i$ denote the chosen consecutive stationary gang assignment of the respective tasks. Intuitively, any column and row entry in Figure 3.5 in which the two colored windows intersect represents an interference configuration. If for a column (consecutive assignment of τ_k) there exists at least one row (consecutive assignment of τ_i) in which both windows intersect then τ_i interferes with τ_k for the consecutive assignment under consideration in at least one possible configuration.

our objective is to classify the worst-case interference behaviour of task τ_i on τ_k irrespective of the concrete assignments of A_i^ℓ and A_k^ℓ

Intuitively, our objective is to classify the worst-case interference behaviour of task τ_i on τ_k irrespective of the concrete assignments of A_i^ℓ and A_k^ℓ. In the provided example, it can be observed that all consecutive assignments of τ_k suffer interference from τ_i and thus irrespective of the concrete gang assignment A_i^ℓ.

In the following Lemma 3.10 we prove that in general there are at most $E_k + E_i - 1$ out of the M consecutive stationary gang assignments of τ_k, in which τ_i interferes with τ_k. To determine whether τ_i exhibits self-suspension behaviour with respect to the interference on τ_k or not additionally depends on the assignments of all of τ_i's higher-priority tasks. In the subsequent Lemma 3.12, we then bound the number of consecutive stationary gang assignments of τ_k in which τ_i can not exhibit self-suspension behaviour from below and use this lower-bound to derive an upper-bound.

Lemma 3.10. *Given a rigid gang task τ_k under analysis, each higher-priority rigid gang task τ_i is in the interference domain of the processors in A_k^*, i.e., $\tau_i \in I(A_k^*)$, in at most $E_i + E_k - 1$ of the M-many consecutive stationary gang assignments.*

Proof. Let the tasks in $\mathbb{T}$ be indexed according to their priorities in increasing order and let the consecutive stationary gang assignments for some higher-priority tasks $i < k$ be given by the following processor indices:

$$A_i^* = \{j \bmod M,\ (j+1) \bmod M,\ \dots,\ (j+E_i-1) \bmod M\} \tag{3.15}$$

where $j \in \{0, 1, \dots, M-1\}$ is already given (fixed). Furthermore, let

$$\{\ell + h \bmod M,\ (\ell+h+1) \bmod M,\ \dots,\ (\ell+h+E_k-1) \bmod M\} \tag{3.16}$$

denote the processor indices of a consecutive stationary gang assignment of task τ_k after the h-iteration for some arbitrary initial $\ell \in \{0, 1, \dots, M-1\}$ (we only need this to show that this works for an arbitrary initial position and can be set to 0 for sake of comprehension). Then let h' denote the first iteration such that $(\ell + h' + E_k - 1) \bmod M \equiv j - 1 \bmod M$ (we shift the window of consecutive stationary gang assignments of task τ_k to the border of the window of A_i^*, i.e., the two consecutive stationary gang assignments intersect in the next iteration for the first time. By the above condition it follows that

$$(\ell + h') \bmod M \equiv (j - E_k) \bmod M. \tag{3.17}$$

We have to iterate further z assignments unil the index of the first processor in the allocation of τ_k, i.e., $(\ell + h' + z) \bmod M \equiv (j + E_i - 1) \bmod M$ coincides with the index of the last processor in the assignment of task τ_i. More formally, we seek to find the smallest $z > 0$ such that:

$$(\ell + h' + z) \bmod M \equiv (j + E_i - 1) \bmod M \tag{3.18}$$

$$((\ell + h') \bmod M) + (z \bmod M) \equiv (j + E_i - 1) \bmod M \tag{3.19}$$

$$(j - E_k + z) \bmod M \equiv (j + E_i - 1) \bmod M \text{ by Eq. (3.17)} \tag{3.20}$$

which implies that $z = E_i + E_k - 1$, i.e., z many consecutive stationary gang assignments yield an intersection of both tasks. □

It is complex to determine whether or not τ_i exhibits self-suspension behaviour with respect to the interference on τ_k, since this depends on the assignments of all of τ_i's higher-priority tasks. However, we can prove a lower-bound for the number of assignments, which can have self-suspension behaviour and derive an upper-bound by substracting that lower-bound from the number of all consecutive assignments M, in the following Lemma 3.11 and Lemma 3.12.

Lemma 3.11. *Let A_k^* and A_i^* be two consecutive stationary gang assigments of task τ_k under analysis and a higher-priority task τ_i. If $A_k^* \subseteq A_i^*$ then τ_i has no self-suspension inducing behaviour with respect to τ_k according to Definition 3.9.*

Proof. We prove this lemma by contradiction. Let τ_i have self-suspension inducing behavior with respect to τ_k by assumption then according to Definition 3.9 there exists a task τ_ℓ with higher priority than τ_i (and subsequently higher priority than τ_k) such that $A_\ell^* \cap A_i^* \neq \emptyset$ and $A_\ell^* \cap A_k^* = \emptyset$. This implies that $A_k^* \not\subseteq A_i^*$, which contradicts the assumption $A_k^* \subseteq A_i^*$. □

In the next lemma, we formally prove an upper-bound of the number consecutive stationary gang assignments of a task τ_k under analysis, in which a higher-priority task τ_i has self-suspension inducing behavior with respect to task τ_k.

Lemma 3.12. *For a rigid gang task τ_k under analysis, there are at most* $\min\{2E_k - 1, E_i + E_k - 1\}$ *many consecutive stationary gang assignments, in which a higher-priority task τ_i has self-suspension behavior with respect to task τ_k.*

Proof. For the requirement of the condition $A_k^* \subseteq A_i^*$ to hold, it must be that $E_i \geq E_k$. Then it is evident that $E_i - E_k$ many of the M consecutive stationary assignments A_k^* satisfy this necessary containment property.

Moreover from Lemma 3.10, we know that at most $E_k + E_i - 1$ many consecutive stationary gang assignments cause an intersection of consecutive stationary gang assignments of task τ_i and task τ_k, and can thus potentially have self-suspension inducing behaviour. We hence substract $\max\{E_i - E_k, 0\}$, namely the number of consecutive stationary gang assignments in which self-suspension behavior of τ_i is impossible, from the above. In the case that $E_k \leq E_i$ we have $(E_k + E_i - 1) - E_i + E_k = 2E_k - 1$. Since $2E_k - 1 \leq E_i + E_k - 1$ implies that $E_k \leq E_i$, we can write $\min\{2E_k - 1, E_i + E_k - 1\}$. □

The results from Lemma 3.10 and Lemma 3.12 will be used for the approximation, namely speedup factor analysis of Algorithm 1 and is described in-depth in Section 3.3.4.2 and the following Section 3.3.4.3.

3.3.4.2 *Consecutive Stationary Gang Assignment*

In this section, we present an assignment algorithm, which is capable of providing provable performance guarantees with respect to an optimal rigid gang scheduling algorithm by using the previously derived worst-case interference analyses.

Partitioning Strategy. In the partitioning algorithm of normal tasks, several approaches, namely *first-fit*, *best-fit*, or *worst-fit* are used to assign an element, subject to partition, to a bin. All of these strategies are dependent on an order of the elements that are subject to partitioning, i.e., each different ordering of elements results in a different partitioning. In the context of scheduling algorithms many strategies are common to obtain an ordering such as *Largest-Utilization First (LUF)* or *deadline-monotonic (DM)* orderings.

Deadline Monotonic. We use deadline-monotonic priority assignment and index the tasks in increasing deadline order. That is, task τ_i has a higher priority than task τ_k if index $i < k$. Due to the additional restrictions described above, it is possible to prove interference bounds and in consequence approximation guarantees in terms of schedulability for any stationary gang assignment algorithm, which uses the following algorithm as a basis.

Algorithm 1 DM Stationary Rigid Gang Schedulability Analysis and Assignment.

1: Sort task set $\mathbb{T}$ such that $D_i \leq D_j$ for $i < j$ (ties are broken arbitrarily);
2: **for** k in $\{1, 2, \dots, n\}$ **do**
3:     **for** ℓ in $\{0, 1, \dots, M-1\}$ **do**
4:         $\psi_k(A_k^\ell)$:= generate ψ_k given the candidate A_k^ℓ from Definition 3.11;
5:         $\psi_k^{sus}(A_k^\ell)$:= transform from $\psi_k(A_k^\ell)$ using Definition 3.10;
6:         **if** $\tau_k \cup \psi_k^{sus}$ is schedulable according to any self-suspension aware uniprocessor schedulability test (from Corollary 3.6, 3.7 and 3.8) **then**
7:             assign $A_k^* \leftarrow A_k^\ell$;
8:             break;
9:     **return** no feasible consecutive stationary gang assignments can be found;
10: **return** feasible consecutive stationary gang assignment for each task;

Algorithm Design. We first sort the tasks according to the relative deadlines to have a deadline-monotonic partitioning strategy. Starting from the highest-priority task, i.e., shortest relative deadline, we iteratively consider the consecutive stationary gang assignment candidates $A_k^0, A_k^1, \dots, A_k^{M-1}$ and check whether task τ_k is schedulable for the considered consecutive stationary gang assignment. If for some $\ell \in \{0, 1, \dots, M-1\}$, the candidate A_k^ℓ is feasible, the consecutive gang assignment is determined for that task τ_k; otherwise, we move to the next candidate. If none of the M possible consecutive gang assignments is feasible, this assignment step fails and the algorithm returns failure.

In Corollary 3.5, we assume that all stationary gang assignments A_i^* are already chosen for all tasks τ_i with higher priority than task τ_k. To apply Corollary 3.5 for a worst-case response time analysis, the self-suspension induced interfering tasks from Definition 3.9 and non self-suspending interfering tasks from Definition 3.5 must be obtained.

To facilitate an efficient implementation, we use a matrix representation to indicate whether a task τ_i is assigned on processor P_j. Let ρ denote an $|\mathbb{T}| \times |\mathbb{P}|$ binary stationary gang assignment matrix in which

$$\rho(i,j) := \begin{cases} True & P_j \in A_i^* \\ False & P_j \notin A_i^* \end{cases} \tag{3.21}$$

Given the matrix ρ, the algorithm constructs the interference matrix

$$\Gamma(i,j) := \begin{cases} True & A_i \cap A_j \neq \emptyset \\ False & \text{otherwise} \end{cases} \tag{3.22}$$

by the boolean matrix multiplication $\rho \cdot \rho^T$, where ρ^T is the transpose matrix of ρ. That is, the multiplication operation of two elements is replaced with the *logical and* operation and the addition operation of two elements is replaced with a *logical or* operation. More precisely, each entry in the interference matrix is computed as follows:

$$\Gamma(i,j) = \bigvee_{m=0}^{M-1} \rho(i,m) \wedge \rho(j,m)$$

where an entry $\Gamma(i,j)$ is true only if task τ_i and task τ_j share at least one processor in their stationary gang assignments. The asymptotic time-complexity for the matrix multiplication is given by $\mathcal{O}(n^2 M)$ and the space complexity is given by $\mathcal{O}(nM)$. The complexity can be improved by using a state-of-the art binary matrix multiplication algorithm to sub-cubic complexity. The transformation of the higher-priority tasks τ_i with respect to the task under analysis τ_k into ψ_k, which is later needed to construct ψ_k^{sus}, can be done by the following operation:

$$\begin{cases} \tau_i \in \psi_k & \text{if } \bigvee_{\ell=0}^{i-1} \Gamma(\ell,i) \wedge \overline{\Gamma(\ell,k)} \\ \tau_i \notin \psi_k & \text{otherwise} \end{cases} \tag{3.23}$$

Time Complexity. Hereinafter, the asymptotic time complexity of Algorithm 1 is elaborated on. The interfering task set generation in Line 4 requires $\mathcal{O}(i)$ operations for each task τ_i for $i \in \{1, \ldots, k-1\}$ and therefore $\mathcal{O}(k^2)$ in total for one invocation. In the subsequent transformation step in Line 5, it is required to calculate the right-hand side of Eq. (3.8), which can either be done in $\mathcal{O}(1)$ if only $R_i - C_i$ is taken or $\mathcal{O}(i)$ if both terms are evaluated in Eq. (3.8) for each task τ_i in the interfering task set ψ_k. In Line 6, the utilized schedulability test from Corollary 3.6, Corollary 3.7, and Corollary 3.8 have $\mathcal{O}(kD_k)$ complexity. After considering the outer and inner loop in Line 2 and Line 3 and the deadline-monotonic ordering, the overall time complexity is given by $\mathcal{O}(n^3 M + n^2 M D_n)$.

In this section, we derive firstly a pseudo-polynomial time sufficient schedulability test in Theorem 3.13 and secondly a polynomial-time sufficient schedulability test for Algorithm 1 in Theorem 3.14. Afterwards, we analyze the approximation factor of Algorithm 1, namely the *speed up factor* with respect to any optimal rigid gang scheduling algorithm in Theorem 3.16.

Theorem 3.13. *A constrained-deadline rigid gang task $\tau_k := (C_k, E_k, D_k, T_k) \in \mathbb{T}$ is guaranteed to be feasibly schedulable by deadline-monotonic stationary rigid gang scheduling according to Algorithm 1 on M identical processors if $\exists t : 0 < t \leq D_k$ such that*

$$C_k + \sum_{i=1}^{k-1} \min\{2E_k - 1, E_k + E_i - 1\} \cdot \frac{C_i}{M} + \sum_{i=1}^{k-1} (E_k + E_i - 1) \cdot \left\lceil \frac{t}{T_i} \right\rceil \frac{C_i}{M} \leq t. \quad (3.24)$$

holds and the schedulability of the tasks $\tau_1, \ldots, \tau_{k-1}$ has already been verified, where the tasks are indexed in priority order.

Proof. Let $\mathbb{T} := \langle \tau_1, \ldots, \tau_n \rangle$ denote the priority ordered constrained-deadline rigid gang task set and let the consecutive stationary gang assignments $A_1^*, \ldots, A_{k-1}^*$ of the higher-priority tasks $\tau_1, \ldots, \tau_{k-1}$ be determined, and let all task be verified to be feasibly schedulable according to Algorithm 1.

Suppose that task τ_k is the first task such that Algorithm 1 can not find any feasible consecutive stationary gang assignment A_k^* for τ_k to be deemed schedulable. Then it must be that all self-suspension aware uniprocessor schedulability test from Corollary 3.6, 3.7 and 3.8 failed for all of the M possible consecutive stationary gang assignments $A_k^\ell \in A_k$. Since any feasible suspension-aware uniprocessor analysis can be used, we here choose the *suspension as blocking* based analysis from Corollary 3.6, due to its simplicity. More formally, for each possible consecutive stationary gang assignments A_k^ℓ for $\ell \in \{0, \ldots, M-1\}$, the condition

$$C_k + \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \min\{C_i, S_{i,k}\} + \left\lceil \frac{t}{T_i} \right\rceil C_i > t \quad (3.25)$$

holds for all $0 < t \leq D_k$, where the transformed self-suspension aware task set $\psi_k^{sus}(A_k^\ell)$ depends on the concrete consecutive stationary gang assignment A_k^ℓ as is explained in detail in the previous Section 3.3.4.1.

Based on the failure assumption in Eq. (3.25) that no consecutive stationary gang assignment A_k^ℓ for $\ell \in \{0, 1, \ldots, M-1\}$ suffices the schedulability condition, a superimposed necessary condition is derived by summing up the terms in the left-hand side and the right-hand side, accordingly. This yields that for all times $0 < t \leq D_k$ the following inequality holds.

a superimposed necessary condition is derived by summing up the terms in the left-hand side and the right-hand side, accordingly

$$MC_k + \sum_{\ell=0}^{M-1} \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \min\{C_i, S_{i,k}\} + \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i > Mt. \quad (3.26)$$

Since only the first summand in Eq. (3.26) depends on self-suspension induced behaviour, we split the inequality into two summands as follows

$$MC_k + \sum_{\ell=0}^{M-1} \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \min\{C_i, S_{i,k}\} + \sum_{\ell=0}^{M-1} \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i > Mt. \quad (3.27)$$

For the first summand in Eq. (3.27), we use the upper-bound for the number of consecutive stationary gang assignments of τ_k with respect to any assignments of higher-priority tasks τ_i in which at most $\min\{2E_k - 1, E_k + E_i - 1\}$ tasks have self-suspension induced behaviour as stated in Lemma 3.12. In consequence, we have that

$$\sum_{\ell=0}^{M-1} \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \min\{C_i, S_{i,k}\} \leq \sum_{\tau_i \in \psi_k} \min\{2E_k - 1, E_k + E_i - 1\} \cdot C_i \tag{3.28}$$

where in Eq. (3.28) we use the inclusion $\psi_k^{sus}(A_k^\ell) \subseteq \psi_k$.

With respect to the second summand in Eq. (3.27), the upper bound for the number of consecutive stationary gang assignments of A_k^ℓ that result in interference from a higher-priority task $\tau_i \in \psi_k^{sus}(A_k^\ell)$, irrespective of the already fixed concrete assignment A_i^*, from Lemma 3.10, we have that

$$\sum_{\ell=0}^{M-1} \sum_{\tau_i \in \psi_k^{sus}(A_k^\ell)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq \sum_{\tau_i \in \psi_k} (E_k + E_i - 1) \cdot \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{3.29}$$

where in Eq. (3.29) we use the inclusion property $\psi_k^{sus}(A_k^\ell) \subseteq \psi_k$.

By injecting the results from Eq. (3.29) and Eq. (3.28) into Eq. (3.27), we have the following necessary condition

$$C_k + \sum_{i=1}^{k-1} \min\{2E_k - 1, E_k + E_i - 1\} \cdot \frac{C_i}{M} + \sum_{i=1}^{k-1} (E_k + E_i - 1) \cdot \left\lceil \frac{t}{T_i} \right\rceil \frac{C_i}{M} > t. \tag{3.30}$$

for the non-schedulability of task τ_k according to Algorithm 1. Conversely, the negation of Eq. (3.30) yields a sufficient schedulability test for task τ_k when using Algorithm 1, which concludes the proof. □

The sufficient schedulability analysis in Eq. (3.24) has pseudo-polynomial time complexity. In the following theorem, we show how to derive a sufficient polynomial time test by reduction of Eq. (3.24) to an *integer-linear program* and deriving a hyperbolic bound using the *k2U*-Framework by Chen et al. [CHL15b]. While such constructed sufficient polynomial time tests are more pessimistic, the complexity reduction can be beneficial in contexts of online admission tests and is thus conducted in the following.

integer-linear program
k2U framework

Theorem 3.14. *A constrained-deadline rigid gang task* $\tau_k := (C_k, E_k, D_k, T_k) \in \mathbb{T}$ *is guaranteed to be feasibly schedulable by deadline-monotonic stationary rigid gang scheduling according to Algorithm 1 on M identical processors if the following condition holds*

$$\left(\frac{C_k'}{D_k} + 1\right) \cdot \prod_{j=1}^{k-1} \left(\frac{E_k + M - 1}{M} \cdot \frac{C_j}{T_j} + 1\right) \leq 2 \tag{3.31}$$

and tasks $\tau_1, \ldots, \tau_{k-1}$ *are already verified to be feasibly schedulable.*

Proof. Let the priority-ordered partial task set $\langle \tau_1, \ldots, \tau_{k-1} \rangle \subseteq \mathbb{T}$ be partitioned into $\mathbb{T}_1 := \{\tau_i \in \langle \tau_1, \ldots, \tau_{k-1} \rangle \mid T_i \geq D_k\}$ and $\mathbb{T}_2 := \{\tau_i \in \langle \tau_1, \ldots, \tau_{k-1} \rangle \mid T_i < D_k\}$.

Starting from the sufficient condition in Eq. (3.24) from Theorem 3.13 and the task set partitioning yields

$$C'_k := C_k + \sum_{\tau_i \in \mathbb{T}_1 \cup \mathbb{T}_2} \min\{2E_k - 1, E_k + E_i - 1\} \cdot \frac{C_i}{M} + \sum_{\tau_i \in \mathbb{T}_1} \frac{(E_k + E_i - 1) \cdot C_i}{M} + \sum_{\tau_i \in \mathbb{T}_2} (E_k + E_i - 1) \left\lceil \frac{t}{T_i} \right\rceil \frac{C_i}{M} \leq t. \tag{3.32}$$

Clearly any arbitrary number of evaluated test points in the range $(0, D_k]$ that suffice Eq. (3.32) imply schedulability. For sake of simpler notation, assume that $|\mathbb{T}_2| = k - 1$ and $\langle \tau_1, \tau_2, \ldots, \tau_k \rangle$ is indexed such that $\tau_i \prec \tau_j$ if the task's last release times (no later than D_k) satisfy $t_i \leq t_j$, where $t_i := \lfloor D_k / T_i \rfloor \cdot T_i$ for $i \in \{1, \ldots, k-1\}$ and $t_k = D_k$. Then τ_k is schedulable if

$$\exists t_j \in \{t_1, t_2, \ldots, t_k\} \quad C'_k + \sum_{i=1}^{k-1} \frac{E_k + E_i - 1}{M} \cdot \left\lceil \frac{t_j}{T_i} \right\rceil C_i \leq t_j \tag{3.33}$$

holds. Since each t_i denotes the last release time at or before D_k, we know that the next release $t_i + T_i > D_k$ must be strictly more than the deadline. Moreover since $t_k = D_k \geq t_j$ we have that $t_i + T_i > D_k \geq t_j$ or $\frac{t_i}{T_i} + 1 \geq \frac{t_j}{T_i}$ equivalently. Since the left-hand side is an integer due to the definition of t_i and the fact that the ceiling function is a monotone, we have that $\frac{t_i}{T_i} + 1 \geq \left\lceil \frac{t_j}{T_i} \right\rceil$.

The ordered set $\langle t_1, \ldots, t_k \rangle$ is split at t_j into $\langle t_1, \ldots, t_{j-1} \rangle$ and $\langle t_j, \ldots, t_k \rangle$ such that $t_j > t_i$ for $t_i \in \langle t_1, \ldots, t_{j-1} \rangle$ and $t_j \leq t_i$ for $t_i \in \langle t_j, \ldots, t_k \rangle$. This together with the before observation yields

$$C'_k + \sum_{i=1}^{j-1} \frac{E_k + E_i - 1}{M} \cdot \left\lceil \frac{t_j}{T_i} \right\rceil C_i + \sum_{i=j}^{k-1} \frac{E_k + E_i - 1}{M} \cdot \left\lceil \frac{t_j}{T_i} \right\rceil C_i \tag{3.34}$$

$$\leq C'_k + \sum_{i=1}^{j-1} \frac{E_k + E_i - 1}{M} \cdot \left(1 + \frac{t_i}{T_i}\right) \cdot C_i + \sum_{i=j}^{k-1} \frac{E_k + E_i - 1}{M} \cdot \frac{t_i}{T_i} \cdot C_i \tag{3.35}$$

$$= C'_k + \sum_{i=1}^{k-1} \frac{E_k + E_i - 1}{M} \cdot \frac{t_i}{T_i} \cdot C_i + \sum_{i=1}^{j-1} \frac{E_k + E_i - 1}{M} \cdot C_i \tag{3.36}$$

$$\leq C'_k + \sum_{i=1}^{k-1} \frac{E_k + E_i - 1}{M} \cdot t_i \cdot u_i + \sum_{i=1}^{j-1} \frac{E_k + E_i - 1}{M} \cdot t_i \cdot u_i \tag{3.37}$$

Since $E_i \leq \max\{E_1, \ldots, E_{k-1}\}$, we have that τ_k is schedulable if $\exists t_j \in \{t_1, t_2, \ldots, t_k\}$ such that

$$C'_k + \sum_{i=1}^{k-1} \frac{E_k + \max\{E_1, \ldots, E_{k-1}\} - 1}{M} t_i u_i + \sum_{i=1}^{j-1} \frac{E_k + \max\{E_1, \ldots, E_{k-1}\} - 1}{M} t_i u_i \leq t_j \tag{3.38}$$

holds. By construction, we know that $t_i \in \mathbb{N}$ and the idea of the k2U framework is to solve the following optimization problem

$$\min C_k^* \tag{3.39}$$

$$C_k^* + \sum_{i=1}^{k-1} \alpha \cdot t_i^* \cdot u_i + \sum_{i=1}^{j-1} \beta \cdot t_i^* \cdot u_i \geq t_j^* \qquad \forall j \in \{1, \ldots, k\} \tag{3.40}$$

$$t_j^* \geq 0 \qquad \forall j \in \{1, \ldots, k\} \tag{3.41}$$

by using $\alpha = \beta = \frac{E_k + \max\{E_1, \ldots, E_{k-1}\} - 1}{M}$. Intuitively, the least upper-bound for an admissible C_k^* is searched for such that any C_k^* less than that value implies the existence of some $t_j \in \{t_1, \ldots, t_k\}$, which satisfies Eq. (3.32) that is a sufficient schedulability test. The optimization problem can be further conditioned, since $t_k^* = D_k$ is fixed, we can evaluate $j = k$ and directly write

$$C_k^* = D_k - \sum_{i=1}^{k-1} (\alpha + \beta) \cdot t_i^* \cdot u_i \tag{3.42}$$

and the integer-linear program (ILP) is henceforth posed as

$$\max \sum_{i=1}^{k-1} (\alpha + \beta) \cdot t_i^* \cdot u_i \tag{3.43}$$

$$t_k - \sum_{i=j}^{k-1} \beta \cdot t_i^* \cdot u_i \geq t_j^* \qquad \forall j \in \{1, \ldots, k-1\} \tag{3.44}$$

$$t_j^* \geq 0 \qquad \forall j \in \{1, \ldots, k-1\} \tag{3.45}$$

For $j = k - 1$ Eq. (3.44) yields

$$t_k^* - t_{k-1}^* = \beta \cdot t_{k-1}^* \cdot u_{k-1} \implies \frac{t_k^*}{t_{k-1}^*} = 1 + \beta u_{k-1} \tag{3.46}$$

and injecting this result into Eq. (3.42) yields

$$C_k^* = t_k^* - \frac{\alpha + \beta}{\beta} \cdot \sum_{i=1}^{k-1} t_{i+1}^* - t_i^* = t_k^* - \frac{\alpha + \beta}{\beta} \cdot (t_k^* - t_1^*) \tag{3.47}$$

$$\frac{C_k^*}{t_k^*} = 1 - (\frac{\alpha}{\beta} + 1) \cdot (1 - \frac{t_1^*}{t_k^*}) = 1 - (\frac{\alpha}{\beta} + 1) \cdot \left(1 - \frac{1}{\prod_{j=1}^{k-1} \beta \frac{C_j}{T_j} + 1}\right) \tag{3.48}$$

$$= \left(\frac{\alpha}{\beta} + 1\right) \cdot \frac{1}{\prod_{j=1}^{k-1} \beta \frac{C_j}{T_j} + 1} - \frac{\alpha}{\beta} \tag{3.49}$$

In consequence, we conclude that if

$$\frac{C_k'}{D_k} \leq \left(\frac{\alpha}{\beta} + 1\right) \cdot \frac{1}{\prod_{j=1}^{k-1} \beta \frac{C_j}{T_j} + 1} - \frac{\alpha}{\beta} \tag{3.50}$$

then the existence of a $t_j \in \{t_1, \ldots, t_k\}$ in Eq. (3.32) is guaranteed and thus the schedulability.

After substitution, we reach

$$\left(\frac{C'_k}{D_k}+1\right)\cdot\prod_{\tau_i\in\mathbb{T}_2}\left(\frac{E_k+\max\{E_1,\ldots,E_{k-1}\}-1}{M}\cdot\frac{C_j}{T_j}+1\right)\leq 2 \tag{3.51}$$

where

$$C'_k := C_k+\sum_{\tau_i\in\mathbb{T}}\min\{2E_k-1,E_k+E_i-1\}\cdot\frac{C_i}{M}+\sum_{\tau_i\in\mathbb{T}_1}\frac{(E_k+E_i-1)\cdot C_i}{M} \tag{3.52}$$

which proves the theorem. □

3.3.4.3 *Approximation Analysis*

In this section, we seek to evaluate the approximation quality of our proposed sporadic fixed-priority constrained-deadline stationary gang task scheduling algorithm with the consecutive stationary gang assignment algorithm in Section 3.3.4.

Definition 3.12. *A schedulability test $\mathbb{A}$ for a scheduling algorithm is a mapping from the task set to the binary decision of whether the task set is guaranteed to be feasibly schedulable by the scheduling algorithm, i.e.,*

$$\mathbb{A}:\mathbb{T}\mapsto\{\textit{feasible},\textit{infeasible}\} \tag{3.53}$$

In contrast to classic approximation algorithm analyses that quantify the approximation quality in terms of the *output* of the algorithms, we here quantify the approximation quality in terms of *inputs* to the schedulability tests, i.e., resource augmentation factors such as processor speeds, which are referred to as *speed-up factors* in the research literature. This difference is due to the fact that the output of a schedulability test is a binary yes or no answer, which is not sensible to approximate.

speed-up factors

the output of a schedulability test is a binary yes or no answer, which is not sensible to approximate

Definition 3.13 (Speed-Up Function)**.** *A speed-up function $\gamma:\mathbb{T}\mapsto\mathbb{T}'$ is a scaling that is applied task-wise to all parameters sensitive to speed, e.g., $\gamma(\tau_i=(C_i,\ldots))=(C_i/s,\ldots)$ where $s\in\mathbb{R}_{>0}$ is a constant. The speed is referred to as the unit speed if $s=1$.*

Speed-up factors relate the performance of two schedulability tests to another, i.e., let $\mathbb{A}\neq\mathbb{B}$ denote two sufficient schedulability tests, which are suitable for the task set $\mathbb{T}$ then $\mathbb{B}$ is said to have a speed-up factor s with respect to $\mathbb{A}$ if for any legal task set $\mathbb{T}$ – under the task model – the following implication holds.

$$\mathbb{A}(\mathbb{T})=\text{feasible}\implies\mathbb{B}(\gamma(\mathbb{T}))=\text{feasible} \tag{3.54}$$

By application of the inverse speed function $\gamma^{-1}(\cdot)$ to the argument, the direction of implication changes to

$$\mathbb{A}(\gamma^{-1}(\mathbb{T}))=\text{feasible}\implies\mathbb{B}(\mathbb{T})=\text{feasible} \tag{3.55}$$

Often times, the speed-up factor is stated with respect to a hypothetical exact optimal schedulability test $\mathbb{A}^*$ for a class of scheduling algorithms. In our case,

we are interested in an optimal schedulability test for rigid gang scheduling algorithms of sporadic constrained-deadline task sets upon M identical processors. Let $\mathbb{A}^*_{gang}(\mathbb{T})$ denote the **exact schedulability test** of an optimal rigid gang scheduling algorithm, and let $\mathbb{A}^*_{ord}(\mathbb{T})$ denote the **exact optimal schedulability test** of an optimal scheduling algorithm if the gang constraint is lifted, i.e., each thread in a gang can execute as an individual ordinary task. Our proof strategy is to first show that for any sporadic constrained-deadline rigid gang task set $\mathbb{T}$, the following implications hold

$$\mathbb{A}^*_{gang}(\mathbb{T}) = feasible \implies \mathbb{A}^*_{ord}(\mathbb{T}) = feasible \implies \text{Lemma 3.15 applied to } \mathbb{T} \quad (3.56)$$

Secondly, we show by contrapositive in Theorem 3.16 that if there exists a task set $\mathbb{T}$ such that Theorem 3.13 is infeasible then Lemma 3.15 applied to $\gamma^{-1}(\mathbb{T})$ is also infeasible. With reference to Eq. (3.56), we have then proved that $\mathbb{A}^*_{gang}(\gamma^{-1}(\mathbb{T}))$ is also infeasible, which then yields the speed-up factor. To use the necessary condition of $\mathbb{A}^*_{ord}$ is unfortunately pessimistic, however a necessary condition for $\mathbb{A}^*_{gang}$ is not known.

Lemma 3.15. *A sporadic constrained-deadline rigid gang taskset $\mathbb{T}$ is not schedulable by any multiprocessor scheduling algorithm by running the M processors at any processing speed $s > 0$ if the following condition holds:*

$$\max\left\{\max_{\tau_i\in\mathbb{T}} \frac{C_i}{D_i}, \sum_{\tau_i\in\mathbb{T}} \frac{E_i\cdot C_i}{M\cdot T_i}, \max_{t>0} \sum_{\tau_i\in\mathbb{T}} \max\left\{0, \left\lfloor\frac{t-D_i}{T_i}\right\rfloor + 1\right\} \cdot \frac{C_i\cdot E_i}{M\cdot t}\right\} > s \quad (3.57)$$

Proof. We prove each of the condition in the above *max* operator individually.

Since the E_i-many sub-jobs of each rigid gang task must execute simultaneously and each sub job has to execute for at most C_i time units, we know that a rigid gang task τ_i is unschedulable at processor speed s if $C_i > s \cdot D_i$ or $C_i/D_i > s$ respectively. Clearly if at least one task is not schedulable then the complete task set is unschedulable.

The second condition is straightforward since the total utilization is clearly a lower-bound for schedulability.

A necessary condition for the schedulability of the task set is to ensure that in any interval $[t_0, t_0 + t)$ for $t > 0$, the total execution time demand of all jobs that arrived before t_0 but have deadlines at or before time $t_0 + t$ does not exceed the maximal theoretical execution capacity in the interval $[t_0, t_0 + t)$. As we focus on constrained-deadline sporadic rigid gang task systems, we can quantify the necessary condition by the maximal demand that each gang task can demand in any time interval of length $t > 0$. Due to the co-scheduling constraint to execute all sub jobs simultaneously, this is identical to E_i-many superimposed demand-bound functions described by Baruah et al. in [BMR90], i.e.,

$$dbf_i(t) := E_i \cdot \max\left\{0, \left(\left\lfloor\frac{t-D_i}{T_i}\right\rfloor + 1\right) \cdot C_i\right\} \quad (3.58)$$

The exact maximal theoretical execution capacity for rigid gang tasks may be less than $s \cdot t \cdot M$ due to the co-scheduling constraint. The former however provides

a safe bound in the sense that if the total execution time demand is larger than $s \cdot t \cdot M$ then it is also larger than an exact maximal theoretical execution capacity for rigid gang tasks. In conclusion, we have that if there exists $t > 0$ such that $\sum_{\tau_i \in \mathbb{T}} dbf_i(t) > s \cdot t \cdot M$ then at least one job of at least one task misses its deadline, i.e., the task set is unschedulable. □

Note that the necessary condition in Eq. (3.57) is identical to the necessary condition for schedulability of classical multiprocessor scheduling for ordinary constrained-deadline task systems [Che16; CBU18]. That is, as if each sporadic rigid gang task τ_i is decomposed into E_i ordinary sporadic real-time tasks, each with WCET equal to C_i, relative deadline equal to D_i, and minimum inter-arrival time equal to T_i.

Theorem 3.16. *Let $\alpha_i = \min\{2E_k - 1, E_k + E_i - 1\}/E_i$ and $\beta_i = (E_k + E_i - 1)/E_i$. If Algorithm 1 does not return a feasible stationary gang assignment for some task set $\mathbb{T}$, then no multiprocessor scheduler can find a feasible rigid gang schedule when all M processors are slowed down by $1/(1 + \alpha + 2 \cdot \beta)$ where $\alpha \geq \alpha_i$ and $\beta \geq \beta_i$ for $i \in \{1, \ldots, k-1\}$.*

Proof. By the contrapositive of the sufficient condition Eq. (3.24) in Theorem 3.13 for the schedulability of task τ_k using Algorithm 1, we have that for all times $t \in (0, D_k]$ and thus for $t = D_k$ in particular

$$C_k + \sum_{i=1}^{k-1} \min\{2E_k - 1, E_k + E_i - 1\} \cdot \frac{C_i}{M} + \sum_{i=1}^{k-1} (E_k + E_i - 1) \cdot \left\lceil \frac{D_k}{T_i} \right\rceil \frac{C_i}{M} > D_k \quad (3.59)$$

holds. By definition of the ceiling function, we have $\left\lceil \frac{D_k}{T_i} \right\rceil \leq 1 + \frac{D_k}{T_i}$, which yields

$$\begin{aligned} & C_k + \sum_{i=1}^{k-1} \alpha_i \cdot \frac{E_i \cdot C_i}{M} + \sum_{i=1}^{k-1} \beta_i \cdot \left(\frac{D_k}{T_i} + 1 \right) \frac{C_i \cdot E_i}{M} > D_k \\ = & C_k + \sum_{i=1}^{k-1} (\alpha_i + \beta_i) \cdot \frac{E_i \cdot C_i}{M} + \sum_{i=1}^{k-1} \beta_i \cdot \frac{D_k}{T_i} \frac{C_i \cdot E_i}{M} > D_k \end{aligned} \quad (3.60)$$

for $\alpha_i := \min\{2E_k - 1, E_k + E_i - 1\}/E_i$ and $\beta_i := (E_k + E_i - 1)/E_i$ respectively. Dividing both sides of Eq. (3.60) by the relative deadline D_k, yields

$$\frac{C_k}{D_k} + \sum_{i=1}^{k-1} (\alpha_i + \beta_i) \cdot \frac{E_i \cdot C_i}{M \cdot D_k} + \sum_{i=1}^{k-1} \beta_i \cdot \frac{C_i \cdot E_i}{M \cdot T_i} > 1 \quad (3.61)$$

Let $0 < \alpha_i \leq \alpha$ and $0 < \beta_i \leq \beta$ then we have that

$$\frac{C_k}{D_k} + (\alpha + \beta) \cdot \sum_{i=1}^{k-1} \frac{E_i \cdot C_i}{M \cdot D_k} + \beta \cdot \sum_{i=1}^{k-1} \frac{C_i \cdot E_i}{M \cdot T_i} > 1. \quad (3.62)$$

By replacing $\frac{C_k}{D_k}$, $\sum_{i=1}^{k-1} \frac{E_i \cdot C_i}{M \cdot D_i}$ and $\sum_{i=1}^{k-1} \frac{C_i \cdot E_i}{M \cdot T_i}$ in Equation (3.62) with the maximum of these three values, we obtain

$$\max \left\{ \frac{C_k}{D_k}, \sum_{i=1}^{k-1} \frac{E_i \cdot C_i}{MD_i}, \sum_{i=1}^{k-1} \frac{C_i \cdot E_i}{M \cdot T_i} \right\} > \frac{1}{(1 + \alpha + 2 \cdot \beta)}. \quad (3.63)$$

Further, since $D_i \leq D_k$ due to deadline-monotonic scheduling policy for $i = 1, 2, \ldots, k-1$, we know that

$$\max_{t>0}\left\{\sum_{i=1}^{k-1} \max\left\{0, \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1\right\} \cdot \frac{C_i \cdot E_i}{Mt}\right\} \tag{3.64}$$

$$\geq \sum_{i=1}^{k-1} \max\left\{0, \left\lfloor \frac{D_k-D_i}{T_i} \right\rfloor + 1\right\} \cdot \frac{C_i \cdot E_i}{MD_k} \geq \sum_{i=1}^{k-1} \frac{C_i \cdot E_i}{MD_k}. \tag{3.65}$$

Therefore, by the above discussions and the necessary condition in 3.15, we know that

$$\max\left\{\frac{C_k}{D_k}, \max_{t>0}\left\{\sum_{\tau_i \in \mathbb{T}} \max\left\{0, \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1\right\} \frac{C_i \cdot E_i}{M \cdot t}\right\}, \sum_{\tau_i \in \mathbb{T}} \frac{C_i \cdot E_i}{M \cdot T_i}\right\} \tag{3.66}$$

$$> \frac{1}{(1+\alpha+2\beta)} = \gamma(\cdot) = \frac{1}{s}. \tag{3.67}$$

As a result the speed-up factor s is given by $1 + \alpha + 2\beta$. □

An immediate observation is if the gang sizes E_i for $i \in \{1, \ldots, k-1\}$ satisfy the inequality $E_k \leq \eta \cdot E_i$. In that case, we have

$$\alpha_i := \min\{\frac{2E_k - 1}{E_i}, \frac{E_k + E_i - 1}{E_i}\} \leq \min\{2\eta - \frac{1}{M}, \eta + 1 - \frac{1}{M}\} \tag{3.68}$$

$$\beta_i := \frac{E_k + E_i - 1}{E_i} \leq \eta + 1 - \frac{1}{M} \tag{3.69}$$

and therefore

$$s \leq 1 + 2\min\{2\eta - \frac{1}{M}, \eta + 1 - \frac{1}{M}\} + \eta + 1 - \frac{1}{M} \tag{3.70}$$

$$= \min\{4 + 3\eta, 3 + 4\eta\} - \frac{2}{M} \tag{3.71}$$

An observation from the above parametric speedup factor is that the best case in terms of the speed-up factor is if the gang sizes of lower-priority tasks are a lot smaller than the gang sizes of the higher-priority tasks. However even in the extreme case $E_k = 1$ and $E_i = M$ we have

best case in terms of the speed-up factor is if the gang sizes of lower-priority tasks are a lot smaller than the gang sizes of the higher-priority tasks

$$s \leq 3 < 3 + \frac{2}{M} \leq \min\{4 + 3\eta, 3 + 4\eta\} - \frac{2}{M} \tag{3.72}$$

Please note that the provided speedup factor is not tight and only provides an upper-bound. However, it can be observed that the approach works *provably* better for smaller gang sizes.

it can be observed that the approach works provably *better for smaller gang sizes*

A number of potential pitfalls that can occur when the speedup factor metric is used to describe the performance of an algorithm has been detailed by Chen et al. [CBH+17a], most notably for our situation that a good speedup factor does not necessarily imply a good overall performance but a good worst-case performance. This becomes problematic, when the algorithm is designed with a speedup factor in mind, e.g., by introducing restrictions that decrease average performance but ensure a certain level of performance in an artificial worst-case

setting. We note that our provided algorithm in Section 3.3.4 does not have any of the potential pitfalls regarding the analysis of speedup factors discussed in [CBH+17a]. However, our heuristic restriction which assigns the gang to a consecutive group of processors can be a source of pessimism. Whether the optimal solution under this restriction has good speedup factors or not is an open problem. Note that the provided speedup factor is parametric with respect to the size of the gangs (see [CBH+17a] for further explanation and motivation for the concept of parametric speedup factors).

3.3.5 EVALUATION

In this section, we present evaluations with synthetically generated gang task sets to evaluate our proposed algorithm (denoted as *DM-OUR* here) against the current state-of-the art by Dong and Liu [DL17] for sporadic implicit-deadline gang task systems under global EDF. Specifically, we compare to the optimized schedulability test in [DL17], denoted as *DONG-OPT*, based on the acceptance ratio, i.e., the number of schedulable task sets compared to the number of tested task sets.

We also evaluate our algorithm for sporadic constrained-deadline gang task systems under different settings of gang sizes, but without comparison due to the absence of research results for constrained-deadline gang tasks. In these experiments, we seek to explore how much the imposed constraints in terms of stationary gang assignments and fixed-priority scheduling algorithms impact the schedulability of the tested task sets.

3.3.5.1 *Experimental Setup*

We generate synthetic task sets of sporadic gang tasks with implicit- and constrained-deadlines in the following way. To generate the task sets, we use the UUniFast algorithm [BB05] to draw n samples of $x_i = E_i \cdot C_i/MT_i$ uniform at random where $x_i \in (0, 1]$ such that $\sum_{i=1}^{n} x_i = x$ for $x \in \{0.05, 0.1, 0.15, \ldots, 1\}$. Moreover, the periods T_i are drawn from a log-uniform distribution in the range of $[10, 100]$ ms.

The generated task sets are classified by the range of admissible gang sizes into *light*, *moderate*, and *heavy*. We differentiate two different settings for these gang sizes:

1. **Setting I** - with variable gang sizes: In the first setting, each *light* gang task can have a gang size in $[1, M/8]$, a *moderate* task can gang size in $[1, M/4]$, and a *heavy* task can have gang size in $[M/8, M/2]$.
2. **Setting II** - with fixed gang sizes: In this setting, a fixed gang size number is assigned to each task of a category. Namely, each *light* task has gang size $M/8$, each *moderate* task has gang size $M/4$ and each *heavy* task has a gang size $3M/8$.

We avoid the generation of too heavy tasks, since in these cases the scheduling problem is degraded to uniprocessor scheduling.[1] With respect to constrained-deadlines, we only demonstrate our proposed algorithm by a case of variable gang sizes (Setting I) in Figure 3.9 and a case of fixed gang sizes (Setting II) in Figure 3.10.

3.3.5.2 *Evaluation Results*

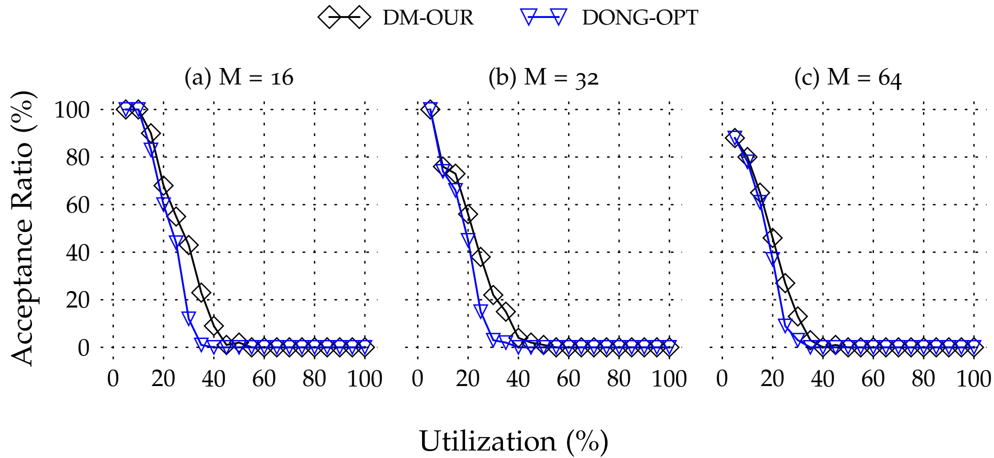

Figure 3.6: Acceptance ratio for *light* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according *Setting II*.

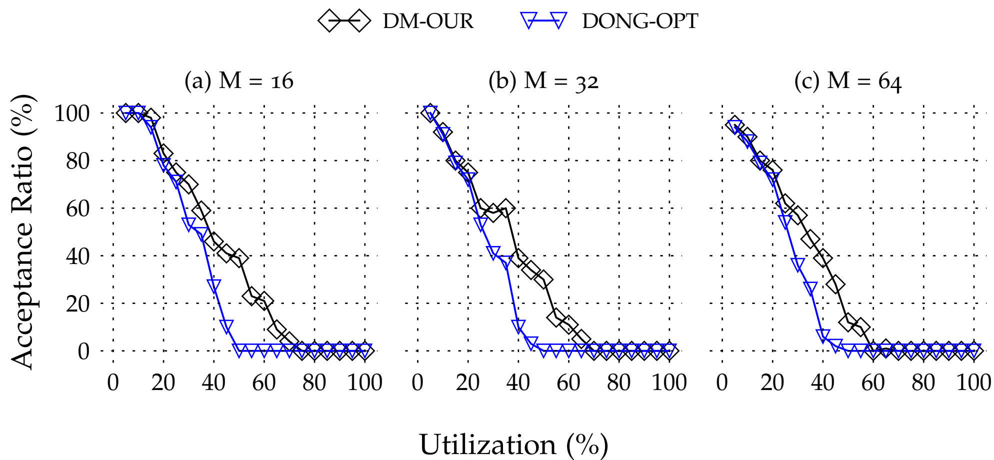

Figure 3.7: Acceptance ratio for *moderate* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*.

1 Dong and Liu [DL17] also performed their evaluations for gang size in $[5M/8, M]$ for all tasks. This configuration is not considered here as this setup implies that there is no possibility to concurrently execute two gang tasks in parallel due to the imposed gang size. The problem becomes equivalent to uniprocessor scheduling by viewing all processors as one virtual group. In this case, preemptive EDF is the optimal solution and the classical timing analysis for uniprocessor EDF scheduling can be applied.

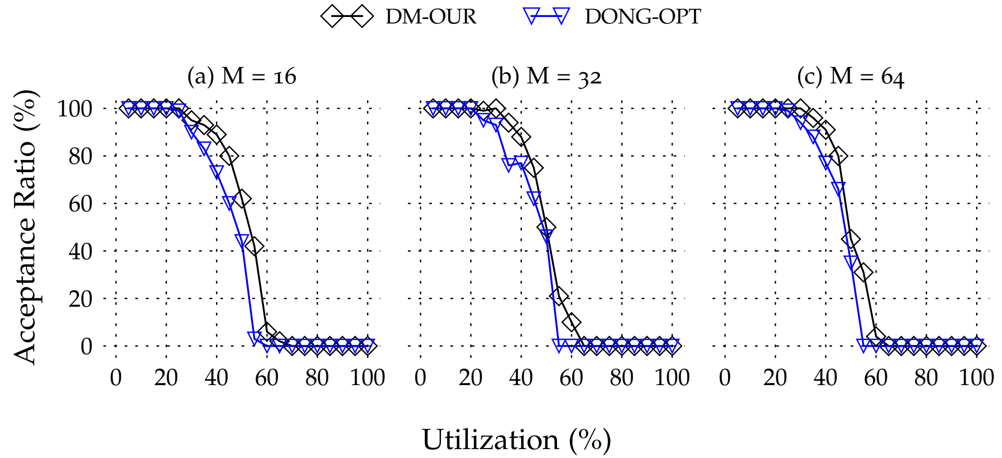

Figure 3.8: Acceptance ratio for *heavy* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*.

3.3.5.3 *Evaluation results for implicit-deadline task sets*

For sporadic implicit-deadline gang task systems, we compare our algorithm (*DM-OUR*) with the approach by Dong and Liu [DL17] (*DONG-OPT*) under the setting with variable gang sizes, in which each configuration is evaluated with 100 task sets and 20 tasks for each task set. In all conducted experiments shown in Figures 3.6, 3.7, and 3.8, our algorithm *DM-OUR* outperforms *DONG-OPT* for all evaluated scenarios under the setting with variable gang sizes. The most significant improvement of *DM-OUR* compared with *DONG-OPT* is demonstrated for the *moderate* task set in Figure 3.7 where up to 40% can be achieved for 50% normalized utilization. The smallest improvement can be observed for *heavy* gang task sets, where *DM-OUR* slightly outperforms *DONG-OPT*. This is due to the fact that the heavier the task sets are, the more similar the schedulability is to the uniprocessor schedulability problem. This also implies that the stationary gang scheduling has less choices for gang assignments. Since EDF is an optimal uniprocessor schedulability, the trouble to deal with the heavy gang task sets comes from the adopted schedulability tests. For *DM-OUR*, we have to consider more tasks in Ψ_k and for *DONG-OPT* their analysis becomes less pessimistic as the multiplicative of $1/M$ in their analysis decreases.

3.3.5.4 *Evaluation results for constrained-deadline task sets*

For constrained-deadlines, we show our schedulability test for light, moderate, and heavy task sets for gang sizes compliant to Setting I in Figure 3.9 and gang sizes compliant to the Setting II described in Figure 3.10, in which each configuration is tested with 100 task sets and 20 tasks per task set. The behavior of Setting I is almost similar to the results in Figures 3.6, 3.7, and 3.8 but with lower acceptance ratios.

For constrained-deadlines with fixed numbers of gang sizes as explained in Setting II, a similar trend can be observed. However, moderate as well as heavy task sets almost show the same acceptance ratio and the acceptance ratio

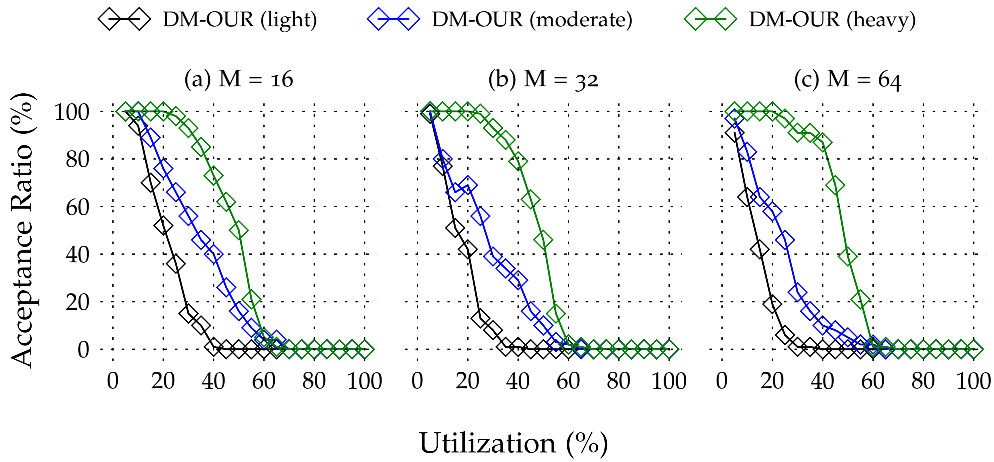

Figure 3.9: Acceptance ratio for *light* sporadic constrained-deadline gang task sets according to *Setting I*. The deadline is chosen randomly between 70% − 100% of the minimum inter-arrival time.

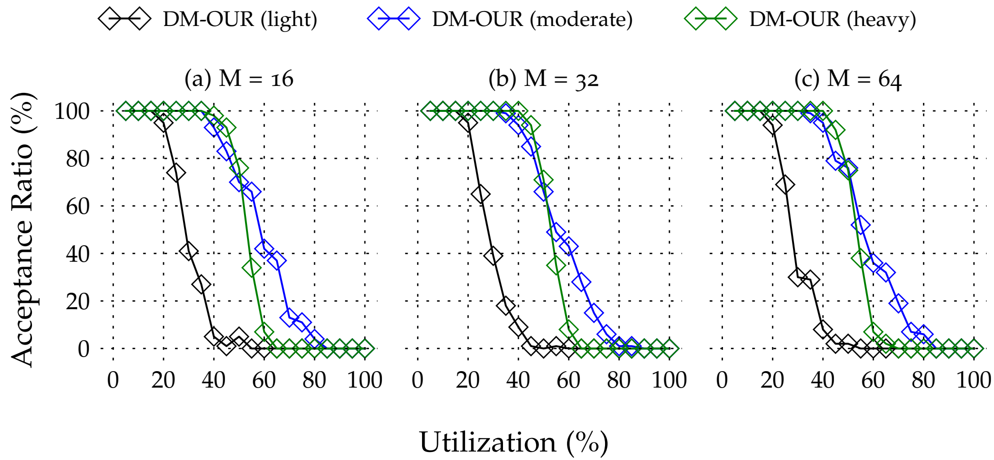

Figure 3.10: Acceptance ratio for *light*, *moderate*, *heavy* sporadic constrained-deadline gang task sets according to *Setting II*. The deadline is chosen randomly between 70% − 100% of the minimum inter-arrival time.

of light tasks also increases. This further supports the assumption, that the increased number of tasks with self-suspension behaviour decreases the overall schedulability. This is explained by the fact that it is less likely to have self-suspension behaviour of interfering tasks if all tasks have the same gang size.

3.3.5.5 *Summary of Evaluation Results*

In summary, the evaluations demonstrate, that the restriction of fixed-priority stationary gang scheduling does not significantly sacrifice the schedulability of sporadic implicit-deadline rigid gang task systems, in comparison to the state-of-the-art. In contrast, the schedulability could be improved slightly without even considering performance benefits of implementations in real systems, e.g., reduced context switches and migrations.

3.4 SIMULTANEOUS PROGRESSION SWITCHING PROTOCOLS

not only the distributed structure, but also the link switching of NoCs have imposed a great challenge in the design and analysis for real-time systems

Inter-core communication is a central challenge in many-core systems for which Network-on-chips (NoCs) have been demonstrated to scale well and to provide good overall performance. However, not only the distributed structure, but also the link switching of NoCs have imposed a great challenge in the design and analysis for real-time systems, where timing verification is mandatory. NoC protocols like wormhole switching are designed with scalability and flexibility in mind, thus the existing link switching protocols usually consider each single link to be scheduled independently. The flexibility of such link-based arbitration allows each packet to be distributed over multiple routers but also increases the number of possible link states, i.e., the number of flits in a buffer, which have to be considered in the worst-case timing analysis for real-time systems. In contrast, a shared bus is very simple and precisely analyzable with respect to the worst-case response time of each injected message; however, a bus does not scale well.

novel timing predictable architecture and design of a two-dimensional NoC

In this section, we present a novel timing predictable architecture and design of a two-dimensional NoC system, which is suitable for real-time multicore systems. More precisely, to achieve timing predictability by design, we propose a family of less flexible switching protocols, called *simultaneous progression switching protocols* (SP2), in which the links used by a flow *either* all simultaneously transmit one flit (if it exists) of this flow *or* none of them transmits any flit of this flow. Conceptually, our proposed design can be interpreted as a time-multiplex of shared bus systems, where the bus consists of a subset of links of the network-on-chip that may change over time. Based on this *simultaneous progression* property, we reduce the schedulability of the NoC to the discrete time uniprocessor self-suspension scheduling problem, using the same insight presented for stationary rigid gang scheduling presented the previous Section 3.3. By implementation of the protocol and the routers, any non-minimal route is deadlock-free, which helps to make use of the path diversity.

The remainder of this section is organized as follows. In Section 3.4.1, the formal model and problem statement of our studied network-on-chip is presented. In Section 3.4.2, we formally elaborate the mismatch of the uniprocessor execution model with the pipelined transmission model of flits in network-on-chips and motivate and explain the *simultaneous progression* property of SP2 hereinafter. Following, in Section 3.4.3, a conceptual implementation for the SP2 protocol is constructed and described. In Section 3.4.5 a worst-case traversal time for each flow under analysis is presented. Lastly, in Section 3.4.6, the conceptual implementation of the SP2 protocol is evaluated by means of numerical experiments and theoretical considerations.

3.4.1 System Model & Problem Definition

Network-on-Chips (NoCs) are characterized by the topology, routing protocol, arbitration, buffering, flow control mechanism, and switching protocol.[2] In this dissertation, we consider a NoC as a collection of cores $\mathbb{A}$, routers $\mathbb{V}$, and links $\mathbb{L}$. Each router is connected to at least one other router by two physically separate simplex links, i.e., up-link and down-link. We assume that all the routers, and links are homogeneous, i.e., the transmission rate and processing capability are identical, except for the position dependent number of ports of the routers.

Throughout our analyses, we assume that the cores and switches are synchronized perfectly with respect to time, which means that there is no clock drift in the NoC. We will explain later how the global synchronization can be achieved in our protocol implementation. Otherwise, the clock drift must be considered carefully. One solution is to introduce additional delays and interferences to pessimistically bound the impact due to clock drifts. Moreover, we assume that the NoC is a *discrete time system*, i.e., the NoC operates in the granularity of a fixed time unit that is referred to as *cycle*. Strongly related is the term phit and flit, which denote the integral transmission unit in the network and corresponds to the number of bits that can be transferred in a single cycle.

NoC operates in the granularity of a fixed time unit that is referred to as cycle

3.4.1.1 *Messages and Periodic/Sporadic Flows*

A periodic (sporadic) flow $\tau_i \in \mathbb{T}$ generates an infinite sequence of flow instances, called messages, with the following parameters:

- T_i is the minimum inter-arrival time or period of the flow τ_i, i.e., for a periodic flow one message is released exactly every T_i time units and for a sporadic flow two subsequent messages are separated by at least T_i.
- $\mathbb{L}_i$ is an arbitrary static routing path of the flow τ_i, i.e., $l_{i_1}, l_{i_2}, \ldots, l_{i_{\eta_i}}$ is the ordered sequence of the η_i links, which a message of τ_i has to be transmitted on. We assume that a physical link cannot be used more than once in the static routing path.
- $C_i \in \mathbb{N}$ is the largest (worst-case) number of phits of any message generated by flow τ_i. Please recall that *phit*-many bits are transmitted over each link in the network in a cycle. Therefore all temporal analyses and considerations are conducted with respect to cycles.
- $D_i \in \mathbb{N}$ is the relative deadline of the flow τ_i in cycles. That is, when a message is injected at time a_i, its absolute deadline is $a_i + D_i$, at which the last phit of the message has to reach the destination. Our protocols are not restricted to any specific relation of the minimum inter-arrival time and the relative deadline D_i, but our timing analysis focuses on constrained-deadline flows, where for each flow $D_i \leq T_i$ is satisfied.

arbitrary static routing path

$C_i \in \mathbb{N}$ is the largest (worst-case) number of phits of any message

absolute deadline is $a_i + D_i$, at which the last phit of the message has to reach the destination

2 Our notation of *flows* is equivalent to *tasks* and our notation of *messages* is equivalent to *jobs* in the classical notation of the real-time systems community.

3.4.1.2 *Problem Definition*

A first analytical approach to determine the worst-case response time of sporadic traffic flows in wormhole switched fixed-priority network-on-chips was given by Mutka [Mut94] and Hary and Ozguner [HO97]. Both of them are based on the schedulability analysis for uniprocessor sporadic real-time tasks under fixed-priority scheduling developed in [LSD89; JP86]. To analyze the worst-case response time of the flow τ_i, they considered the complete path $\mathbb{L}_i$ as a single shared resource, i.e., a uniprocessor. This shared resource may not always be available for τ_i, and they modeled the unavailability by *only* considering the higher-priority flows that use any link in $\mathbb{L}_i$, called *direct interference*. They concluded that the problem is equivalent to the fixed-priority uniprocessor scheduling, which was disproved by Kim et al. [KKH+98], who showed that the flow τ_i can suffer from the interference due to flow τ_j even if $\mathbb{L}_i$ and $\mathbb{L}_j$ have no intersection, called *indirect interference*. By extending the notion of interference sets developed by Kim et al. [KKH+98], Lu et al. [LJS05] proposed to discriminate between flows that could not interfere with each other to reduce the pessimism of the analysis.

direct interference

indirect interference

However, both of the approaches in [KKH+98; LJS05] assume that the synchronous release of the first messages of the sporadic real-time flows is the worst-case, i.e., similar to the critical instant theorem in classical uniprocessor fixed-priority scheduling proposed. This statement was later disproved in 2008 by Shi and Burns [SB08], where jitter terms were added to model the asynchronous release of the first messages of the sporadic real-time flows. Based on the results of this work, Kashif and Patel proposed a link-based analysis called stage-level analysis [KGP14; KP16] to achieve a tighter analysis. Both analyses were proved to be unsafe by Xiong et al. [XLW+16] using simulations. It was discovered that a flit of a higher-priority flow may induce interference more than once, i.e., on multiple switches, thus rendering the conjectures made by Shi and Burns [SB08] and Kashif and Patel [KGP14; KP16] false. This behavior is referred to as *multi-point progressive blocking* by Indrusiak et al. [IBN18]. The state of the art with respect to fixed-priority wormhole switched networks-on-chips with infinite buffers is represented by [IBN16; XWL+17]. Unfortunately, the infinite buffer assumption is infeasible in real systems, thus back-pressure effects that occur due to limited buffer sizes in the switches have to be considered. In the work of Indrusiak et al. [IBN18], the authors incorporate buffer sizes into the worst-case response time analysis. They "*chose to provide intuitions, insight and experimental evidence on the proposed analysis and its improvements, rather than theorems or proofs*." Thus, further counterexamples may be found. Nikolíc et al. [NTI+19] presented an improved analysis over the results in[IBN18; XWL+17].

multi-point progressive blocking

infinite buffer assumption is infeasible in real systems, thus back-pressure effects that occur due to limited buffer sizes in the switches have to be considered

Problem Definition. The fact that almost all proposed analyses have been found to be flawed, suggests that the scheduling algorithm and architecture are too complex to be reasonably analyzed. Further evidence for this claim is that in the analyses provided by Indrusiak et al. [IBN18], increased buffer sizes lead to increased worst-case response times. We consider the interrelated problem of the scheduler design and the corresponding scheduling analysis with respect to real-time constraints.

the scheduling algorithm and architecture are too complex to be reasonably analyzed

Scheduler Design. The fundamental algorithmic complexity of the NoC scheduling problem results from the underlying *job shop scheduling problem with fixed machines*, which was shown to be NP-hard for three or more jobs by Sotskov et al. [SS95]. The classical job shop scheduling problem is defined by a set of machines M, a set of Jobs J, and a set of operations O. Each job J consists of a sequence of n_j operations $O_{j1}, O_{j2}, \ldots O_{nj}$, which need to be scheduled in that order. For each operation O_{jk}, P_{jk} units of time are required for the processing on a dedicated machine M_{kj} with $M_{jk} \neq M_{jk'}$ for $k \neq k'$. Given a problem instance and a constructed schedule, the makespan of that schedule refers to the latest finishing time of all operations O. In this job shop scheduling variant, each job J is given a set of machines $M_j \subseteq M$, which the job needs to be processed on simultaneously.

job shop scheduling problem with fixed machines

The connection between the NoC scheduling problem and the above makespan job shop scheduling problem with fixed machines is as follows. Each hard real-time message (defined as an instance of a flow) in the NoC has to be successfully transmitted from its source to its destination before its deadline. Consider the special case of a flow set $\mathbb{T}$ such that all periodic flows have the same period. Further, let the message of a flow τ_j be given by a sequence of n_j flits that correspond with the operations $O_{j1}, O_{j2}, \ldots O_{nj}$ and the processing times P_{jk} respectively. Moreover, we associate the machines M_j with the links used in the route for each flow. Then, the flow set $\mathbb{T}$ is schedulable if a schedule with a makespan, which is no more than the flows' periods is found.

The rigid gang scheduling problem and the NoC, i.e., job shop, scheduling problem are related, they differ however in the sense that in a NoC the links used by a flow have to be defined from the source node to the destination node, i.e., a specific subset of machines must be used where as in gang scheduling any subset of machines can be used. The studied *scheduler design* problem attempts to answer the question of how the switching mechanism (scheduling algorithm) should be designed such that:

rigid gang scheduling problem and the NoC, i.e., job shop, scheduling problem are related
scheduler design *problem*

- The progression model provably matches with the execution model of uniprocessors;
- real-time guarantees can be formally verified;
- routing does not have to obey minimal-path routing, or specific deadlock avoidance schemes, and can thus be optimized with respect to real-time constraints.

Schedulability Test. Analogously, the *schedulability test* problem studied in this paper is defined as follows: We are given a NoC, defined as a collection of cores $\mathbb{A}$, routers $\mathbb{V}$, and links $\mathbb{L}$. For a given set $\mathbb{T}$ of sporadic or periodic flows on the NoC and a switching mechanism, the objective is to validate whether the messages (instances of the flows) can meet their deadlines.

3.4.2 SIMULTANEOUS PROGRESSION PROPERTY

In this section, we formally show why the link-based arbitration problem in wormhole switched network-on-chip does not match the uniprocessor execution

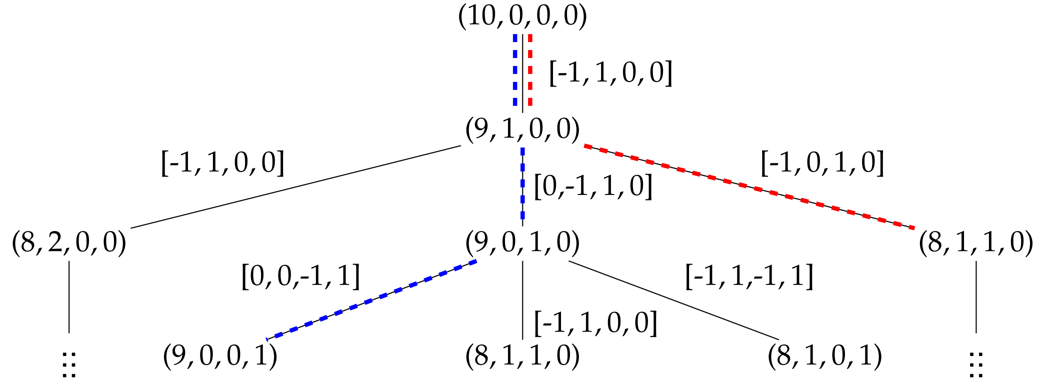

Figure 3.11: Progressions of a message which involves 2 cores and 2 routers, i.e., 3 links, when $C_i = 10$. The numbers associated with an edge indicate the number of buffered phits at the respective routers or cores. The red dashed path illustrates the beginning of the *fastest* progression and the blue dashed path illustrates the beginning of a *slowest* progression.

model and illustrate the subsequent problems in response-time analyses using uniprocessor scheduling theory.

In order to explain the mismatch, we focus on the possible buffer states of one instance (i.e., message) of a flow τ_i under analysis at each core. Let $\vec{B}_i$ denote the state vector of the number of flits that are buffered in the cores and routers involved in $\mathbb{L}_i$. Suppose that there are η_i links involved in the path $\mathbb{L}_i := \langle l_1, \ldots, l_{\eta_i} \rangle$. Note that the first element in $\vec{B}_i$ denotes the number of flits of τ_i in the source core, left to be sent, and the last element in $\vec{B}_i$ denotes the number of flits that have been received at the destination core. In the following analysis, we assume that the NoC is fully synchronized in time. Therefore, in each time unit, a buffered flit can be forwarded to the next node (a core or a router). Since the NoC works in discrete time, we can observe the changing of the vector $\vec{B}_i$ over time when considering only the time units at which the message is sent. When a flit is sent in a time unit at the j-th link $l_j \in \mathbb{L}_i$, then the number of flits of the j-th entry in $\vec{B}_i$ is reduced by 1 and the number of flits of the $(j+1)$-th entry is incremented by 1. For notational brevity, let $\vec{y}_j$ be a vector of $\eta_i + 1$ elements in which all the elements are 0 except the j-th element that is -1 and the $(j+1)$-th element that is 1, e.g., $\vec{y}_1 + \vec{y}_3$ implies that the first and the third link transmit one flit in this time unit and the number of flits buffered at the interconnected nodes are incremented and decremented respectively.

Definition 3.14 (Progression)**.** *Consider a buffer state $\vec{B}_i$ at time t, in which all elements in $\vec{B}_i$ are non-negative integers. Suppose element z_j is either 0 or 1 for $j = 1, 2, \ldots, \eta_i$ and* ***at least one of them is*** *1. Specifically, when z_j is 1, the j-th link sends one phit forward. Let $\vec{Y}$ be $\sum_{j=1}^{\eta_i} z_j \vec{y}_j$.*

For a buffer state $\vec{B}_i$, z_j for $j = 1, 2, \ldots, \eta_j$, and a vector $\vec{Y}$, the change of the buffer state is ***valid*** *if*

- *all elements in $\vec{B}_i + \vec{Y}$ are non-negative integers that do not exceed the buffer capacities $\vec{B}$, and*
- *the j-th element in $\vec{B}_i$ is $\geq z_{j+1}$ for $j = 1, 2, \ldots, \eta_i - 1$.*

If the change of the buffer state is valid, we say that the flow makes a ***progression*** *in this time unit.* □

In each time unit, a link may or may not be utilized to send a flit of f_i in the switching mechanism. Therefore, there are $2^{\eta_i} - 1$ combinations of the vectors of $\vec{y}$. Note that progressions do not have to take place in two consecutive time units. If the message is not sent in the next time unit, there is no progression of the message. As an illustrative example, consider $C_i = 10$ and that the message is sent from a core A_1 via two routers V_1 and V_2 to core A_2. If one flit is sent in a time unit, we get $\vec{B_i} = (C_i - 1, 1, 0, 0)$. Now, there are three possibilities for the next time unit when the NoC transmits a flit or multiple flits of the message:

- $\vec{B_i} = (C_i - 1, 0, 1, 0)$: In this case, A_1 does not send any flit but R_1 sends a flit to V_2. The progression is due to $\vec{Y} = (0, -1, 1, 0)$.
- $\vec{B_i} = (C_i - 2, 2, 0, 0)$: That is, A_1 sends one flit to V_1 but V_1 does not send a flit to V_2, i.e., $\vec{Y} = (-1, 1, 0, 0)$.
- $\vec{B_i} = (C_i - 2, 1, 1, 0)$: That is, A_1 sends one flit to V_1 and V_1 sends a flit to V_2, which means that the progression is due to $\vec{Y} = (-1, 1, 0, 0) + (0, -1, 1, 0) = (-1, 0, 1, 0)$.

This particular example is illustrated in the first three levels of the tree in Figure 3.11. In each of the above states, the next progression has to be considered. We only illustrate the progressions that are possible when $\vec{B_i}$ is $(9, 0, 1, 0)$.

Definition 3.15 (A Complete Series of Progressions)**.** *A complete series of progressions is a sequence of progressions defined in Definition 3.14, one after another, starting from* $\vec{B_i} = (C_i, 0, 0, \ldots, 0)$ *and finishing with* $\vec{B_i} = (0, 0, \ldots, C_i)$*.* □

A safe analysis of the worst-case response time or the schedulability for sending the message should consider all possible *complete series of progressions* of τ_i. If we only account for the number of time units when the message of τ_i is transmitted, it is not difficult to see that the *slowest* one only sends one phit forward per progression, in which the switching mechanism results in $C_i \cdot \eta_i$ iterations of progressions. Moreover, the *fastest* one sends one phit (if available) forward for all cores and routers involved in the path $\mathbb{L}_i$ per progression, which results in $C_i + \eta_i - 1$ iterations of progressions. Due to the simultaneous forwarding of phits at all cores and routers involved, we call that progression sequence a *simultaneous progression*.

complete series of progressions

slowest progression

fastest progression

Due to the simultaneous forwarding of phits at all cores and routers involved, we call that progression sequence a simultaneous progression

The correspondence of the scheduling problem for wormhole switched fixed-priority network-on-chip to the uniprocessor fixed-priority scheduling problem has not been proven and is conjectured by us to not hold in general, based on the following reasoning. In a uniprocessor system, if a job is executed for x time units, the execution time of the job is reduced by x time units. However, sending x flits can result in different series of progressions in the NoC, with varying number of flits transmitted, violating that execution assumption.

sending x flits can result in different series of progressions in the NoC, with varying number of flits transmitted, violating that execution assumption

Despite that mismatch, the results in [Mut94; HO97; KKH+98; LJS05; SB08; KGP14; KP16; XLW+16; IBN16; XWL+17; IBN18; NTI+19; NHE19] assumed that the corresponding uniprocessor scheduling problem can use $C_i + \eta_i - 1$ as the

worst-case execution time of the corresponding sporadic task to represent a flow. This implicitly assumes that the flow takes the fastest complete series of progressions. Such uniprocessor analyses are only valid when the other iterations of progressions are accounted for correctly. However, the fastest complete series of progressions may not always be possible.

To ensure the correctness of the analysis, some additional time units should be included. Many patches have been provided to account for such additional time units after the series of flaws found in [Mut94; HO97; KKH+98; LJS05; SB08; NIP16; KP16; XLW+16]. The recent counter examples provided by Xiong et al. [XLW+16] are due to backpressures when the buffer space is limited. However, we conjecture that the existing analyses that assume uniprocessor equivalence are not rigorous even when the buffer space is unlimited due to the mismatch explained above and non-systematic means to account for the model differences.

a correct proof should explain how to safely account for the number of iterations in the progressions of the flows

These series of flaws in the literature suggests that the scheduling algorithm and network architecture may be too complex to be correctly analyzed adopting uniprocessor real-time scheduling theory and its assumptions. However, such a correspondence to uniprocessor scheduling theory is potentially very difficult to achieve due to the large space of progression. Furthermore, safe approximations and upper bounds are also missing in the literature. In both cases, a correct proof should explain how to safely account for the number of iterations in the progressions of the flows and map them to the corresponding execution time in the constructed instance of the uniprocessor scheduling problem. Since the wormhole switching protocol was not designed with predictability constraints in mind, designing new protocols that can be safely analyzed without losing too much flexibility or efficiency can be an alternative. Therefore, we choose to provide a scheduling algorithm based on the *simultaneous progression* property.

3.4.3 PROTOCOL IMPLEMENTATION

Instead of proving the complex scenarios in the standard wormhole switching, we propose another protocol which has only *one complete series of progressions*. These less flexible switching protocols, called SP2, achieves timing predictability by enforcing that a flow τ_i is eligible to transmit on its route **if and only if** it can be allocated all the links in $\mathbb{L}_i$ in-parallel. In other words, the links used by a flow τ_i *either* all simultaneously transmit one flit of this flow (if it exists) *or* none of them transmits any flit of this flow. As a result, for a progression of τ_i in a time unit, some links in $\mathbb{L}_i$ may be reserved even though there is no flit to be transmitted over this link in this time unit (a behaviour similar to *processor spinning*). In order to meet the deadline of a message of a flow τ_i, that arrives at time a_i, the concept of simultaneous progression requires to have $C_i + \eta_i - 1$ time units to use all the links in $\mathbb{L}_i$ simultaneously before the absolute deadline at $a_i + D_i$. Notably, such a *simultaneous progression* has similarities to the rigid gang scheduling problem, which is explained and analyzed in detail in Section 3.3.

some links in $\mathbb{L}_i$ may be reserved even though there is no flit to be transmitted

In this section, we presume those provided analytical results and focus this section on the conceptualization of a network-on-chip implementation, which satisfies the SP2 properties and derive how the response-time analyses must be

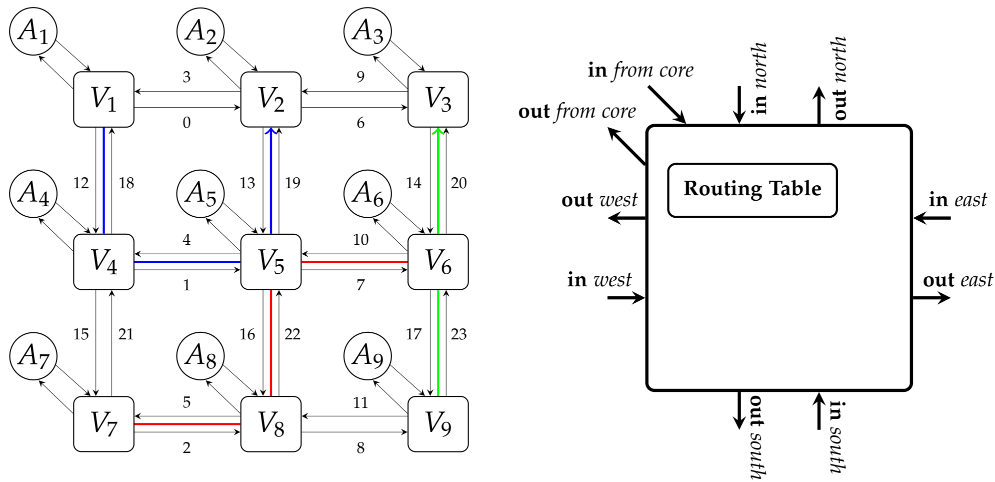

Figure 3.12: The arbitration and data message transmission can be done sequentially within a single data net, or interleaved using a separate arbitration net. Each cluster can transmit a flit in each cycle to the centralized arbiter core.

Algorithm 2 Behavioural description of the arbitration routine in SP2

1: $allocated \leftarrow 00\ldots0$;
2: **for each** flow with id i (in priority order) in Table 3.1 as *table* **do**
3:     **if** $table[i].epochs > 0$ **then**
4:         **if** $allocated \wedge table[i].path == 0$ **then**
5:             $table[i].epochs \leftarrow table[i].epochs - 1$;
6:             $allocated \leftarrow allocated \vee table[i].path$;

adapted accordingly. The connection of simultaneous progression arbitration and the rigid gang scheduling problem, which is analyzed in Section 3.3, is formally given as follows; we can consider that each of the links in $\mathbb{L}$ is a processor, each flow is a task, the links $\mathbb{L}_i$ form a gang for flow τ_i, and the execution time is $C_i + \eta_i - 1$.

In general, SP2 can be implemented with any strategy, which ensures the simultaneous progression property, but we will focus on work-conserving fixed-priority priority-based SP2 scheduling with message-level-fixed priorities, i.e., all messages of a flow τ_i have the same priority. Thus, whenever two messages intend to use one link at the same time, the higher-priority message is scheduled and the lower-priority message is suspended. Whenever a message is suspended in one of its links, it is suspended on all of its links.

Due to the fact that in SP2 this reservation scheme eliminates *circular waiting* it is thus deadlock-free. Therefore, in SP2 scheduling for network-on-chips, the path diversity is not constrained to deadlock-free minimal routing schemes like *X-Y Routing* but can utilize any arbitrary routing scheme.

circular waiting

3.4.4 ARBITRATION IMPLEMENTATION

The architectural implementation of the priority-based SP2 providing the *all-or-nothing* property requires to rethink previous router designs. In state-of-the-art

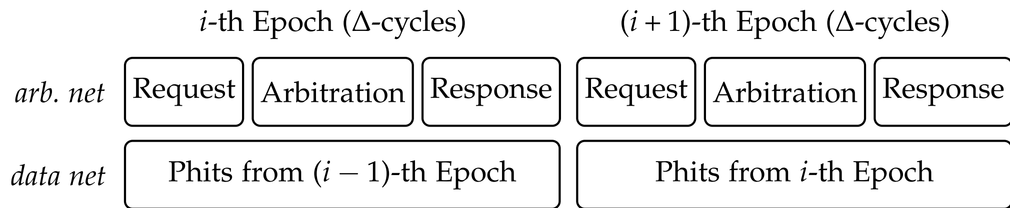

Figure 3.13: The arbitration and communication is pipelined using two distinct nets, i.e., the arbitration and the data net.

wormhole switching protocols, the decision at each router is local, whereas the *all-or-nothing* property requires global decision making. The simultaneous progression switching protocols are a general description of a family of concrete protocols for which there could be different possible implementations.

In this section, we provide a centralized arbitration implementation, which is achieved by using two dedicated networks, namely, one for arbitration (called the *arbitration net*) and one for data transmission (called the *data net*).

discrete time domain

epoch

The arbitration net and data net both operate in the *discrete time domain*, with a fixed *epoch*, denoted as Δ. In general, the separation of the network into dedicated arbitration and data nets, can be accomplished using only one physical net by time-multiplexing and the interleaved sending of arbitration and data messages or by separating the network using space multiplexing or frequency multiplex, which allows for parallel sending of arbitration and data messages. An exemplary implementation of the data net and arbitration net, using two physically separated parallel nets, is illustrated in Figure 3.13. Note that the arbitration (in the arbitration net) of the k-th epoch is responsible for ensuring the *all-or-nothing* property of the messages to be sent in the data net of the $(k+1)$-th epoch. Specifically, we consider the control state of both nets to be stable and available at epoch boundaries, i.e., $0, \Delta, 2\Delta, \ldots$, etc.

centralized-priority arbitration

we dedicate one core in the network-on-chip to the arbitration

In this section, we describe the conceptual design of the *centralized-priority arbitration* implementation of the proposed SP2 protocol. We discuss our design on the presumption that a regular topology such as a $2D$-Mesh in Figure 3.12 or a $2D$-Torus is used. In order to implement centralized arbitration, we dedicate one core in the network-on-chip to the arbitration, that is, the arbitration logic is either implemented in software, which is run on that core, or implemented in hardware logic. With reference to Figure 3.12, the core A_5 would be the dedicated *arbiter core*.

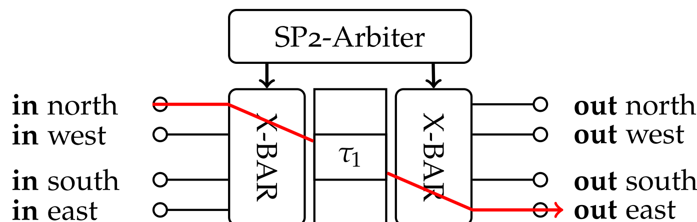

Figure 3.14: A possible implementation of the SP2 arbitration using phit-sized virtual channels, and crossbars.

3.4.4.1 *Router Design*

The number of buffers and virtual channels within a NoC router impact the hardware complexity, energy consumption and switching speeds and must thus be considered. By the SP2 property, each flow requires a single phit-sized virtual channel in each router along its static path. From the hardware perspective, this comes at a cost, since each virtual channel must be arbitrated separately, which requires additional *multiplexers* and *de-multiplexers* in the router. An exemplary arbiter design within an SP2 router is shown in Figure 3.14, where the *arbiter logic* within the router controls the two crossbar switches to switch input ports to the respective virtual channels, and the virtual channels to the output ports. This implements the SP2 logic, due to the reason that the incoming link of a switch (on a flow's path) is only granted to that flow, if and only if the outgoing link is granted to that flow as well, as is illustrated in the fastest progression in Fig 3.11.

multiplexer

de-multiplexer

In the initial design, a router requires at most as many virtual channels (of phit-sized buffers) as the number of flows, which cross that router, such that each flow can transmit without additional blocking. The reduction of limited virtual channels can be traded off with additional latency in SP2 by implicitly requesting a buffer slot along a link for transmission in the arbitration message. A flow is only eligible to send if the channel and a buffer on all links is granted on each router, which can be achieved since SP2 is a global scheduling scheme. To that end, the central arbiter core must maintain the state of all available buffer slots at each router. On the downside, this approach leads to priority-inversion, since lower-priority flows from the direct contention domain – flows, which share at least one link with a flow under consideration – cause interference.

reduction of limited virtual channels can be traded off with additional latency in SP2

3.4.4.2 *Arbiter Design*

Table 3.1: An exemplary *arbitration table* at the core A_5 of the network in Figure 3.12. The additional color annotation is used to visually identify the three exemplary flows.

Flow (id)	Epochs	Path
1 (red)	...	001000010000000000001010
2 (blue)	...	010000000000100000010000
3 (green)	...	000000000000000000010010
⋮		
n	...	...

Every flow, which is admitted to the system, is stored in the centralized arbiter in a *look-up table* with a unique *flow id*, which encodes the flow's priority, number of remaining *epochs*, required by each flow to be fully transmitted without any interference, and a bit vector *path*, which encodes the flow's static routing path. An exemplary *arbitration table* at the core A_5 of the network in Figure 3.12 is shown in Table 3.1, with an additional color annotation to visually identify the three exemplary flows. The *path* field consists of 24 bits, where each index in the bit string – starting from index 0 – corresponds to the respective link identifier in

look-up table

flow id

path vector

arbitration table

the network, e.g., the red highest-priority flow τ_1 uses link l_2, l_{22}, l_7, l_{20}. A flow is considered *ready* if the epoch field is non-zero.

flow ready

3.4.4.3 *Arbitration Phase*

The set of all flows, which can communicate in a given epoch, is determined by the centralized arbiter in the following way. The arbitration phase recurs periodically every epoch and is structured by the sequence of *request*, *arbitration*, and *response states*, as illustrated in Figure 3.13, and explained hereinafter. During the arbitration phase, each router in the network is configured to be in the so called *arbitration state*, which configures the routers to switch according to arbitration state specific routing paths, detailed later in this section.

arbitration phase recurs periodically every epoch
request state
arbitration state
response state

Request *(Arbitration State)*. In the *arbitration state*, each router is configured to route any received phit according to the static *arbitration route* described later in Section 3.4.4.4. At the router we require an additional 2×4-bit register *direction* (corresponding to *north*, *south*, *west*, *east*) for each flow, crossing that router, to memorize the ingress and egress direction of the arbitration request. The directions *from core*, do not need to be additionally stored, since the uniquely connected processing element of a switch is responded to by default except for the arbiter core. All *source cores* that need to communicate – at the time of arbitration – send the flow id's of the flows, which are *ready-to-transmit*, to the arbiter core using the previously mentioned static routing.

north port
south port
west port
east port
from core port
source core
ready-to-transmit

Arbitration *(Arbitration State)*. The arbiter core waits until all requests are received, which time defines the duration of the request stage and is analyzed later in this section. At the beginning of the arbitration stage, all flows that are admitted to transmit in the subsequent transmission state, are determined by following the procedure described in Algorithm 2. The SP2 property is ensured, since the look-up table in the router as illustrated in Table 3.1 is processed in priority order and subsequently the links in the network are allocated in priority order. The allocation is premised under the additional constraint that all required links are only granted if they can be granted all at once – as ensured by the *and* operation in Line 4 in Algorithm 2. The available remaining links for lower-priority flows is then updated in Line 6 in the bit-wise *or* operation.

Response *(Arbitration State)*. In the *response phase*, the arbiter core sends back the *flow id* of the flows that can communicate in the next epoch to the source cores in an arbitration response message. The arbitration response message is routed in the opposite direction with respect to the request message. As an additional advantage of the static and contention-less routes arbitration specific routing paths, the arbiter core is able to send a *synchronization message* along the arbitration response to minimize the *clock skew* for all cores in the network.

response phase
synchronization message
clock skew

Send *(Transmission State)*. In the *transmission state*, the routing tables in the routers are re-enabled. Then at the router, all phits from the buffer on the granted links are switched. For instance, the red flow τ_1 is granted links l_2, l_{22}, l_7, l_{20} and the blue flow τ_3 is granted the links l_1, l_{12}, l_{19}. Then exemplary, at the router V_5, the phit at the buffer for flow τ_3 at *in west* is switched to the buffer at *out north*.

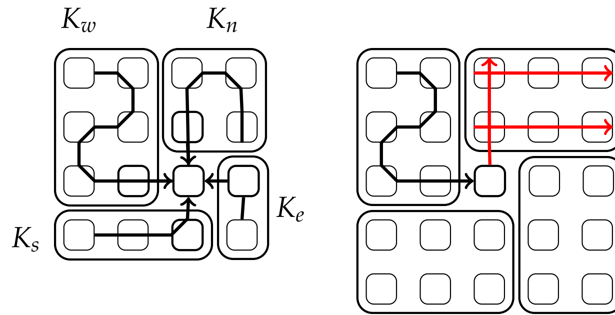

Figure 3.15: Clustering and routing for each router in the *arbitration state* for the $N \times N$ 2D-Mesh topology for even and odd dimensions. Even dimensions lead to irregular clusters, whereas odd dimensions lead to regular clusters.

Analogously, the phit at the buffer for flow τ_1 at *in south* is switched to the buffer *out east*.

3.4.4.4 *Arbitration State Routing*

The time duration of an *epoch* is given by the time duration of the arbitration phase, which consists of the request, arbitration, and response phase. Since the *epoch* duration is also the unit of arbitration, i.e., preemptions can only occur at integral epoch boundaries, the epoch duration should be as small as possible. The time duration of the request and response phase is determined by the longest possible transmission latency of the respective *request* and *response messages*, which are sent from each core to the central arbiter core. Consequently, the maximum transmission latency of those messages should be minimized, which entails i) the routing paths and ii) the message's bit encoding.

preemptions can only occur at integral epoch boundaries

request message

response message

For our proposed centralized arbiter, we assume that all the cores are connected in a N×N 2D-Mesh, whilst the arbitration logic is implemented in software in the *center core*, as illustrated in Figure 3.15. We emphasize that the routing in the arbitration net (and the clusters) is done statically. Especially, we note that the *all-or-nothing* property does not have to hold for the arbitration net. The key concept for the routing paths is to partition the 2D-Mesh or 2D-Torus into 4 clusters of routers such that each cluster sends exactly one phit per cycle without contention to the router of the arbiter core simultaneously. That is, the *north*, *west*, *east*, and *south* ingress ports of the arbiter core router, e.g., V_5, are connected to exactly one cluster by the cluster heads. The cluster head is the router that connects to the ingress ports of the arbiter router directly, e.g., V_2, V_6, V_8, V_4.

routing in the arbitration net (and the clusters) is done statically

Cluster Construction. Starting from the router of the arbiter core, the routers in *north*, *west*, *east*, and *south* direction belong to the respective clusters K_n, K_w, K_e, and K_s. Every router that is to the right – relative to the outward pointing direction vectors – of the routers of the clusters is added to the cluster, which is illustrated in Figure 3.15 by the red arrows. The clustering depends on the dimension of the $N \times N$ 2D-Mesh (or Torus), i.e., odd dimensions yield 4 evenly sized partitions and even dimensions result in 4 distinct cluster sizes.

Routing Paths. For each of the 4 clusters, a *zig-zag* path is constructed, which is

zig zag path

a path that contains all routers in the cluster. By this construction, each router has an in-degree of at most 1, i.e., there is no contention at any router, due to multiple phits waiting to be transmitted sequentially. The number of cycles that are required to transport a single phit from the last router on the zig zag path to the arbiter router in a cluster is given by the number of routers on the path, which is given by the cluster sizes. A *zig zag* path is constructed for a rectangular west cluster K_w by following the west direction of the cluster head until a boundary is reached (skip if no hop in west direction is possible), one hop north, following the east direction until a boundary is reached (skip if no hop in west direction is possible), one hop north, and so on until all routers in the partition are reached. Paths for the other clusters are constructed analogously by rotation of the operation, e.g., for K_n, the respective directions are replaced by the follows substitutions; $north \rightarrow east, east \rightarrow south, south \rightarrow west, west \rightarrow north$.

In the case of even dimensions, there are four distinct cluster sizes, which are given as follows:

$$K_w = \frac{N^2}{4} + \frac{N}{2} \qquad K_n = \frac{N^2}{4} \tag{3.73}$$

$$K_s = \frac{N^2}{4} - 1 \qquad K_e = \frac{N^2}{4} - \frac{N}{2} \tag{3.74}$$

Since the latency is determined by the largest cluster size, i.e.,

$$\max\{K_w, K_n, K_s, K_e\} = \frac{N^2}{4} + \frac{N}{2} \tag{3.75}$$

Corollary 3.17. *If N is even then after* $\frac{N^2}{4} + \frac{N}{2} + 1$ *cycles, each router in the system has transmitted a phit to the central arbiter core.*

In the case of odd dimensions, the cluster sizes are identical and given by

$$K_w = K_n = K_s = K_e = \left\lceil \frac{N}{2} \right\rceil \cdot \left\lfloor \frac{N}{2} \right\rfloor \tag{3.76}$$

Corollary 3.18. *If N is odd then after* $\lceil \frac{N}{2} \rceil \cdot \lfloor \frac{N}{2} \rfloor + 1$ *cycles, each router in the system has transmitted a phit to the central arbiter core.*

One additional cycle is due to the fact that even the closest non-arbiter core requires at least one hop to reach the central arbiter core.

flows per core

Message Encoding. Each router is in exactly one cluster and each core is connected to exactly one router, which injects *flows per core* (FPC) many flows into the network. Each core injects the id's of the flows waiting to be transmitted such that

$$m_{req} := FPC_{max} \cdot log_2(|\mathbb{T}|)\ [bit] \tag{3.77}$$

denotes the longest request message injected in any cluster.

arbitration response message

clear to send

After the arbitration is finished, an *arbitration response message* is sent to each cluster, containing the clusters' router ids and the flow ids, which are *clear to send*. Notably, each router can be admitted and thus notified to switch through at

most 4 flow ids simultaneously (under the SP2 property), since only the *in west* to *out north*, or *in south* to *out east*, and the opposite directions can be switched simultaneously. Thus, in the worst-case each router in the cluster receives 4 flow ids resulting in

$$m_{resp} := 4 \cdot log_2(|\mathbb{T}|) \cdot |K_{max}| \; [bit] \tag{3.78}$$

Given the sizes in bit of the response and request message we can compute the duration in terms of clock cycles as follows:

$$\Delta_{resp} := \frac{m_{resp}}{phit} \cdot \begin{cases} \lceil \frac{N}{2} \rceil \cdot \lfloor \frac{N}{2} \rfloor + 2 \; [cycles] & \text{if } N \in \mathbb{N} \text{ is odd} \\ (\frac{N}{2} + 1) \cdot \frac{N}{2} + 2 \; [cycles] & \text{otherwise} \end{cases} \tag{3.79}$$

Similarly,

$$\Delta_{req} := \frac{m_{req}}{phit} \cdot \begin{cases} \lceil \frac{N}{2} \rceil \cdot \lfloor \frac{N}{2} \rfloor + 2 \; [cycles] & \text{if } N \in \mathbb{N} \text{ is odd} \\ (\frac{N}{2} + 1) \cdot \frac{N}{2} + 2 \; [cycles] & \text{otherwise} \end{cases} \tag{3.80}$$

The remaining Δ_{arb} depends on the arbitration routine, which is executed on the arbiter core. The runtime must be obtained from worst-case execution time and worst-case response-time analysis on the core. In the remainder of the analysis we presume a determined Δ_{arb} such that $\Delta = \Delta_{req} + \Delta_{arb} + \Delta_{resp}$.

3.4.5 Response-time & schedulability analysis

In this section, we elaborate on the response-time analyses for a determined *epoch*, denoted as Δ, and the restriction that all arbitration events such as flow arrival times, transmission grants, and preemptions are only granted at epoch intervals. In order to analyze the discrete-time response-time of a flow, the worst-case traversal time (WCTT), i.e., the transmission time of each flow without any interference must be computed.

Definition 3.16 (Worst-Case Traversal Time)**.** *Let $C_i \in \mathbb{N}$ denote the length of a message as an integer multiple of a phit, that is, the number of cycles required to fully transmit a message over a single link. Then, the worst-case traversal time of a message C_i is*

$$WCTT_i = \begin{cases} \left(\left\lceil \frac{C_i}{\Delta} \right\rceil + (\eta_i - 1) \right) \cdot \Delta & \text{if } C_i + \eta_i > \Delta + 1 \\ \Delta & \text{otherwise} \end{cases} \tag{3.81}$$

number of cycles to progress through the static path $\mathbb{L}_i$ from the source core to the destination core (without any interference).

We need to make a distinction of the worst-case traversal time dependent on the message sizes and path length, because if a message can be transmitted in one epoch then this results in a $WCTT_i$ of Δ.

The network interfaces at the cores decouple the computation and communication. Hence, if a message of flow τ_i is released at time a_i, the arbitration

phase starts at the next epoch boundary. The waiting time until the message is eligible to compete for arbitration is given by $\lceil \frac{a_i}{\Delta} \rceil \cdot \Delta - a_i$ which is no more than Δ. Recurring on these results, we can state the worst-case response time analysis for each flow τ_i in analogy to the analysis presented in Section 3.3.3.3 as follows.

Corollary 3.19. *A sporadic constrained-deadline flow set $\mathbb{T}$ is fixed-priority schedulable using SP2 – with an implementation given epoch duration Δ and a given priority order – if for each flow $\tau_i \in \mathbb{T}$ the following two conditions are satisfied:*

1. *All higher-priority flows of τ_i have already been validated to be schedulable, which can be achieved by the iterative application of this rule in decreasing priority order.*
2. *The flow τ_i is schedulable according to any fixed-priority self-suspension aware uniprocessor schedulability analysis, where the higher-priority task set of τ_i is transformed according to the following equation: For each $\tau_j \in hp(\tau_i)$*

$$\tau_j' = \begin{cases} (WCTT_j,\ T_j - \Delta,\ D_j - 2\Delta,\ R_j - WCTT_j) & \text{if } \tau_j \text{ has self-suspension behaviour} \\ (WCTT_j,\ T_j - \Delta,\ D_j - 2\Delta) & \text{otherwise} \end{cases} \tag{3.82}$$

The release-jitter of higher-priority flows which is due to the delayed arbitration, is accounted for by the adjusted period. Moreover, one epoch of interference, which is due to the delay in the arbitration net, must be accounted for, which yields an adjusted deadline of $D_j - 2\Delta$.

3.4.6 EVALUATION

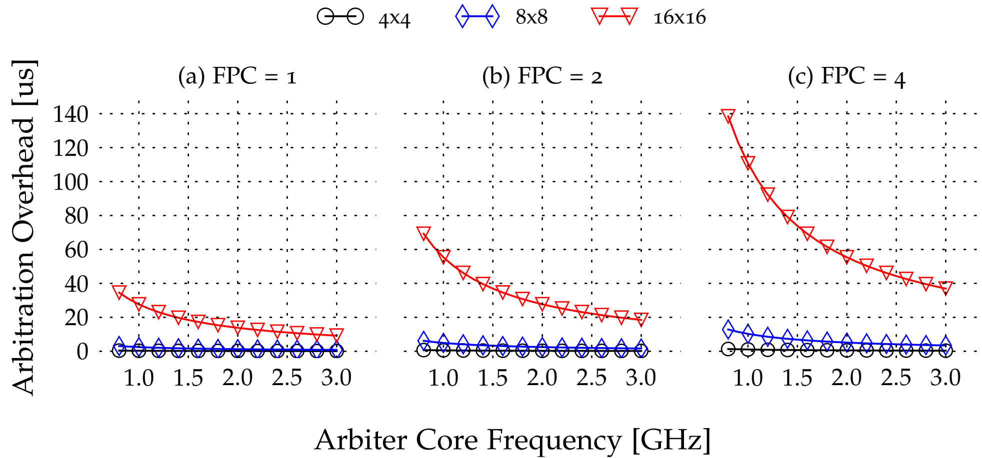

Figure 3.16: Overhead for the arbitration algorithm in μs for varying number of flows per core (FPC), 2D-Mesh dimensions, and core frequencies from 800 MHz to 3000 MHz.

In our experiments, the number of flows with a unique priority on each source core is varied with the total number of cores in the network. Specifically, we assume that each core sends up to a fixed number (denoted as flows per core (FPC), e.g., 1, 2, or 4) of *flow ids* in one arbitration epoch. Then, we asses the

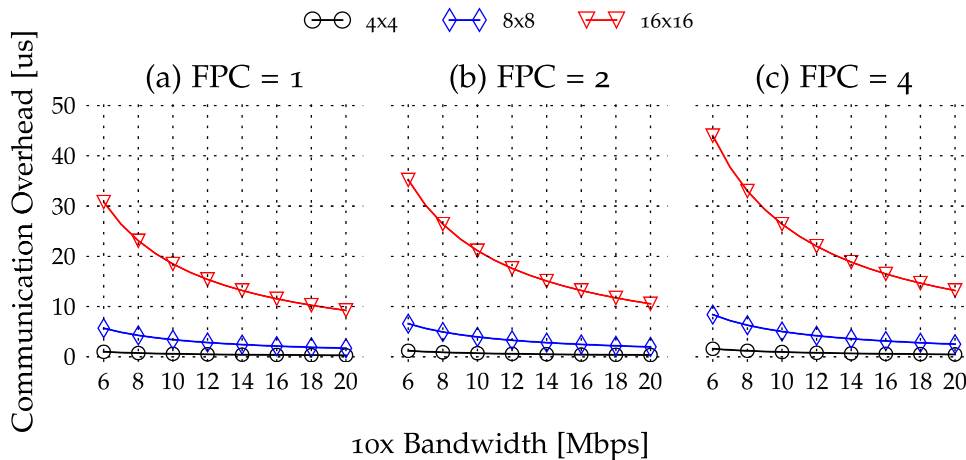

Figure 3.17: Overhead for the arbitration communication in μs for varying number of flows per core (FPC), 2D-Mesh dimensions, and link bandwidths. Note that the communication overhead is independent from the arbiter core frequency.

Table 3.2: Number of arbiter core cycles required for the arbitration algorithm in a N×N 2D-Mesh network-on-chip with 1,2,4, and 8 flows per core.

N / #cycles	1-FPC	2-FPC	4-FPC	8-FPC
4	300	552	1064	2092
8	2465	4920	10269	20672
12	9618	20204	40549	80960
16	27643	55389	110748	224600

arbitration overhead as well as the communication overhead for different core frequencies and link bandwidths. In these experiments, we implemented the SP2 arbitration algorithm in software and measured the required time for varying core frequencies. We emphasize that in a real realization the arbitration may also be done in hardware.

Arbitration Overhead Measurements. The arbitration overhead is calculated by measuring the execution time of the arbitration algorithm explained in Algorithm 2. This arbitration algorithm was prototyped in C++ and run on an Intel Xenon Gold 6126 CPU core at 2.6 GHz. Based on these measured execution times and the known core frequency of the measuring system, we normalized and estimated the overhead for different core frequencies in the range of $800, 900, \ldots, 3000$ MHz as shown in Figure 3.16.

Communication Overhead Measurements. The communication overhead is determined by calculating the required number of cycles such that all arbitration requests (from all source cores) are transmitted to the arbiter and the responses are received at the source cores. Using the results from Eq. (3.80) and Eq. (3.79), the cumulative time that is required for the arbitration communication is given by the sum of the equations divided by the link bandwidth in $10^6 bps$. The results are presented in Figure 3.17 for different link bandwidths accordingly.

Overhead Results. As shown in Figure 3.16, the overheads for the communication and arbitration increase with the number of flows per core as well as with

the dimension of the 2D-Mesh. For 256 flows (4 FPC) in a 8×8 mesh, the communication overhead is 64 μs for 10Mbps, which suggests that the communication overhead for 4×4 and 8×8 2D-Meshes is low enough such that an arbitration within the data plane could be a feasible alternative. The communication overhead for 1024 flows (4 FPC) in a 16×16 mesh and 10 Mbps links is 320 μs but can be reduced to 16 μs for 200 Mbps links. The data suggest that the overhead can become too large for higher dimensions and large number of flows. In these cases, other approaches for the arbitration net are required.

Concluding Remark. As an assessment of this proposed initial conceptual implementation, one can observe that the reduction of all possible series of progressions to only allow the fastest progression – and thus an uniprocessor equivalent execution model – yields good analyzability, since uniprocessor scheduling theory is well understood and very precise analyses exist. Additionally, complex buffer contention scenarios do not need to be considered.

Another advantage is that only a phit-sized buffer is required at each router for each flow that is crossing it. However, at the cost that each phit-sized buffer implements a virtual channel in order to implement the preemption of SP2. Likely, our protocol can be augmented on top of existing wormhole-switched networks in case that real-time verification is mandatory and the network dimension and traffic load is small enough. From the negative perspective, the scaling issue is the most predominant one, i.e., in our suggested implementation, the epoch duration is in the order of $\mathcal{O}(n^2)$, which suggests that for larger network sizes, larger number of flows, and longer arbitration messages, the schedulability may be too degraded.

3.5 CONCLUSION

With reference to the hypothesis of this dissertation that either the parameter uncertainty and hardware peculiarities must be considered in the scheduling algorithm design and associated formal schedulability analyses, or the predictability of the hardware must be increased, the focus in this chapter is on the design of arbitration protocols and scheduling algorithms that increase predictability to allow for safe worst-case response time analyses.

- At first, a specialization of the rigid gang scheduling problem for hard real-time systems is proposed, in which each gang task is restricted to execute on a assigned subset of processors. We presented how the corresponding schedulability test problem can be reduced to a uniprocessor suspension-aware schedulability test. Furthermore, we showed how to derive specific consecutive stationary rigid gang assigments for preemptive deadline-monotonic gang scheduling that admits a parametric speed-up factor with respect to an optimal rigid gang scheduling algorithm. While the presented scheduling algorithm is task-level fixed-priority, extensions to job-level fixed-priority scheduling algorithms such as EDF are possible by adopting the proper suspension-aware schedulability tests and suspension inducing behaviour analysis. As future work, we plan to implement a fixed-priority stationary gang scheduler in a real-time operating systems

and evaluate the scheduling overheads and investigate potential benefits with respect to improved worst-case execution time and resource contention patterns.

- Secondly, we propose a family of simultaneous progression switching protocols for real-time NoC arbitration that is described by the *all-or-nothing* property and provides increased predictability at the cost of decreased average case performance. A possible implementation, including router design, and arbitration algorithm is given. While the evaluations hint to scalability issues in our proposed implementation for larger networks, for smaller to medium sized NoCs, our proposal suggests to be a beneficial first step towards the design and analysis of timing-predictable switching protocols of NoCs that allow for safe response-time analyses. Notably, any non-minimal route in simultaneous progression switching protocols is deadlock-free, since the *simultaneous progression* property prevents circular waiting at the buffers. Therefore, the path diversity can be better utilized in order to distribute the load over the links such that contention is reduced.

4 HIERARCHICAL PARALLEL DAG SCHEDULING

Cyber-physical systems have shifted from uniprocessor to multiprocessor system designs, resulting in multiple challenges for the design and verification methodology of parallel real-time applications. In particular, fine-grained parallel task models, and appropriate scheduling algorithms, which are amenable to formal response-time analyses, and robust with respect to parameter uncertainty, are mandatory.

Hierarchical scheduling provides temporal and spatial isolation, which is beneficial for the robustness of the real-time system design, with respect to parameter uncertainty in the control flow, and worst-case execution times. Decomposing the scheduling problem into an *inter-task* and *intra-task scheduling problem*, the response-time behaviour of each DAG job can be formally verified, on the basis of a precisely specified promised service contract, which is guaranteed to be honored. Consequently, possible worst-case execution time underestimations, resulting from unsound pWCET techniques, can be observed, and limited to the failure of a single reservation system. Henceforth, appropriate counter-measures can be initiated in the real-time system to handle the failure gracefully.

worst-case execution time underestimations, resulting from unsound pWCET techniques, can be observed, and limited to the failure of a single reservation system

In this chapter, we propose two hierarchical scheduling-based solutions to, firstly address uncertainty in the control flow, and thus precedence constraints of parallel DAG tasks. This uncertainty, is modeled with probabilistic precedence constraints, formalized in the proposed *probabilistic conditional DAG task* model. A bounded tardiness constraint, for the execution of a probabilistic conditional DAG task, is considered under k-consecutive deadline miss constraints in Section 4.4 *Probabilistic Conditional-DAG Scheduling*. Secondly, we present the *parallel-path progression* concept in Section 4.5 *Parallel Path Progression Scheduling*. This concept allows to schedule highly parallel DAG structures provably efficient, and in some cases even optimally, on gang, and ordinary reservation systems. Starting from Section 4.1 *Motivation*, in which hierarchical scheduling approaches are motivated for parallel task and parallel DAG task scheduling in particular, the related work regarding parallel real-time scheduling is given in Section 4.2 *Related Work*. The common task models, definitions, and the general hierarchical scheduling approach is explained in Section 4.3 *Hierarchical DAG Scheduling & DAG Task Model*. Lastly, the chapter is concluded with a summary of the results in Section 4.6 *Conclusion*.

probabilistic conditional DAG task

4.1 MOTIVATION

The overarching objective in real-time scheduling of parallel DAG tasks, is to efficiently utilize the parallelism, provided by multiprocessors, for task set execution with inter- and intra-task parallelism, and to verify temporal constraints.

Based on the dissertation hypothesis, we believe that the decoupling of parallel application design and scheduling on the one hand, and real-time operating system scheduling and service contracts on the other hand, is the most promising approach to implement real-time aware parallel applications. This is, due to the temporal, and spatial isolation, as well as the easy integration with any readily available scheduling algorithm, provided by the real-time operating system. For instance, partitioned or global scheduling algorithm variants of task-level fixed-priority, or earliest-deadline first (EDF) scheduling algorithms can be used. Above the modular decomposition, of the scheduling problem, a key property of using hierarchical scheduling is that the problem of task model parameter uncertainty, such as worst-case execution time underestimations by using e.g., pWCET techniques, can be observed and limited to the failure of a single reservation system.

hierarchical scheduling approach for parallel applications is also very suitable for implementation and usability

Beyond, the property of robustness, and modularity, the hierarchical scheduling approach for parallel applications is also very suitable for implementation and usability. This is evident by the fact that the de-facto standard programming and scheduling model for parallel computing, namely OpenMP, and the real-time extension OmpSs [DAB+11] are using hierarchical scheduling. On the higher-level, the worker threads, which implement the reservations, are scheduled by the real-time operating system; on a lower-level, the subjobs are managed by the respective runtime environment, which is responsible to implement the internal DAG scheduling algorithm. The concrete implementation, of the hierarchical scheduling algorithm, is not standardized. E.g., the approaches of OpenMP and OmpSs differ in that, OpenMP implements fork-join parallelism with a master thread, which creates a so called team of parallel threads on encountering a parallel region; In contrast, OmpSs uses a pool of worker threads to serve the subjobs as soon as they become ready, which is similar to *list-scheduling*. In both frameworks, the application source code is written in a high-level programming language, which is instrumented with a set of directives, which together with library routines, and a provided runtime environment, are used to describe and execute the parallel application.

list scheduling

In the context of cyber-physical systems, the hierarchical scheduling of parallel DAG tasks, is further complicated by the following problems:

control flow and execution times of the application depend on the captured sensory information

- Beyond the hardware induced parameter uncertainties in terms of the worst-case execution time, many relevant cyber-physical systems, such as autonomous driving systems, are composed of applications, in which the control flow and execution times of the application depend on the captured sensory information. For instance, the number of objects, which must be tracked with the radar, or the number of objects, which must be identified and processed within an image for trajectory planning, result in different program executions and subsequently, highly variable execution times. These applications are reported to be subject to multiple conditional branches and control flow instructions, which change the structure of a DAG task during runtime as stated by Melani et al. [MBB+15]. However, for many applications in those systems, e.g., closed-loop feedback controllers, worst-case centric provisioning with a safe but very pessimistic upper bound is not required, due to the inherent controller robustness towards timing

non-idealities like bounded tardiness, and deadline misses. To that end, the uncertain execution behaviour and response-time of conditional control flow can be accepted and analyzed in a probabilistic sense. Many research efforts have been focused on formalizing and analyzing relaxations of deadline constraints [PMM+19], e.g., weakly hard systems where m out of k task instances must meet the deadlines. Moreover, Maggio et al. [MHM+20] and Vreman et al. [VCM21] investigate the closed-loop control system stability under consecutive deadline-miss constraints, which further motivates the need for scheduling algorithms, which can guarantee probabilistic bounds on consecutive deadline misses.

conditional control flow can be accepted and analyzed in a probabilistic sense

- With regards to hard real-time hierarchical parallel DAG task scheduling, another challenge is the resource efficiency of the hierarchical scheduling algorithms. Paradoxically, the large number of processors of modern multi-processor architectures, e.g., the *Kalray MPPA* architecture with 256 cores, is detrimental to the resource efficiency if the increased number of processors does not benefit the worst-case response-time analysis.

In this chapter, we propose the following contributions to address the motivated problems. In order to formally describe and verify quantitive guarantees of deadline misses, some quantifications are of importance for soft real-time systems, namely the probability of a deadline miss, probability for k consecutive deadlines misses, and the maximum tardiness of a job. Despite the fact that these guarantees are soft, the precise quantification thereof are hard and challenging, even for the ordinary sequential real-time task models, which are scheduled upon a uniprocessor system. We analyze, optimize and verify the schedulability of probabilistic conditional parallel DAG tasks on identical multiprocessors with respect to quantities such as deadline-miss probabilities, consecutive deadline-miss probabilities and tardiness constraints. When considering the scheduling and analysis of probabilistic conditional parallel DAG tasks, not only inter-task, but also intra-task interference, and multiprocessor *scheduling anomaly* effects, i.e. the early completion of subjobs may lead to longer response-times, must be considered; complicating the analyses for the above mentioned quantities. In the state-of-the-art analyses, the above quantities can only be derived under strict model assumptions, e.g., that a job is aborted whenever a job exceeds its deadline. The reason for this complexity is partly due to inter-task interference, i.e., the preemption and interference patterns of the task system, due to higher-priority jobs, which results in a large number of system states, which must be considered in a response-time analysis.

scheduling anomaly

With regards to the second problem, we propose a subtask-level fixed-priority policy, which allows to account for the fact that multiple paths within a DAG are executed simultaneously, which can thus be removed from the self-interference analysis. This *parallel path progression* property improves Graham's bound substantially, and improves the resource efficiency of the proposed (independent) ordinary-, and gang reservation systems in the hierarchical scheduling. Moreover, a stricter *path monotonic prioritization*, which induces the *path monotonic progression* property, is presented. On the basis, of the *path monotonic progression* property, a self-suspension aware reservation system can be devised.

parallel path progression

path monotonic prioritization

path monotonic progression

4.2 RELATED WORK

The scheduling of parallel real-time tasks with worst-case parameters, e.g., worst-case execution times, upon multiprocessor systems has been extensively studied for different parallel task models. An early classification of parallel tasks with real-time constraints into rigid, moldable or malleable has been described by Goosens et al. [GB10a].

Parallelism can be categorized into inter-task parallelism, which refers to the parallel execution of distinct tasks; each of which executes sequentially and intra-task parallelism which refers to the parallel execution of a single task. Intra-task parallelism requires task models with subtask-level granularity, which can be scheduled in parallel, e.g., fork-join models [LKR10], synchronous parallel task models, or DAG (directed-acyclic graph) based task models [FNN17; BMSS+13; Bar15a; BMS+13; MBB+15]. Early work on parallel task models focused on synchronous parallel task models, e.g., [MBN+14; SAL+11; CLP+13]. Synchronous parallel task models extend the fork-join model [Con63] in such a way that they allow different numbers of subtasks in each (synchronized) segment where the number of subtasks can exceed the number of processors. More recently, the directed-acyclic graph (DAG) task model has been proposed and been subject to scheduling algorithm design and analysis. The DAG task is a more general parallel structure where each task is described by a set of subtasks and their precedence constraints that are represented by a directed-acyclic graph. The parallel DAG task model has been studied for global [BMS+13; NNB19; CA14] and partitioned scheduling [BMSS+13; CBN+18b; BCM19; FNN17].

The formal parallel DAG task model has been shown to correspond to models in parallel computing APIs such as OpenMP by Sun et al. [SGW+17], or Serrano et al. in [SMV+15]. The observation that the OpenMP tasking model resembles the formal sporadic DAG task scheduling model when considering a single real-time task was further reinforced in several other works [SRQ18; MSB+17; VQM15; VRS+16]. However, OpenMP presents limitations with respect to modeling sporadic real-time DAG tasks such as missing directives to specify deadlines or sporadic job releases such that an extension for real-time systems is not trivial. To that end, the OmpSs [DAB+11] programming model and its successor OmpSs-2 were proposed and developed in the context of the P-SOCRATES project [PNY+15]. Similar to OpenMP, OmpSs and OmpSs-2 use a set of directives to instrument an application source code written in a high-level programming language.

The proposed scheduling algorithms and analyses in the literature can be categorized into decompositional and non-decompositional. In the former, the parallel task model is decomposed into a set of sequential task models, which is scheduled and analyzed in their stead, e.g., [JGL+20; BMSS+13; CBN+18b; BMS+13; NNB19]. Non-decompositional approaches consider the peculiarities of the parallel task models, e.g., [LCA+14; UBC+18; Bar15b; NNB19; DL18; BMS+13]. For instance the non-decompositional federated scheduling as proposed by Li et al. [LCA+14] avoids inter-task interference for parallel tasks by exclusive processor allocation. The idea of federated scheduling has been extended in various forms, e.g., in [JGL+17; JGL+21; DGA20; UBC+18; Bar15b; Bar15a]. In the context of

parallel DAG tasks Ueter et al. proposed a reservation scheme to schedule sporadic arbitrary-deadline DAG tasks [UBC+18] with real-time constraints and Buttazzo et al. [BBW11] studied DAG task scheduling on partitioned reservation systems.

For sequential stochastic tasks a plethora of prior work concerning probabilistic analyses exists, e.g., [SYM+11; HTA19]. Recent work has focused on the improvements of efficiency in convolution-based probabilistic deadline-miss analysis approaches. Von der Brüggen et al. [BPC+18] propose efficient convolutions over multinomial distributions by exploiting several state space reduction techniques, approximations using Hoeffding's and Bernstein's inequality, and unifying equivalence classes. Chen et al. [CBC18a] propose the efficient calculation of consecutive deadline-misses using Chebyshev's inequality and moment-generating functions and optimizations thereof. More recent advances by Markovic et al. in [MNP22; MPN21] further improve the computational complexity for the convolution-based approaches. There have also been efforts to use reservation servers to schedule probabilistic sequential tasks. For example, Palopoli et al. [PFA+16] have shown how to calculate the probability of a deadline miss for periodic real-time tasks scheduled using the constant bandwidth server (CBS). The authors have reduced the computation to the computation of a steady state probability of an infinite state discrete time markov chain with periodic structure. Other approaches to tackle the probabilistic analysis of real-time tasks is real-time queuing theory by Lehoczky et al. [Leh96], which is an extension of classical queuing theory to systems with deadlines. An initial work that analyzed the probabilistic response-times of parallel DAG tasks was proposed by Li [LAG+14]. Li extended prior work on federated scheduling [LCA+14] by facilitating queuing theory to devise federated scheduling parameters such that each task's tardiness is bounded and soft real-time requirements are met. More recent work on the probabilistic response-time analysis of parallel DAG tasks is by Ben-Amor et al. [BMC16; BCM+20a; BCM+20b]. The authors have studied the probabilistic response-time analysis of parallel DAG tasks upon multiprocessor systems using partitioned fixed-priority scheduling at the subtask-level. In their model each subtask is described by a probabilistic worst-case execution time and static precedence constraints between them. Based on the above, the authors derive probabilities for subtask response-times using convolution based approaches and compose an overall response-time. However the approaches by Ben-Amor et al. [BMC16; BCM+20a; BCM+20b] are limited to the response-time analysis of a single probabilistic DAG task and can not easily be extended to multiple tasks due to more complex inter-task interference.

Probabilistic analyses for parallel DAG tasks

Li extended prior work on federated scheduling [LCA+14] by facilitating queuing theory

convolution based approaches

Motivated by the conditional execution behaviour of modern parallel applications, e.g., autonomous driving or robotics, where the branching decisions and execution times are dependent on sensory information, e.g., in object detection, and control applications, the conditional DAG task model has been proposed. A plethora of research concerning the real-time schedulability of this model has been conducted by e.g., [MBB+15; Bar15c; CLJ+19]. Most recently, the computational complexity of the scheduling of conditional DAG with real-time constraints has been investigated by Marchetti et al. [MMS+20], and Baruah et al. in [Bar21; BM21]. In light of the observations that many applications that motivate the

conditional DAG task model, e.g., closed-loop feedback controllers hard real-time system engineering (with a safe but very pessimistic upper bound) is not required due to the inherent controller robustness towards timing non-idealities like jitter and deadline misses. To that end, the uncertain execution behaviour and response-time of conditional parallel tasks can be *accepted* and analyzed in a probabilistic sense. Many research efforts have been focused on formalizing and analyzing relaxations of deadline constraints [PMM+19], e.g., weakly hard systems where m out of k task instances must meet the deadlines. Moreover, Maggio et al. [MHM+20] and Vreman et al. [VCM21; VPM+22] investigate the closed-loop control system stability under consecutive deadline-miss constraints, which motivates scheduling algorithms that can guarantee probabilistic bounds on consecutive deadline misses to the application. To that end, soft real-time applications that can tolerate bounded number of deadline-misses, probabilistic task models and response-time analyses for these kind of parallel conditional DAG tasks are of interest.

uncertain execution behaviour and response-time of conditional parallel tasks can be accepted *and analyzed in a probabilistic sense*

Improvements in the response-time analyses can be categorized into inter-task and intra-task interference improvement

Improvements in the response-time analyses can be categorized into inter-task and intra-task interference improvement. A plethora of real-time scheduling algorithms and response-time analyses thereof have been proposed in the literature, e.g., for generalized parallel task models [SAL+11], and for DAG (directed-acyclic graph) based task models [HJG+19; DL17; ZDB+20; FNN17; BMSS+13; Bar15a; BMS+13; MBB+15]. For DAG-based task models, improvements in the response-time analyses can be categorized into analyses that improve inter-task interference, e.g., in [DL18; FNN17], or intra-task interference as e.g., in [LCA+14; HJG+19; HLG21; ZDB+20]. In general, intra-task interference analyses build upon the interference analysis along the execution of the envelope path. In federated scheduling [LCA+14], the intra-task interference of the envelope execution is upper-bounded by the workload of the non-envelope subjobs divided by the number of processors. The corresponding response-time analysis requires no information about the internal structure of the DAG except for the total volume and the longest path. This analysis was improved by He et al. [HJG+19], who proposed a specific intra-vertex priority assignment for list-scheduling that must respect the topological ordering of the vertexs within the DAG. This priority assignment and the inspection of the DAG structure results in a less pessimistic upper-bound for a task's self-interference of the envelope path compared to federated scheduling. These results are further improved and extended by Zhao et al. [ZDB+20], where subjob dependencies are explicitly considered along the execution of the envelope path to more accurately bound self-interference. Most recently, He et al. [HLG21] improve their prior work by lifting the topological order restrictions in their intra-vertex priority assignments, which further improved the results by Zhao et al. [ZDB+20].

inspection of the DAG structure results in a less pessimistic upper-bound for a task's self-interference

4.3 HIERARCHICAL DAG SCHEDULING & DAG TASK MODEL

In this section, the *fundamental* parallel DAG task model, used in this chapter, is formally described. Subsequently, the most fundamental definitions, and techniques for analysis are presented. Hereinafter, the hierarchical scheduling ap-

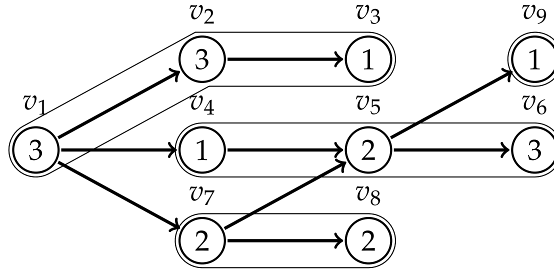

Figure 4.1: An exemplary directed-acyclic graph (DAG) with subtasks $v_1, v_2, \ldots, v_9$. The numbers within the vertices denote the subtasks' worst-case execution time. The arrows represent the precedence constraints indicating that the release of a subjob depends on the finishing of all incident subjobs.

proach, and the properties which are inherent to the studied reservation systems are examined.

A sporadic arbitrary-deadline (or constrained-deadline) directed-acyclic graph (DAG) task $\tau_i := (G_i, D_i, T_i)$ is defined by a DAG G_i, minimal inter-arrival time T_i, and relative deadline D_i. Each task releases an infinite sequence of task instances, called jobs. We use J_i^ℓ to denote the ℓ-th job of task τ_i, and a_i^ℓ, f_i^ℓ, and $d_i^\ell = a_i^\ell + D_i$ to refer to the arrival time, finishing time, and (absolute) deadline of job J_i^ℓ.

task $\tau_i := (G_i, D_i, T_i)$ is defined by a DAG G_i, minimal inter-arrival time T_i, and relative deadline D_i

The task's DAG G_i is defined by the tuple (V_i, E_i), where V_i denotes the finite set of subtasks and the relation $E_i \subseteq V_i \times V_i$ denotes the *precedence constraints* among them, such that there are no cyclic precedence constraints. To be mathematically precise, each job J_i^ℓ is associated with an instance of the DAG G_i^ℓ with corresponding ℓ-th subjobs v_j^ℓ where v_j is a subtask in V_i. A subjob of the ℓ-th job of task τ_i, namely v_j^ℓ for $v_j \in V_i$, is released when all ℓ-th subjobs v_k^ℓ for $(v_k, v_j) \in V_i$ have finished execution. To reduce this notation, we drop the index of the task as well as of the job when analyzing one specific job. That is, we refer to $G = (V, E)$ and $v_j \in V$ to denote a subjob of a specific DAG job; an exemplary DAG is illustrated in Figure 4.1.

precedence constraints

In hard real-time systems, tasks must fulfill timing requirements, i.e., each job J_i^ℓ must finish its *total volume* between the arrival of a job at a_i^ℓ and that job's absolute deadline at $a_i^\ell + D_i$. That is, a task τ_i is said to meet its deadline if each job meets its deadline, i.e., $f_i^\ell \leq a_i^\ell + D_i$ for all $\ell \in \mathbb{N}$.

total volume

Definition 4.1 (Volume)**.** *The volume $vol_i : V_i \to \mathbb{R}_{\geq 0}$ specifies the worst-case execution time of each subtask $v_j \in V_i$, which means that no subjob (instance) v_j^ℓ ever executes for more than $vol_i(v_j)$ time-units on the execution platform, but may finish earlier. Moreover, the volume of any subset of subtasks $W \subseteq V_i$ is $vol(W) := \sum_{v_j \in W} vol_i(v_j)$. In particular, the* total volume *of a task τ_i is given by $C_i := vol_i(V_i)$.*

On the basis of a volume function, the *length* of a DAG, i.e., the volume of the longest path in a DAG can be defined as follows.

length

Definition 4.2 (Length)**.** *The length L_i of a DAG G_i is defined based on the volume as $L_i := \max\{vol_i(\pi) \mid \pi \text{ is a complete path in } G_i\}$, i.e., the longest path with respect to the cumulative worst-case execution time.*

Recall, that in *arbitrary-deadlines*, we do not make any assumptions about the relation of deadline and inter-arrival time, whereas *constrained-deadlines* implies that the relative deadline is no more than the inter-arrival time.

Definition 4.3 (Complete Path). *Let a DAG $G = (V, E)$ then for each subtask $v_j \in V$, the set of predecessors of v_j and the set of successor of v_j is given by $pred(v_j) := \{v_i \in V \mid (v_i, v_j) \in E\}$ and $succ(v_j) := \{v_i \in V \mid (v_j, v_i) \in E\}$, respectively. A complete path is a strictly ordered set of subtasks $\pi := \langle v_1, \dots, v_n \rangle$ such that $pred(v_1) = \emptyset$, $succ(v_n) = \emptyset$ and $v_k \in pred(v_{k+1})$ for all $k \in \{1, \dots, n-1\}$.*

In this chapter, we propose a specific two-layered hierarchical scheduling approach for parallel DAG tasks. On the lower level, reservation systems (or threads), which comply with the sporadic task model are scheduled on the physical processors by any appropriate scheduling policy. On the higher level, the attached DAG job and its subjobs are dispatched and serviced by the reservations in a temporally and spatially isolated environment for the promised amount of service. Notably, reservation systems can be co-scheduled with other task systems on the same set of physical processors, using readily available response-time analyses.

In the context of this chapter, a reservation system for a DAG task τ_i, is defined by a set of m_i reservations, namely $(E_i^1, D_i, T_i), \dots, (E_i^{m_i}, D_i, T_i)$; each of which has a fixed promised budget of $E_i^j \leq D_i$ amount of time for $j \in \{1, \dots, m_i\}$. Each of the m_i reservations are released simultaneously, with the release of the to-be served DAG job, such that each reservation offers E_i^j amount of service, for any of the subjobs of the attached DAG job, during the interval $[a_i, d_i)$; consequently the inter-arrival times of two subsequent instances of the reservation system is at least T_i. Note, that each reservation system instance serves exactly one DAG task instance, and that the service is only offered at a time $t \in [a_i, d_i)$, if the respective reservation is scheduled at that time t.

Reservation Properties. The formal properties, and the internal dispatching mechanism of the subjobs of the attached DAG job, are formally stated in the following definitions.

Definition 4.4 (Parallel Service). *A reservation system instance provides $m_i \in \mathbb{N}$ parallel reservations, such that at any time, at most m_i reservations can provide service to the attached DAG job, concurrently.*

Definition 4.5 (Attached Service). *An instance of the reservation system serves exactly one DAG job of a DAG task. This means that an instance of the $m_i \in \mathbb{N}$ parallel reservations that serve the ℓ-th job J_i^ℓ of DAG task τ_i all arrive synchronous at time a_i^ℓ and the deadline is given by d_i^ℓ. Note that if the next DAG job arrives before the previous one is finished, a new instance of reservations is released and the accumulate DAG job is attached to it.*

Definition 4.6 (Sustained Service). *The service of a reservation is provided whenever the reservation system is eligible to execute, i.e., is scheduled, irrespective of the actual service requests of the attached DAG job.*

A negative consequence of *sustained service* is that the reservation system may block the processor without executing any workload. This issue is described in Section 4.5.6, and some solutions are discussed to improve resource utilization.

Definition 4.7 (Internal Scheduling). *In* preemptive subtask-level fixed-priority list scheduling *(List-FP) of the provided service by a reservation system, a task instance (job) of a DAG task $G = (V, E)$ with a fixed-priority assignment of each subjob $v \in V$ is serviced according to the following rules:*

- *A subjob arrives to the reservation system's internal ready list, if all preceding subjobs have executed until completion, i.e., the subjob arrival time a_i for each subjob v_i is given by $\max \{f_j \mid v_j \in pred(v_i)\}$. An arrived, but not yet finished subjob, is considered* pending.

pending subjob

- *At any time t, all reservations that can provide service at time t, are busy executing the highest-priority pending subjobs. A lower-priority subjob is preempted if necessary, i.e., a higher-priority subjob is released.*

Analysis Framework. All the forthcoming analyses, of the proposed hierarchical scheduling approaches, consist of a two-stage analysis, namely:

1. Firstly, the schedulability analysis of the reservation systems upon M identical multiprocessors; hence it is guaranteed that the promised service can be provided within the interval $[a_i, d_i)$.
2. And secondly – premised on provided service in the prior stage – the response-time analysis of the attached DAG job.

With regards to the first analysis step, it is to emphasize that our individual reservations are intentionally modeled as sequential sporadic arbitrary-deadline tasks and thus any scheduling algorithm and any appropriate schedulability analysis can be used, e.g., global fixed-priority scheduling, or partitioned earliest-deadline first. Therefore, the research focus in this chapter is on the provision of the reservation budgets, and the response-time analysis of a DAG job, given the contracted service, provided by the reservations.

research focus in this chapter is on the provision of the reservation budgets

The response-time analyses of a specific DAG job of interest, are based on the envelope concept (or critical path), which is formally introduced as follows.

Definition 4.8 (Envelope). *Let S be any concrete schedule of the subjobs $V = \{v_1, \ldots, v_\ell\}$ of a given DAG job of some DAG task $G = (V, E)$. Let each subjob $v_k \in V$ have the arrival time a_k and finishing time f_k in S. We define the envelope of G in S as the collection of arrival and finishing time intervals $[a_{k_1}, f_{k_1}), [a_{k_2}, f_{k_2}), \ldots, [a_{k_p}, f_{k_p})$ for some $p \in \{1, \ldots, \ell\}$ backwards in an iterative manner as follows:*

1. *$k_i \neq k_j \in \{1, \ldots, \ell\}$ for all $i \neq j$.*
2. *v_{k_p} is the subjob in V with the maximal finishing time.*
3. *$v_{k_{i-1}}$ is the subjob preceding v_{k_i} with maximal finishing time, for all $i \in \{p, p-1, \ldots, 2\}$.*
4. *v_{k_1} is a source vertex, i.e., has no predecessor.*

Based on the envelope, we call the ordered set $\pi_e := \langle v_{k_1}, v_{k_2}, \ldots, v_{k_p} \rangle$ the envelope path of G in S. We note that the definition of an envelope for a DAG job may not be unique if there are subjobs with the same finishing times. In that case, ties can be broken arbitrarily.

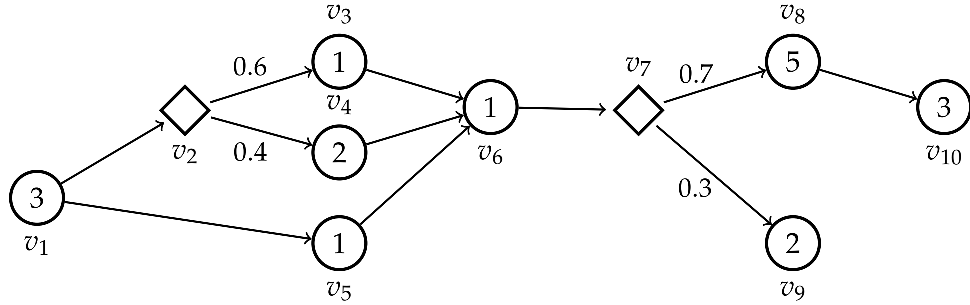

Figure 4.2: An exemplary probabilistic conditional DAG task in which each conditional vertex (diamond) denotes that only one of its adjacent subjobs is released (with the annotated probability) during runtime. In this specific example four different DAG structures can be instanced during runtime.

The arrival times of each subjob depend on the finishing time of all preceding subjobs, and therefore the arrival sequence for all subjobs is dependent on a concrete schedule and thus may be different for different concrete schedules. Consequently, any complete path in G can be the envelope path in a concrete schedule, if no further properties such as subtask priorities and path volume are considered. The key property of the envelope is that it contiguously partitions the execution interval $[a_i, f_i]$ of the DAG job, and in each sub interval $[a_{k_p}, f_{k_p})$, the subjob v_{k_p} is pending and eligible to execute, since all its precedence constraints are satisfied by the formal construction.

key property of the envelope is that it contiguously partitions the execution interval

4.4 PROBABILISTIC CONDITIONAL-DAG SCHEDULING

In this section, we analyze the probabilistic response-time of parallel conditional DAG tasks, which are not forced to be immediately aborted, when the deadline is missed. The hierarchical scheduling algorithm can be integrated with hard real-time workloads, as well as any number of probabilistic parallel DAG tasks. In order to formally describe and verify quantitive guarantees of deadline misses, some quantifications are of importance for soft real-time systems, namely the *probability of a deadline miss*, *probability for k-consecutive deadlines misses* and the *maximum tardiness* of a job.

In this contribution, we aim to analyze, optimize, and verify the schedulability of probabilistic conditional parallel DAG tasks on identical multiprocessors, with respect to quantities such as deadline-miss probabilities, consecutive deadline-miss probabilities and tardiness constraints. When considering the scheduling and analysis of probabilistic conditional parallel DAG tasks, not only inter-task, but also intra-task interference, and multiprocessor scheduling anomaly effects must be considered, complicating the analyses for the above mentioned quantities.

4.4.1 TASK AND PROBLEM MODEL

A probabilistic conditional directed-acyclic graph (pC-DAG) is defined by the tuple $G = (V, E)$. The vertices $V = V_\Diamond \cup V_\circ$ are comprised of finitely many regular

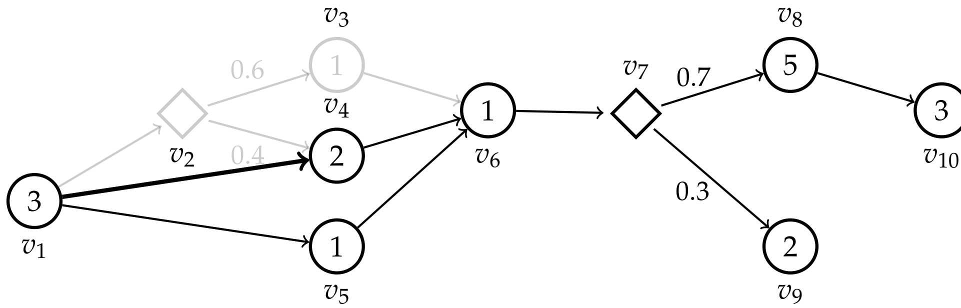

Figure 4.3: An exemplary probabilistic conditional DAG task in which each conditional vertex (diamond) denotes that only one of its adjacent subjobs is released (with the annotated probability) during runtime. In this specific example four different DAG structures can be instanced during runtime.

vertices $V_\circ$ and finitely many decision vertices $V_\Diamond$; an exemplary probabilistic conditional DAG is illustrated in Figure 4.2. The edges $E \subseteq V \times V$ denote the directed precedence constraints between two vertices such that there are no directed cycles in the graph. The main difference to a regular DAG is given by the partitioning of vertices into decision and regular vertices, which are defined as follows.

Decision Vertex. Each *decision vertex* $v_j \in V_\Diamond$ denotes a branching decision to exactly one (of possibly many) succeeding *conditional branches* and satisfies the following properties:

decision vertex

conditional branch

- To be well-defined, a decision vertex must have exactly one preceding regular vertex and more than one succeeding regular vertex, since the branching condition is determined by the preceding vertex during execution.
- A decision vertex is not executable, but rather represents a branching option to one of the conditional paths, which is determined by the computation of the preceding regular vertex.
- Each decision vertex $v_j \in V_\Diamond$ is a random event with the sample space $\Omega_j = \{(v_j, v_k) \mid (v_j, v_k) \in E\}$; that is determined by the set of all edges that originate in decision vertex v_j and end in a regular successor vertex v_k, which represents the source vertex of a conditional branch.
- At each decision vertex, exactly one conditional branch must be taken, i.e., for each $v_j \in V_\Diamond : \sum_{\omega \in \Omega_j} \mathbb{P}(\omega) = 1$.

In the following, we assume that the probability for each outcome $\mathbb{P}(\omega)$ for $\omega \in \Omega_j$ is given by either application knowledge or measurements.

Regular Vertex. In contrast to a decision vertex, a *regular vertex* $v_i \in V_\circ$ denotes an executable entity, namely a subtask, which is annotated with its worst-case execution time $vol(v_i)$. That means, no subjob executes for more than $vol(v_i)$ time units.

regular vertex

Source Vertex. All vertices in V of a pC-DAG G, which do not have any predecessor, are marked as *source vertices* of that graph.

In order to assure a well-defined runtime behaviour of our pC-DAG model, we first formalize a *branching decision* and then define how a branching decision

branching decision

determines the runtime behaviour.

Definition 4.9 (Branching Decisions)**.** *Let $G = (V_\Diamond \cup V_\circ, E)$ denote a pC-DAG then*

$$B_j := \{(v_i, v_j, v_k) \mid (v_i, v_j) \wedge (v_j, v_k) \in E\} \quad (4.1)$$

denotes the set of all possible branching decisions at a decision vertex $v_j \in V_\Diamond$. Furthermore, an element $b_{j,k} \in B_j$ denotes the branching decision to vertex v_k at decision vertex v_j, i.e., $b_{j,k} = (v_i, v_j, v_k) \in B_j$, if the two edges (v_i, v_j) and (v_j, v_k) exist.

residue pC-DAG

Each branching decision results in an altered *residue pC-DAG*, which is formalized as follows.

Definition 4.10 (Substitution)**.** *A substitution of a concrete branching decision $b_{j,k} = (v_i, v_j, v_k)$ for $b_{j,k} \in B_j$ is a transformation of a pC-DAG $G = (V, E)$ to another pC-DAG $G' = (V', E')$ denoted by $G \xrightarrow{b_{j,k}} G'$ where G' is constructed by the application of the following rules:*

1. *Determine source vertices of G.*
2. *Remove decision vertex v_j from the graph G.*
3. *Remove all edges $(*, v_j), (v_j, *)$ from the graph G.*
4. *Add an edge (v_i, v_k) to the graph G.*
5. *Remove all vertices and edges that are not reachable from any* source *vertex in G.*

A sequence of substitutions is considered ***complete*** *if the final pC-DAG G' does not contain any decision vertices anymore, i.e., the result of a complete substitution is a simple DAG.*

An example of a substitution is given based on the illustrated pC-DAG in Figure 4.2 at decision vertex v_2 such that $B_2 = \{b_{2,3} = (v_1, v_2, v_3), b_{2,4} = (v_1, v_2, v_4)\}$; without loss of generality, we choose $b_{2,4}$ to be the taken branching decision. By application of rule 1, we determine v_1 as the only source vertex in G; By application of rule 2-4, v_2 is removed from V, edges $(v_1, v_2), (v_2, v_3)$, and (v_2, v_4) are removed from E and edge (v_1, v_4) is added to E; By application of rule 5, v_3 is removed from V, since v_3 is not reachable by the source vertex v_1. The resulting pC-DAG G'; after the substitution; is shown in Figure 4.3.

We consider a set $\mathbb{T}$ of sporadic constrained-deadline ($D_i \leq T_i$) pC-DAG tasks in a multiprocessor system of M identical (homogeneous) processors. Recall, that a job is said to meet its deadline if it finishes at most D_i time units after its release, and said to miss its deadline otherwise.

Definition 4.11 (pC-DAG Job)**.** *The ℓ-th job of a pC-DAG task $\tau_i = (G_i, D_i, T_i)$ is denoted by $J_{i,\ell} = (G_{i,\ell}, d_{i,\ell}, a_{i,\ell})$ where $G_{i,\ell}$ denotes a complete substitution of the pC-DAG G_i, and $a_{i,\ell}$ and $d_{i,\ell} = a_{i,\ell} + D_i$ denote the arrival time and absolute deadline of the job $J_{i,\ell}$. The concrete complete substitution $G_{i,\ell}$ of all possible complete substitutions, is based on the taken branching decisions during the execution of the ℓ-th job. We emphasize, that the graph $G_{i,\ell}$ of pC-DAG job is a simple DAG, since no decision vertices are contained anymore in a complete substitution. For that reason we synonymously use the term DAG job, to refer to the concrete complete substitution $G_{i,\ell}$ of the ℓ-th job.*

Table 4.1: Tabular representation of the probabilities of the parameters total volume and length for the probabilistic conditional DAG task illustrated in Figure 4.2.

Probability	Length	Total Volume
0.42	13	14
0.18	7	8
0.28	14	15
0.12	8	9

The probability for a specific complete substitution during the runtime of a job is described in the following definition.

Definition 4.12 (Valid complete Branching Sequence). *A sequence of branching decisions $(b_{j_1,k_1}, \dots, b_{j_n,k_n})$ is a valid complete branching sequence of the pC-DAG $G := G^{(0)}$ if the following conditions are satisfied:*

- $v_{j_{z+1}} \in V_\diamond^{(z)}$ *of* $G^{(z)}$, *where* $G^{(z-1)} \xrightarrow{b_{j_z,k_z}} G^{(z)}$ *for* $z \in \{1, \dots, n-1\}$
- $G^{(n)}$ *is a simple DAG, i.e.,* $V_\diamond^{(n)} = \emptyset$

invalid- & valid complete branching decisions

This distinction between *invalid* and *valid complete branching decisions* is necessary, since some decision vertices may be removed during a substitution of another decision vertex, which implies a zero-probability for such a runtime evolution. During the runtime of the ℓ-th job, at most $|B_{j_1}| \cdot |B_{j_2}| \cdot \ldots \cdot |B_{j_n}|$ many different complete substitutions (simple DAGs) can be instanced (if no decision vertex is ruled out by another decision vertex).

Definition 4.13 (DAG Job Probability). *Let $b_{j_1,k_1} = (v_{i_1}, v_{j_1}, v_{k_1}), b_{j_2,k_2}, \dots, b_{j_n,k_n}$ denote a valid complete branching sequence of a pC-DAG G such that $G^{(n)}$ is an ordinary DAG. The probability that a DAG job evolves into $G^{(n)}$ during execution is given by $\mathbb{P}(G^{(n)}) = \mathbb{P}((v_{j_1}, v_{k_1})) \cdot \mathbb{P}((v_{j_2}, v_{k_2})) \cdot \ldots \cdot \mathbb{P}((v_{j_n}, v_{k_n}))$.*

For each pC-DAG task, a joint cumulative distribution function for the DAG-dependent parameters *volume* and *length* as defined in Definition 4.1 and Definition 4.2 is inferred. This is done by the enumeration of all valid branching decisions, which yield a complete substitution $G^{(n)}$ for some $n \in \{1, \dots, |V_\diamond|\}$. The corresponding volume C and length L are then collected to derive the cumulative distribution function. For instance, the joint distribution function of the pC-DAG illustrated in Figure 4.2, is given by the calculation of the probability for each of the valid complete branching decisions and its respective parameter values as summarized in Table 4.1. More precisely, the instance illustrated in Figure 4.4 represents the case where both upper edges are chosen for which the probability is $0.7 \cdot 0.6 = 0.42$. The associated length is 13 and the associated volume is 14. By similar reasoning, choosing the edges with probability $0.7 \cdot 0.4$, $0.3 \cdot 0.6$, and $0.3 \cdot 0.4$ yield 0.28, 0.18 or 0.12 realization probability of the associated DAGs. Consequently, we derive the joint cumulative distribution function for the pC-DAG G, i.e., $\mathbb{P}(C \leq u, L \leq v)$ as follows:

$$\begin{aligned} &\mathbb{1}(u-14) \cdot \mathbb{1}(v-13) \cdot 0.42 + \mathbb{1}(u-8) \cdot \mathbb{1}(v-7) \cdot 0.18 \\ &+ \mathbb{1}(u-15) \cdot \mathbb{1}(v-14) \cdot 0.28 + \mathbb{1}(u-9) \cdot \mathbb{1}(v-8) \cdot 0.12 \end{aligned} \tag{4.2}$$

where $\mathbb{1}$ is 1 if $x \geq 0$ and 0 otherwise.

We note that the analyses in the forthcoming sections are not limited by our model, and can be applied to (possibly) other pC-DAG task models or probabilistic DAG models, as long as the joint distribution function of the pC-DAG task regarding the volume and length can be derived.

4.4.2 PROBABILISTIC DEADLINE MISS DESCRIPTION

In the instances that a job misses its deadline, a mechanism must be devised, which decides the actions taken upon deadline-miss events

In the instances that a job misses its deadline, a mechanism must be devised, which decides the actions taken upon deadline-miss events. A common mechanism is the immediate abortion of every job, which exceeds its deadline in order to avoid any interference of subsequent jobs. From the practical side, this approach is inefficient in the sense that all computation results and state changes are discarded, and may even incur additional costs to revoke state changes for consistency reasons. This approach is especially inefficient if the required amount of execution time to finish the remaining workload is rather small and a late result is still useful. For that reason, it is beneficial to let jobs execute, and potentially finish their remaining workload before a final abortion. On the flip side, this additional workload directly impacts the response-time of subsequent jobs and thus increases the chance of subsequent deadline misses.

it is beneficial to let jobs execute, and potentially finish their remaining workload before a final abortion

tardy

Formally, a job is *tardy* if it is not finished at, or before, its absolute deadline. To avoid unbounded interference of subsequent jobs, whilst allowing a tardy job to likely finish, we aim to track and bound the amount of workload, which tardy jobs can carry into the scheduling window of the next job.

Definition 4.14 (Tardy Workload)**.** *Let $\delta_i(\ell)$ denote the cumulative tardy workload of the first ℓ instances (jobs) of a constrained-deadline pC-DAG task, sampled at the absolute deadline of the ℓ-th job. Let $G_{i,\ell}$ denote the corresponding complete substitution then the tardy workload of the first ℓ jobs can be stated as:*

$$\delta_i(\ell) = \max\{\delta_i(\ell-1) + vol(G_{i,\ell}) - work_i(S, d_{i,\ell-1}, d_{i,\ell}), 0\} \quad \text{for } \ell > 1 \tag{4.3}$$

where $work_i(S, d_{i,\ell-1}, d_{i,\ell})$ denotes the amount of executed workload of task τ_i during the window from the absolute deadline of the $(\ell-1)$-th job and the absolute deadline of the ℓ-th job in a given schedule S. The tardy workload of the first job is given by

$$\delta_i(1) = \max\{vol(G_{i,1}) - work_i(S, t_{i,1}, d_{i,1}), 0\}$$

since by definition the system starts with the release of the first job.

In summary, the ℓ-th complete substitution, i.e., DAG $G_{i,\ell}$ of pC-DAG task τ_i meets its deadline if $\delta_i(\ell) = 0$ and misses its deadline if $\delta_i(\ell) > 0$. In pursuance of improving the problem of unbounded interference, we introduce a threshold value $\rho_i > 0$ for the admissible amount of tardy workload of each job of a task τ_i. In the following we consider schedules S' instead of S where the tardy workload is aborted each time it exceeds the threshold value. This provides the property that in S' the upper bound $\delta_i(\ell) \leq \rho_i$ is valid for all tasks τ_i and all $\ell \in \mathbb{N}$.

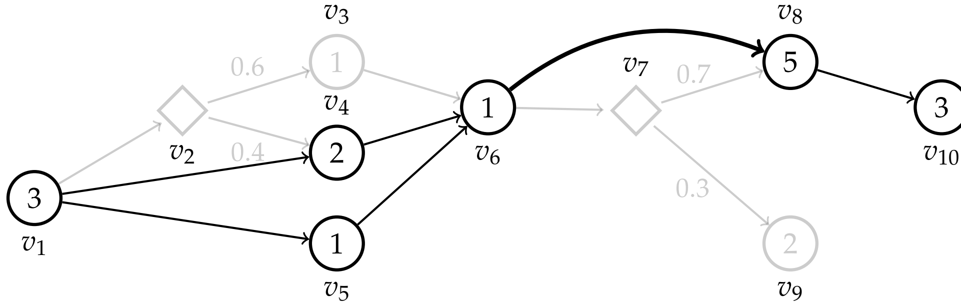

Figure 4.4: A complete substitution of the pC-DAG shown in Figure 4.2 where the valid branching decisions $b_{2,4}$ and $b_{7,8}$ are chosen. The probability for this complete substitution G' is given by $0.4 \cdot 0.7 = 0.282$ with $C = 15$ and $L = 14$.

To clarify the definitions, consider two subsequent job releases with bound $\rho_i = 2$ at time 0 and 10 with their respective absolute deadlines at 8 and 18 and 100% processor share. Further assume that the execution time demand of the first job is 10 and 6 for the second. Assuming that at the time of the job's deadline, 8 units of work are finished, 2 units of work are carried into the execution window (10 to 18) of the second job. Since the bound allows 2 units of workload of a prior job to be executed, the first job is not aborted and finishes at time 12. The tardiness of that job is thus given by $12 - 8 = 4$. The second job then executes from 12 to 18 and in this case meets its deadline.

Definition 4.15 (Consecutive Deadline Misses)**.** *Any sequence of k consecutive jobs $J_{i,\ell}, J_{i,\ell+1}, \ldots, J_{i,\ell+k-1}$ for $\ell > 0$ is subject to k consecutive deadline misses if the following conditions hold:*

- *All jobs in the sequence miss their deadline*
- *Either $\ell = 1$ or the previous job $J_{i,\ell-1}$ does not miss its deadline.*

For each task we define a function $\theta_i : \mathbb{N} \rightarrow [0, 1]$ to specify that we tolerate k consecutive deadline misses for a given probability of at most $\theta_i(k)$. Based on the prior definitions, we formally state a probabilistic k consecutive deadline miss constraint as follows.

Definition 4.16 (k Consecutive Deadline Miss Constraint)**.** *Let*

$$\phi_i(\ell, k) := \mathbb{P}(\delta_i(\ell) > 0, \ldots, \delta_i(\ell + k - 1) > 0 \mid \ell = 1 \text{ or } \delta_i(\ell - 1) = 0) \tag{4.4}$$

denote the probability that the sequence of k consecutive jobs $J_{i,\ell}, J_{i,\ell+1}, \ldots, J_{i,\ell+k-1}$ suffers from k consecutive deadline misses. Then a probabilistic conditional DAG task τ_i is said to satisfy the constraint $\theta_i(k)$ if

$$\sup_{\ell \geq 1} \{\phi_i(\ell, k)\} \leq \theta_i(k), \tag{4.5}$$

i.e., for any initial ℓ the probability $\phi_i(\ell, k)$ does not exceed the threshold $\theta_i(k)$.

Table 4.2: Summary of used notation in this section.

Symbol	Meaning
$V_\Diamond$	Set of decision vertices in V
$V_\circ$	Set of regular vertices in V
B_j	$\{(v_i, v_j, v_k) \mid (v_i, v_j), (v_j, v_k) \in E, v_j \in V_\Diamond, v_i, v_k \in V_\circ\}$
$b_{j,k}$	Choosing decision $(v_i, v_j, v_k) \in B_j$
$\mathbb{P}(\omega)$	Probability of an event $\omega \in \Omega$
$\mathbb{T}$	Taskset of pC-DAGs
$\mathbb{K}$	Set of reservations
C_i	Total volume of a DAG G_i
$\delta_i(\ell)$	Bounded tardy workload of the ℓ-th job
L_i	Longest path of DAG G_i
κ_i	Reservation system for task τ_i
Ω_j	Sample space of a decision node v_j
$v_i \prec v_j$	v_i precedes v_j
$vol(v_i)$	Worst-case execution time of regular subjob v_i
E_i	Reservation budget
m_i	Number of in-parallel reservations
Φ_i^n	$\mathbb{R}_{>0} \to \mathbb{R}_{>0},\ E_i \mapsto \mathbb{P}(R_i^1 > D_i)_{\mid m_i = n}$

4.4.3 Objective, Discussion of Guarantees & Limitations

The objective of the forthcoming analyses is the design of a hierarchical scheduling policy such that the probability of k consecutive deadline misses of a pC-DAG is no more than a user specified upper bound given the requirements described in Section 4.4.4 are satisfied. While the hierarchical scheduling approach allows to detect and contain temporal violations such as consecutive deadline misses and thus allows for the initiation of error handling, we do not devise any specific error handling mechanisms in this contribution. Also we do not guarantee any quality of service beyond the probability bound of occurrence given by $\theta_i(k)$ in this work.

In our approach, the ℓ-th job of a pC-DAG task τ_i is guaranteed to meet its deadline if the bounded tardy workload $\delta_i(\ell) = 0$ and misses its deadline if $\delta_i(\ell) > 0$. All tardy jobs are counted for the consecutive deadline misses, irrespective of whether the tardy job can finish the execution within its tardy workload bound or not. In the case that a deadline is missed, it is allowed to continue the execution of the tardy workload until the tardy workload bound ρ_i is exceeded, which then results in the abortion of the ℓ-th job. Our approach does not guarantee if a job, that is not finished at its deadline, can finish within its tardy workload bound ρ_i or is aborted before completion.

The tardiness of aborted jobs is undefined since their executions are incomplete and thus no tardiness bound can be stated for them. Nonetheless, in the case that a tardy job can finish its workload within its tardy workload bound and is thus

not aborted, a probabilistic description of the tardiness and a maximal tardiness can be stated (Corollary 4.6), which is helpful in the assessment of resulting performance degradation from the application perspective.

4.4.4 HIERARCHICAL SCHEDULING ANALYSIS

In this section we propose a hierarchical scheduling policy to schedule the instances of probabilistic pC-DAG tasks such that response-times can be analyzed without considering inter-task interference and thus only the random process of an individual pC-DAG task must be considered. In our parallel reservation system, each pC-DAG task τ_i is serviced individually according to a reservation system, which satisfies the properties stated in Section 4.3 with the following refinements, which are required due to the fact that multiple DAG jobs can be backlogged.

1. **FIFO.** The DAG jobs of a pC-DAG task are assigned to the provided services in first-in-first-out (FIFO)-manner. Furthermore, we assume that at each time all assigned reservations only serve the subjobs of a single DAG job by the FIFO-policy. We emphasize that a subjob may migrate from one reservation to another, i.e., the service provided by the reservation system is a global scheduling scheme.
2. **Activation of Service.** The first release of the parallel reservation system must be synchronous with the release of the first DAG job. All subsequent reservation system releases are periodic in case of pending workload and synchronous with the next DAG job otherwise.

With respect to the property of *sustained service*, in this section, we focus on *spinning parallel reservation systems* defined as follows.

spinning parallel reservation systems

Definition 4.17 (Spinning Reservation System)**.** *A* reservation system *consists of $m_i \in \mathbb{N}$ reservation servers that provide E_i (equal) amount of service each and that is replenished every $P_i > 0$ time units and provides it service. More specifically, to provide the service, a multiset of $m_i \in \mathbb{N}$ distinct reservations are activated, in which each of them guarantees a service of E_i time units over an interval of length P_i. Whenever a reservation is scheduled during an interval, the corresponding interval is either providing service or is spinning if nothing is waiting to be serviced.*

The execution time budget E_i of the reservation systems are determined off-line and are static. Moreover, the deadline-compliant schedulability of the reservation system must be verified beforehand such that service guarantees can be verified. Thus for the remainder of this section, we assume the existence of a feasible schedule S upon a set of identical multiprocessors, meaning that all reservations satisfy the stated properties. Based on the promised service, we then build our probabilistic analysis.

Definition 4.18 (Work)**.** *Let $work_i(S, t_1, t_2)$ denote the amount of workload generated from DAG jobs of task τ_i that was* executed *(processed) during the time interval from t_1 to t_2 in a given schedule S.*

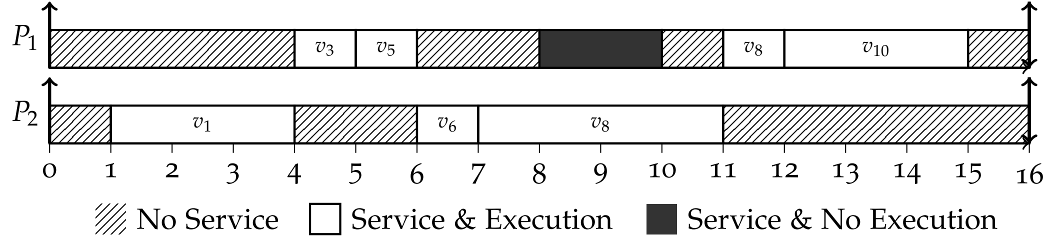

Figure 4.5: An exemplary schedule of the pC-DAG illustrated in Figure 4.2 where the decisions are taken according to Figure 4.4. The resulting DAG job with its subjobs (vertices) are executed on 2-in parallel spinning reservations with budget $E_i = 8$ each. One spinning reservation is partitioned to processor P_1 and the other is partitioned to processor P_2 where they are scheduled according to some scheduling policy. The dashed areas represents that the reservations are preempted by higher-priority reservations or are depleted and therefore do not provide service. The unused time interval from 8 to 10 in the first reservation indicates that the reservation provides the service but the DAG is not executed in the reservation due to either its precedence constraints or lack of workload.

Based on this definition, the worst-case response-time of the ℓ-th job $J_{i,\ell}$ of a pC-DAG task τ_i that arrives at $a_{i,\ell}$ is given by the smallest $t' \geq a_{i,\ell}$ such that

$$work_i(S, a_{i,\ell}, t') \geq vol(G_{i,\ell}) + backlog_i(S, a_{i,\ell}), \tag{4.6}$$

where $backlog_i(S, a_{i,\ell})$ is the amount of unfinished workload at time $a_{i,\ell}$ of jobs of τ_i released before $a_{i,\ell}$. Note that $backlog_i(S, a_{i,\ell}) = 0$ if there are no previous deadline misses since we assume constrained deadlines, i.e., $D_i \leq T_i$ in our system model. In the following we express the processed work in terms of provided service and develop a response-time bound in Theorem 4.3.

Definition 4.19 (Service). *Let $serv_i(S, t_1, t_2)$ denote the amount of service that is promised to the DAG jobs from τ_i during the time interval from t_1 to t_2 in the schedule S.*

Please note that $work_i(S, t_1, t_2) \leq serv_i(S, t_1, t_2)$ since *promised* service of the reservation system does not equate to *consumed* service for serving τ_i.

Based on the definition of an envelope (cf. Definition 4.8), we are able to formally state the following lemma.

Lemma 4.1. *Given a schedule S of the taskset $\mathbb{T}$. We consider a task $\tau_i \in \mathbb{T}$ with an m_i-in-parallel reservation system. Let a DAG job of τ_i with envelope $[s_{k_1}, f_{k_1}), \ldots, [s_{k_p}, f_{k_p})$. Then the amount of work that is finished during the interval from $f_{k_{q-1}}$ to f_{k_q} for $q \in \{2, \ldots, p\}$ is lower bounded by*

$$work_i(S, f_{k_{q-1}}, f_{k_q}) \geq serv_i(S, f_{k_{q-1}}, s_{k_q}) + serv_i(S, s_{k_q}, f_{k_q}) - (m_i - 1) \cdot c_{k_q} \tag{4.7}$$

where v_{k_q} is the subjob from the envelope starting at time s_{k_q} and finishing at f_{k_q}.

Proof. In the proof we split the work at time s_{k_q} and upper-bound each partial summand of $work_i(S, f_{k_{q-1}}, f_{k_q})$, i.e.,

$$work_i(S, f_{k_{q-1}}, s_{k_q}) + work_i(S, s_{k_q}, f_{k_q})$$

individually. In the first step, we prove that between finish and start of two consecutive subjobs in the envelope, the provided service is fully utilized by the DAG instance, i.e.,

$$work_i(S, f_{k_{q-1}}, s_{k_q}) = serv_i(S, f_{k_{q-1}}, s_{k_q})$$

holds for all $q \in \{2, \ldots, p\}$. Given the work-conserving properties of the algorithm used to dispatch subjobs to the provided service, we know that an eligible subjob is scheduled whenever service is available. Since by definition s_{k_q} is the earliest time that v_{k_q} is able to execute and $f_{k_{q-1}}$ is the earliest time that v_{k_q} is eligible to be executed, all services during $f_{k_{q-1}}$ to s_{k_q} must have been used to execute other (non-envelope path) subjobs.

Secondly, we show that $work_i(S, s_{k_q}, f_{k_q})$, that is from start to finish of a subjob in the envelope, can be upper-bounded by $\max\{serv_i(S, s_{k_q}, f_{k_q}) - (m_i - 1) \cdot c_{k_q}, c_{k_q}\}$.

Notice that due to the sequential execution of the subjob v_{k_q}, there can only be unused service during the execution of v_{k_q}. Moreover, during the execution of v_{k_q}, at most $m_i - 1$ reservations may be unused. Since the maximal duration for execution of v_{k_q} is c_{k_q}, we conclude

$$work_i(S, s_{k_q}, f_{k_q}) \geq serv_i(S, s_{k_q}, f_{k_q}) - (m_i - 1) \cdot c_{k_q}$$

□

Based on the following extension of the above lemma, we can calculate the response-time of a DAG job.

Lemma 4.2. *Under the same conditions stated in Lemma 4.1, it follows that for all* $0 \leq t \leq f_{k_p}$ *the following property holds:*

$$work_i(S, t_G, t_G + t) \geq serv_i(S, t_G, t_G + t) - (m_i - 1) \cdot L_i \tag{4.8}$$

where t_G *denotes the release time of a DAG job of* τ_i *with DAG structure* G.

Proof. The main part to prove this lemma is already done in Lemma 4.1. It is however left to consider the case that t is not a time instant of the envelope.

Similar to the proof of Lemma 4.1 we can show that $work_i(S, f_{k_{q-1}}, t) = serv_i(S, f_{k_{q-1}}, t)$ for all $t \in [f_{k_{q-1}}, s_{k_q}]$ and that $work_i(S, s_{k_q}, t) \geq serv_i(S, s_{k_q}, t) - (m_i - 1) \cdot c_{k_q}$ for all $t \in [s_{k_q}, f_{k_q}]$. Furthermore, by the same reasoning $work_i(S, t_G, t) = serv_i(S, t_G, t)$ holds for all $t \in [t_G, s_{k_1}]$.

We obtain the desired result by splitting the interval $[t_G, t]$ into parts already described above and estimating all of them at the same time. To formalize this, we define $\mu := (t_G, s_{k_1}, f_{k_1}, \ldots, s_{k_p}, f_{k_p})$. For $q \in \{1, \ldots, 2p + 1\}$ we denote by $\mu(q)$ the q-th entry of μ and by $\mu^t(q). = \min\{\mu(q), t\}$ the q-th entry bounded by t. A decomposition of $work_i(S, t_G, t_G + t)$ using the partition μ yields

$$\sum_{q=1}^{p} work_i(S, \mu^t(2q - 1), \mu^t(2q)) + \sum_{q=1}^{p} work_i(S, \mu^t(2q), \mu^t(2q + 1)) \tag{4.9}$$

For the first summand, we know that the following inequality holds

$$\sum_{q=1}^{p} work_i(S, \mu^t(2q-1), \mu^t(2q)) \geq \sum_{q=1}^{p} serv_i(S, \mu^t(2q-1), \mu^t(2q)) \tag{4.10}$$

by the fact that the *work* is lower bounded by the *service*. The second summand from above is lower bounded by the following condition

$$\sum_{q=1}^{p} work_i(S, \mu^t(2q), \mu^t(2q+1)) \geq \sum_{q=1}^{p} \left(serv_i(S, \mu^t(2q), \mu^t(2q+1)) - (m_i - 1) \cdot c_{k_q} \right) \tag{4.11}$$

Combining both of the inequalities yields the lower bound

$$\sum_{q=1}^{2p} serv_i(S, \mu^t(q), \mu^t(q+1)) - (m_i - 1) \sum_{q=1}^{p} c_{k_q} \tag{4.12}$$

$$= serv_i(S, t_G, t_G + t) - (m_i - 1) \sum_{q=1}^{p} c_{k_q} \geq serv_i(S, t_G, t_G + t) - (m_i - 1) \cdot L_i$$

which concludes the proof. □

Definition 4.20 (Supply Bound Function)**.** *For a task $\tau_i \in \mathbb{T}$ the minimal service that is provided by the associated parallel reservation system during an interval of length $t \geq 0$ is described by the supply bound function $sbf_i(t)$.*

The supply bound function is a lower bound for the service provided to a DAG task for all feasible schedules S, i.e., $\forall S : serv_i(S, t_G, t_G + t) \geq sbf_i(t)$. This leads us to the following theorem.

Theorem 4.3 (Response-Time Bound)**.** *Consider a task $\tau_i \in \mathbb{T}$ and assume that the reservation system of τ_i is m_i-in-parallel and its minimal service is described by $sbf_i(t)$. Let G be the DAG job of $J_{i,j}$ of a pC-DAG task τ_i. Then the response-time of $J_{i,j}$ is upper-bounded by $\min\{t > 0 \mid sbf_i(t) \geq W_i^G\}$ where $W_i^G := C_i + (m_i - 1) \cdot L_i + backlog_i(S, t_G)$ for notational brevity.*

Proof. Let $t' := \min\{t > 0 \mid sbf_i(t) \geq W_i^G\}$. We prove the theorem by contraposition. Assume that t' does not bound the response-time then $t' < f_{k_p}$, where f_{k_p} is the last entry in the envelope of G in S. In this case ($t' < f_{k_p}$) Lemma 4.2 applies and yields:

$$work_i(S, t_G, t_G + t') \geq serv_i(S, t_G, t_G + t') - (m_i - 1) \cdot L_i \tag{4.13}$$

$$\geq sbf_i(t') - (m_i - 1) \cdot L_i \tag{4.14}$$

By the definition of t' we have $sbf_i(t') \geq C_i + (m_i - 1) \cdot L_i + backlog_i(S, t_G)$. Hence, $work_i(S, t_G, t_G + t') \geq C_i + backlog_i(S, t_G)$, which implies that the job is finished at time t', i.e., $t' \geq f_{k_p}$ contradicting the assumption. □

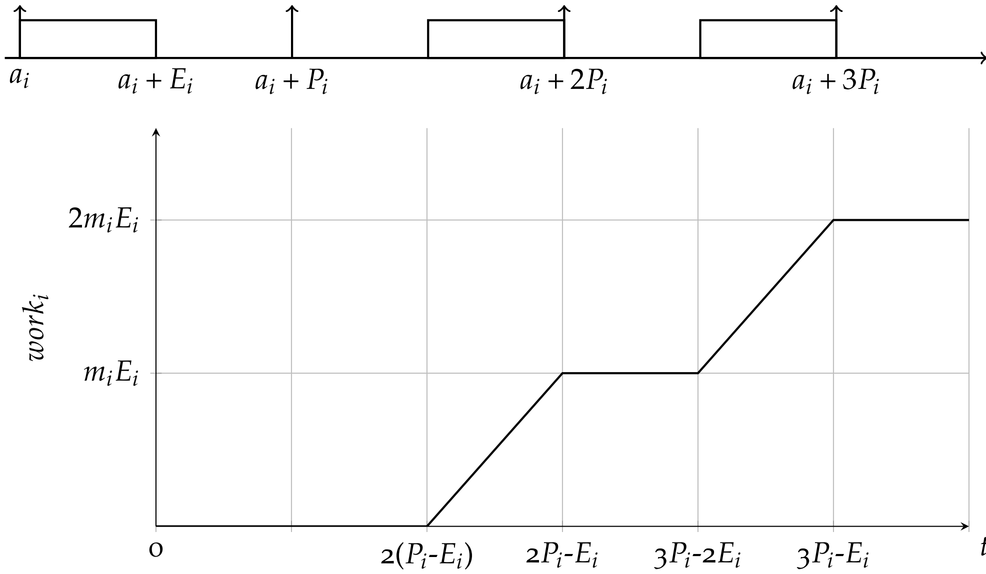

Figure 4.6: Supply Bound Function $sbf(t)$ of the a reservation system with m_i in-parallel service and E_i reservation budget each. The supply-bound function denotes the minimal guaranteed service in any interval of length t for any feasible schedule.

We emphasize that the reservations and respective supply-bound function are not enforced to any specific function. The complexity of the calculation of the response-time depends only on the supply bound function. For instance, Figure 4.6 illustrates the supply bound function of *our spinning reservation system* from Definition 4.17. As depicted, there may be no service provided to the task for up to $2 \cdot (P_i - E_i)$ time units in the worst case.

We note that the first activation of the reservation system serving task τ_i can either occur synchronous with the release of the first job of τ_i or periodically in an asynchronous manner. In the first case, the reservation system can stop assigning new reservation servers if there is no pending or unfinished job of τ_i, as long as it starts assigning new reservations if new jobs arise in the ready queue.

the first activation of the reservation system serving task τ_i can either occur synchronous with the release of the first job of τ_i or periodically in an asynchronous manner

If we assume a reservation server as in Definition 4.17 then the response-time or service-time of a DAG job G is described by the following theorem. Before we present the formal proof we give an intuition. The worst-case service provided by the reservations is depicted in Figure 4.6. In the figure every time when service is provided, it is done on m_i resources simultaneously. Hence, the total time which τ_i has to be served, until the pending workload W_i^G is finished is given by $\frac{W_i^G}{m_i}$. This takes up to $\left\lceil \frac{W_i^G}{m_i \cdot E_i} \right\rceil + 1$ service cycles. To account for the time that the service cycles do not provide any service, we have to multiply the number of service cycles with the maximal amount of time that τ_i is not served during a service cycle, i.e., $(P_i - E_i)$. In total, the response-time is upper bounded by Eq. (4.15).

Theorem 4.4 (Service Time)**.** *Let G denote the DAG job of a pC-DAG task τ_i under analysis. We assume that for τ_i we have a reservation system as in Definition 4.17 with*

m_i equal sized in-parallel services $E_i \leq P_i$. We can give an upper bound R_G on the response-time of the job of τ_i by

$$R_G = \left(\left\lceil \frac{W_i^G}{m_i \cdot E_i} \right\rceil + 1 \right) (P_i - E_i) + \frac{W_i^G}{m_i} \tag{4.15}$$

where $W_i^G := C_i + (m_i - 1)L_i + backlog_i(S, t_G)$ for notational brevity.

Proof. For the proof we deliberately assume that $C_i > 0$ since otherwise no work has to be done and $R_G = 0$ is already a trivial response-time bound. Using Theorem 4.3, we have to find the minimal $t > 0$ such that $sbf_i(t) = W_i^G$. Let $sbf_i(t)$ denote the worst-case service (supply bound function) as illustrated in Figure 4.6. Let the function $g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ with

$$g(t) := \left(\left\lceil \frac{t}{m_i E_i} \right\rceil + 1 \right) (P_i - E_i) + \frac{t}{m_i} \tag{4.16}$$

such that the composition $sbf \circ g$ is the identity and the function g yields the minimal value of the inverse image of $sbf_i(t)$, i.e., $g(t) = \min\{sbf_i^{-1}(t)\}$ holds for all $t > 0$. Hence if $g(t)$ satisfies the properties, we obtain $g(W_i^G) = \min\{t > 0 \mid sbf_i(t) \geq W_i^G\}$. We can verify that $g(t)$ satisfies the required properties by inspection of the supply bound function. □

In general, if we know an upper bound b on the backlog of the previous job, we can state the response-time bound from Eq. (4.15) independent from the previous schedule, by

$$R'_G(b) = \left(\left\lceil \frac{V_i^G(b)}{m_i E_i} \right\rceil + 1 \right) (P_i - E_i) + \frac{V_i^G(b)}{m_i} \tag{4.17}$$

where $V_i^G(b) := C_i + (m_i - 1) \cdot L_i + b$. Based on Eq. (4.17), we bound the response-time for the case that the preceding job has a deadline miss and for the case that the preceding job has *no* deadline miss.

Corollary 4.5. *Under the assumptions of Theorem 4.4, $R'_G(\rho_i)$ is an upper bound on the response-time of G if the preceding job has a deadline miss, and $R'_G(0)$ is an upper bound if the preceding job has no deadline miss.*

Proof. This follows directly from Theorem 4.4 by using either $backlog_i(S, t_G) \leq \rho_i$ (in case of a deadline miss) or $backlog_i(S, t_G) = 0$ (in case of *no* deadline miss). □

Based on these results, a bounded tardiness (upper bound on the finishing time minus the absolute deadline of every job of a task) can be stated as follows:

Corollary 4.6 (Bounded Tardiness)**.** *The tardiness for jobs of pC-DAG task τ_i that are not aborted, i.e., can finish all tardy workload before exceeding the tardy workload bound ρ_i is at most $\max\{R'_G(\rho_i) - D_i, 0\}$ if*

$$R'_G(\rho_i) - D_i \leq P_i + \left(\left\lfloor \frac{\rho_i}{E_i} \right\rfloor + 1 \right) \cdot (P_i - E_i) + \rho_i$$

Proof. Since any finishing time of any job can be no more than an upper bound for the response-time, by Corollary 4.5, we know that the tardiness is bounded by $\max\{R'_G(\rho_i) - D_i, 0\}$. We, however, have to validate the required assumptions for the tardiness bound and to whether a job is aborted or not.

Since the amount of executed tardy workload is bounded by ρ_i before the abortion of that job, the finishing time is bounded as follows: Let $D_i \leq T_i$ and t_i denote the release of the job that has tardy workload at time $t_i + D_i$. By our definition of reservations, the budget is replenished periodically for every P_i time units if there is unfinished pending workload.

Although the tardy workload is agnostic of the internal structure of the workload, the reservation system guarantees to provide the minimum progress. Suppose that $t_i + D_i + \lambda$ is the time for the next replenishment of the reservation system after $t_i + D_i$ with $0 \leq \lambda \leq P_i$. Starting from $t_i + D_i + \lambda$, at least E_i amount of computation can be served for a period of P_i. In consequence, the servicing of the tardy job is aborted no earlier than

$$
\begin{aligned}
& t_i + D_i + \lambda + \left\lfloor \frac{\rho_i}{E_i} \right\rfloor \cdot P_i + \left(P_i - E_i + \rho_i - \left\lfloor \frac{\rho_i}{E_i} \right\rfloor \cdot E_i \right) \\
& \leq t_i + D_i + P_i + \left(\left\lfloor \frac{\rho_i}{E_i} \right\rfloor + 1 \right) \cdot (P_i - E_i) + \rho_i
\end{aligned}
$$

□

4.4.5 RESERVATION ANALYSIS AND OPTIMIZATION

In this section we devise the analysis and optimization algorithm to generate reservation systems, which provably respect the upper bounds for k consecutive deadlines misses in a probabilistic sense. We emphasize that, in order to co-design the k consecutive deadline-miss constraints with the reservations configurations, time-efficient algorithms are required to calculate the probabilities for k consecutive deadline misses for any given reservation configuration.

4.4.5.1 *Analysis of Reservation Systems*

Based on the finite sample space of DAG jobs (complete substitutions) G of the pC-DAG tasks τ_i we define the random variables $R_i^1 := (G \mapsto R'_G(\rho_i))$ and $R_i^0 := (G \mapsto R'_G(0))$, which yield for each DAG job the response-time bounds from Corollary 4.5 with and without a previous deadline miss. According to Definition 4.16, the constraint for k consecutive deadline misses is fulfilled if $\sup_{\ell \geq 1} \phi_i(\ell, k) \leq \theta_i(k)$, is satisfied where $\phi_i(\ell, k)$ is given by

$$\mathbb{P}(\delta_i(\ell) > 0, \ldots, \delta_i(\ell + k - 1) > 0 \mid \ell = 1 \text{ or } \delta_i(\ell - 1) = 0)$$

Using Bayes' Theorem, we can formulate $\phi_i(\ell, k)$ as

$$\mathbb{P}\left(\delta_i(\ell + k - 1) > 0 \mid \delta_i(\ell + k - 2) > 0, \ldots, \delta_i(\ell) > 0\right) \cdot \phi_i(\ell, k - 1) \tag{4.18}$$

The probability that the $(\ell + k - 1)$-th job does not meet its deadline, does not decrease if the amount of tardy workload of the preceding job is increased. Therefore,

if $\delta_i(\ell+k-2) = \rho_i$, then the probability for a deadline miss of $J_{i,\ell+k-1}$ is maximal. In this case, the amount of tardy workload of the other jobs $\delta_i(\ell+k-3), \ldots, \delta_i(\ell)$ is irrelevant for the amount of tardy workload of $J_{i,\ell+k-1}$. More specifically,

$$\begin{aligned} & \mathbb{P}\left(\delta_i(\ell+k-1) > 0 \mid \delta_i(\ell+k-2) > 0, \ldots, \delta_i(\ell) > 0\right) \\ \leq \; & \mathbb{P}\left(\delta_i(\ell+k-1) > 0 \mid \delta_i(\ell+k-2) = \rho_i\right) \end{aligned}$$

holds and we can thus upper bound the probability for k consecutive deadline misses $\phi_i(\ell, k)$ by

$$\phi_i(\ell, k) \leq \mathbb{P}\left(\delta_i(\ell+k-1) > 0 \mid \delta_i(\ell+k-2) = \rho_i\right) \cdot \phi_i(\ell, k-1). \tag{4.19}$$

Then by Corollary 4.5 we know that

$$\mathbb{P}\left(\delta_i(\ell+k-1) > 0 \mid \delta_i(\ell+k-2) = \rho_i\right) \leq \mathbb{P}\left(R_i^1 > D_i\right)$$

and for the probability of the first job (without previous deadline miss, i.e., $\delta_i(\ell-1) = 0$ or $\ell = 1$)

$$\phi_i(\ell, 1) = \mathbb{P}\left(\delta_i(\ell) > 0\right) \leq \mathbb{P}\left(R_i^0 > D_i\right).$$

Combining the above results yields a bound on the probability of k consecutive deadline misses:

$$\begin{aligned} \phi_i(\ell, k) &\leq \mathbb{P}\left(R_i^1 > D_i\right) \cdot \phi_i(\ell, k-1) \\ &\leq \mathbb{P}\left(R_i^1 > D_i\right)^2 \cdot \phi_i(\ell, k-2) \\ &\ldots \\ &\leq \mathbb{P}\left(R_i^1 > D_i\right)^{k-1} \cdot \phi_i(\ell, 1) \\ &\leq \mathbb{P}\left(R_i^1 > D_i\right)^{k-1} \cdot \mathbb{P}\left(R_i^0 > D_i\right) \end{aligned}$$

Since $\mathbb{P}\left(R_i^0 > D_i\right) \leq \mathbb{P}\left(R_i^1 > D_i\right)$, we also derive a simplified bound for the probability of k consecutive deadline misses of task τ_i by

$$\phi_i(\ell, k) \leq \mathbb{P}\left(R_i^1 > D_i\right)^k. \tag{4.20}$$

Since the bound does not depend on a specific ℓ we know that

$$\sup_{\ell \geq 1} \phi_i(\ell, k) \leq \mathbb{P}\left(R_i^1 > D_i\right)^{k-1} \cdot \mathbb{P}\left(R_i^0 > D_i\right).$$

to derive upper bounds on response-times for queuing systems it must be shown that the system is stable

As a prerequisite to derive upper bounds on response-times for queuing systems it must be shown that the system is stable. Informally speaking this means that the maximal time between two met deadlines is finite. We first give a formal definition of stability and then show that our devised reservation-based queuing system is stable by construction.

Definition 4.21 (Stability). *A spinning reservation system κ_i is considered stable if for all $\ell \geq 0$ with $\delta_i(\ell) = 0$ it is almost certain that there exists $k > 0$ such that $\delta_i(k+\ell) = 0$. More formally, the probability for k consecutive deadline misses approaches 0 for $k \to \infty$, i.e.,*

$$\lim_{k \to \infty} \sup_{\ell \geq 1} \phi(\ell, k) = 0 \tag{4.21}$$

Theorem 4.7 (Stability). *A reservation system κ_i is stable if $\mathbb{P}(R_i^1 > D_i) < 1$.*

Proof. The probability for k consecutive deadline misses is bounded by

$$\sup_{\ell \geq 1} \phi_i(\ell, k) \leq \mathbb{P}\left(R_i^1 > D_i\right)^k$$

according to Eq. (4.20). If $\left(R_i^1 > D_i\right) < 1$, then $\mathbb{P}\left(R_i^1 > D_i\right)^k \to 0$ for $k \to \infty$. This concludes the theorem. □

In consequence we do not have to especially consider stability concerns in the design of the reservation systems other than k consecutive deadline constraints.

4.4.5.2 *Distribution Function Calculation*

In this section, we show how to practically calculate the upper bounds on the response-time.

First, we define the as per pC-DAG task $\tau_i \in \mathbb{T}$ and the associated reservation system $\kappa_i \in \mathbb{K}$ auxiliary random variable X_i as

$$X_i := \frac{C_i + (m_i - 1) \cdot L_i + \rho_i}{m_i \cdot E_i} = \frac{V_i^G}{m_i E_i} \tag{4.22}$$

where C_i and L_i are random variables for which the joint distribution function can be computed from the pC-DAG task model, as previously described in Section 4.4.1. As a consequence, the distribution function $\mathbb{P}(X_i \leq u)$ can be directly computed.

C_i and L_i are random variables for which the joint distribution function can be computed from the pC-DAG task model

With reference to Corollary 4.5, the distribution function of R_i^1 can be written as follows:

$$\mathbb{P}(R_i^1 \leq u) = \mathbb{P}\left((P_i - E_i) \cdot (\lceil X_i \rceil + 1) + E_i \cdot X_i \leq u\right) \tag{4.23}$$

Our objective is to compute the cumulative distribution function of R_i^1 based on the cumulative distribution function of X_i. To that end, let $dom(X_i) \in \mathbb{R}$ denote all values that X_i can take. Then, we define the set of constant values

$$I_i := \{\ell \in \mathbb{N} \mid \lfloor \inf(dom(X_i)) \rfloor \leq \ell \leq \lceil \sup(dom(X_i)) \rceil\} \tag{4.24}$$

Moreover, for notational brevity let $\psi(X_i) := (P_i - E_i) \cdot (\lceil X_i \rceil + 1) + E_i \cdot X_i$ then by the fact that $\lceil X_i \rceil \mapsto \ell + 1$ for every $X_i \in (\ell, \ell + 1]$, the domain of $\psi(X_i)$ can be partitioned into sub-domains as follows:

$$\bigcup_{\ell \in I_i} \{(P_i - E_i) \cdot (\ell + 2) + E_i \cdot X_i, \ell < X_i \leq \ell + 1\}$$

Therefore, for a given $\ell < X_i \leq \ell + 1$, the condition $(P_i - E_i) \cdot (\lceil X_i \rceil + 1) + E_i \cdot X_i \leq u$ can be rearranged to $(P_i - E_i) \cdot (\ell + 2) + E_i \cdot X_i \leq u$, which results in

$$X_i \leq \frac{u - (P_i - E_i) \cdot (\ell + 2)}{E_i} \tag{4.25}$$

By the σ-additivity property of distribution functions, algebraic rearrangements yield the following result:

$$\mathbb{P}(R_i^1 \leq u) = \mathbb{P}\left((P_i - E_i) \cdot (\lceil X_i \rceil + 1) + E_i \cdot X_i \leq u\right) \tag{4.26}$$

$$= \sum_{\ell \in I_i} \mathbb{P}\left(X_i \leq \frac{u - (P_i - E_i) \cdot (\ell + 2)}{E_i},\ \ell < X_i \leq \ell + 1\right) \tag{4.27}$$

4.4.5.3 *Optimization of Reservation Systems*

Algorithm 3 Calculation of Reservation Systems

Require: $\mathbb{T} := \{\tau_1, \ldots, \tau_n\}$, $\theta_1(k_1), \theta_2(k_2), \ldots, \theta_n(k_n)$, $m_1^*, m_2^*, \ldots, m_n^*$;
Ensure: Reservation systems, $\mathbb{K} := \{\kappa_1, \ldots, \kappa_n\}$, which satisfy $\theta_i(k_i)$;

```
1: Initialize reservations 𝕂 ← ∅;
2: for each task τ_i for i in {1, ..., n} do
3:     for m_i in {1, 2, ..., m_i^*} do
4:         E_i ← min{E_i | (Φ_i^n)^{k_i} ≤ θ_i(k_i)};
5:         if E_i could not be found then
6:             continue;
7:         else
8:             𝕂 ← 𝕂 ∪ κ_i := (m_i, E_i, D_i, T_i);
9:             break;
   return 𝕂;
```

In this section, we present Algorithm 3 to calculate spinning reservation systems for the scheduling of probabilistic constrained-deadline conditional DAG tasks. Under the consideration of probabilities of, upper bounds for the maximal number k_i of tolerable consecutive deadline misses, and given tardy workload bounds; the objective, is to find minimal numbers of in-parallel reservations m_i and, associated minimal amounts of service time E_i. For each probabilistic constrained-deadline conditional DAG task the algorithm determines all feasible configurations (m_i, E_i) by iterating through the number of in-parallel reservations $m_i \in [1, m_i^*]$ and search for the smallest required reservation service to still comply with the consecutive deadline-miss constraints.

Theorem 4.8 (Monotonicity)**.** *The functions*

$$\Phi_i^n : \mathbb{R}_{>0} \to \mathbb{R}_{>0},\ E_i \mapsto \mathbb{P}(R_i^1 > D_i)_{|m_i = n} \tag{4.28}$$

which yield the probabilities of an upper bound of a deadline-miss, for a fixed number $m_i \in \mathbb{N}$ of reservations with respect to the service time E_i, are monotonically decreasing.

Proof. For easier readability let

$$Y_i := \frac{C_i + (m_i - 1) \cdot L_i + \rho_i}{m_i} = X_i \cdot E_i \tag{4.29}$$

for which the distribution function is independent of E_i for every fixed m_i. Note that X_i is the auxiliary variable from Eq. (4.22). According to the definition of $\mathbb{P}(R_i^1 > D_i)$ in the beginning of this section, we have to prove that

$$\mathbb{P}\left(\left(\left\lceil \frac{Y_i}{E_i} \right\rceil + 1\right) \cdot (P_i - E_i) + Y_i > D_i\right) \tag{4.30}$$

$$\geq \mathbb{P}\left(\left(\left\lceil \frac{Y_i}{E_i + \epsilon} \right\rceil + 1\right)(P_i - (E_i + \epsilon)) + Y_i > D_i\right) \tag{4.31}$$

for any positive arbitrary increment $\epsilon \geq 0$ and any realizations of $Y_i \geq 0$. Let an arbitrary realization $Y_i \geq 0$ satisfy

$$\left(\left\lceil \frac{Y_i}{E_i + \epsilon} \right\rceil + 1\right) \cdot (P_i - (E_i + \epsilon)) + Y_i > D_i \tag{4.32}$$

In this case Y_i satisfies

$$\left(\left\lceil \frac{Y_i}{E_i} \right\rceil + 1\right) \cdot (P_i - E_i) + Y_i > D_i \tag{4.33}$$

as well which yields the assumption by the property of distribution functions. □

Due to the monotonicity of the functions Φ_i^n as shown in Theorem 4.8, it is possible to find the minimal amount of reservation service to guarantee compliance with the consecutive deadline-miss constraints by using binary search in the interval $(0, D_i]$. We emphasize that m_i^* is an upper bound specified by the user, which can be set to an arbitrary fixed number, or determined as the point where an increase in the number of in-parallel reservations does not yield a *significant* decrease in the amount of required service to satisfy the deadline-miss probability constraints.

it is possible to find the minimal amount of reservation service to guarantee compliance with the consecutive deadline-miss constraints by using binary search in the interval $(0, D_i]$

4.4.6 PARAMETER SPACE EXPLORATION

Owing to the absence of published research results, which we can directly compare to, we instead opt to explore the design space with respect to the minimal reservation service compared to the associated number of in-parallel reservations for different bounds of deadline-miss probabilities, and tardy workload bounds, as well as measures of the resource usage improvements, compared to hard real-time reservation based DAG scheduling as proposed in [UBC+18].

In the first part of this section, we explore the configurations of reservation systems, i.e., the number of in-parallel reservations m_i and the reservation budgets E_i, which comply with:

we explore the configurations of reservation systems

- upper bounds for a single deadline miss probability $\theta_i(1)$
- and given tardy workload bounds ρ_i

Note that we investigate single deadline miss probabilities, since consecutive deadline misses follow immediately by exponentiation.

Choosing reservation budgets close to the replenish period may impair the schedulability of the whole task set, since processors may not be shared by more

than one task or scheduled with best-effort tasks. On the other hand, decreasing the reservation budgets increases the amount tardy workload of jobs that could not finish within their provisioned service.

A key practical design question is, whether there is a maximal number of in-parallel reservations such that the minimal required reservation budget does not decrease anymore

Theorem 4.8 proves that there is a minimal reservation budget for a given number of in-parallel reservations m_i, which satisfies a given upper bound of deadline-miss probabilities. A key practical design question is, whether there is a maximal number of in-parallel reservations such that the minimal required reservation budget does not decrease anymore. Since we are not able to provide analytic solutions to the above question, we instead opt to design numerical studies and showcase the behaviour for different scenarios.

4.4.6.1 *Synthetic pC-DAG Distribution Function Generation*

We generate discrete joint cumulative distribution functions, which are used to represent the resulting pC-DAG jobs as described in Section 4.4.1 in a computationally efficient way. To do so, we generate a set of 100 simple DAGs, representing DAG jobs (complete substitutions during runtime), and assign probabilities to each of these 100 DAGs; using the UUniFast algorithm [BB05], such that the probabilities add up to 100%. Each of the 100 DAGs is generated with 50 to 100 (drawn uniformly at random) regular vertices and each regular vertex is assigned an integral worst-case execution time between 1 to 100 (drawn uniformly at random) time units. Despite not generating explicit decision vertices, the 100 DAGs, used to represent the different complete substitutions, roughly correspond to pC-DAGs with 4 decision vertices with 3 branching decisions each, or 6 decision vertices with 2 branching decisions each. After the generation of all regular vertices (subjobs) and their corresponding worst-case execution times, the precedence constraints are generated. To that end, we generate an upper tri-diagonal adjacency matrix, where a non-zero entry $e_{i,j}$ in the matrix, denotes an edge from v_i to v_j. For each of the 100 DAGs, the entry is non-zero with a probability from 30% to 70% (drawn uniformly at random) in the case of *high inherent parallelism pC-DAGs* and 70% to 90% in the case of *low inherent parallelism pC-DAGs*, to qualitatively influence the generation of DAGs with varying degrees of length to volume ratios. Based on the 100 DAGs and the corresponding internal structures, the volume and length parameters are computed. Based on these, the joint cumulative density function $\mathbb{P}_i(u, v)$ is calculated.

We restrict ourselves to implicit-deadlines, i.e., $D_i = T_i$ where T_i is drawn uniformly at random from the range of 105% to 120% of the longest critical path of all 100 DAGs, such that each task is not unschedulable (in hard real-time sense) by construction.

4.4.6.2 *Discussions and Optimizations of Reservation Parameters*

We constrain our parameter space to those tuples $(E_i(m_i), m_i)$, which comply with upper bounds for a single deadline miss probability $\theta_i(1) \leq 5\%, 10\%, 20\%$. Since $\theta_i(k_i)$ decreases for $k_i > 1$, we only search the parameter space for a single deadline miss instead of consecutive deadline misses.

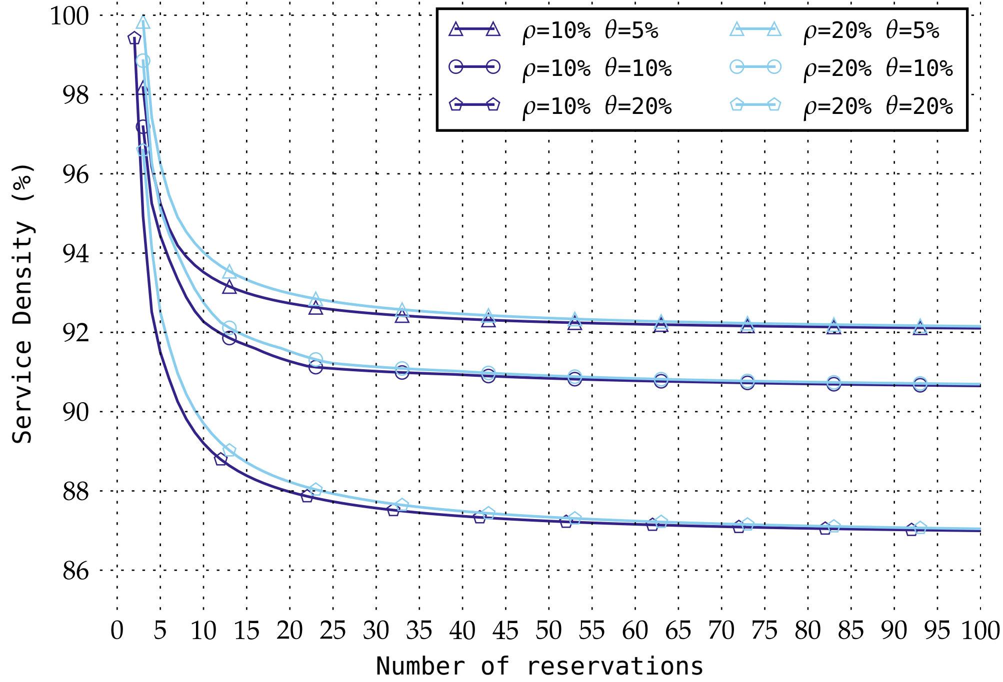

Figure 4.7: Service density (E_i/D_i) with respect to the associated number of in-parallel reservations of equal probability upper bounds for a single deadline-miss of 5%, 10%, 20%. Furthermore, the tardy workload bound ρ_i is set to 10% and 20% of the task's deadline. The expected contribution of the critical-path length to the overall volume is 19%.

Moreover, we explore the tardy workload bounds ρ_i to 10% and 20% relative to the task's deadline, i.e., $0.1 \cdot D_i$, $0.2 \cdot D_i$, to investigate the influence of the admissible tardy workload on the reservation system configurations. In our numerical study, we define the minimal **service density** as follows:

$$S_i(m_i) := \frac{E_i(m_i)}{D_i} \cdot 100\% \tag{4.34}$$

In our showcased scenarios, we set the replenishment period of the reservation servers P_i to the task's period T_i to demonstrate a rather pessimistic case. For instance, when we let E_i be equal to $0.5P_i$ then this yields a lower-bound of 50% service density in our implicit-deadline case.

we only focus on two extreme cases of pC-DAG tasks, namely a task with very high and a task with very low inherent parallelism

In our explorations, we only focus on two extreme cases of pC-DAG tasks, namely a task with very high and a task with very low inherent parallelism. More precisely, tasks where the expected volume is much larger than the expected length of the critical path, i.e., $\mathbb{E}\{C_i\} \gg \mathbb{E}\{L_i\}$, are considered *highly parallel*. Conversely, tasks where the contribution of the expected length of the critical path to the expected volume is large, i.e., $\mathbb{E}\{C_i\} \approx \mathbb{E}\{L_i\}$ are considered *highly sequential*. In both demonstrated extreme cases, we change the number of in-parallel reservations m_i in the range from 1 to 100 and plot the corresponding service density $S_i(m_i)$.

with increasing admissible deadline-miss probability, the resource usage in terms of service density and number of in-parallel reservations decreases

High Parallelism. Figure 4.7 shows the results for the *highly parallel* task where the critical path length constitutes 19% of the overall workload. In general it can be seen that with increasing admissible deadline-miss probability, the resource usage in terms of service density and number of in-parallel reservations decreases. However, for all deadline-miss probability and tardy workload constraints the resource usages converge. Additionally, it can be seen that with increasing tardy workload bounds, the resource usages increase if the deadline-miss probabilities are to be met.

the decrease in resource usage is smaller than in the case of the highly parallel task

Low Parallelism. In contrast, Figure 4.8 shows the results for the *highly sequential* task where the critical path length constitutes 56% of the overall workload. It can be seen that the decrease in resource usage is smaller than in the case of the highly parallel task, which is due to the fact that increasing parallel reservations is less beneficial due to the dominant sequential execution requirement of the task. Nonetheless, with increasing admissible deadline-miss probability, the resource usage in terms of service density and number of in-parallel reservations decreases as in the case of the highly parallel task.

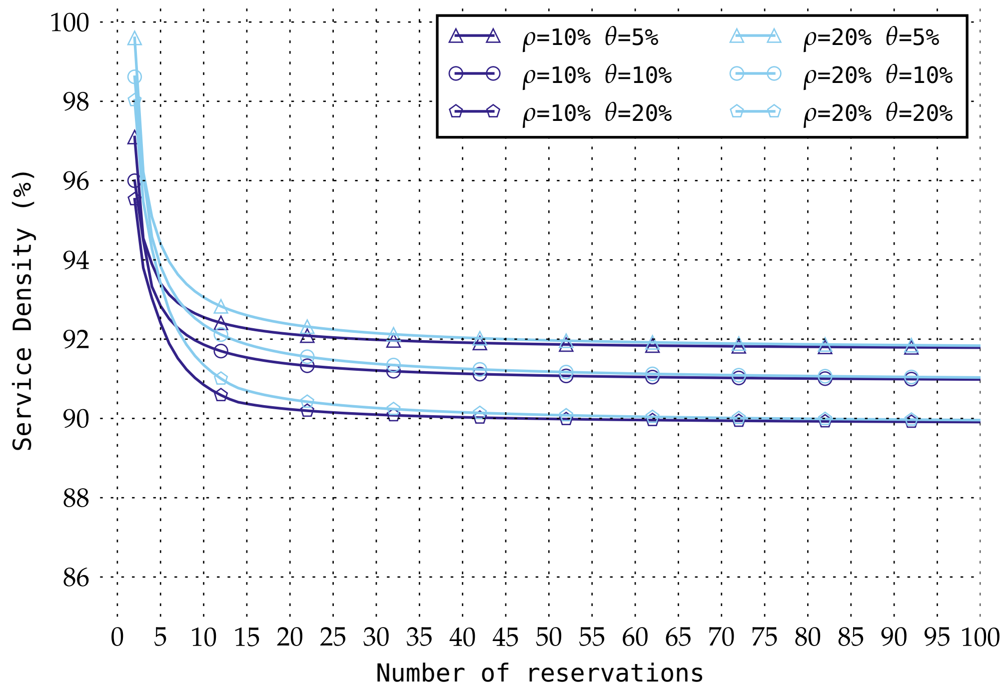

Figure 4.8: Service density (E_i/D_i) with respect to the associated number of in-parallel reservations of equal probability upper bounds for a single deadline-miss of 5%, 10%, 20%. Furthermore, the tardy workload bound ρ_i is set to 10% and 20% of the task's deadline. The expected contribution of the critical-path length to the overall volume is 56%.

4.4.6.3 *Resource Savings*

To assess the resource savings of our proposed approach, we generate 100 joint cumulative distribution functions that represent the probabilistic characteristics

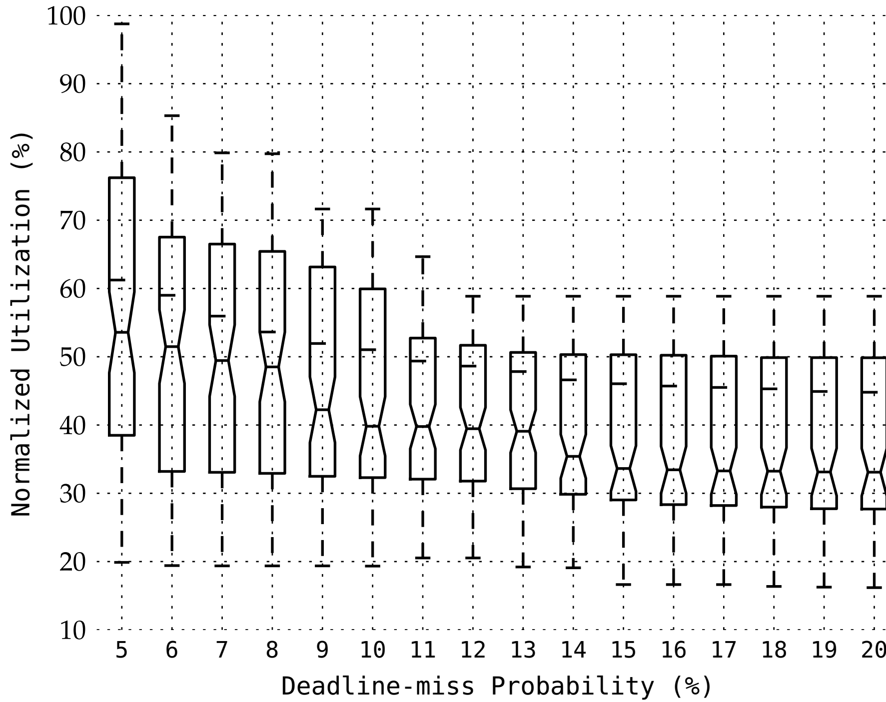

Figure 4.9: Required reservation resources compared for synthetically generated pC-DAGs for each deadline-miss probability $5\%, 6\%, \cdots, 20\%$ compared to a hard real-time reservation system.

of all possible complete pC-DAG substitutions of each of the 100 generated pC-DAGs. For each pC-DAG, 50 DAGs are generated each of which consists of $10 - 100$ (uniformly sampled) subjobs, which have an integer worst-case execution time of $1 - 100$ (uniformly sampled). The edge probabilities are chosen between 0.3 to 0.9, resulting in DAGs with varying degrees of parallelism, i.e., length to volume ratios, in the range of 40% to 70%. Additionally, the tardy workload bound is set to 0. Please note that we are unable to report the number of decision vertices in our experiments, since we generated a collection of 50 DAGs to derive a joint cumulative distribution function. For each deadline-miss probability $5\%, 6\%, \ldots, 20\%$ the minimal required reservation resources $m_i \cdot E_i$ are compared to an equivalent hard-real time DAG reservation as proposed in [UBC+18]. We report the required $m_i \cdot E_i$ for the probabilistic case in this paper divided by the required $m_i \cdot E_i$ for the deterministic case in [UBC+18], denoted as *Normalized Utilization*, using the box plot in Figure 4.9. It can be observed that the normalized required resources, decrease with increasing deadline-miss probabilities.

All demonstrated cases show that by relaxing the strict deadline constraints to probabilistic upper bounds, it is possible to substantially decrease the required resource usage compared to hard real-time equivalent solutions.

by relaxing the strict deadline constraints to probabilistic upper bounds, it is possible to substantially decrease the required resource usage

4.5 PARALLEL PATH PROGRESSION SCHEDULING

Paradoxically, the large number of processors of modern multiprocessor architectures is detrimental to the resource efficiency of hard real-time systems if the increased number of processors does not benefit the worst-case response-time analysis.

In this contribution, we consider the progress of multiple complete paths in a DAG instead of only the envelope in the response-time analysis. In general, intra-task interference analyses build upon the interference analysis along the execution of the envelope (also known as critical path or key path). The number of parallel paths, which we will analyze, can amount to the degree of maximum parallelism, e.g., the processor or reservation count. More precisely, inversely to prior approaches, we do not track the execution progress by the analysis of envelope path interference, but track analyzable simultaneous progress of a collection of *many parallel complete paths* alongside the envelope path using intra-task prioritization. By virtue of this approach, we only have to account for the interference of subjobs that do not belong to any of these parallel paths for a response-time bound. We extend the *parallel path progression* concepts and corresponding response-time analyses to hierarchical scheduling to significantly improve the worst-case response-time for high-parallel DAG use cases. Moreover, a stricter *path monotonic progression* property is proposed, which allows to construct self-suspension aware reservation systems, which can improve resource efficiency.

4.5.0.1 *Parallel Path Progression Concepts*

In this section, we examine the required properties to achieve parallel path progression on M processors, dedicated to execute a single job of a DAG task. By that, we avoid any inter-task interference and solely focus on intra-task interference.

Definition 4.22 (n-Path Collection)**.** *Let a DAG $G = (V, E)$ then the enumeration of all possible complete paths is denoted as*

$$\Psi(G) := \{\pi \mid \pi \text{ is a complete path according to Definition 4.3 in } G\}$$

Any subset of paths $\psi \in \mathcal{P}(\Psi(G))$ from the powerset of $\Psi(G)$ is called a path collection. *Further, a path collection $\psi \in \mathcal{P}(\Psi(G))$ is called an n-*path collection *if $|\psi| = n$, i.e., a collection of n-many complete paths.*

maximal number of paths, which can be executed in parallel, is limited by the number of processors M

For notational convenience, in the remainder of this chapter, we will use π_* to denote the longest path in G, i.e., $vol(\pi_*) \geq vol(\pi)$ for all $\pi \in \Psi(G)$. It is a fact that the maximal number of paths, which can be executed in parallel, is limited by the number of processors M. Therefore, we constrain our solution space to n-path collections, where $n \in \{1, \ldots, M\}$. Based on a concrete n-path collection ψ, the set of subtasks, which belong to at least one of the paths in ψ is defined by $V_s(\psi) := \pi_{\psi_1} \cup \ldots \cup \pi_{\psi_n}$ for each $\pi_{\psi_1}, \ldots, \pi_{\psi_n} \in \psi$. Please note that we use the subscripts to index the paths belonging to the path collection. Conversely, the

complement set of subtasks, which do not belong to any of the selected paths, is denoted by $V_s^c(\psi) := \{v \in V \mid v \notin V_s(\psi)\}$.

We propose a *parallel path progression prioritization*, which assigns each subtask a priority based on the membership of the above sets, which is formalized in the following definition. Later in this section, we explain how this prioritization can be used to better analyze the self-interference by explicitly considering the parallel execution of paths in ψ in the response-time analysis.

parallel path progression prioritization

Definition 4.23 (Parallel Path Progression Prioritization)**.** *Let $V_s(\psi)$ denote the set of subtasks from an n-path collection ψ of a DAG $G = (V, E)$. A fixed-priority policy for all subtasks $v \in V$ is a* parallel path progression prioritization *if and only if $\Pi(v_i) < \Pi(v_k)$ for any two $v_i \in V_s(\psi)$ and $v_k \in V_s^c(\psi)$, where $\Pi(v_i)$ denotes the priority of subtask v_i.*

Note that in our notation for the priorities, a higher value implies a higher priority, i.e., $\Pi(v_i) > \Pi(v_k)$ implies that v_i has a higher priority than v_k. A sufficient policy to satisfy the parallel path progression prioritization property is to only use two distinct priority-levels.

We clarify the introduced notation and definitions collectively in the following example. The path enumeration $\Psi(G)$ of the DAG illustrated in Figure 4.1 consists of six paths $\{\pi_1, \pi_2, \ldots, \pi_6\}$. The individual paths are: $\pi_1 := \langle v_1, v_2, v_3\rangle$, $\pi_2 := \langle v_1, v_4, v_5, v_9\rangle$, $\pi_3 := \langle v_1, v_4, v_5, v_6\rangle$, $\pi_4 := \langle v_1, v_7, v_5, v_9\rangle$, $\pi_5 := \langle v_1, v_7, v_5, v_6\rangle$, and $\pi_6 := \langle v_1, v_7, v_8\rangle$.

A 2-path collection ψ from the powerset $\mathcal{P}(\Psi(G))$ is for instance given by $\psi := \{\pi_2, \pi_3\}$. Subsequently, $V_s(\psi) = \pi_2 \cup \pi_3 = \{v_1, v_4, v_5, v_6, v_9\}$ and $V_s^c(\psi) := \{v_2, v_3, v_7, v_8\}$. If for instance all subjobs $v_i \in V_s(\psi)$ are assigned priority $\Pi(v_i) = 1$ and conversely all subjobs $v_i \in V_s^c(\psi)$ are assigned priority $\Pi(v_i) = 2$, then this prioritization is a valid *parallel path progression prioritization*.

4.5.0.2 *Parallel Path Progression Scheduling*

In this section, we look at a single DAG job that is scheduled on M dedicated processors by a work-conserving preemptive subtask-level fixed-priority list scheduling algorithm in conjunction with the parallel path progression prioritization. We elaborate how this prioritization aids the analysis of the parallel progression of a path collection, and in consequence the response-time analysis of the DAG job.

Definition 4.24 (List-FP)**.** *In a* preemptive list-FP schedule *on M dedicated processors, a task instance (job) of a DAG task $G = (V, E)$ with a fixed-priority assignment of each subjob $v \in V$ is scheduled according to the following rules:*

- *A subjob arrives to the ready list if all preceding subjobs have executed until completion, i.e., the subjob arrival time a_i for each subjob v_i is given by* $\max\{f_j \mid v_j \in pred(v_i)\}$. *An arrived but not yet finished subjob is considered pending.*
- *At any time t, the M highest-priority pending subjobs are executed on the M processors and a lower-priority subjob is preempted if necessary.*

Please note that the only difference from Definition 4.7 is that M dedicated processors are considered here. The extension to hierarchical scheduling is explained later on and for now we analyze the response-time of a single DAG job using all the previously introduced properties. Let a subtask-level fixed-priority list schedule S on M dedicated processors be generated for a single job J where all subjobs are prioritized according to the rule described in Definition 4.23.

For the response-time analysis, we analyze the time interval $[a_J, f_J)$ between the arrival time and finishing time of J, by interval partitioning into busy and non-busy times. Any point in time $t \in [a_J, f_J)$ is called *busy* if an envelope subjob is executed in S at time t. Conversely, any point in time $t \in [a_J, f_J)$ is called *non-busy* if the envelope subjob is not executed. By the construction of the envelope according to Definition 4.8, it must be that at any point in time $t \in [a_J, f_J)$ an envelope subjob is pending, i.e., has arrived and not yet finished. In conjunction with the simple fact that t can be exclusively either *busy* or *non-busy* we know that the response-time of DAG job J is given by the cumulative amount of time spent in either of these two states.

Any point in time $t \in [a_J, f_J)$ is called busy *if an envelope subjob is executed in S at time t. Conversely, any point in time $t \in [a_J, f_J)$ is called* non-busy *if the envelope subjob is not executed*

In contrast to prior work, we further partition the envelope intervals $[a_{k_i}, f_{k_i})$ depending of whether the corresponding subjob v_{k_i} belongs to the set of the chosen n-path collection, i.e., $v_{k_i} \in V_s(\psi)$ or not, assuming an envelope path $\pi_e := \langle v_{k_1}, v_{k_2}, \ldots, v_{k_p} \rangle$ in S. The intuition of this approach is to tie the execution of an envelope subjob to the execution of subjobs of the path collection ψ, which is used and explained in the forthcoming analyses.

The intuition of this approach is to tie the execution of an envelope subjob to the execution of subjobs of the path collection ψ

Theorem 4.9 (Preemptive Response-Time Bound)**.** *The response-time of a DAG job J with an arbitrary n-path collection $\psi = \{\pi_{\psi_1}, \ldots, \pi_{\psi_n}\} \in \mathcal{P}(\Psi(G))$ (of n at most M) that is scheduled on M dedicated homogeneous processors using preemptive List-FP scheduling is bounded from above by*

$$R_J \leq vol(\pi_*) + \frac{vol(V_s^c(\psi))}{M - n + 1} \tag{4.35}$$

Proof. By the definition of the envelope (cf. Definition 4.8), we know that the interval $[a_J, f_J)$ in a concrete preemptive *List-FP schedule S*, can be partitioned into contiguous intervals $[a_{k_1}, f_{k_1} = a_{k_2}), \ldots, [f_{k_{n-1}} = a_{k_n}, f_{k_n})$, where $[a_{k_i}, f_{k_i})$ denotes the arrival and finishing time of subjob v_{k_i} for all $i \in \{1, \ldots, p\}$ in the envelope path $\pi_e := \langle v_{k_1}, v_{k_2}, \ldots, v_{k_p} \rangle$.

Busy Time. Considering each envelope subjob interval $[a_{k_i}, f_{k_i})$ individually for $i \in \{1, \ldots, p\}$, the amount of *busy* time is given by the execution time of v_{k_i}, which is by definition no more than $vol(v_{k_i})$. The cumulative amount of *busy* time in $[a_J, f_J)$ can be obtained by adding up the interval's individual *busy* times resulting in $vol(\pi_e)$, which is no more than the longest path $vol(\pi_*)$.

Non-Busy Time. Since our scheduling policy is work-conserving we have that whenever an envelope subjob v_{k_i} is not executing during $[a_{k_i}, f_{k_i})$ then all M processors must be busy executing non-envelope subjobs. Since v_{k_i} can be exclusively either in $V_s(\psi)$ or $V_s^c(\psi)$, we analyze the set

$$\{t \in [a_J, f_J) \mid \text{envelope subjob is not executing}\} \cap [a_{k_i}, f_{k_i})$$

for both cases individually:

- Let $v_{k_i} \in V_s(\psi)$ and by assumption not execute at time t then at most $n-1$ processors execute subjobs from $V_s(\psi)$. That is because all subjobs in $V_s(\psi)$ stem from n different paths, which implies that there can never be more than n-many subjobs from $V_s(\psi)$ pending concurrently, in general. Moreover, since by assumption $v_{k_i} \in V_s(\psi)$ and is not executing at t, at most $n-1$ subjobs from $V_s(\psi)$ are pending. Conversely, we know that at least $M-(n-1)$ processors execute subjobs from $V_s^c(\psi)$, since otherwise v_{k_i} would be executed contradicting the case assumption.
- In the other case, let $v_{k_i} \in V_s^c(\psi)$ and by assumption not be executing at time t then it must be that no processor is executing any subjob from $V_s(\psi)$. That is because if any of the lower-priority subjobs in $V_s(\psi)$ would be executing, then the higher-priority envelope subjob $v_{k_i} \in V_s^c(\psi)$ would be executing as well, which contradicts the case assumption. Conversely, we know that all M processors are exclusively used to execute subjobs from $V_s^c(\psi)$.

In summary, we have that during all *non-busy times* $t \in [a_J, f_J)$, we have that at least $M-(n-1)$ processors execute subjobs from $V_s^c(\psi)$. The total volume of subjobs from $V_s^c(\psi)$ during $[a_J, f_J)$ is at most $vol(V_s^c(\psi))$. The maximal cumulative amount of *non-busy times* is achieved by evenly distributing the workload and is thus no more than $vol(V_s^c(\psi))/(M-(n-1))$. □

Sustainability of our response-time analysis. Many multiprocessor hard real-time scheduling algorithms and schedulability analyses presented in the literature are not sustainable, which means that they suffer from timing anomalies. These anomalies describe the counter-intuitive phenomena that a job that was verified to always meet its deadline can miss its deadline by augmenting resources, e.g., to execute the job on more processors or to decrease the execution-time (early completion). In Corollary 4.10, we show that our response-time bound is sustainable with respect to the number of processors and the subjob execution-time. This is a beneficial property in dynamic environments, where available processors and execution times vary, and ultimately simplifies implementation efforts and improves robustness in real systems.

Corollary 4.10 (Sustainability)**.** *The response-time bounds in Theorem 4.9 holds true for a DAG job with $G = (V, E)$ even if any subjob $v \in V$ completes before its worst-case execution time or if the number of processors is increased.*

Proof. This comes directly from the observation that the volume of the envelope path $vol(\pi_e)$ as well as the length of *non-busy* intervals can only decrease if the worst-case execution time of any subjob decreases or the number of processors is increased. Since the response-time is upper-bounded by the sum of times that an envelope job is executed and the sum of times that no envelope job is executed, the corollary is proved. □

4.5.1 DAG VERTEX COVERAGE

The general problem to find the minimal number of vertex-disjoint paths – such that all vertexes of a graph G are covered – is an NP-complete problem as can be

Table 4.3: Summary of used Notation.

Symbol	Meaning
$vol(v) \in \mathbb{R}$	Volume (worst-case execution time) of subtask v
$\Pi(v) \in \mathbb{N}$	Fixed-priority value of subtask v
$\Psi(G)$	All paths from source vertex to sink vertex in G
$\pi \in \Psi(G)$	Path from source to sink vertex in G
$\pi_e \in \Psi(G)$	Envelope path of G in a schedule
$\pi_* \in \Psi(G)$	Longest path in G
$\mathcal{P}(\Psi(G))$	Path collections of G, i.e., power-set of all paths
$\psi \in \mathcal{P}(\Psi(G))$	A path collection
$C \in \mathbb{R}$	Total volume $vol(V)$ of a DAG G
$V_s(\psi)$	Set of vertexes (subtasks) in the path collection ψ
$vol(\pi_*) \in \mathbb{R}$	Volume of longest path π_*
$V_s^c(\psi)$	Path-collection complement $\{v \in V \mid v \notin V_s(\psi)\}$
$\mathcal{G}_i$	Gang reservation system for $\tau_i \in \mathbb{T}$
$\mathcal{O}_i$	Ordinary reservation system for $\tau_i \in \mathbb{T}$
$w \in \mathbb{N}$	Upper-bound for the minimal size of a path cover
$\mathbb{T}$	DAG task set

shown by reduction to the HamiltonianCycle problem. For directed graphs G however, it was shown by Gallai and Milgram in 1960 that the minimal number of vertex-disjoint paths to cover all vertexes of a directed graph G is no more than the size of the maximal independent set of G, generalizing the results from Dilworth [Dil90] and König-Egevary [Dem79].

In the case that the directed graph is also acyclic, i.e., a DAG, then the largest independent set of G can be computed in polynomial time by the known reduction to the maximal matching problem in bipartite graphs by using, e.g., the Hopcroft-Karp algorithm. For instance, the set of vertex-disjoint paths that cover the DAG illustrated in Figure 4.1 is given by $U = \{\langle v_1, v_2, v_3\rangle, \langle v_4, v_5, v_6\rangle, \langle v_7, v_8\rangle, \langle v_9\rangle\}$, which is calculated as described above. In our proposed path collection algorithm, we require a collection of paths from source to sink vertices of G, which fully cover G, which are not necessarily vertex-disjoint. Hence, we here explain the explicit algorithmic construction as required in Line 1 of our proposed Algorithm 4 to obtain $w \in \mathbb{N}$ many paths that cover G. Please note that for our algorithm, the knowledge of the numerical value w is sufficient without knowing the explicit paths. To prove the existence however, we construct the w-many paths explicitly.

For each $i \in \{1, \ldots, |U|\}$, we initialize the i-th path π_i with the vertex-disjoint path $u_i = \langle u_{i_1}, \ldots, u_{i_n}\rangle \in U$ and apply the following steps successively until all π_i are paths according to Definition 4.3:

- If the left-most vertex in π_i is not a source vertex of G, then pick any $u_h = \langle u_{h_1}, \ldots, u_{h_m}\rangle \in U$ such that $\langle v_{u_{h_z}}, v_{u_{i_1}}\rangle$ in E for some $z \in \{1, \ldots, m\}$ and extend the path to $\pi_i = \langle u_{h_1}, \ldots, u_{h_z}\rangle \circ \pi_i$.
- If the right-most vertex in π_i is not a sink vertex of G then pick any tuple $u_h = \langle u_{h_1}, \ldots, u_{h_m}\rangle \in U$ such that $\langle v_{u_{i_n}}, v_{u_{h_z}}\rangle$ in E for some $z \in \{1, \ldots, m\}$ and update the path to $\pi_i = \pi_i \circ \langle u_{h_z}, \ldots, u_{h_m}\rangle$

For instance, with reference to the provided example, we start at $u_4 = \langle 9 \rangle$ with $\pi_4 = \langle v_9 \rangle$, which is a sink vertex in G and identify tuple $u_2 = \langle 4,5,6 \rangle$ since $(v_5, v_9) \in E$ and the path is updated to $\pi_4 = \langle v_4, v_5, v_9 \rangle$. Since v_4 is not a source vertex in G, we continue and identify $u_1 = \langle 1,2,3 \rangle$ due to $(v_1, v_4) \in E$ and update $\pi_4 = \langle v_1, v_4, v_5, v_9 \rangle$. Since v_1 is a source vertex, the procedure is terminated. Repeating the procedure yields the four final paths $\pi_1 = \langle v_1, v_2, v_3 \rangle, \pi_2 = \langle v_1, v_7, v_8 \rangle, \pi_3 = \langle v_1, v_4, v_5, v_6 \rangle$, and $\pi_4 = \langle v_1, v_4, v_5, v_9 \rangle$, which collectively cover all vertexes $v \in V$. Please note that while in this example, $w = 4$ is the minimal number of paths to cover G, the algorithm in general only provides a safe upper-bound.

4.5.2 WEIGHTED MAXIMUM COVERAGE

Another related algorithm is the WEIGHTED MAXIMUM COVERAGE [NWF78] problem. Hereinafter, we map the problem of finding an n-path collection ψ for a DAG G that maximizes $vol(V_s(\psi))$ (minimizes $vol(V_s^c(\psi))$) to that problem as follows:

- **Input**: A problem instance I of the WEIGHTED MAXIMUM COVERAGE problem is given by a collection of sets $S := \{S_1, \ldots, S_m\}$, a weight function ω, and a natural number k. Each set $S_i \subseteq U$ is a subset from some universe U for each $i \in \{1, \ldots, m\}$ and each element $s \in S_i$ is associated with a weight as given by the function $\omega(s)$.
- **Objective**: For a given problem instance I, the objective is to find a subset $S' \subseteq S$ such that $|S'| \leq k$ and $\sum_{s \in \{\cup \{S_i \in S'\}\}} w(s)$ is maximized.

It was shown by Nemhauser et al. [NWF78] that any polynomial time approximation algorithm of the WEIGHTED MAXIMUM COVERAGE problem has an asymptotic approximation ratio with respect to an optimal solution that is lower-bounded by $1 - 1/e$ unless $P = NP$, where e is Euler's number. This approximation ratio can be achieved by a greedy strategy that always chooses the set which contains the largest weights of not yet chosen elements. Despite WEIGHTED MAXIMUM COVERAGE and our problem not being equivalent, we use the same approximation strategy for the n-path Collection Approximation in Algorithm 4.

4.5.3 APPROXIMATION ALGORITHM

On the basis of the existence of a path collection of size w, it is possible to analyze the approximation quality of Algorithm 4. We firstly present our proposed algorithm and thereafter prove the approximation factor.

n-Path Collection Approximation Algorithm. From Line 1 to Line 3 in Algorithm 4, the upper-bound w of the minimal number of paths to fully cover the is computed. If the number of processors M is sufficient to allow the parallel execution of all w paths, i.e., $w \leq M$ then those paths are chosen for the path collection.

In the other case, from Line 4 to 17, in each iteration $n \in \{1, \ldots, M\}$, the longest path π_n^* with respect to the current iteration's volume function vol' is chosen.

Algorithm 4 n-Path Collection Approximation (nPCA)

Require: DAG $G = (V, E)$, No. CPU M, WCET vol.
Ensure: An approximately optimal n-path collection ψ

1: $\psi^* \leftarrow \text{PathCoverage}(G) := \{\pi_{\psi_1}, \ldots, \pi_{\psi_w}\}$
2: **if** $w \leq M$ **then**
3: **return** (ψ^*, w);
4: create $\psi_0 \leftarrow \emptyset$;
5: $z \leftarrow \infty$;
6: $vol' \leftarrow vol$;
7: **for each** $n \in \{1, \ldots, M\}$ **do**
8: create $\psi_n \leftarrow \psi_{n-1}$;
9: $\pi_n^* \leftarrow$ use DFS(G) to search the max. vol' path;
10: $\psi_n \leftarrow \psi_n \cup \pi_n^*$;
11: **for each** $v \in \pi_n^*$ **do**
12: update $vol'(v)$ to 0;
13: $z' \leftarrow (C - vol(V_s(\psi_n))) \;/\; M - (n-1)$;
14: **if** $z' < z$ **then**
15: solution $(\psi^*, n^*) \leftarrow (\psi_n, n)$;
16: update $z \leftarrow z'$;
17: **return** solution (ψ^*, n^*);

After the path is chosen, all volumes of that path's subjobs are set to 0 to indicate that the subjobs have already been covered. By this strategy, we always choose the path, which contains the largest amount of volume of not yet chosen subjobs in each iteration. Moreover, in each n-th iteration, it is probed in Line 14 if the solution ψ_n strictly improves the prior solution ψ_{n-1} with one path less. At the end of the M-th iteration, an n^*-path collection ψ^* is found that yields formal guarantees as stated in Theorem 4.11. The maximal bipartite matching can be obtained in $\mathcal{O}(|V|)$ using the Hopcroft-Karp algorithm. The time-complexity of nPCA is dominated by the for-loop and the depth-first search (DFS) in Line 9 that is invoked in each of the iterations, resulting in $\mathcal{O}(M \cdot |V||E|)$ time complexity.

Theorem 4.11 (nPCA). *The worst-case response-time of a DAG job J (makespan) on M dedicated processors using parallel path progression scheduling and for which the n^*-many paths are calculated according to Algorithm 4, is at most*

$$R_{opt} \cdot \begin{cases} 1 + \frac{M}{M-n^*+1} \cdot \left(1 - \frac{1}{w}\right)^{n^*} \leq 2 - \frac{1}{w} & w > M \geq n^* \\ 1 & M \geq w \end{cases} \tag{4.36}$$

where w refers to the solution of the algorithm described in Section 4.5.1.

Proof. We prove this theorem for the cases $M \geq w$ and $M < w$ individually.

Case 1. In the first case, i.e., from Line 1 to Line 3, let $M \geq w$. From the discussion in Section 4.5.1, we know that each vertex $v \in V$ of the DAG $G = (V, E)$ is covered by at least one of those w-many paths as computed by the algorithm described

in Section 4.5.1, which are returned in Line 3. In consequence, the response-time bound is given by

$$R_J \leq vol(\pi_*) + \frac{0}{M-(w-1)} \leq R_{opt} \tag{4.37}$$

which is upper-bounded by an optimal response-time, since the longest path in G is a lower-bound of any DAG job's response-time.

Case 2. Please note that the obtained w is not a minimal solution, but an upper-bound of it. That is, there may exist n-many paths under the constraints $w > M \geq n \geq 1$ such that the DAG's total volume C can be covered. However such a minimal solution is not known to be computable in polynomial-time. We can however prove the approximation ratio of an optimal response-time with respect to that upper-bound.

Step 1. We prove by contradiction, that for each iteration $n \in \{1, \ldots, M\}$ the following inequality holds

$$vol^{(n)}(\pi_n^*) \geq \frac{C - vol(V_s(\psi_{n-1}))}{w} \tag{4.38}$$

where $\psi_0 := \emptyset$ and $vol(V_s(\psi_0)) = 0$. We use $vol^{(n)}$ to refer to vol' in the n-th iteration for better clarity in this proof. Assume for contradiction that there exists an iteration $n \in \{1, \ldots, M\}$ such that

$$\forall \pi_n \in \Psi(G) \quad w \cdot vol^{(n)}(\pi_n) + vol(V_s(\psi_{n-1})) < C \tag{4.39}$$

holds. Then it must hold in particular that

$$w \cdot vol^{(n)}(\pi_n^*) + vol(V_s(\psi_{n-1})) < C \tag{4.40}$$

where – by the algorithmic strategy – π_n^* is chosen such that for all paths $\pi \in \Psi(G)$, the inequality $vol^{(n)}(\pi_n^*) \geq vol^{(n)}(\pi)$ is satisfied. Thus, $w \cdot vol^{(n)}(\pi_n^*) \geq vol^{(n)}(\cup_{i=1}^{w} \pi_i)$ for any arbitrary collection of w-many paths. Consequently, we have that if Eq. (4.40) holds then

$$vol^{(n)}(\bigcup_{i=1}^{w} \pi_i) + vol(V_s(\psi_{n-1})) < C \tag{4.41}$$

holds as well. It is easy to see that based on the algorithm

$$vol^{(n)}(\bigcup_{i=1}^{w} \pi_i) = vol(\bigcup_{i=1}^{w} \pi_i) - vol(V_s(\psi_{n-1})) \tag{4.42}$$

holds, since if the volume of a vertex in the n-th iteration differs from the initial volume then that vertex must be covered by any of the collected paths during the prior iterations, i.e., $V_s(\psi_{n-1})$. Then using the identity of Eq. (4.42) in Eq. (4.41) yields that Eq. (4.39) leads to the condition $vol(\cup_{i=1}^{w} \pi_i) < C$ for any arbitrary collection of w-many paths, which contradictions the existence of a solution of a cover with w-many paths.

Step 2. In a second step, we now claim and prove by induction that

$$vol(V_s^c(\psi_n)) = C - vol(V_s(\psi_n)) \leq \left(1 - \frac{1}{w}\right)^n \cdot C \tag{4.43}$$

For $n = 1$, Eq. (4.43) reduces to $w \cdot vol(V_s(\psi_1)) \geq C$, which holds true since we know that there exists a finite w such that $vol(\pi_{\psi_1} \cup \ldots \cup \pi_{\psi_w}) = C \leq \sum_{i=1}^{w} vol(\pi_{\psi_i}) \leq w \cdot vol(V_s(\psi_1)) = w \cdot vol(\pi_*)$, since ψ_1 only contains the longest path in G. In the induction step $n \to n+1$, we have

$$C - vol(V_s(\psi_{n+1})) = C - (vol(V_s(\psi_n)) + vol'(\pi^*_{n+1})) \tag{4.44}$$

Using Eq. (4.38), we conclude that

$$\begin{aligned}
\text{Eq. (4.44)} &\leq C - vol(V_s(\psi_n)) - \frac{C - vol(V_s(\psi_n))}{w} && (4.45)\\
&\leq (C - vol(V_s(\psi_n))) \cdot \left(1 - \frac{1}{w}\right) && (4.46)\\
&\leq \left(1 - \frac{1}{w}\right)^n \cdot C \cdot \left(1 - \frac{1}{w}\right) && (4.47)
\end{aligned}$$

Conclusion. Using Eq. (4.43) yields that after the n-th iteration of nPCA, the maximum response-time using the computed n-path collection ψ_n is at most

$$R_J \leq vol(\pi_*) + \frac{C}{M} \cdot \frac{M}{M - n + 1} \cdot \left(1 - \frac{1}{w}\right)^n \tag{4.48}$$

Due to the fact that $R_{opt} \geq \max\{vol(\pi_*), C/M\}$ and the minimal response-time solution (ψ^*, n^*) returned by nPCA in Line 17 using Eq. (4.48) we have that

$$\begin{aligned}
R_J &\leq R_{opt} \cdot \min_{n \geq 1} \left\{1 + \frac{M}{M - n + 1} \cdot \left(1 - \frac{1}{w}\right)^n\right\}\\
&\leq R_{opt} \cdot \min_{n \geq 1} \left\{1 + n \cdot \left(1 - \frac{1}{w}\right)^n\right\} && (4.49)\\
&\leq R_{opt} \cdot \left(2 - \frac{1}{w}\right) && (4.50)
\end{aligned}$$

Please note that Eq. (4.49) is due to the fact that under the constraints ($M \geq n \geq 1$) the function $M/(M - n + 1)$ is strictly decreasing with respect to M for $n \geq 1$. Due to the constraints, the minimal feasible M is bounded from below by n and is thus no more than $n/(n - n + 1) = n$.

In addition, we have the parametric bound based on the results w and n^* generated by the algorithm namely

$$R_J \leq R_{opt} \cdot \left(1 + \frac{M}{M - n^* + 1} \cdot \left(1 - \frac{1}{w}\right)^{n^*}\right) \tag{4.51}$$

Finally, we have $R_J \leq$ Eq. (4.51) $\leq$ Eq. (4.50) concluding the proof. □

Discussion. For stated makespan problem, it is in fact irrelevant if there exists an $n < w \leq M$ such that G can be covered with n-many paths, since only a single DAG job is considered. That is, if sufficient processors are available then R_J is optimal, otherwise the iteration is started. Then, if an optimal n is found, the makespan is also optimal.

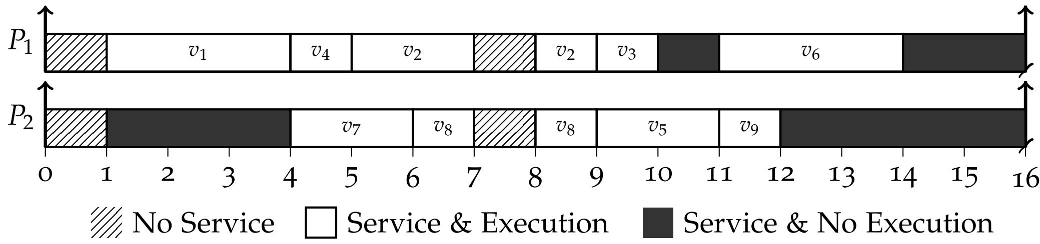

Figure 4.10: Exemplary schedule for a 2-gang reservation system with a 2-path collection of the DAG illustrated in Figure 4.1 with a deadline of 16 time units.

In the scheduling problem with other DAG jobs however, the non minimal solution does matter, since processor over-allocation should be avoided. Note however that our solution is never worse than the greedy strategy, since in the hierarchical scheduling allocations we consider the full-cover solution with w-many paths separately. In the evaluation, we can also show that for many DAG task sets, the provided bound can significantly reduce the allocation requirements compared to only the greedy strategy.

our solution is never worse than the greedy strategy, since in the hierarchical scheduling allocations we consider the full-cover solution with w-many paths separately

4.5.4 HIERARCHICAL SCHEDULING EXTENSION

We extend the properties of parallel path progression to a system with inter-task interference using a hierarchical scheduling approach. We propose and discuss two reservation schemes, namely a *gang reservation system* in Section 4.5.4.1 and an *ordinary reservation system* in Section 4.5.4.2, and provide resource provisioning rules and response-time analyses. The hierarchical scheduling problem consists of two interconnected problems:

- Service provisioning of the respective *gang*-reservation or *ordinary*-reservation systems such that a DAG job can finish within the provided service.
- Verification of the schedulability of the provisioned reservation systems by any existing analyses, which support the respective task models, e.g., sporadic arbitrary-deadline gang tasks or sporadic arbitrary-deadline ordinary sequential tasks.

For the remainder of this section, we assume the existence of a feasible schedule upon M identical multiprocessors for the studied reservation system model and focus on the service provisioning problem. We assume that a reservation system satisfies the following four properties stated in Section 4.3 regardless of the specific reservation model.

4.5.4.1 *Gang Reservation System*

In gang scheduling, a set of threads is grouped together into a so called gang with the additional constraint that all threads of a gang must be co-scheduled at the same time on available processors. It has been demonstrated that gang-based parallel computing can often improve the performance [FR92; Jet97; WP19]. Due to its performance benefits, the gang model is supported by many parallel computing standards, e.g., *MPI*, *OpenMP*, *Open ACC*, or *GPU computing*. Motivated

Algorithm 5 Approximate Minimal Waste Gang

Require: $\tau_i := (G_i, D_i, T_i)$, No. CPU M, vol, width w.
1: $\mathbb{G} \leftarrow \emptyset$;
2: **for each** $m_i \in \{1, \dots, \min\{w, M\}\}$ **do**
3: $(\psi^*, n^*) \leftarrow$ call Algorithm 4 with parameters G_i, m_i, vol;
4: **if** $E_i \leftarrow$ using Eq. (4.52) with $(m_i, \psi^*, n^*) \leq D_i$ **then**
5: $\mathbb{G} \leftarrow \mathbb{G} \cup (m_i, E_i, \psi^*, n^*)$;
6: **return** $\mathcal{G}_i \in \mathbb{G}$ such that $\mathcal{G}_i$ minimizes Eq. (4.54)

by the practical benefits and the conceptual fit of parallel path progressions in our approach and the gang execution model, we propose an m-Gang reservation system as follows.

Definition 4.25 (m-Gang Reservation System)**.** *A sporadic arbitrary-deadline m_i-gang reservation system $\mathcal{G}_i$, which serves a sporadic arbitrary-deadline DAG task $\tau_i := (G_i, D_i, T_i)$ is defined by the tuple $\mathcal{G}_i := (m_i, E_i, D_i, T_i)$ such that $m_i \cdot E_i$ service is provided during the arrival- and deadline interval, with the gang scheduling constraint that all reservations must be co-scheduled at the same time.*

Hence, the provisioning problem of $\mathcal{G}_i$ for a DAG task τ_i is to find m_i and E_i such that, given the parallel path progression property, and the gang scheduling constraint, each DAG job can complete within one of the m_i reservations before its absolute deadline.

Theorem 4.12 (Gang Reservation Provisioning)**.** *Each job J_i^ℓ of a sporadic arbitrary-deadline DAG task $\tau_i := (G_i, D_i, T_i)$ can complete its total volume C_i within its respective gang reservation instance of the m_i parallel reservations of size E_i before its absolute deadline if*

$$vol(\pi_*) + \frac{vol(V_s^c(\psi))}{m_i - n + 1} \leq E_i \leq D_i \tag{4.52}$$

holds for any n- path collection ψ of at most m_i and the gang reservation system $\mathcal{G}_i$ is verified to be schedulable, i.e., able to provide all service before the absolute deadline.

Proof. Since in an m_i-gang all reservations provide service simultaneously, the arrival and finishing time window $[a_J, f_J]$ of any DAG job J of task τ_i can be partitioned into *busy*, *non-busy*, and *non-service* intervals in which none of the m_i gang reservations are scheduled and thus provide no service. In analogy to previous proofs, the response-time is no more than the cumulative length of these sets, where the cumulative length of non-service times is upper-bounded by $D_i - E_i$ given the assumption that the m_i gang is schedulable. In consequence, if Eq. (4.52) holds, then

$$R_J \leq vol(\pi_*) + \frac{vol(V_s^c(\psi))}{m_i - n + 1} + D_i - E_i \leq D_i \tag{4.53}$$

which concludes the proof. □

Gang-Reservation Provisioning. Finding the best provisioning for specific gang reservation systems depends on the concrete schedulability problem at hand and the other tasks that are to be co-scheduled. Hence, it may be beneficial to trade decreased budgets for increased gang size in some concrete scenarios. Determining such specific provisions is however beyond the scope of this work. Instead, in a more generic optimization attempt, we seek to find a provisioning that minimizes the unused gang service (waste), which is described as $m_i \cdot E_i - C_i$. Due to the gang restriction, increasing the number of reservations m_i beyond the number of processors M that the reservations are going to be executed upon is not possible. Moreover, increasing the gang size beyond the width of the DAG can only increase the waste, since the width describes the maximal inherent parallelism. Due to the constrained search space of $m_i \in \{1, \ldots, \min\{w, M\}\}$ an exhaustive search can be applied to find the values of m_i, E_i with $E_i \leq D_i$ and ψ, n, which minimizes the waste objective

we seek to find a provisioning that minimizes the unused gang service (waste)

$$m_i \cdot vol(\pi_*) + m_i \cdot \frac{C_i - vol(V_s(\psi))}{m_i - n + 1} - C_i \tag{4.54}$$

as shown in Algorithm 5. For illustration of the algorithm, an exemplary result for the DAG in Figure 4.1 with longest path volume 10, total volume 18, relative deadline 16 and $M = 3$ is calculated. Since, the width w of the DAG can be observed to be 4, only gang sizes of $m_i \in \{1, 2, 3\}$ are viable candidates. The algorithm returns a 2-gang with the 2-path collection $V_s(\psi) = \{v_1, v_7, v_5, v_6, v_2, v_3\}$ that yields a reservation budget of $10 + (18 - 14)/(2 - 2 + 1) = 14 \leq 16$ and waste of $2 \cdot 14 - 18 = 10$. Figure 4.10 shows an exemplary schedule of this provisioned 2-gang system and the internal DAG job scheduling, using the gang reservations.

4.5.4.2 *Ordinary Reservation System*

Despite the practical benefits of gang scheduling, the analytic schedulability due to the co-scheduling constraint, is reduced compared to an equivalent *ordinary reservation system* without such constraints, defined as follows.

ordinary reservation system

Definition 4.26 (m-Ordinary Reservation System)**.** *A sporadic arbitrary-deadline m_i-ordinary reservation system $\mathcal{O}$, which serves a sporadic arbitrary-deadline DAG task $\tau_i := (G_i, D_i, T_i)$ is defined by the tuple $\mathcal{O}_i := (E_i^1, \ldots, E_i^{m_i}, D_i, T_i)$ such that $\sum_{p=1}^{m_i} E_i^p$ service is provided during the arrival- and deadline interval.*

Notably, despite the tuple representation, an ordinary reservation system is represented by unrelated sporadic arbitrary-deadline tasks, which can all execute independently, e.g., using partitioned scheduling.

Theorem 4.13 (Ordinary Reservation Provisioning)**.** *Each job J_i^ℓ of a sporadic arbitrary-deadline DAG task $\tau_i := (G_i, D_i, T_i)$ can complete its total volume C_i within its respective ordinary reservation instance $\mathcal{O}_i$ before its absolute deadline if firstly*

$$vol(\pi_*) + \frac{vol(V_s^c(\psi)) + (n-1) \cdot D_i}{m_i - n + 1} \leq \frac{\sum_{p=1}^{m_i} E_i^p}{m_i - n + 1} \tag{4.55}$$

holds for any n-path collection ψ where n is at most m_i and all; and secondly, the ordinary reservation system $\mathcal{O}_i$ is verified to be schedulable, i.e., each individual reservation is able to provide all service before the absolute deadline.

Proof. Let S be a schedule of m_i ordinary reservations, which are verified to be feasibly schedulable. That is, each individual reservation is guaranteed to provide E_i^p service to the DAG job $J := J_i^\ell$ during the interval $[a_J, a_J + D_i)$ for $p \in \{1, \ldots, m_i\}$. Let ψ denote the n-path collection and $V_s(\psi)$ denote the corresponding set of subjobs. Let $m_i(t) \in \{0, 1, \ldots, m_i\}$ the number of reservations, which provide service at time t and let $h(t) \in \{0, \ldots, m_i(t)\}$ the number of reservations providing service to subjobs in $V_s^c(\psi)$ at time t.

We prove this theorem by contrapositive, i.e., we assume that DAG J misses its deadline and prove that this leads to the violation of Eq. (4.55)

We prove this theorem by contrapositive, i.e., we assume that DAG job J misses its deadline and prove that this leads to the violation of Eq. (4.55). Let J miss its deadline at time $d_J := a_J + D_i$ in S and let v_{k_p} denote a subjob that is executing at time d_J and is not yet finished. Please note that at least one such subjob must exists, since otherwise the total volume is finished, which contradicts the assumption of a deadline miss.

Using the envelope construction rules in Definition 4.8, an *incomplete envelope path* $\pi_e := \langle v_{k_1}, v_{k_2}, \ldots, v_{k_p} \rangle$ is derived starting from subjob v_{k_p}. We partition the interval $[a_J, d_J)$ into contiguous sub intervals I_{k_i} namely $[a_{k_1}, f_{k_1}), [a_{k_2}, f_{k_2}), \ldots, [a_{k_p}, d_J)$ for each incomplete envelope subjob. Moreover, we partition each subjob interval into *busy times* defined as $\alpha_{k_i} := \{t \in I_{k_i} \mid v_{k_i} \text{ is executed}\}$ and *non-busy times* defined as $\beta_{k_i} := \{t \in I_{k_i} \mid v_{k_i} \text{ is not executed}\}$ for each $i \in \{1, \ldots, p\}$. Please note that in our partitioning, the case of no service, i.e., $m_i(t) = 0$ is considered a *non-busy time*.

To measure the cumulative amount of time spent in either state, we define

$$|\alpha_{k_i}| = \int_{I_{k_i}} [t \in \alpha_{k_i}] \, dt \text{ and } |\beta_{k_i}| = \int_{I_{k_i}} [t \in \beta_{k_i}] \, dt \tag{4.56}$$

where $[pred]$ denotes the iverson bracket, which evaluates to 1 if the predicate is true and 0 otherwise. Each point in time $t \in [a_J, d_J]$ is exclusively either a busy or a non-busy time and since by assumption J misses its deadline, we have that

$$D_i < \sum_{i=1}^{p} |\alpha_{k_i}| + |\beta_{k_i}| \tag{4.57}$$

We analyze the amount of busy times and non-busy times separately as follows.

Busy Time. The cumulative amount of *busy* times in each interval I_{k_i} is given by $|\alpha_{k_i}| \leq vol(v_{k_i})$ for $i \in \{1, \ldots, p-1\}$ and $|\alpha_{k_p}| < vol(v_{k_p})$ since the subjob v_{k_p} has not yet finished execution by definition. In summary, the cumulative amount of busy times during I_{k_i} is given by

$$\sum_{i=1}^{p} |\alpha_{k_i}| < \sum_{i=1}^{p} vol(v_{k_i}) = vol(\pi_e) \leq vol(\pi_*) \tag{4.58}$$

Non-Busy Time. We further partition the *non-busy* interval into a *parallel path* case if the incomplete envelope subjob $v_{k_i} \in V_s(\psi)$ and a *non-parallel path* case if $v_{k_i} \in V_s^c(\psi)$. Since our scheduling policy is work-conserving we have that whenever an envelope subjob v_{k_i} is not serviced at time $t \in I_{k_i}$ then all $m_i(t)$ reservations must be servicing non-envelope jobs (or no service is available at all).

Non-Parallel Path: Let by assumption $v_{k_i} \in V_s^c(\psi)$ then for any time $t \in \beta_{k_i}$ each of the $m_i(t)$ reservations is either exclusively servicing subjobs from $V_s^c(\psi) \setminus v_{k_i}$ (or no service is provided at all). This is due to the fact that $V_s(\psi)$ subjobs have lower-priority than $V_s^c(\psi)$ subjobs and thus the servicing of a subjob from $V_s(\psi)$ would imply the servicing of all pending $V_s^c(\psi)$ subjobs, which contradicts the assumption that pending $v_{k_i} \in V_s^c(\psi)$ is not serviced. In consequence of this implication we have that $\beta_{k_i} \subseteq \{t \in I_{k_i} \mid h(t) = m_i(t)\}$ and thus $|\beta_{k_i}|$ can be over-approximated as

$$|\beta_{k_i}| \leq \int_{I_{k_i}} [h(t) = m_i(t)] \, dt \tag{4.59}$$

We introduce the auxiliary function $\bar{m}_i(t) = m_i - m_i(t)$ to formalize the non-service at time t, which yields

$$|\beta_{k_i}| \leq \int_{I_{k_i}} [h(t) + \bar{m}_i(t) = m_i] \, dt \tag{4.60}$$

Each $t \in I_{k_i}$, which satisfies the predicate also satisfies $(h(t) + \bar{m}_i(t))/m_i = 1$. Moreover, since by definition $h(t) + \bar{m}_i(t) \geq 0$ for any $t \in I_{k_i}$ (regardless of the predicate being satisfied or not), we further approximate the length of β_{k_i} to

$$|\beta_{k_i}| \leq \int_{I_{k_i}} \frac{h(t) + \bar{m}_i(t)}{m_i} \, dt \tag{4.61}$$

Parallel Path: By assumption, let the incomplete envelope subjob $v_{k_i} \in V_s(\psi)$ not being serviced by any $m_i(t)$ at time $t \in I_{k_i}$. We use $n(t) \in \{1, \ldots, n\}$ to denote the number of pending subjobs from $V_s(\psi)$ at time t. By case assumption, we know that for each $t \in \beta_{k_i}$ at most $n(t) - 1$ subjobs from $V_s(\psi)$ are serviced by $m_i(t)$ reservations at time t. Additionally, we know that pending $V_s^c(\psi)$ subjobs are prioritized before $V_s(\psi)$ subjobs, i.e.,

$$h(t) \geq m_i(t) - \min\{m_i(t), n(t) - 1\} \tag{4.62}$$

$$\geq m_i(t) - (n(t) - 1) \geq m_i(t) - (n - 1) \tag{4.63}$$

In consequence of this implication we have that

$$\beta_{k_i} \subseteq \{t \in I_{k_i} \mid h(t) \geq m_i(t) - (n - 1)\} \tag{4.64}$$

and thus

$$|\beta_{k_i}| \leq \int_{I_{k_i}} [h(t) \geq m_i(t) - (n - 1)] \, dt \tag{4.65}$$

Using $\bar{m}_i(t) = m_i - m_i(t)$ yields the inequality $h(t) + \bar{m}_i(t) \geq m_i - (n - 1)$. With the same reasoning as in the previous case, the length can be approximated by

$$|\beta_{k_i}| \leq \int_{I_{k_i}} \frac{h(t) + \bar{m}(t)}{m_i - (n - 1)} \, dt \tag{4.66}$$

In conclusion we have that

$$|\beta_{k_i}| \leq \begin{cases} \text{Eq. (4.66)} & \text{if } v_{k_i} \in V_s(\psi) \\ \text{Eq. (4.61)} & \text{if } v_{k_i} \in V_s^c(\psi) \end{cases} \tag{4.67}$$

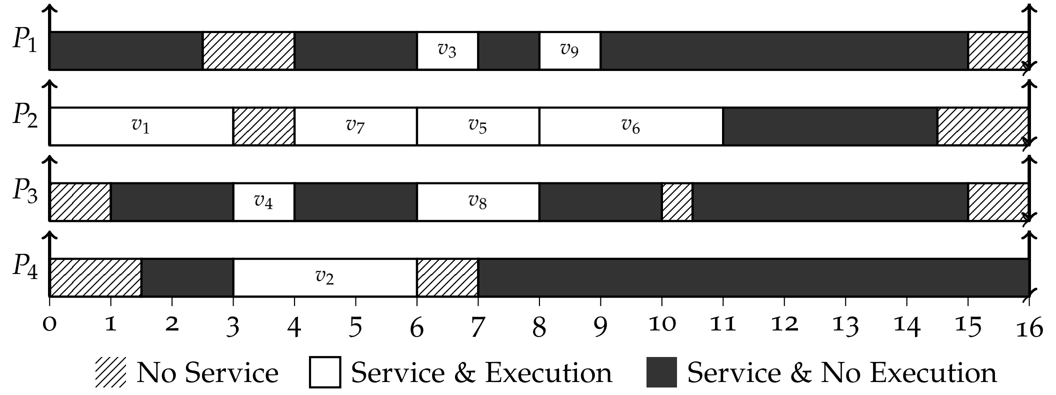

Figure 4.11: Exemplary schedule for an ordinary reservation system of the DAG illustrated in Figure 4.1 with a deadline of 16. A reservation system that consists of 4 equally sized reservations of 13.5 time units using a 3-path collection computed by nPCA.

and since Eq. (4.66) $\geq$ Eq. (4.61), we reach the conclusion that

$$\sum_{i=1}^{p} |\beta_{k_i}| \leq \int_{a_J}^{d_J} \frac{h(t) + \bar{m}(t)}{m_i - (n-1)} \, dt \tag{4.68}$$

By definition, $\int_{a_J}^{d_J} h(t)\, dt \leq vol(V_s^c(\psi))$ holds and thus

$$\sum_{i=1}^{p} |\beta_{k_i}| \leq \frac{vol(V_s^c(\psi))}{m_i - n + 1} + \int_{a_J}^{d_J} \frac{\bar{m}_i(t)}{m_i - n + 1} \, dt \tag{4.69}$$

The contract of the reservation system for a job of DAG task τ_i promises to provide $E_i^1, \ldots, E_i^{m_i}$ service during the arrival time of the DAG job J and its deadline $[a_J, d_J)$. Therefore, each of the m_i individual reservations does not provide service for at most $D_i - E_i^p$ for $p \in \{1, \ldots, m_i\}$ amount of time, which implies that

$$\sum_{i=1}^{p} |\beta_{k_i}| \leq \frac{vol(V_s^c(\psi))}{m_i - n + 1} + \frac{\sum_{p=1}^{m_i} D_i - E_i^p}{m_i - n + 1} \tag{4.70}$$

In conclusion and with reference to Eq. (4.58), we have that a deadline miss of J implies that

$$D_i < vol(\pi_*) + \frac{vol(V_s^c(\psi))}{m_i - n + 1} + \frac{\sum_{p=1}^{m_i} D_i - E_i^p}{m_i - n + 1} \tag{4.71}$$

which proves the theorem. □

Ordinary-Reservation Provisioning Algorithm. Similar to the problem of gang reservation provisioning, trading the number of reservations with the individual reservation sizes may be beneficial for the *schedulability* of concrete task sets. Again, to provide a baseline solution that can be further refined for concrete application scenarios, we search for reservation systems that minimize the cumulative allocated service under the constraints of equal budgets $E_i^p = E_i^{p+1}$ for $p \in \{1, \ldots, m_i - 1\}$ and $E_i^1 \leq D_i$, which is described in Algorithm 6. The key

Algorithm 6 Minimal Service Ordinary Reservations

Require: Task $\tau_i := (G_i, D_i, T_i)$, No. CPU M, vol_i.
Ensure: Minimal Service m_i-ordinary Reservations $\mathcal{O}_i$

1: $\psi^*, w \leftarrow \text{PathCover}(G) := \left\{\pi_{\psi_1^*}, \ldots, \pi_{\psi_w^*}\right\}$
2: $vol' \leftarrow vol;$
3: create $\psi_0 \leftarrow \emptyset;$
4: **for each** $n \in \{1, \ldots, \min\{w, M\}\}$ **do**
5:     **if** $n = w$ **then**
6:         **if** $vol(V_s(\psi_*)) - vol(V_s(\psi_{n-1})) > D_i$ **then**
7:             $E_i^* \leftarrow (vol(\pi_*) + (w-1) \cdot D_i)/w;$
8:             **return** solution $(E_i^*, w, \psi_*);$
9:     **else**
10:         create $\psi_n \leftarrow \psi_{n-1};$
11:         $\pi_n^* \leftarrow$ use DFS(G) to search the max. vol' path;
12:         $\psi_n \leftarrow \psi_n \cup \pi_n^*;$
13:         **for each** $v \in \pi_n^*$ **do**
14:             update $vol'(v)$ to 0;
15:         **if** $vol(V_s(\psi_n)) - vol(V_s(\psi_{n-1})) > D_i$ **then**
16:             $E_i^* \leftarrow (vol(\pi_*) + vol(V_s^c(\psi_n)) + (n-1)D_i)/n;$
17:             $n^* \leftarrow n;$
18: **for each** $m_i \in \{n^*, \ldots, M\}$ **do**
19:     $E_i' \leftarrow ((m_i - n^*) \cdot vol(\pi_*) + n^* \cdot E_i^*)/m_i;$
20:     **if** $E_i' \leq D_i$ **then**
21:         **return** solution $(E_i', m_i, \psi_n);$
22: **return** infeasible;

intuitions are; firstly the number of parallel reservations m_i is bound by the minimum of the number of available processors M and the width of the serviced DAG w, and secondly that the cumulative allocated service can only increase with increasing m_i. Therefore in a first stage, we set m_i to the minimal attainable value, which is n under the constraints $m_i \geq n$, resulting in the inequality

$$vol(\pi_*) + C_i - vol(V_s(\psi)) + (n-1) \cdot D_i \leq \sum_{p=1}^{n} E_i^p$$

as subject to optimization. In Algorithm 6, we compute the path-cover and width w and apply for each $n \in \{1, \ldots, \min\{w, M\}\}$ the iterative nPCA principle to search the configuration that minimizes the service by analyzing if in the n-th iteration the following improvement condition

$$-vol(V_s(\psi_n)) + (n-1) \cdot D_i < -vol(V_s(\psi_{n-1})) + (n-2) \cdot D_i$$

holds, which simplifies to $vol(V_s(\psi_n)) - vol(V_s(\psi_{n-1})) > D_i$. In case that the calculated $n^* < w$ leads to a deadline constraint violation, i.e., $E_i^* > D_i$, we iterate $m_i \in \{n^*, \ldots, M\}$ until the deadline is met and return the respective solution. If $n^* = w$ then the deadline constraint can only be violated if $vol(\pi_*) > D_i$ holds, i.e., if the DAG is not schedulable by default.

Discussion. In contrast to the gang reservation system, increasing the size of the path collection in the ordinary reservation system may not be beneficial due to the additional $(n-1) \cdot D_i$ term. In order for improvements over a single path collection to be possible, the following condition must hold

$$(m-n+1)vol(\pi_*) + C_i - vol(V_s(\psi_n)) + (n-1) \cdot D_i < (m-1)vol(\pi_*) + C_i$$

which simplifies to

$$(n-1) \cdot (D_i - vol(\pi_*)) + vol(\pi_*) < vol(V_s(\psi_n)) \tag{4.72}$$

While the left-hand side's increase is always $D_i - vol(\pi_*)$ the right-hand side's increase is diminishing (under nPCA). Therefore, if $n = 2$ does not yield an improvement then neither does any $w > n > 2$. In general, for an n-path collection to be an improvement over the single path collection and respective reservation system, the following condition must be satisfied

$$(n-1) \cdot (D_i - vol(\pi_*)) < \sum_{k=2}^{n} vol'(\pi_k^*) \tag{4.73}$$

where vol' refers to the iteration based volume function in nPCA. It can be observed that very tight deadlines, i.e., $D \approx vol(\pi_*)$ and DAG structures with few overlapping paths benefit the most from this approach. In that regard the DAG used in the following example does not benefit from the parallel path concepts and only serves to illustrate the reservation principle. Note that since in Algorithm 6, the calculated cumulative allocated service no more than the cumulative allocated service of a single path reservation system, the speed up factors of $3 + 2\sqrt{2}$ under partitioned and global EDF scheduling of the arbitrary-deadline ordinary reservation system with respect to any optimal scheduler as shown by Ueter et al. [UBC+18] still apply.

since in Algorithm 6, the calculated cumulative allocated service no more than the cumulative allocated service of a single path reservation system, the speed up factors of $3 + 2\sqrt{2}$ under partitioned and global EDF scheduling of the arbitrary-deadline ordinary reservation system with respect to any optimal scheduler as shown by Ueter et al. [UBC+18] still apply

Example. An exemplary 4-ordinary reservation system using a 3-path collection for the DAG shown in Figure 4.1 with deadline 16 is illustrated in Figure 4.11. In this example, each reservation is equal in size which results to $E_i^p = 13.5$ for $p \in \{1, \ldots, 4\}$ according to Eq. (4.55) with $D_i = 16$, $n = 3$, $m_i = 4$, and $vol(V_s^c(\psi)) = 4$. Please notice that the service can be provided arbitrarily depending on a concrete schedule as long as the promised service is provided within the arrival and deadline interval.

4.5.5 EVALUATION

In the forthcoming evaluations, we assess the performance of our proposed parallel path progression concepts using synthetically generated DAG task sets to allow for a systematic and exhaustive exploration. Firstly, we evaluate the makespan of our approach (*OUR*) compared to the state-of-the-art approach as represented by He et al. [HLG21] (*HE*). We use federated scheduling [LCA+14] (*FED*) to assess if *OUR* can leverage the the potential of parallel execution of

parallel paths, since a similar performance of *OUR* and *FED* indicates that either most of the workload is on the longest path or the number of processors is insufficient, i.e., our bound degrades to Grahams bound.

Secondly, we evaluate our proposed gang and ordinary reservation systems provisioning strategies from Algorithm 5 (*GA*) and Algorithm 6 (*ORD*) for relative resource over-provisioning against the resource allocation of semi-federated scheduling [JGL+17] (*SEM*) and reservation-based federated scheduling [UBC+18] (*UE*) with respect to the DAG's total volume, i.e., the lower-bound.

Thirdly, we evaluate if our path cover algorithm can outperform the iterative path collection selection in the approximation algorithm and in what parametric scenarios.

4.5.5.1 *Experiment-Data Generation*

In order to systematically evaluate the presented algorithms and to assess the performance for different classes of DAG structures, we generated 300 DAGs using the *layer-by-layer* and the *Erdős–Rényi* generation method for each configuration of generation parameters.

layer-by-layer
Erdős–Rényi

Layer-by-Layer Generation. The internal structure of the DAG under evaluation strongly impacts the performance of the evaluated analyses. The layer-by-layer method offers a parameterized generation process to randomly generate DAGs whose structure can be attributed to the generation parameters *min parallelism*, *max parallelism*, *min layer*, *max layers*, and *connection probability*. In each DAG's generation, the number of layers is chosen uniformly from the range $5-10$ and $10-15$ representing *min layer* to *max layer*. In each layer, the number of subtasks referred to as *parallelism* is drawn uniformly from the ranges of $5-10$, $10-15$, $10-25$, and $10-30$ representing the range of *min parallelism* to *max parallelism*. Please note that the minimal number of paths to fully cover a DAG is never more than the maximum parallelism in any of the generated layers. The connection of subtasks at a layer ℓ is only allowed by subtasks from layer $\ell-1$. Each newly generated subtask in a layer is connected with a subtask from the previous layer with probability *connection probability*, which is drawn at random from the ranges $5\%-10\%$, $10\%-20\%$, $20\%-30\%$, $40\%-50\%$, $50\%-60\%$, and $40\%-80\%$ resulting in 48 different configurations for which we generated a set of 300 DAGs each. Each subtask is assigned an integer worst-case execution time drawn uniformly from the range 10 to 100.

Erdős–Rényi Generation. In addition, we generated DAG task sets using the Erdős–Rényi method that is paremeterized with *min vertex* to *max vertex* and *connection probability*. In the generation process, at first, a number of vertexes is drawn uniformly in the ranges of $10-100$ and $100-150$. For each class of connection probabilities $5\%-10\%$, $15\%-20\%$, $25\%-30\%$, $35\%-40\%$, and $45\%-50\%$, a *connection probability* is drawn uniformly that is used for the generation of a single DAG belonging to that class. Secondly, an upper-triagonal adjacency matrix is generated where each entry a_{ij} in the matrix, i.e., the directed edge (v_i, v_j), is drawn uniformly with the probability *connection probability*. We generated 300 DAGs for each combination of configurations.

Table 4.4: Difference of the path cover bound w compared to the minimal number of paths calculated by the greedy approach for a full cover for the layer-by-layer DAG sets.

Layers	Paral.	Prob. (%)	max Δ	Improved (%)
5- 15	5-30	5-10	7	26
5-15	5-30	10-30	6	23
5-15	5-30	40-80	3	5

Table 4.5: Difference of the path cover bound w compared to the minimal number of paths calculated by the greedy approach for a full cover for the Erdős–Rényi DAG sets.

No. Vertex	prob. (%)	max Δ	Improved (%)
10-100	5-10	9	99%
10-100	15-20	5	96%
10-100	25-30	4	80%
10-100	35-40	2	63%
10-100	45-50	2	51%
100-150	5-10	6	75%
100-150	15-20	5	77%
100-150	25-30	3	66%
100-150	35-40	2	56%
100-150	45-50	2	38%

Deadline and Period Generation. For each of the generated DAG sets described before, we generated *easy*, *medium*, and *hard* to schedule deadlines. That is, the open interval of deadlines $D := (vol(\pi_*), C)$ defines deadlines such that the DAG is not infeasible by default or trivially schedulable. The interval is then partitioned into three equi-sized intervals D_1, D_2 and D_3 representing the first, second, and third fraction of the interval. A deadline is considered *hard* if it is drawn uniform at random from the interval D_1, *medium* if it is drawn uniform at random from D_2, and *easy* if it is drawn uniform at random from D_3 respectively. Since the compared to methods only support constrained-deadlines except for reservation-based federated scheduling, which is a special case of our ordinary reservation system, we only evaluated constrained deadlines. We draw $\alpha \in [a, b] \subset [1, b]$ for $[a, b]$ in $\{[1, 1.2], [1.2, 1.5], [1, 2], [2, 3], [1, 1.8], [1, 3]\}$ and set the period $T = \alpha \cdot D$.

4.5.5.2 *Path Cover Experiments*

In the path cover experiment, we compare the calculated bound by the path cover algorithm that is described in Section 4.5.1 against the minimal number of paths required by the greedy approach – described in Section 4.5.3 – to completely cover the DAG. The motivation for this experiment is to show that there are actually cases for the evaluated DAGs in which this bound is better than the iterative solution and in consequence, the resource allocation can be significantly improved. Both algorithms are evaluated on the DAGs sets described

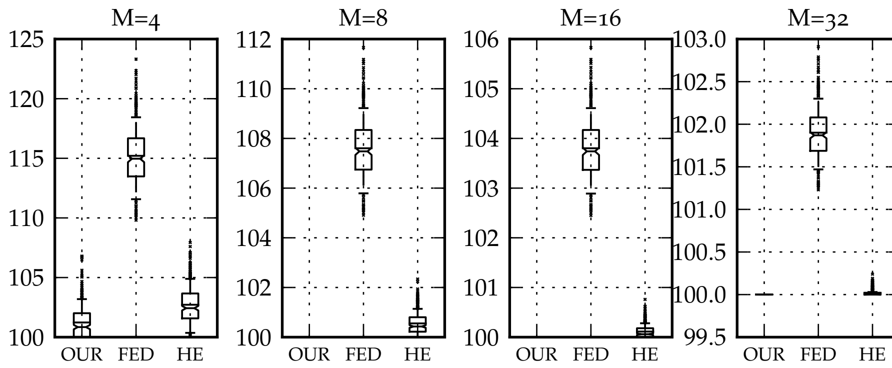

Figure 4.12: Relative makespan of DAGs generated by the Erdős–Rényi method with $100 - 150$ vertexes and a connection probability of $45 - 50\%$.

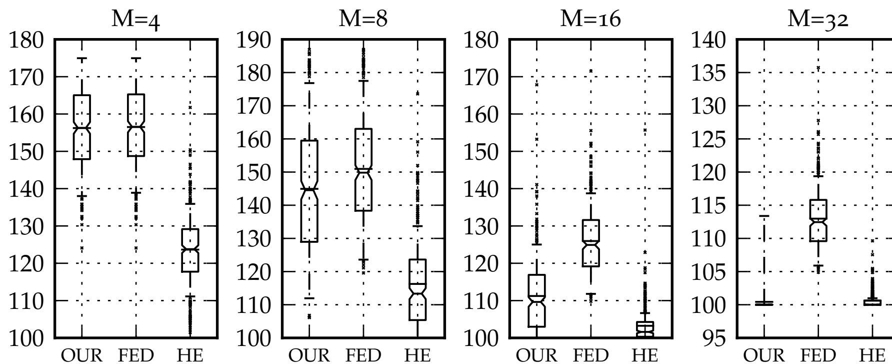

Figure 4.13: Relative makespan of DAGs generated by the layer-by-layer method with $10 - 15$ layers, parallelism of $10 - 30$, and a connection probability of $50 - 60\%$.

in Section 4.5.5.1 for which the results are aggregated and shown in Table 4.4 and in Table 4.5 respectively. The column *improved* shows the percentage of DAGs in the evaluated set for which the path cover determines less paths that the iterative solution. The column *max* shows the maximal absolute difference of the required paths, i.e., how many paths, the path cover algorithm requires less.

It can be observed that for the Erdős–Rényi set all DAG sets can be significantly improved and that sets in the range of $5 - 40\%$ connection probability have at least 56% improvements. For the layer-by-layer DAG sets, the improvements are still existent however only roughly a quarter of DAGs could benefit and higher connection probabilities reduce the improvements.

4.5.5.3 *Makespan Experiment*

We evaluate the makespan, i.e., the worst-case response-time of a single DAG job on $4, 8, 16, 32$ processors exclusively for all configurations described in Section 4.5.5.1 and present the makespan normalized to a theoretical lower-bound of $\max\{C/M, vol(\pi_*)\}$, i.e., 100% implies a tight result. Since the evaluations showed similar results, only a few representative figures are shown in the box plots in Figure 4.12, and Figure 4.13.

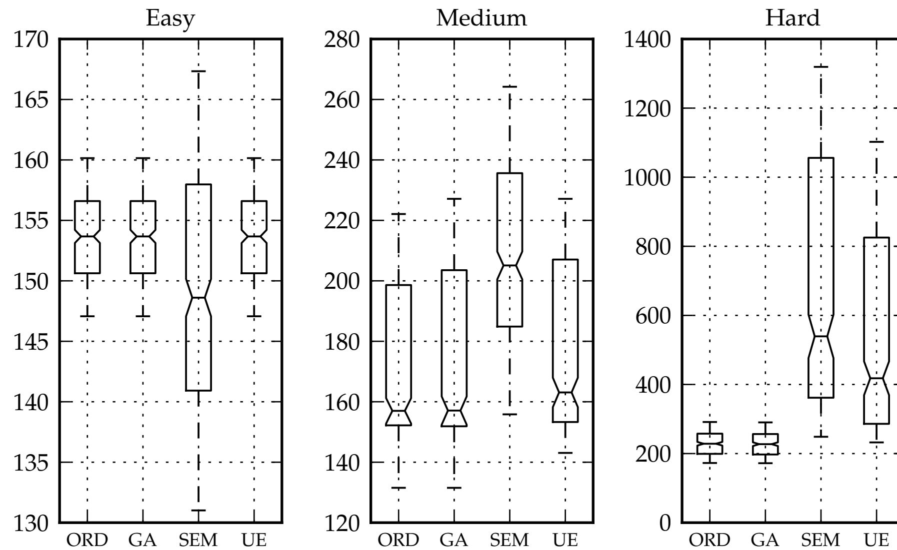

Figure 4.14: Over-provisioning for Erdős–Rényi with $100 - 150$ vertexes, $35 - 40\%$ connection probability and $\alpha = [1.2, 1.5]$.

Observations. From all recorded results, it can observed that our method does not strictly dominate the approach by *HE*, but can improve the makespan in case of high parallelism, i.e., large number of processors. Intuitively, the makespan of our approach is better if the majority of the workload of the DAG task is distributed on at most M paths, which *likely* increases with the number of available processors. Therefore, the improvements depend on the DAG parameters and number of processors. Otherwise, the approach by *HE* is better in analyzing the subtask interference more accurately and thus able to provide a better makespan. Notably, our approach is however able to provide tight results for many of the evaluated cases. A representative case is shown in Figure 4.12, where a DAG set with 100-150 vertexes and $45 - 50\%$ connection probability is evaluated. It can be observed that *OUR* provides a tight results when the number of provided processors is 8, whereas *HE* provides slightly larger (non-optimal) makespan values. Another representative case is shown in Figure 4.13 for a DAG set generated by the layer-by-layer method with $10 - 15$ layers, a parallelism of $10 - 30$, and a connection probability of $50 - 60\%$. Note that, federated scheduling (*FED*) coincides with our analysis if only one path is considered. It can be seen that *OUR* does not significantly improve *FED* up to 16 processors, which suggests that the the majority of the workload of the DAG task is distributed on more paths. However, when 32 processors are provided for the tasks with *parallelism* of at most 30 then the makespan of *OUR* is tight in most cases.

4.5.5.4 *Reservations Over-Provisioning Experiment*

In our hierarchical scheduling approach, the schedulability depends on the schedulability analysis used for the reservation systems. We are only interested in the evaluation of the resource allocation of the reservations systems as the schedulability depends on the performance of the schedulability analyses of the scheduling algorithms used to schedule the reservation systems. We compare the resource allocation of each approach per job activation (meaning over a period T)

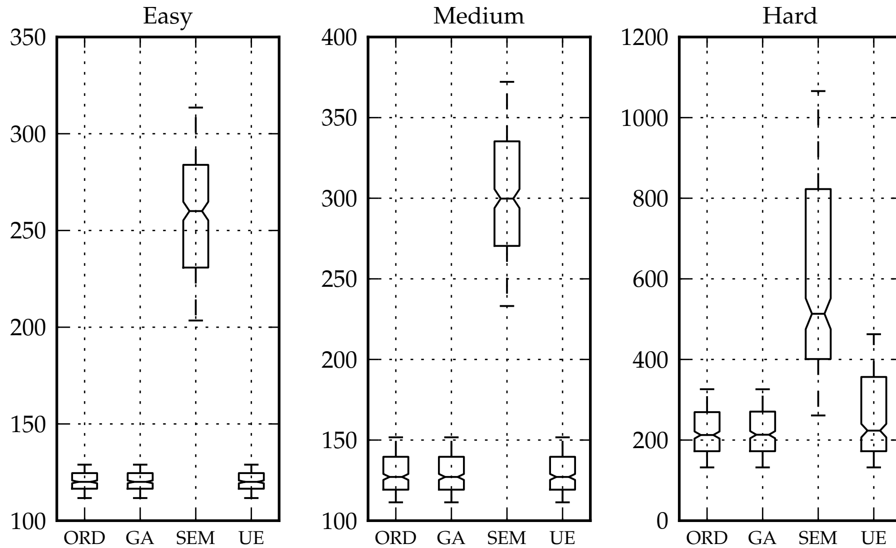

Figure 4.15: Over-provisioning for layer-by-layer 10 − 15 layers, parallelism 10 − 30%, connection probability 50 − 60% and $\alpha = [2, 3]$.

of our ordinary reservation (ORD), our gang (GA), semi-federated scheduling (SEM), and reservation-based federated scheduling (UE) relative to the DAGs total volume. That is, an over-provision value of 100% indicates a tight allocation. We assume that a sufficient number of processors is available such that a reservation system can be found for every generated deadline. A few representative results are shown in the box plots in Figure 4.14 and Figure 4.15.

Observation. It can be seen that *ORD* and *GA* improve the resource allocation in all scenarios and significantly for hard to scheduled DAG tasks. With increasing period to deadline ratio, the larger the improvements of the reservation based scheduling approaches to semi-federated scheduling are. Interestingly, *ORD* and *GA* show similar resource allocation in all scenarios, which demonstrates that the less restrictive reservation scheme of *ORD* does not incur larger resource demands.

4.5.6 Reclamation & Suspension

In the preceding analyses and provisioning algorithms of the gang- and ordinary reservation systems, the property of *sustained service* – that is, that the service of a reservation is provided whenever the reservation system is scheduled, irrespective of whether there are insufficient number of pending subjobs to be served at that time – is required. The following question that is to be examined and discussed in the remainder of this section is whether this pessimism can be reduced.

4.5.6.1 *Reclamation*

A possible and robust option to reclaim resources is to attach soft real-time or best-effort workload to each hard real-time parallel DAG reservation system with a lower priority than the to-be served DAG job. In the formal response-time and

provisioning analyses, the only required property is that the to-be served DAG job can claim the promised service whenever a subjob is pending and the reservation is scheduled for execution. As can be seen, this property is not violated by the background workload, due to the lower priority.

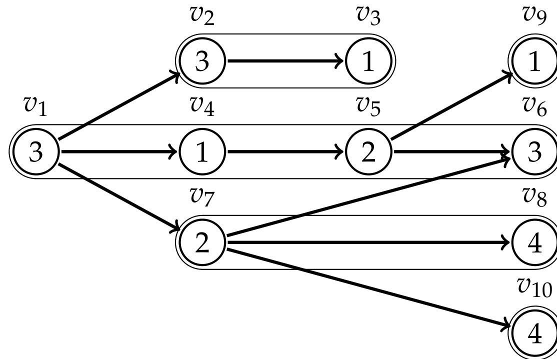

Figure 4.16: An exemplary directed-acyclic graph (DAG) with subtasks $v_1, v_2, \ldots, v_{10}$. The numbers within the vertices denote the subjob's worst-case execution time and the arrows represent the precedence constraints of the subjobs. The path-monotonic decomposition is highlighted by the border around the respective subtasks.

4.5.6.2 *Suspension*

In a suspension-aware reservation system, a reservation suspends itself whenever there is no pending subjobs waiting to be served

Another possible solution is to use the concept of self-suspension within the hierarchical scheduling policy, i.e., suspension-aware reservation systems. In a suspension-aware reservation system, a reservation suspends itself whenever there is no pending subjobs waiting to be served and thus resource utilization is improved at the cost of increased response-time analysis complexity.

The formal construction and technical implementation of such a suspension-aware reservation system is however non-trivial, due to the problems of

- determining a consistent rule to suspend and resume a reservation within the reservation system;
- and to upper-bound the amount of time that a reservation may be in a suspended state

This is due to the fact, that before runtime it is uncertain which subjobs are executed upon which specific reservation.

path-monotonic decomposition

Under some specific circumstances, that we call a *path-monotonic decomposition*, it is possible to tie specific subtasks to a specific reservation within the reservation system and achieve a similar but stricter path-monotonic progression property than the parallel path progression property in the reservation systems. This path-monotonic progression property allows to know at design time, which paths are to be executed on which reservation during runtime.

Definition 4.27 (Path-monotonic Decomposition)**.** *A path-monotonic prioritization and decomposition of a DAG $G = (V, E)$ is defined by an algorithm in a two stage procedure as follows:*

- *Determine the volume of the longest path that each vertex $v \in V$ lies on, which is denoted as $vol(\pi(v))$. For algorithmic processing, a unique source vertex v_0 and a unique sink vertex v_∞ is augmented to G. Then using the idea of the Floyd-Warshall algorithm yields the largest volume path from each vertex u to any vertex v for $u, v \in V$ denoted as $vol(u, v)$. Subsequently, it is possible to determine the length of a longest path that goes through a vertex $u \in V$ by*

$$vol(\pi(v)) := vol(v_0, u) + vol(u, v_\infty) - vol(u) \tag{4.74}$$

Then, all vertexes v with the same $vol(\pi(v))$ are collected, e.g., with reference to the DAG illustrated in Figure 4.16 and Table 4.6 the sets $\{v_1, v_4, v_5, v_6, v_7, v_8, v_{10}\}$ and $\{v_2, v_3, v_9\}$ can be inferred with volume 9 and 7, respectively. While this partition, is already sufficient to obtain a path-monotonic prioritization that satisfies Eq. (4.75), further considerations are required for the subtask to reservation partition as some vertexes of the same partition lie on different paths, i.e., can execute in parallel.

- *In a second step, the partitions are further refined such that for any two vertices u, v in a partition, the property $v \notin paral(u)$ and $u \notin paral(v)$ holds, where $paral(u) := \{v \in V \mid \nexists \text{ path from } u \text{ to } v \text{ or from } v \text{ to } u \text{ in } G\}$. Intuitively, this property implies that all vertices in a partition lie on the same path, i.e., execute sequentially. Recurring back to the example, the procedure yields the partition $\{v_1, v_4, v_5, v_6\} \cup \{v_7, v_8\} \cup \{v_{10}\}$ of the first set. We assign the priority in increasing order i.e., $\Pi(v) = 1$ for $v \in \{v_1, v_4, v_5, v_6\}$, $\Pi(v) = 2$ for $v \in \{v_7, v_8\}$, and $\Pi(v) = 3$ for $v \in \{v_{10}\}$. Similarly for $\{v_2, v_3, v_9\}$, we have $\{v_2, v_3\} \cup \{v_9\}$ and $\Pi(v_2) = \Pi(v_3) = 4$, and $\Pi(v_9) = 5$. Note that the priorities are going to be used for the suspension-aware reservations that are executing the specific tied subtasks.*

The refined partitions, e.g., $V_1 := \{v_1, v_4, v_5, v_6\} \cup V_2 := \{v_7, v_8\} \cup V_3 := \{v_{10}\} \cup V_4 := \{v_2, v_3\} \cup V_5 := \{v_9\}$ obtained by the algorithm are called the path-monotonic decomposition of G. After the procedure, each $v \in V$ in the DAG $G = (V, E)$ satisfies:

$$vol(\pi(v_i)) > vol(\pi(v_j)) \implies \Pi(v_i) < \Pi(v_j) \tag{4.75}$$

Suspension Algorithm. To exemplify the proposed suspension-aware reservations, the example in Figure 4.17 is given. Each partition of the decomposition is attached to exactly one suspension-aware reservation. In ❶, only the subjob v_9 is tied to the reservation and since v_9 is not pending at time $t = 0$, the reservation suspends itself. Similarly, the reservations in ❷, ❸, ❹ suspend. At time $t = 7$, the subjob v_9 is released and the respective reservation resumes ❺. In the provided example, the reservation suffers from no external interference such that the pending subjob is serviced immediately. When v_9 is finished ❻, the reservation is paused. By this algorithm, the sustained service requirement that each subjob can claim the resources when necessary is still satisfied.

Suspension-Aware Reservation Provisioning. Unlike to the prior described ordinary reservation systems, in a path-monotonic decomposition, each partition requires its own reservation that it is tied to. That is, the number of required reservations is identical to the number of partitions in the decomposition implying $n = m < M$, where M is the number of available processors. Despite this

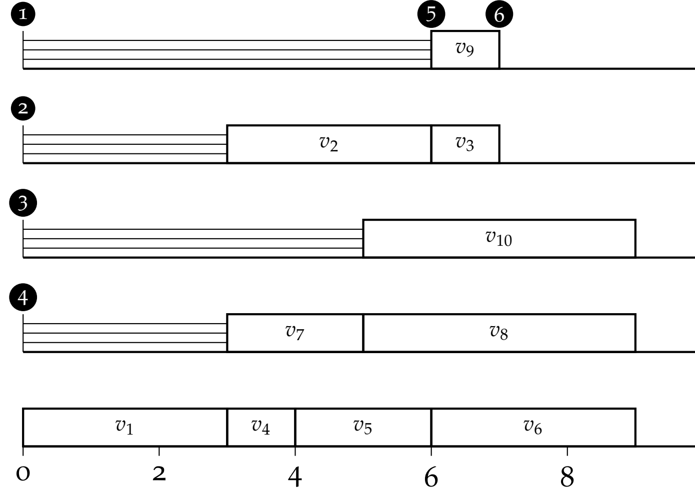

Figure 4.17: Exemplary schedule for the DAG illustrated in Figure 4.16 on five suspension-aware reservations for the path-monotonic decomposition $\{v_1, v_4, v_5, v_6\}, \{v_7, v_8\}, \{v_{10}\}, \{v_2, v_3\}, \{v_9\}$. The reservation that services the subjobs $\{v_1, v_4, v_5, v_6\}$ is assigned the lowest priority 1 and $\{v_9\}$ is assigned the highest priority 5.

disadvantage, please note that since *path-monotonic prioritization* policy is a specialization of the proposed *parallel-progress prioritization* the prior response-time analyses and reservation provisioning schemes (without suspension) are still applicable. Hence, Eq. (4.77) is an immediate corollary for the path-monotonic decomposition with $m = n$.

Definition 4.28 (Suspension-Aware Reservation System)**.** *A sporadic arbitrary-deadline suspension-aware reservation system for a path-monotonic decomposition of a DAG task τ_i, $V_1, \ldots, V_m$, is defined by*

$$\mathcal{O}_i^s := \left\{(E_i^1, S_i^1, D_i, T_i), \ldots, (E_i^m, S_i^m, D_i, T_i)\right\} \tag{4.76}$$

where the j-th reservation serves subjobs from V_j for $j \in \{1, \ldots, m\}$. Each reservation system serves exactly one *sporadic arbitrary-deadline DAG task τ_i. The reservation budget E_i^j denotes the promised service during a release and deadline interval provided by the j-th reservation and S_i^j denotes an upper-bound of cumulative self-suspension time of that reservation. The relative deadline D_i and minimal inter-arrival time T_i of each reservation is inherited by the to be serviced task.*

We devise the parameters of the suspension-aware reservation system $\mathcal{O}_i^s$ of a DAG task τ_i with path-monotonic decomposition $V_1, \ldots, V_m$, according to the following rules:

1. Calculate the budgets $E_i^1, \ldots, E_i^m$ such that $\forall j \in \{1, \ldots, m\}$ $vol(V_j) \leq E_i^j$ and

$$vol(\pi^*) + (m-1) \cdot D_\tau \leq \sum_{j=1}^{m} E_i^j \tag{4.77}$$

Table 4.6: A possible path-monotonic prioritization of the DAG illustrated in Figure 4.1.

SubJob	Paral	Priority	$vol(\pi(v_i))$
v_1	$\emptyset$	1	9
v_2	$v_4, v_5, v_6, v_7, v_8, v_9, v_{10}$	4	7
v_3	$v_4, v_5, v_6, v_7, v_8, v_9, v_{10}$	4	7
v_4	$v_2, v_3, v_7, v_8, v_{10}$	1	9
v_5	$v_2, v_3, v_7, v_8, v_{10}$	1	9
v_6	$v_2, v_3, v_9, v_8, v_{10}$	1	9
v_7	v_2, v_3, v_4, v_5, v_9	2	9
v_8	$v_2, v_3, v_4, v_5, v_9, v_6, v_{10}$	2	9
v_9	$v_2, v_3, v_6, v_7, v_8, v_{10}$	5	7
v_{10}	$v_2, v_3, v_4, v_5, v_9, v_6, v_8$	3	9

2. The budgets and suspension-times are calculated as $S_i^j := E_i^j - vol(V_j)$ and $E_i^j := vol(V_j)$.
3. The j-th reservation is assigned a fixed-priority according to the priorities of the subjobs in V_j as illustrated in the Table 4.6.

Scheduling Policy. We assume that the suspension-aware reservation system $\mathcal{O}_i^s$ is scheduled by global fixed-priority scheduling (G-FP) on M processors. The reason is, that in G-FP, at any time the M highest-priority jobs are scheduled, which implies that whenever a reservation, which services subjobs of a certain priority level is scheduled, all other reservations with higher-priority are guaranteed to be scheduled as well, if they have work pending and are thus not suspended. This property is required to ensure that the parallel path progression and the path-monotonic path progression property holds despite subjobs being tied to specific reservations. In the prior analyses, it was assumed that the highest-priority subjob can migrate to execute on any active reservation, which property is now equally ensured by the G-FP scheduling algorithm.

In the remainder of this section, the theorems and formal proofs for the path-monotonic progression is given.

Lemma 4.14. *Let a DAG G be prioritized according to the path-monotonic prioritization and decomposition policy then the decomposition with the lowest priority is the envelope path in a List-FP schedule S on $m \geq 1$ dedicated processors.*

Proof. We prove this theorem by cases and show that under our path-monotonic prioritization policy, it is not possible that the longest path does not compose the envelope.

Let $\pi_z := \{v_{z_1}, v_{z_2}, \ldots, v_{z_n}\}$ for some $n \in \{1, \ldots, |V|\}$ denote the subjobs of the longest path. Given a feasible *List-FP* schedule S, we construct the envelope of G according to the rules described in Definition 4.8 and identify that $\pi_e := \{v_{e_1}, v_{e_2}, \ldots, v_{e_p}\}$ for some p in $\{1, \ldots, |V|\}$ composes the envelope. Let $[a_{e_1}, f_{e_1}), [a_{e_2}, f_{e_2}), \ldots, [a_{e_p}, f_{e_p})$ and $[a_{z_1}, f_{z_1}), [a_{z_2}, f_{z_2}), \ldots, [a_{z_n}, f_{z_n})$ denote the arrival- and finishing time intervals of the respective subjobs in π_e and π_z and by assumption $\pi_e \neq \pi_z$. Without loss of generality we will assume that

the last subjobs are different, i.e., $v_{e_p} \neq v_{z_n}$ in the analysis. Otherwise, the analysis considers the paths $v_{e_1}, \ldots, v_{e_{p-i}}$ and $v_{z_1}, \ldots, v_{n-i}$ for the smallest $i \in \{1, \ldots, \min\{p-1, n-1\}\}$ such that $v_{e_{p-i}} \neq v_{z_{n-i}}$. Please note that the relative path-length differences are invariant to this truncation.

- For the *first case*, we assume that $\pi_e \cap \pi_z = \emptyset$. In this case, it must be that $f_{e_p} > f_{z_n}$ since otherwise v_{z_n} would be in the envelope due to the envelope definition that in case of equality of finishing times, the subjob with lower-priority is chosen. By the path-monotonic prioritization we know that priority $\Pi(v_{z_n}) < \Pi(v_{e_p})$. Moreover, it must be that the arrival times $a_{z_1} = a_{k_1}$, since otherwise at least one common predecessor must exist, which violates the case assumption. Let $exe(S, \pi_i, x, y)$ denote the amount of workload of subjobs in π_i that are executed in S during $[x, y)$. At any time t during the interval $[a_{z_1}, f_{z_n})$, subjobs of π_z and π_e are pending and ready for execution. By the prioritization it must be that whenever subjobs of π_z are executing then subjobs of π_e are executing as well, i.e.,

$$\begin{aligned} exe(S, \pi_e, a_{z_1}, f_{z_n}) &\geq exe(S, \pi_z, a_{z_1}, f_{z_n}) = vol(\pi_z) \\ vol(\pi_e) &> vol(\pi_z) \qquad (\text{since } f_{e_p} > f_{z_n}) \end{aligned}$$

  which contradicts the assumption that π_z is the longest path.

- In the *second case*, we assume that $\pi_e \cap \pi_z \neq \emptyset$, i.e., there exists at least one subjob that exists in both paths. Among all those subjobs that exist in both paths, we will use the subjob that finishes latest in S denoted by v_*. We split both paths at this subjob, i.e., $\pi_z = \{v_{z_1}, \ldots, v_*, \ldots, v_{z_n}\}$ and $\pi_e = \left\{v_{k_1}, \ldots, v_*, \ldots, v_{k_p}\right\}$. By the longest path property, it must be that $vol(\pi_e^*) \leq vol(\pi_z^*)$ holds for the sub paths $\pi_e^* := \left\{v_*, \ldots, v_{e_p}\right\}$ and $\pi_z^* := \{v_*, \ldots, v_{z_n}\}$. If that was not the case then, $\left\{v_{z_1}, \ldots, v_*, \ldots, v_{e_p}\right\}$ would be a longer path than π_z, which contradicts the assumption that π_z is the longest path.

  By the same argument as in the prior case, we have that $f_{e_p} > f_{z_n}$ and analyze the schedule S during the time interval that starts at a_* and finishes at f_{z_n}. At any time t during the interval $[a_*, f_{z_n})$, subjobs of π_z and π_e are pending and ready for execution. By the prioritization it must be that whenever subjobs of π_z are executing then subjobs of π_e are executing as well, i.e.,

$$\begin{aligned} exe(S, \pi_e^*, a_*, f_{z_n}) &\geq exe(S, \pi_z^*, a_*, f_{z_n}) = vol(\pi_z^*) \\ vol(\pi_k^*) &> vol(\pi_z^*) \end{aligned}$$

  which contradicts the assumption that π_z is the longest path.

□

By this lemma, we know that if the longest path π_z has the lowest priority then the envelope path at runtime is identical to the longest path π_z. This has the advantage that this information is available before runtime. With reference to the provided example in Figure 4.16, the path $\langle v_1, v_4, v_5, v_6 \rangle$ is the envelope path in any generated List-FP schedule. Interestingly, this principle can be extended as follows.

Definition 4.29 (Monotone Path Progression). *We say that the progression of a path π_i in a schedule S is monotone with respect to another path π_j if and only if $[a_{\pi_i}, f_{\pi_i}) \subseteq [a_{\pi_j}, f_{\pi_j})$, where a_{π_*} denotes the arrival time of the first job in the path π_* in S and f_{π_*} denotes the finishing time of the last job in that path.*

Theorem 4.15 (Monotone Parallel Path Progression). *Let $\psi := \{\pi_{\psi_1}, \ldots, \pi_{\psi_m}\}$ denote the paths that are extended from the path-monotonic decomposition of a DAG G. Then for any two paths π_{ψ_i} and π_{ψ_j} where $i < j$, a List-FP schedule S on m dedicated processors satisfies $[a_{\pi_{\psi_1}}, f_{\pi_{\psi_1}}) \supseteq [a_{\pi_{\psi_2}}, f_{\pi_{\psi_2}}) \supseteq \ldots \supseteq [a_{\pi_{\psi_n}}, f_{\pi_{\psi_n}})$.*

Proof. We will prove that $[a_{\pi_{\psi_1}}, f_{\pi_{\psi_1}}) \supseteq [a_{\pi_{\psi_2}}, f_{\pi_{\psi_2}}) \supseteq \ldots \supseteq [a_{\pi_{\psi_n}}, f_{\pi_{\psi_n}})$ iteratively. Let S_1 be a List-FP schedule of a given DAG job $G = (V, E)$ with subjobs $V = \{v_1, \ldots, v_\ell\}$. Let each subjob $v_k \in V$ have the arrival time a_k and finishing time f_k in the given *initial* schedule S_1.

By Lemma 4.14, we know that the envelope path in S_1 is given by the subjobs of the longest path π_{ψ_1}, i.e., the envelope is given by $[a_{1_1}, f_{1_1}), \ldots, [a_{1_n}, f_{1_n})$. We reduce this initial schedule S_1 in an iterative manner as follows:

- Calculate the envelope of S_i as described in Definition 4.8 and collect all subjobs that compose the envelope path denoted by π_e^i.
- We construct the reduced schedule S_{i+1} by removing all intervals $[a_k, f_k)$ from the schedule S_i that belong to subjobs $v_k \in \pi_e^i$ and $v_k \notin \{\pi_{\psi_{i+1}}\} \cup \ldots \cup \{\pi_{\psi_n}\}$ for $i \in \{0, \ldots, n-1\}$.

Since only subjobs with the lowest-priority, under the condition that they are not used in any other remaining higher-order path are removed in the reduction procedure, the schedule for the remaining subjobs remains unaltered. In particular, the only way that a lower-priority subjob can change the execution behaviour of a higher-priority subjob is due to precedence constraints, which is however inhibited by the reduction rule to retain all subjobs that are used by the remaining higher-order paths. In the reduced schedule S_{i+1} only those subjobs that are in either of the paths $\pi_{\psi_{i+1}}, \ldots, \pi_{\psi_n}$ remain in the reduced schedule.

All properties required to apply Lemma 4.14 to the reduced schedule S_{i+1} are met and thus by the same arguments $\pi_{\psi_{i+1}}$ is the envelope path π_e^{i+1} of S_{i+1}. Moreover, since in the reduction procedure subjobs are removed, we know that the last finishing time in the envelope of S_{i+1} is no more than the last finishing time of the envelope of S_i, i.e., $f_{\pi_{\psi_{i+1}}} \leq f_{\pi_{\psi_i}}$. Also it must hold that $a_{\pi_{\psi_{i+1}}} \geq a_{\pi_{\psi_i}}$, because if that was not the case, then v_{i+1_1} precedes v_{i_1}. This would contradict the assumption that π_{ψ_i} is the longest path in S_i, since the path could be prolonged by subjob v_{i+1_1}.

By repeating the argument for all $i \in \{1, \ldots, n-1\}$, the theorem is proved. □

The advantage of this approach is that $\pi_e^i := \pi_{\psi_i}$, i.e., the schedule dependent envelope path π_e^i of the reduced schedule S_i is given by the schedule independent path π_{ψ_i} which is known beforehand allowing to tie subtasks to reservations. On the downside however, in order to maintain the *monotone parallel path progression* property, early completions are forbidden, i.e., each subjob must execute for its worst-case execution time.

4.6 CONCLUSION

With reference to the hypothesis of this dissertation, the focus in this chapter is on the design and formal verification of real-time scheduling algorithms of fine-grained parallel task models on multicore architectures for which the derived theoretical guarantees are based on properties that are exposed as formal contracts that are to be honored. Those properties can be monitored and enforced in the systems in light of uncertainty, and can be used modularly to compose safe and tight analyses, as well as for optimization of the scheduler design, schedulability test problem, and other reliability characteristics of the real-time systems.

- Firstly, we studied the structural uncertainty of parallel applications that we modeled as probabilistic conditional DAG task (pC-DAG) and proposed a sustainable resource reservation system that allows to guarantee probabilistic upper bounds of k consecutive deadline misses. In addition, we provided an algorithm to optimize the reservations systems with respect to the above quantities and showed that resource usage for scheduling pC-DAGs – under k-consecutive deadline miss constraints – is significantly improved compared to conservative reservation systems. In the future we intend to improve the tightness of our proposed bounds and further evaluate the effectiveness of the approach in a real system.
- Secondly, in this chapter, we propose and analyze an intra-task prioritization that induces properties such as *parallel path progression* in any generated schedule that allow to improve the resource efficiency significantly for gang and ordinary reservation systems. On the basis of that property, we propose a sustainable scheduling algorithm and analysis that for hierarchical scheduling for gang-based and ordinary reservation systems for sporadic arbitrary-deadline DAG tasks. For these reservations, we provide algorithms that approximately provision optimal reservation systems with respect to the service they require. We evaluated our approach using synthetically generated DAG task sets and demonstrated that our approach can improve the state of the art in high-parallelism scenarios while demonstrating reasonable performance for low-parallelism scenarios. Moreover, we hint at how to improve the active idling issue of the proposed reservation systems with an additional *path-monotonic progression* property which admits a suspension-aware reservation design. In future work, we plan to improve and evaluate the suspension-aware reservation design and reclamation mechanisms more thoroughly.

5 REGULATOR-BASED ADAPTIVITY

Apart from temporal constraints, safety-critical real-time systems are often subjected to other constraints, such as robustness to soft errors, which are caused by transient faults. Transient faults are caused by environmental factors such as cosmic radiation and electromagnetic interference [Bau05]. The fault rates are non-negligible due to the increased sensitivity by the high integration density of modern system-on-a-chip. In consequence, safety-critical real-time systems must be designed with error-handling techniques to detect errors and allow for appropriate system recovery mechanisms if necessary.

Full error protection measures with hardware and software redundancy is often too costly in terms of resource usage to be considered a viable design option. To that end, *software-based fault-tolerance* techniques are a prominent choice, due to the ability to trade-off increased error protection with increased execution time. Then, by facilitating the different job variants, which provide different levels of protection against errors, the inherent robustness towards limited numbers of errors in many relevant safety-critical applications can be used to reduce resource usage at the cost of temporarily degraded quality of service (QoS).

software-based fault-tolerance

trade-off increased error protection with increased execution time

In this chapter, an adaptive state-based policy is presented. More precisely, an error-history based job variant selection strategy, which explicitly considers the error probability to choose an *optimal job variant* (to be released) with respect to the long-term average system utilization is presented. Above that, each reachable state, is verified to comply with weakly-hard error-constraints [1], and the deadline compliance of each task is guaranteed. In Section 5.1 *Motivation*, the objective, challenges, and application of our regulator-based adaptivity is thoroughly motivated, which is then formally introduced and explained in Section 5.4 *Automata-based Regulator* for the weakly-hard error constraints. In Section 5.2 *Related Work*, the relevant related work is presented. In Section 5.3 *System Model*, the studied system model is elaborated. In Section 5.5 *Reinforcement Learning Based Approach*, the reinforcement learning extension of our approach for unknown fault and error probabilities is presented. Lastly, all proposed approaches are evaluated in Section 5.6 *Evaluation*. In Section 5.7 *Conclusion*, the results of this chapter are summarized.

each reachable state, is verified to comply with weakly-hard error-constraints

5.1 MOTIVATION

Safety-critical real-time systems are often subjected to reliability constraints, such as limited numbers of soft errors, which are caused by transient faults. For instance, in a trajectory planning module, which is implemented by a collection

1 Weakly-hard refers to k-consecutive and (m, k) deadline miss constraints, but we use the term for soft errors here deliberately.

of tasks, using sensory data to produce trajectory data; transient faults may lead to soft errors, which could cause catastrophic consequences such as missing or faulty trajectories, leading to potentially catastrophic vehicle steering.

error-bound models

never more than k-consecutive jobs are erroneous

Many experimental research results have shown that errors in safety-critical real-time systems do not necessarily cause catastrophic system malfunction under the assumption of *error-bound models* such as either k-consecutive errors constraints or (m, k)-constraints. The former models requires that never more than k-consecutive jobs are erroneous, whereas the latter requires that at least m jobs out of any k consecutive jobs finish without error. For instance, Vreman et al. [VCM21; MHM+20] analyzed and showed control stability under k-consecutive deadline miss constraints. With regards to (m, k) constraints, it was shown that robotic applications can still successfully finish their tasks under a limited number of errors [CBC+16; YCC18]. While the original concept of (m, k)-constraints [HR95] was designed for allowing limited deadline misses [CKZ19; HQE20], the concept of (m, k)-constraints is equally applicable in the context of soft errors.

explicit output comparison

redundant multi-threading

job modes for each task are constructed, each of which provides different levels of assurance

detected mode

reliable mode

unreliable mode

Software-based fault-tolerance techniques such as *explicit output comparison* (EOC) [GGB13], control flow checking by using software signatures [OSM02], and *redundant multi-threading* [CBC18b] provide good flexibility, due to the capability to dynamically trade-off error protection with additional runtime. That is, different job modes for each task are constructed, each of which provides different levels of assurance. For instance, in the *detected* mode an error in the job can be detected, in the *reliable* mode an error can be detected and corrected, and in the *unreliable* mode errors are not detectable and no correction is provided. In turn, the worst-case execution times of the respective modes increase with the level of assurance, i.e., *unreliable* jobs have very short wort-case execution times, whereas *reliable* jobs have significantly larger worst-case execution times, and *detected* jobs have a larger worst-case execution time than *reliable* jobs.

weakly-hard soft error constraints

deeply red pattern

evenly distributed pattern

Weakly-hard soft error constraints allow to formally specify the conditions of robustness, i.e., the minimal required quality-of-service, which assures proper system function. By choice of the different job modes, these *weakly-hard soft error constraints* can be assured, e.g., to guarantee (m, k)-constraints. Most relevant techniques rely on static patterns like the *deeply red pattern* [KS95] (known as the R-pattern) or the *evenly distributed pattern* [QH00] (known as the E-pattern) to enforce reliable, i.e., error-free, executions.

While weakly-hard constraints increase the number of admissible system states, according to the specification, a consequential challenge for real-time systems is to guarantee timeliness for an increasing number of states. Since in hard real-time systems, each system state, i.e., the worst-case state, needs to be considered, increasing the state space is challenging the analysis precision. This is also the reason why static patterns such as the evenly distributed pattern – which evenly spreads reliable executions across any k-consecutive job releases – are a good choice with respect to worst-case response-time analyses of the task set. The job modes are selected in a static manner, such that the weakly-hard error constraints are satisfied for any possible sequence of actual errors observed in the system. And due to the static pattern, the worst-case interference in the response-time analysis can be precisely derived.

However, as errors are *rare events*, a strict worst-case provision of job variants to guarantee the k-consecutive or (m,k)-constraints is wasteful on system utilization as more reliable instances are instantiated than are actually necessary by observed errors in the system. Chen et al. [CBC+16] proposed a more adaptive approach by tracking the momentarily number of errors and to only choose a costly reliable job when necessary. Despite that approach allowing for some adaptivity, it might still lead to pessimistic resource usage, since the actual probability – that a job is erroneous – is not accounted for.

In the literature of fault tolerant systems, e.g., [CBC+16; NQ06; NZ20], one common objective is to minimize the system utilization in order to minimize the energy consumption, thermal strain, and response-times. In terms of energy consumption, the processor can be modeled by a *busy* (a job is executed) and an *idle* (nothing is executed) state which determines the energy consumption. In the *busy* state, the consumed power can be decomposed into *static* and *dynamic* power consumption. When the processor is idle, it (ideally) only consumes the static power. In addition, keeping the processor in a *busy* state for a long continuous interval can increase the processor's temperature, which results in higher energy consumption for cooling, and in turn, increases the static leakage power consumption [SLD+03; LDS+07] – due to the super linear relationship between temperature and static leakage power. Consequently, the power consumed in the idle state is much lower than in the busy state.

In terms of energy consumption, the processor can be modeled by a busy *(a job is executed) and an* idle *(nothing is executed) state which determines the energy consumption*

In spite of the challenges with respect to hard real-time constraints, weakly-hard error constraints, dynamic system evolution, and resource utilization, which must all be satisfied simultaneously; we propose the following contributions to address the motivated problems.

We propose a formalization of all *compliant system states*, and *compliant system evolutions*, using a *deterministic finite automata* (DFA). For this automata, the transition system depends on, the stochastic external cause for error, and the assurance level of a selected job mode in that state. Hence, the class of state-based selection policies, in which job modes with appropriate level of assurance are chosen, such that any system evolution is enforced to be within a feasible region, is defined. In consequence, any state in the feasible region complies with the weakly-hard error constraints.

deterministic finite automata

On the basis of this formalization, an optimization for resource utilization, is devised among the class of compliant state-based selection policies. That is, the objective is to choose job modes, such that any sequence of states remains within the feasible region, and the job mode selection minimizes the long term average worst-case execution time. This optimization is based on the current state and the error probability, which we solve analytically (for a known error probability), and by means of reinforcement learning if the error probability must be estimated during operation. Moreover, we ensure that any sequence of job modes is not worse than the static R-Pattern with respect to worst-case response-time analysis.

5.2 RELATED WORK

With regards to the design of automatic control systems, several controller design techniques have been proposed, which are capable of tolerating delayed [Ram99; KGC+12] or dropped signal samples [HSJ08; BS15; GDD19]. Hence, in the case of an erroneous control signal, that respective sample can be dropped and the prior sample is used for compensation [Ram99; HSJ08; BS15]. Another series of fault tolerance techniques, rely on the so-called (m, k) models, which was originally developed for guaranteeing limited deadline misses for firmed real-time systems, or so-called weakly-hard real-time systems [BBL01]. Under this constraint, a task has to meet at least m deadlines (or can miss at most m deadlines), in any sequence of k consecutive jobs, which has henceforth been used in several works, e.g., in [CKZ19; HQE20; SKT20; VPM+22]. Following, (m, k)-constraint models have been applied to the domain of fault tolerance, as well to describe the robustness of control systems, e.g., in [CBC+16; YCC18].

With regards to hard real-time schedulability for task sets with (m, k) deadline-miss constraints, several static patterns are widely applied for different purposes, i.e., the *deep red pattern* (R-pattern) [KS95], the *evenly distributed pattern* (E-pattern) [Ram99], and the *reverse E-pattern* [QH00]. A non-adaptive approach has been proposed by Von der Brüggen et al. in [BCH+16], in which work, it is determined if the system with dynamic real-time guarantees can provide full timing guarantees, or limited timing guarantees, without any online adaptation after fault occurrence.

Regarding adaptive and optimization based approaches, several results have been published. Chen et al. proposed an adaptive approach in [CBC+16], trying to minimize the overall execution time of a task by postponing the execution of safe, but time-consuming execution modes to the last possible point in time. Liang et al. in [LWJ+20] developed new methods and an optimization algorithm to analyze, and improve control stability and system schedulability, subjected to deadline misses and faults. The method is based on the application of two different fault-tolerance techniques, namely redundant execution, using EOC [GGB13] techniques, and re-executions in case of a soft error. With regards to energy-aware optimization, Al Enawy et al. proposed an on-line speed adjustment algorithm in [AA05]; exploiting the slack-time of skipped and completed jobs, to minimize the number of dynamic failures (in terms of (m, k)-firm deadline constraints), while remaining within an energy budget. Wang et al. in [WHK+21] presented a cross-layer approach to improve system adaptability by allowing proactive skipping of task executions. Huang et al. developed an online intermittent-control framework in [HXW+20], which combines formal verification with model-based optimization and deep reinforcement learning. The objective of the proposed framework is to opportunistically skip certain control computation and actuation to save actuation energy and computational resources without compromising system safety. Their focus is however the control safety rather than schedulability and (m, k) robustness.

5.3 SYSTEM MODEL

In this section, the studied system model is explained in detail and the model assumptions are elaborated on. At first in Section 5.3.1, the studied task system and task models are formally introduced and the respective job modes are defined. In Section 5.3.2, the underlying stochastic fault model is introduced and a per-job soft error probability is derived. Lastly, in Section 5.3.3, the hard real-time schedulability problem is stated with respect to our job mode selection strategy.

5.3.1 TASK MODEL

In this chapter, we consider a set of periodic constrained-deadline real-time tasks, i.e., $\mathbb{T} = \{\tau_1, \ldots, \tau_n\}$. Each task can release its jobs in the *reliable*, *detected*, *unreliable*, or the composite mode *detected+reliable*. The composite mode is the sequential execution of the *detected* mode immediately followed by the *reliable* mode, which completes early if no fault was detected. Each task is defined by a tuple $\tau_i := (C_i, D_i, T_i)$ where $C_i \in \{C_i^u, C_i^d, C_i^r, C_i^{d+r}\}$ denotes the set of worst-case execution times (WCETs) of the different job execution modes, i.e., the *unreliable* mode, *detected* mode, *reliable* mode, and *detected+reliable* mode. Throughout this chapter, we assume that for each task $\tau_i \in \mathbb{T}$ the relation $C_i^u < C_i^d < C_i^r < C^{d+r}$ holds, which is due to the additional overheads for *detected* and *reliable* modes, respectively. Moreover, we assume that the overhead for detection and recovery is integrated into the WCETs of the corresponding jobs' execution modes. With regards to the different execution modes, we assume that software-based fault tolerance techniques are used to detect and recover fault-induced soft errors. Each task is allowed to instantiate jobs in the *reliable*, *detected* or *unreliable* mode, with respective implications for soft-errors, and overheads, summarized in the following definition.

task can release its jobs in the reliable, detected, unreliable, *or the composite mode* detected+reliable

we assume that software-based fault tolerance techniques are used to detect and recover fault-induced soft errors

Definition 5.1 (Job Mode)**.** *Each job J_i^ℓ of each task $\tau_i \in \mathbb{T}$ can be executed in the following modes exclusively:*

- ***Unreliable.*** *In the unreliable mode, no additional code instrumentation or overhead is required, but in turn soft-errors are not detectable, i.e., no guarantee can be given at the end of that job's execution whether or not it is error-free. Hence, that job's return values and side-effects are not trustworthy.*
- ***Detected.*** *In the detected mode, the code is instrumented with techniques to verify the correctness of the executed job; e.g., error detection with special encoding of the data, control flow, sanity, or consistency checks [Pra86]. In the detected mode, errors can be detected, however that job's return values and side-effects are not trustworthy, regardless.*
- ***Reliable.*** *In the reliable mode, soft-errors must be detected – using techniques presented in the detected mode – and subsequently recovered or corrected. That is, in order to guarantee a correct result in the reliable mode, either a recovery routine can be issued to guarantee the job's correctness or task replication [HAZ17] can be applied to achieve high reliability. For instance m successive executions (replications) of a job yield a $1 - (p_e)^m$ probability to produce a correct result under*

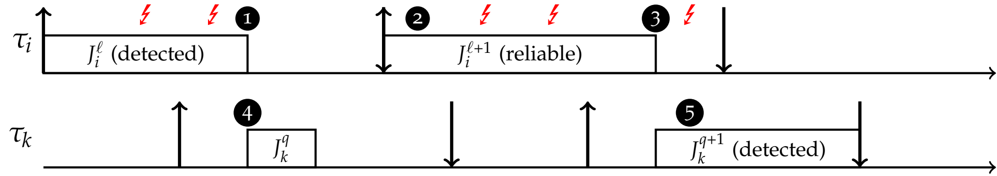

Figure 5.1: An exemplary schedule for two tasks $\tau_i, \tau_k \in \mathbb{T}$ that are subjected to faults, which result in soft-errors if at least one fault occurs during that jobs execution. Job J_k^q is executed in the unreliable mode.

our assumption of an independent error probability for a job of p_e. A probability of, e.g., $1-(p_e)^m \leq 10^{-9}$ is then considered reliable in spite of other system reliability estimates such as mean-time to failure. In consequence, it is guaranteed that the return values and side-effects are trustworthy.

- ***Detected + Reliable.*** *In the composite detected+reliable mode, an immediate compensation in the same instance – in case of a detected error after execution of the detected mode – is issued. In contrast to instantiating a reliable instance right away, conditionally executing a reliable instance only after an error was detected can reduce resource usage in very low error probability environments. In this mode, it is guaranteed that the return values and side-effects are trustworthy.*

red lightning symbols indicate faults induced by external cause

For better illustration, consider the exemplary schedule for two tasks shown in Figure 5.1. The red lightning symbols indicate faults induced by external cause, e.g., cosmic radiation, that lead to soft errors if a job executes during the time of fault if no reliable mode is executed. When the jobs of τ_i, namely $J_i^\ell, J_i^{\ell+1}$ and the jobs of τ_k, namely, J_k^q, J_k^{q+1} arrive, one of the job modes in Definition 5.1 is selected with the described effects and guarantees. Hence, the job modes are *controllable* system inputs, which together with the stochastic cause of faults, determine the manifestation of a soft-error in the respective job. In ❶, the finishing job J_i^ℓ is erroneous, since the detected mode provides no assurance beyond detection, and is subjected to two faults during its execution. Similarly, job J_k^{q+1} in ❺, is erroneous. In contrast in ❹, the job J_k^q executes in the reliable mode and is not subjected to any faults, and thus error-free. The reliable job $J_i^{\ell+1}$, which starts execution in ❷ and is error-free at the finishing time at ❸, despite being subjected to two faults during its execution. We assume that errors may take place at any time – under the poisson assumptions stated in the next section – during a job's execution without breaking the job's control flow.

5.3.2 FAULT AND ERROR MODEL

Transient faults can lead to soft errors, resulting in the incorrect results from the affected jobs. In this chapter, we assume that a job is erroneous if it is subjected to at least one fault during execution. Furthermore, we assume that the probability for each job to suffer from at least one fault to be upper-bounded by a single value

$$\mathbb{P}(\text{at least one fault occurred during execution of any job of task } \tau_i) \leq p_e < 1$$

(5.1)

can be obtained. This fault model assumption is suitable for poison processes, i.e., stochastic processes which satisfy the following properties:

fault model assumption is suitable for poison processes

1. In an interval $[t, t + \Delta t]$, there is at most one *fault event*, where Δt denotes a sufficiently small capture interval.

fault event

2. The probability that a fault event occurred during any interval of length Δt is proportional to Δt.
3. The occurrence of a fault event in an interval of length Δt does not depend on events in the past.

In the poisson model, it does not matter, when exactly a job is scheduled, but only the cumulative amount of time it is executed. Since, the execution time is upper bounded by the worst-case execution time, an upper-bound for Eq. (5.1), can be obtained. For such a fault process, the number of fault events, which can be observed in any interval of length t is poisson distributed with probability density function $\frac{\lambda^k e^{-\lambda t}}{k!}$ (for k-many faults). Consequently, the probability that at least one fault event is observed in any interval of length t, is given by $1 - e^{-\lambda t}$. Since it is safe and sufficient to estimate an upper-bound for p_e, we can define p_e for each task τ_i as $1 - e^{-\lambda \cdot C_i^d}$, where λ is the estimated environmental fault rate, and C_i^d is the worst-case execution time of a detected mode job. This is under the premise that each job executes for at most its worst-case execution time $C_i^u < C_i^d$ in the *unreliable* or *detected* mode, and the replica model for the *reliable* mode, in which each repeated detected mode instance, is also sensitive to faults for C_i^d time units. Hence, the probability that at least one fault occurs during the execution of the respective modes is given by $1 - e^{-\lambda \cdot C_i^u} \leq 1 - e^{-\lambda \cdot C_i^d} = p_e$, respectively. Henceforth, we assume that each job's error probability is constant for each task and upper-bounded by p_e. We will also only use p_e deliberately, without referring to the specific task, since all analyses are task-wise.

the number of fault events, which can be observed in any interval of length t is poisson distributed

5.3.3 SCHEDULABILITY AND SCHEDULING

In this work, we assume an arbitrary preemptive scheduling algorithm to schedule the task set, which satisfies the application's temporal requirements. That means, if tasks are subject to real-time constraints then a real-time capable scheduling algorithm such as rate-monotonic scheduling or EDF is used. If however soft real-time or best-effort is required then a suitable algorithm, e.g., EDF may be used. Please note, the objective of our work is the minimization of each task's average execution time under the corresponding k-consecutive errors and (m, k)-constraints and is orthogonal to the scheduling problem, in the sense that the scheduling decisions and the job mode selection are independent from one another. However, if the task set is subjected to real-time constraints then the approach to adopt the multi-frame task model to analyze the worst-case execution pattern as suggested by Chen et al. [CBC+16] can be used.

scheduling decisions and the job mode selection are independent from one another

Our proposed analytic solution, as well as the reinforcement learning (RL) based solution, generate sequences of job modes, which are never worse than

the job mode sequence as generated by the R-pattern [KS95] for the same (m,k)-constraint as well as the k-consecutive error constraint. In the former, the first $k-m$ jobs are executed in the *detected* mode and the remaining m instances are successively executed in the *reliable* mode, which is explained in more detail in the respective coming sections.

5.4 AUTOMATA-BASED REGULATOR

transitions of the DFA are in general triggered by events, which are beyond the control of the system

The conceptual foundation of this chapter is the automata-based regulator, where a deterministic finite automata (DFA) is used to formalize the state-space of erroneous jobs and the transitions between error states. The transitions of the DFA are in general triggered by events, which are beyond the control of the system such as radiation, which causes faults, and in turn cause an error in the respective job. We study the problem for a set $\mathbb{T}$ of periodic constrained-deadline tasks, where each task needs to satisfy weakly-hard error-constraints. Each task in the task set is considered individually, and each of the studied weakly-hard error-constraints, namely the k-consecutive error constraint and the (m,k)-constraint, are considered individually.

From here, common definitions and theoretic foundations, for both weakly-hard error-constraints are presented. In the subsequent Section 5.4.0.1, the k-error automata is formally introduced and examined as a common theoretical framework for both weakly-hard error constraints. Following, in Section 5.4.1, the analyses for k-consecutive error constraints are presented. In Section 5.4.2, the analyses concerning the (m,k)-constraints are presented.

5.4.0.1 *k-Error Automata Construction*

We indicate the correctness of the ℓ-th job J_i^ℓ of task τ_i by the *character* $c_\ell \in \Sigma$ and use a (possibly infinite) sequence of concatenated characters, i.e., a word $w = c_1 \circ c_2 \circ \ldots \circ c_n$ to indicate the correctness of the job sequence J_i^1 to J_i^n for $n \in \mathbb{N}$.

Definition 5.2 (Correctness Indication). *We indicate the correctness of a job at the end of its execution by an element of the set $\Sigma = \{0,1\}$. That is, an error-free executed job is indicated by a 1 and an erroneously executed job is denoted by a 0.*

For convenience, we denote the sub-word of w which starts at index a and ends at index b as $w(a,b) = c_a \circ \ldots \circ c_b$ for $a < b$. The $w(a,:)$ and $w(:,b)$ are used to denote the sub-word starting at index a to the end or from the beginning to the index b.

To verify the compliance of a task with respect to the weakly-hard error constraints of k-consecutive faults or (m,k) constraints, an infinite sequence of jobs, i.e., any sequence of k-consecutive jobs must be analyzed. While there are infinitely many sub-words (since there may be infinite job releases), there are only 2^k many different outcomes of interest $Q := \{00\ldots0, \ldots, 11\ldots1\}$ for which we define a k-error-automata.

Definition 5.3 (k-Error-Automata). *A k-error-automata $\mathcal{A}_k := (q_s, Q, \Sigma, \delta)$ is defined by a 4-tuple, where $Q := \{0,1\}^k$ denotes the finite set of states of all possible outcomes in any k consecutive job releases. The start $q_s := 11\ldots1 \in Q$ denotes the unique starting state, $\Sigma := \{0,1\}$ denotes the input alphabet, and δ defines the transition system $\delta : (Q, \Sigma) \mapsto Q$ such that for any state $q \in Q := \{00\ldots0, \ldots, 11\ldots1\}$*

$$\delta(q,0) = q(2,:) \circ 0 \in Q \tag{5.2}$$

$$\delta(q,1) = q(2,:) \circ 1 \in Q \tag{5.3}$$

An exemplary 3-error-automata $\mathcal{A}_3$ is illustrated in Figure 5.2. The connection between the schedule and the k-error automata is constructed as follows.

Definition 5.4 (Job Sequence Induced State). *Let any concrete word $w = c_1 \circ \ldots \circ c_\ell$ for $\ell \geq k$, which indicates the outcomes of all finished jobs. A sub-word of length k starting at the j-th job $w(j, j+k-1)$ for $j \in \{1, \ldots, \ell-k+1\}$ induces a state $q \in Q$ in the k-error-automata $\mathcal{A}_k$ denoted as $\psi(w(j, j+k-1)) = q \in Q$ if q's binary representation is identical to $w(j, j+k-1)$.*

As the word w – indicating the correctness of all finished jobs – evolves with the finishing of each released job, the state of the k-error-automata transitions accordingly. More precisely, let the $(j+k)$-th job finish at time f_{j+k} and let the sub-word $w(j, j+k-1)$ denote the latest k-consecutive job outcomes prior to time f_{j+k}. Based on the outcomes of the $(j+k)$-th job as indicated by c_{j+k}, the evolved job sequence induced state in $\mathcal{A}_k$ is given by $\delta(\psi(w(j, j+k-1)), c_{j+k})$. The outcome of the $(j+k)$-th job is determined by the occurrence of an error, which is assumed to be stochastic in nature and beyond our control. That is, under the assumption of a constant error probability p_e we have that $\mathbb{P}(c_{j+k} = 0) = p_e$ and conversely $\mathbb{P}(c_{j+k} = 1) = 1 - p_e$. We can however control the execution mode of the $(j+k)$-th job release, i.e., *unreliable*, *detected*, *reliable* or *detected* followed by *reliable*.

As described in Section 5.3.1, in the *unreliable mode*, the correctness of $(j+k)$-th job is given by $c_{j+k} = 0$ with probability 1. In *detected mode*, if a error occurred then $c_{j+k} = 0$ with probability p_e and $c_{j+k} = 1$ with probability $1 - p_e$ otherwise. In the *reliable mode*, the execution is guaranteed to be correct, i.e., $c_{j+k} = 1$ with probability 1. In the *detected followed by an optional reliable mode*, a reliable instance is only released if an error was detected and the current instance has to be correct in order to ensure the corresponding weakly-hard error constraint. In general, a sub set of states $Q^* \subset Q$ is called *compliant states* if the bit representation of each $q \in Q^*$ satisfies the respective weakly-hard error constraint, which is introduced more formally in later sections.

Our objective is to devise a state-based execution mode selection such that any infinite sequence of outcomes of jobs as indicated by an evolving word w is firstly compliant with the respective weakly-hard error constraints and secondly minimizes the expected execution time of each task. More precisely for any job sequence induced compliant state $\psi(w(j, j+k-1)) \in Q^*$ of $\mathcal{A}_k$, we devise a mode selection strategy

$$\alpha : Q^* \mapsto \{u, d, r, d+r\} \tag{5.4}$$

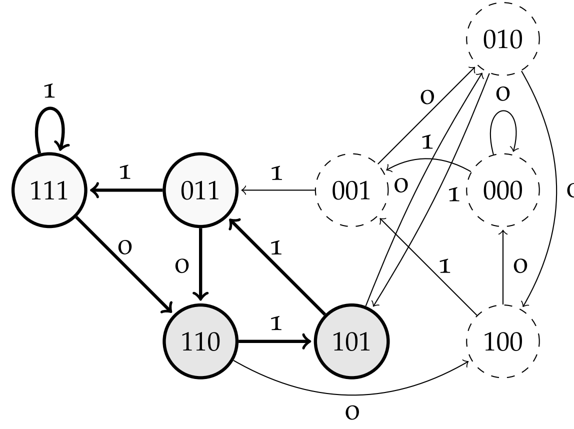

Figure 5.2: An exemplary k-error-automata $\mathcal{A}_k$ and the $(2,3)$-compliant automata $\mathcal{A}_k^*$ is highlighted in bold, where the darker states are *critical states* and the lighter states are *nominal states*.

to choose either an *unreliable*, *detected*, *reliable*, or a *detected* job optionally followed by a *reliable* instance for the $(j+k)$-th job release such that

$$\mathbb{P}(\psi(w(j+1, j+k)) \notin Q^* \mid \psi(w(j, j+k-1) \in Q^*, \alpha(\psi(w(j, j+k-1))) = 0 \quad (5.5)$$

for all $j \in \mathbb{N}$. For short, let $x_j := w(j, j+k-1)$ for some $j \in \mathbb{N}$ then

$$\mathbb{P}(c_{j+k} = 1 \mid \alpha(x_j)) = \begin{cases} 0 & \text{if } \alpha(x_j) = u \\ 1 - p_e & \text{if } \alpha(x_j) = d \\ 1 & \text{if } \alpha(x_j) = r \vee (d+r) \end{cases}$$

Conversely, $\mathbb{P}(c_{j+1} = 0 \mid \alpha(x_j)) = 1 - \mathbb{P}(c_{j+1} = 1 \mid \alpha(x_j))$. Please note that while the job may actually execute correctly even in *unreliable* mode, we have to consider it an error to guarantee compliance with the imposed weakly-hard error constraints, since the outcome is not observable. From a design perspective, we have to design the transition system of $\mathcal{A}_k$ such that only the compliant states Q^* are reachable, which is formally defined as follows.

Definition 5.5 (Compliant Transitions). *A transition system δ of a k-error-automata $\mathcal{A}_k$ is compliant if and only if for any given word w with $j \geq 1$ the following implication holds*

$$\psi(w(j, j+k-1)) \in Q^* \implies \delta(\psi(w(j, j+k-1)), c_{j+k}(\alpha)) \in Q^* \quad (5.6)$$

where Q^* denotes the compliant states.

critical state
nominal state

The set of compliant states can be further partitioned into *critical states* and *nominal states*. In a critical state, an error-free job execution is mandatory for the task to be compliant to its respective weakly-hard error constraint. Conversely, in the nominal state, any outcome is compliant with the constraint.

5.4.1 CONSECUTIVE-ERROR CONSTRAINTS

The automata-based regulator approach will be introduced in full detail for the more complicated (m,k)-constraint in the following Section 5.4.2, but the results

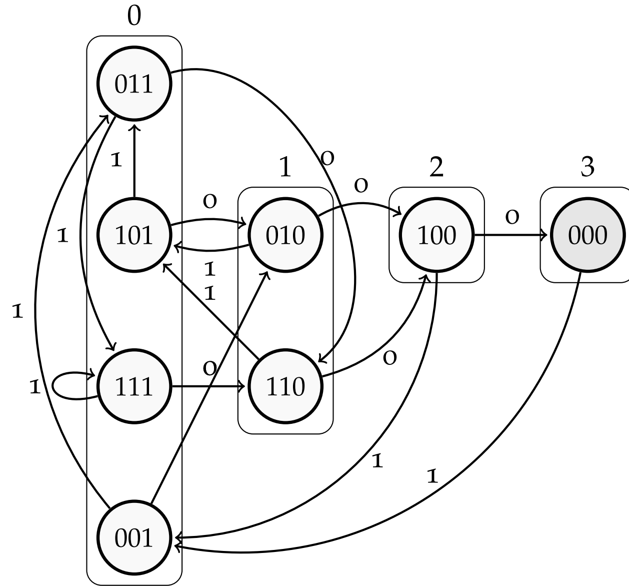

Figure 5.3: An exemplary k-error-automata $\mathcal{A}_k$ and the 3-consecutive error constraint compliant automata $\mathcal{A}_k^*$ is highlighted in bold, where the darker states are *critical states* and the lighter states are *nominal states*.

are equally applicable for the simpler k-consecutive error constraints, which are introduced hereinafter.

By construction of the k-error automata $\mathcal{A}_k$, each state $q \in Q^*$ satisfies the k-consecutive error constraint, however in the critical state a correct execution is mandatory for compliance, as illustrated in Figure 5.3. Moreover, it can be seen that each state $q \in Q^*$ can be partitioned into a superordinate state which counts the number of consecutive errors, i.e., $0, 1, 2$, or 3 in the provided example.

in the critical state a correct execution is mandatory

Definition 5.6 (Critical State)**.** *A compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a critical state with respect to k-consecutive error constraints if there are exactly k erroneously executed jobs in the word $w(j+1, j+k-1)$.*

Definition 5.7 (Nominal State)**.** *A compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a nominal state with respect to k-consecutive error constraints if there are at most $k-1$ consecutively erroneous jobs in $w(j+1, j+k-1)$ ending in $w(j+k-1)$.*

We will use the simpler problem structure of the k-consecutive error constraints to derive and prove an analytic solution to the expected number of executed jobs until the critical state is reached. This metric allows to assess the average quality-of-service which is attainable using our state-based approach for the k-consecutive error constraint and provides an intuition for the optimization potential. To that end, consider the simplified automata which consists of the superordinate states $q \in \{0, 1, \ldots, k\}$ counting the number of consecutive errors illustrated in Figure 5.4. The transitions are solely driven by the stochastic faults and subsequent errors which are realized with probability p_e.

expected number of executed jobs until the critical state is reached

Definition 5.8. *Let $\Omega(q)$ denote the expected number of executed jobs until the* critical state, *as defined in Definition 5.6, is reached – starting in the initial state q for any $q \in \{0, 1, \ldots, k\}$.*

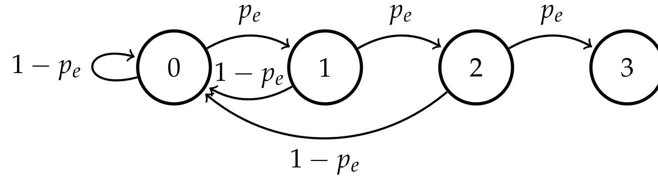

Figure 5.4: 3-consecutive error constraint compliant automata $\mathcal{A}_k^*$.

Since the expected value for a stochastic process with independent random variables, is the sum of each random variable's individual expected value, the following system of recurrence relations can be stated:

$$\begin{aligned}
\Omega(0) &= p_e \cdot (1 + \Omega(1)) + (1 - p_e) \cdot (1 + \Omega(0)) \\
\Omega(1) &= p_e \cdot (1 + \Omega(2)) + (1 - p_e) \cdot (1 + \Omega(0)) \\
&\vdots \\
\Omega(i) &= p_e \cdot (1 + \Omega(i+1)) + (1 - p_e) \cdot (1 + \Omega(0)) \\
&\vdots \\
\Omega(k) &= 0
\end{aligned} \tag{5.7}$$

where $\Omega(k) = 0$, since the *critical state* for k-consecutive errors is reached.

Based on this system of recurrence relations, the system of linear equations can be stated as $\Omega = A \cdot \Omega + [1, 1, \ldots, 0]^T$, which has a unique solution if $I - A$ is non-singular. In that case, the expected number of executed jobs – starting in the initial state $q = 0$ – is given by

$$\Omega(0) = \left((I - A)^{-1} \cdot [1, 1, \ldots, 0]^T\right) \cdot [1, 0, \ldots, 0]^T \tag{5.8}$$

It can be shown that $I - A$ in non-singular, which is stated and proven in the following theorem.

Theorem 5.1. *The matrix* $S := I - A$

$$\begin{bmatrix}
1 - p_e & -(1 - p_e) & 0 & 0 & \ldots & 0 \\
-p_e & 1 & -(1 - p_e) & 0 & 0 & 0 \\
-p_e & 0 & 1 & -(1 - p_e) & 0 & 0 \\
\vdots & 0 & 0 & \ddots & \ddots & \vdots \\
-p_e & 0 & 0 & 0 & 1 & -(1 - p_e) \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{5.9}$$

is non-singular under the premise that $p_e > 0$.

Proof. We prove that the kernel of S is trivial, i.e., if $\alpha \in ker(S) \implies \alpha = 0$ directly by solving $S \cdot [\alpha_1, \ldots, \alpha_n]^T$ on the basis of the special structure of S.

At first, we prove by induction over the row vectors (or more precisely the recursive structure) in S that

$$\alpha_{n-k} = \alpha_n \cdot (1 - p_e)^k + \alpha_1 \cdot \sum_{h=0}^{k-1} p(1 - p_e)^h \tag{5.10}$$

holds for all $k \in \{1, \ldots, n-2\}$, i.e., except for the first and last row.

Induction Base. For $k = 1$, reading the second-to-last row vector of S yields that $-p_e \cdot \alpha_1 + \alpha_{n-1} - (1-p_e) \cdot \alpha_n = 0$ must hold, which in turn yields

$$\alpha_{n-1} = (1-p_e) \cdot \alpha_n + \alpha_1 \cdot \sum_{h=0}^{1-1} p(1-p_e)^h \tag{5.11}$$

$$= (1-p_e) \cdot \alpha_n + \alpha_1 \cdot p_e \tag{5.12}$$

Induction Step. In the induction step, we advance in the row from $n-k$ to $(n-(k+1))$, i.e., $k \rightarrow k+1$, and read from the structure of S that

$$\alpha_{n-(k+1)} = p_e \cdot \alpha_1 + (1-p_e) \cdot \alpha_{n-k} \tag{5.13}$$

$$= p_e \cdot \alpha_1 + (1-p_e) \cdot (\alpha_n \cdot (1-p_e)^k + \alpha_1 \cdot \sum_{h=0}^{k-1} p(1-p_e)^h) \tag{5.14}$$

$$= \alpha_n \cdot (1-p_e)^{k+1} + \alpha_1 \cdot (p_e(1-p_e)^0 + (1-p_e) \cdot \sum_{h=0}^{k-1} p(1-p_e)^h) \tag{5.15}$$

$$= \alpha_n \cdot (1-p_e)^{k+1} + \alpha_1 \cdot \sum_{h=0}^{k} p_e(1-p_e)^h \tag{5.16}$$

In the second step, we inspect the first and last row individually, and conclude that $\alpha_n = 0$ and $\alpha_1 = \alpha_2$. Since from the previous step, we know that for $k \in \{1, \ldots, n-2\}$ Eq. (5.10) holds, we know that in particular for $k = 2$,

$$\alpha_2 = \alpha_1 \cdot \sum_{h=0}^{(n-2)-1} p_e(1-p_e)^h \tag{5.17}$$

holds. Since $\sum_{h=0}^{n-3} p_e(1-p_e)^h$ is non-zero by our assumption that $p_e > 0$ it must be that $\alpha_1 = \alpha_2 = 0$ and therefore $\alpha_{n-k} = 0$ for all $n-k \in \{3, \ldots, n-1\}$. □

It can be shown that for the k-consecutive error constraints, the expected number of executed jobs until the critical state is reached for the first time is given by

$$Eq.\ (5.8) = \Omega(0) = \sum_{h=1}^{k} \left(\frac{1}{p_e}\right)^h \tag{5.18}$$

where $p_e > 0$.

This metric is interesting, since it describes the system evolution if it is driven only by the stochastic occurrence of faults and subsequent errors. Notably, $\Omega(0)$ allows to assess the average quality-of-service which is attainable using our state-based approach for the k-consecutive error constraint.

Additionally, $\Omega(0)$ hints at the possible improvements of our approach with respect to the average worst-case execution time. For instance for an error probability p_e of 10%, the expected number of executed jobs to reach 2-consecutive errors is 110 and 1110 for 3-consecutive errors. From this, it is obvious that assuming worst-case scenarios and thus job mode selection strategies are highly wasteful. However, for very high error probabilities, e.g., 90%, the expected number of job executions to reach 3-consecutive errors is only 4. In these cases, there is less leeway to profit from the other job modes.

for an error probability p_e of 10%, the expected number of executed jobs to reach 2-consecutive errors is 110 and 1110 for 3-consecutive errors

5.4.2 (m,k)-constraints

While a k-error-automata models all error sequences in k consecutive jobs, not each sequence is (m, k) compliant. In order to verify if a task satisfies its (m, k) constraint after the finishing of the ℓ-th job given the indication c_ℓ, it must be tested if every sub-word of length k in $w = c_1 \circ \ldots \circ c_\ell$ for $\ell \geq k$ contains at least m correct executions.

Definition 5.9 (Critical State). *A compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a critical state with respect to (m, k)-constraints if there are only $(m-1)$ correctly executed jobs in the word $w(j+1, j+k-1)$, i.e., the latest previous $(k-1)$ jobs.*

Definition 5.10 (Nominal State). *A compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a nominal state with respect to (m, k) constraints if there are at least m correctly executed jobs among the latest previous $(k-1)$ jobs, i.e., $w(j+1, j+k-1)$.*

Definition 5.11 ((m, k)-Compliant State). *A state $q \in Q$ of a k-error-automata $\mathcal{A}_k$ is called (m, k)-compliant if $\mathbb{1}[q] \geq m$ is satisfied, where the operator $\mathbb{1}$ counts the number of 1's in q's representation. The set of all (m, k)-compliant states is called the (m, k)-compliant state-space denoted by $Q^* \subseteq Q$.*

It can be observed that in order for the transition system to be compliant, we have to enforce an outcome c_{j+k} based on whether $\psi(w(j, j+k-1)$ is a *critical* or *nominal state*. That is, if $\psi(w(j, j+k-1)) \in Q^*$ and *critical* then $c_{j+k} = 1$ must be enforced. In the case that $\psi(w(j, j+k-1)) \in Q^*$ and *nominal* then any $c_{j+k} \in \{0, 1\}$ is a feasible outcome. These observations are formalized in the following corollaries.

Corollary 5.2 (Critical State Transition). *If a compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a critical state then only a correct execution of the $(j+k)$-th job leads to a transition into a compliant state Q^*.*

Proof. The updated word after concatenation of c_{j+1} is given by $w(j+1, j+k)$, i.e., $w(j+1, j+k-1) \circ c_{j+1}$. By definition, the number of correct instances is given by $\mathbb{1}[w(j+1, j+k-1)] = m-1$. Clearly $|w(j+1, j+k)| = k$ and if $c_{j+1} = 0$ then $\mathbb{1}[w(j+1, j+k)] = m-1$ and $\mathbb{1}[w(j+1, j+k)] = m$ if c_{j+1}. □

Corollary 5.3 (Nominal State Transition). *If a compliant state $\psi(w(j, j+k-1)) \in Q^*$ is a nominal state then either execution outcome of the $(j+k)$-th job leads to a transition into a compliant state Q^*.*

Proof. The updated word after concatenation of c_{j+k} is given by $w(j+1, j+k)$, i.e., $w(j+1, j+k-1) \circ c_{j+k}$. By definition, the number of correct instances is given by $\mathbb{1}[w(j+1, j+k-1)] = m$. Clearly $|w(j+1, j+k)| = k$ and if $c_{j+k} = 0$ then $\mathbb{1}[w(j+1, j+k)] = m$ and $\mathbb{1}[w(j+1, j+k)] = m+1$ if c_{j+k} each of which complies with the (m, k) constraints. □

Based on these results, we can formulate properties which must be met by any feasible strategy.

Lemma 5.4 (Compliant Mapping Strategy). *Any mapping strategy α for the k-error-automata $\mathcal{A}_k$ which satisfies the constraints*

$$\alpha(\psi(x_j)) = \begin{cases} r \vee (d+r) & \text{if } \psi(x_j) \text{ is a critical state} \\ u \vee d \vee r & \text{if } \psi(x_j) \text{ is a nominal state} \end{cases} \tag{5.19}$$

leads almost certainly to a compliant transition system for $x_j := \psi(w(j, j+k-1)) \in Q^*$ for all j.

Proof. By the results of Corollary 5.2 and Corollary 5.3, we know that for any induced state $q := \psi(x_j) \in Q^*$, the strategy $\alpha(q)$ must almost certainly enforce a correct outcome of c_{j+k} if q is a critical state and any outcome if q is a critical state to lead to a compliant transition. Clearly, a *reliable* instance or a *detected* instance followed by an optional *reliable* instance in case of an error in case of q being a critical state enforces that $\mathbb{P}(c_{j+k} = 1 \mid \alpha(q)) = 1$. Conversely, if an *unreliable* or a *detected* instance is chosen if q is a nominal state then $\mathbb{P}(c_{j+k} = 0 \mid \alpha(q)) + \mathbb{P}(c_{j+k} = 1 \mid \alpha(q)) = 1$ which thus almost certainly leads to a compliant state. □

An α-induced (m,k)-compliant subset of a k-error-automata $\mathcal{A}_k$ is denoted by $\mathcal{A}_k^*(\alpha)$ and only contains compliant states $Q^* \subseteq Q$ and a compliant transition system $\delta^* \subseteq \delta$ such that for any $q \in Q^*$ the transition $\delta^*(q, c(\alpha(q))) \in Q^*$, which is exemplified in Figure 5.2.

5.4.3 STATES REDUCTION AND MINIMAL AUTOMATA

In the remainder of this section, we propose an algorithm to generate a minimal (m,k)-compliant automata $\mathcal{A}_k^*(\alpha)$, which is necessary to improve the computational complexity of our to be designed expected execution time minimization algorithms. We note that the approach to generate minimal finite-state machines as, e.g., used in Vreman et al. in [VPM22] is applicable for (m,k) constraints as well. However, their generation algorithm is similar to Hopcroft's algorithm [Hop71], which generates all states and merges *equivalent states*, whilst our Algorithm 7 utilizes the specificity of the problem to only generate compliant states right away.

equivalent states

Definition 5.12. *For given (m,k)-constraints, the set of n-step equivalent compliant states of the compliant k-error-automata $\mathcal{A}_k^*$ is given by*

$$[q]_n := \{q, q' \in Q^* \mid (\delta(q,w) = \delta(q',w)) \; \forall w \in \{0,1\}^n\}$$

and we say $q \sim_n q'$ if q and q' are n-step equivalent.

n-step equivalent

We use the *don't care* notation to denote the representative state $[q]_n$, e.g., $* \circ q(2,:) = * \circ q'(2,:)$ for 1-step equivalent states $q \sim_1 q'$ and $** \cdots * \circ q(n+1,:) = ** \cdots * \circ q'(n+1,:)$ for $q \sim_n q'$.

Lemma 5.5. *If there exist $q, q' \in Q^*$ such that $q \sim_{n+1} q'$ then there exist $v, v' \in Q^*$ such that $v \sim_n v'$ or conversely if there are no n-step equivalent states then there are no $(n+1)$-step equivalent states.*

Proof. We prove this lemma constructively, i.e., let $q \sim_{n+1} q'$ then $\delta(q,w) = \delta(q',w)$ for all $w \in \{0,1\}^{n+1}$, which is equivalent to $\delta(q, w(1) \circ w(2,:)) = \delta(\delta(q,w(1)), w(2,:))$ and thus $\delta(\delta(q,w(1)), w(2,:)) = \delta(\delta(q',w(1)), w(2,:))$. Let $v = \delta(q,w(1)) \in Q^*$ and $v' = \delta(q',w(1)) \in Q^*$ then due to the fact that $|w(2,:)| = n$ it must be that $v \sim_n v'$. □

From this lemma it follows that state equivalence must be constructed iteratively until no further n-step equivalent states can be generated from the set of $(n-1)$-step equivalent states for $n \geq 1$. We emphasize that we do not need to consider special constraints on w as e.g., only critical transitions exist for critical states, since only *nominal states* can be equivalent as shown in the following.

Lemma 5.6. *Only* nominal states *can be equivalent states in a compliant non-minimized automata $\mathcal{A}_k^*$.*

Proof. We prove by contradiction that only *nominal states* can be n-step equivalent. Assume that there exist any $q \sim_n q'$ such that q is a critical state and q' is a nominal state, i.e., by definition $\mathbb{1}[q(n+1,:)] = m-1$ and $\mathbb{1}[q(n+1,:)] \geq m$. Since q' is equivalent by assumption we have that $q'(n+1,:) = q(n+1,:)$ and thus $\mathbb{1}[q'(n+1,:)] = m-1$, which implies however that q' is not a nominal state and contradicts the assumption. □

Corollary 5.7. *The initial set of* nominal states *can be minimized to a set of representatives of the form $* \cdots * \circ v$ where v is the shortest v such that $\mathbb{1}[v] = m$ and the prior $k - |v|$ characters are* don't cares.

Proof. This follows from Lemma 5.5 and Lemma 5.6, since we know that states q, q' are merged up to n-step equivalence if $\mathbb{1}[q(n+1,:)] \geq m$ and thus $\mathbb{1}[* \cdots * \circ q(n+1,:)] \geq m$ where n is the maximal equivalence found and thus $v = q(n+1,:)$ $|v| = k - (n+1) + 1 = k - n$, i.e., shortest $|v|$. □

Theorem 5.8 (Minimal Automata)**.** *The minimal number of compliant states Q^* of a (m,k)-compliant $\mathcal{A}_k^*$ is given by*

$$|Q^*| = \frac{k!}{m! \times (k-m)!} \tag{5.20}$$

Proof. The number of compliant states is composed of *critical* and *nominal* states, where the number of *critical* states is given by $\binom{k-1}{m-1}$, since exactly the last $k-1$ characters in a *critical* state q must contain exactly $m-1$ ones.

From Lemma 5.6, we know that $m \leq |v| \leq k-m$ and thus states with $|v| = \ell$ and $\mathbb{1}[v] = m$ are merged into one representative state for $\ell \in \{m, m+1, \ldots, k-m\}$. The number of combinations for each above class is given by the binomial $\binom{\ell}{m}$. However, for each ℓ the number of combinations for $\ell - 1$ must be substracted. This is due to the fact that, by Lemma 5.5, we know that each state is represented by the maximal equivalence representative and the combinations with m ones in the last ℓ characters can be extended to combinations with m ones in the last $\ell + 1$,

Algorithm 7 Generation of minimal compliant $\mathcal{A}_k^*$

Require: Constraint (m, k);
Ensure: Minimal state automata $\mathcal{A}_k^*$;
1: $\mathcal{A}_k^* \leftarrow (q_s, Q^* := \emptyset, \delta := \emptyset, \Sigma := \{0, 1\})$;
2: $q_s \leftarrow \{* * \ldots 1\}$;
3: **for each** $z \in \{0, \ldots, k-m-1\}$ **do**
4: add $q := *_{k-m-z} \circ 1 \circ b(m+z-1, m-1)$ to Q^*;
5: add transition $\delta(q, 1) = q(2, :) \circ 1$ to δ;
6: add transition $\delta(q, 0) = q(2, :) \circ 0$ to δ;
7: **for each** $q \in \{w \in b(k-1, m-1) \mid 1 \circ w\}$ **do**
8: add q to Q^*;
9: add transition $\delta(q, 1) = q(2, :) \circ 1$ to δ;
10: **return** $\mathcal{A}_k^*$;

which should then be covered by the representative of ℓ. In consequence, we have that $|Q^*|$ is given by

$$\binom{k-1}{m-1} + \binom{m}{m} + \sum_{\ell=1}^{k-m} \binom{m+\ell}{m} - \binom{m+\ell-1}{m} = \binom{k}{m} \tag{5.21}$$

which proves the theorem. □

Let $b(z, n)$ denote all bit strings of length z with exactly n ones, which can be recursively defined and computed using dynamic programming. Using the above observations and lemmas, we can generate all *critical* states by $\{w \in b(k-1, m-1) \mid 1 \circ w\}$ and for each *critical* state q, we add a critical transition $\delta(q, 1) = q(2, :) \circ 1$. To generate the minimal set of *nominal* states for (m, k)-constraints, we generate the representatives iteratively using $*_\ell$ to denote a string of ℓ many $*$-characters as follows:

$$\bigcup_{z=0}^{k-m-1} *_{k-m-z} \circ 1 \circ b(m+z-1, m-1) \tag{5.22}$$

For instance, in the case of $(2, 4)$ constraints, the minimal *nominal* states are given by Eq. (5.22) as $* * \circ 1 \circ b(1, 1) = * * 11$, $* \circ 1 \circ b(2, 1) = \{*110, *101\}$. For each merged *critical* state q, the transitions $\delta(q, 0) = q(2, :) \circ 0$ and $\delta(q, 1) = q(2, :) \circ 1$ are added to the transition system of the automata.

5.4.4 MINIMIZATION OF EXPECTED EXECUTION TIME

In this section, we explain our mapping strategy of execution modes to jobs by considering different strategies for *critical* and *nominal* states in detail. Afterwards, an optimization strategy based on the so-called *induced markov chain* is proposed.

induced markov chain

5.4.4.1 *Mapping Strategy*

Our mapping strategy utilizes the design freedom of the different state categories to select the execution mode for next job as described in the following.

Critical State Action. If the current state q is a critical state then the next job has to be executed correctly and therefore either of the following two actions must be taken:

1. Release a *reliable* task instance, i.e., $\alpha(q) = r$.
2. Release a *detected* task instance and only release an immediate follow-up *reliable* task instance, in case of a detected error, i.e., $\alpha(q) = d + r$.

By this mapping, we have enforced that $c(\alpha(q)) = 1$ with probability 1. In the first case, the expected WCET of a job released in state q is either C_r or $(1 - p_e) \cdot C_d + p_e \cdot C_r$. It can be seen that for very low error probabilities p_e, it is better to first run a *detected* instance followed-up by a *reliable* instance.

Nominal State Action. If the current state q is a nominal state then the next job must not be enforced to be executed correctly. Thus, we have the following three options to choose the next job's mode:

1. Release a *reliable* mode instance, i.e., $\alpha(q) = r$ and $c(\alpha(q)) = 1$ with probability 1.
2. Release a *detected* mode instance, i.e., $\alpha(q) = d$ and $c(\alpha(q)) = 1$ with probability $1 - p_e$ and $c(\alpha(q)) = 0$ with probability p_e.
3. Release an *unreliable* mode instance, i.e., $\alpha(q) = u$ and $c(\alpha(q)) = 0$ with probability 1.

Due to the assumed high worst-case execution time of the *reliable* instances, we opt to select either a *detected* or an *unreliable* mode instance in each nominal state q at random. That is, we draw either a *detected* mode instance with probability p_d or an *unreliable* mode instance with probability p_u such that $p_d + p_u = 1$, and the expected average execution time is given by $p_d \cdot C_d + p_u \cdot C_u$. Based on the randomized mode selection, the transitions are stochastic in nature, i.e., $\mathbb{P}(c(\alpha(q)) = 1) = p_d \cdot (1 - p_e)$ and $\mathbb{P}(c(\alpha(q)) = 0) = p_d \cdot p_e + p_u$.

5.4.4.2 *Induced Markov Chain*

Using the mapping strategy α, we can derive an α-induced Markov Chain from the automata $\mathcal{A}_k^*$.

Observation 5.9 (Induced Markov Chain)**.** *The α-induced (m, k)-compliant $\mathcal{A}_k^*(\alpha)$ is a finite discrete-time Markov Chain with transition probability determined by the error probability and the mapping strategy α.*

Due to the state-based mode selection strategy, the probability of being in state q' at time $k+1$, i.e., $\mathbb{P}(x_{k+1} = q')$ only depends on the probability of being in a state q at time k for which $(q, q') \in \delta^*$ holds and the probability of taking a specific transition thereof. Therefore the markov property is trivially satisfied; the

specific transition probabilities are derived based on the error probability p_e and the stochastic state-based mode selection. That is, if q is a nominal state then, by our strategy α, the following state transitions are given:

specific transition probabilities are derived based on the error probability p_e and the stochastic state-based mode selection

$$\mathbb{P}(x_{n+1} = \delta(q, 0)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 0) = p_d \cdot p_e + p_u \quad (5.23)$$

$$\mathbb{P}(x_{n+1} = \delta(q, 1)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 1) = p_d \cdot (1 - p_e) \quad (5.24)$$

and for critical states

$$\mathbb{P}(x_{n+1} = \delta(q, 0)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 0) = 0 \quad (5.25)$$

$$\mathbb{P}(x_{n+1} = \delta(q, 1)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 1) = 1 \quad (5.26)$$

Definition 5.13 (Stationary Distribution)**.** *Let a finite, irreducible Markov Chain be given by $x_{n+1} = A \cdot x_n$, where $x_n \in Q^{*r}$, $A \in \mathbb{F}^{r \times r}$ and $||x_n||_1 = 1$ for all $n \in \mathbb{N}$ and $|Q^*| < \infty$. A probability distribution ξ is said to be a stationary distribution or invariant distribution if*

$$A \cdot \xi = \xi \quad (5.27)$$

We then obtain the corresponding stationary distribution ξ according to Eq. (5.27) by treating ξ as an eigenvector of A with an eigenvalue 1 which can be efficiently numerically solved by e.g., eigenvalue decomposition (spectral theorem [PTV+07]). Let the stationary distribution $\xi^T = [\xi_1, \ldots, \xi_r]$ where the ξ_i correspond to the stationary probability to be in state $q_i \in Q^*$. In consequence the expected average execution time $\mathbb{E}(C)$ is:

$$\sum_{i=1}^{r} \xi_i \cdot (p_d \cdot C_d + p_u \cdot C_u) \cdot [q_i \text{ is nominal}] + \xi_i \cdot \min\{C_r, (1 - p_e) \cdot C_d + p_e \cdot C_r\} \cdot [q_i \text{ is critical}] \quad (5.28)$$

Our formal objective is to minimize $\mathbb{E}(C)$ for each task individually with respect to the parameters p_d ($p_u = 1 - p_d$). What is left to show is that each α-induced (m, k)-compliant Markov Chain always has a stationary distribution.

Theorem 5.10 (Renewal Theorem [GS20])**.** *A finite, irreducible Markov Chain has a unique stationary distribution.*

Definition 5.14 (Irreducibility)**.** *A Markov Chain is irreducible if for any two states, i.e., q, q' there exist $n, n' \in \mathbb{N}_0$ such that $\mathbb{P}(x_{i+n} = q|x_i = q') > 0$ and $\mathbb{P}(x_{i+n'} = q'|x_i = q) > 0$ for some $i \in \mathbb{N}$.*

Theorem 5.11. *The α-induced (m, k)-compliant $\mathcal{A}_k^*(\alpha)$ is a finite and irreducible discrete-time Markov Chain.*

Proof. From Theorem 5.8, it immediately follows that $\mathcal{A}_k^*(\alpha)$ has finite states. Moreover, since the α-induced (m, k)-compliant $\mathcal{A}_k^*(\alpha)$ has non-zero probability for each transition by construction, we only have to prove that any two states q, q' are reachable from one another. We prove this theorem for the non-minimized automata, but since in the minimized automata only equivalent states are merged, it is obvious that the reachability property remains.

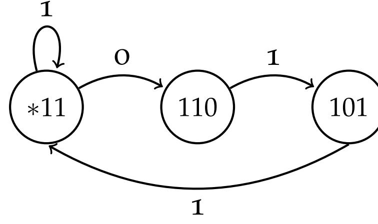

Figure 5.5: A minimal (2, 3)-compliant 3-error-automata $\mathcal{A}_3^*$ as generated by Algorithm 7.

Let q, q' any two states in the non-minimized α-induced Markov Chain $\mathcal{A}_k^*(\alpha)$ then there always exists a sequence of compliant transitions from $q \rightsquigarrow q'$ by decomposition of the transitions into $q \rightsquigarrow 11\dots1$ (k ones) and $11\dots1 \rightsquigarrow q'$. Since for each feasible state $q \in Q^*$ the transition $\delta(q, 1) \in Q^*$ is defined for *critical-* and *nominal states*, the state $11\dots1$ can be reached from any state q by successive 1-transitions. Secondly, for any given (m, k)-constraints, starting from state $11\dots1$, (by construction of the automata) we can use a 0-transition at most $(k-m)$ times and always be in a compliant state. This allows to reach any compliant state $q \in Q^*$ with $\mathbb{1}[q] \geq m$ to be reachable from $11\dots1$, since $q = \delta(11\dots1, q)$ and $\mathbb{1}[q] + [q] = k$ and thus $[q] = k - \mathbb{1}[q] \leq k - m$. □

5.4.4.3 *An Illustrative Example*

To illustrate our approach, we here provide a full example of the previously described task with $(m = 2, k = 3)$-constraints and assume that the execution times of the different job modes are given by $C^u = 1$, $C^d = 1.5$, and $C^r = 3$ and the task has an error probability of $p_e = 0.1$. After minimization according to Algorithm 7, the generated automata is shown in Figure 5.5 The ordered set of the states Q^* is given by $\langle Q^* \rangle = \langle *11, 110, 101 \rangle$ where $*11$ is a *nominal* state and $110, 101$ are *critical* states. By using the mapping strategy described in Section 5.4.4.1, we derive the following non-zero transition probabilities:

$$
\begin{aligned}
&\mathbb{P}(x_{n+1} = *11 | x_n = *11) = p_d \cdot (1 - p_e) = 0.9 \cdot p_d \\
&\mathbb{P}(x_{n+1} = 110 | x_n = *11) = p_e \cdot p_d + p_u = 1 - 0.9 \cdot p_d \\
&\mathbb{P}(x_{n+1} = 101 | x_n = 110) = 1 \\
&\mathbb{P}(x_{n+1} = *11 | x_n = 101) = 1
\end{aligned}
$$

The corresponding transition probability matrix A is:

$$
A = \begin{bmatrix} 0.9 \cdot p_d & 0 & 1 \\ 1 - 0.9 \cdot p_d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.29}
$$

where the row and column indexes are referring to the index of $\langle Q^* \rangle$. By solving the equation $\xi \in ker(A - I)$ such that $||\xi||_1 = 1$, we obtain the values $\xi_1 = 1/(3 - 1.8 \cdot p_d)$ and $\xi_2 = \xi_3 = (1 - 0.9 \cdot p_d)/(3 - 1.8 \cdot p_d)$, which yields the following expected average execution time $\xi_1 \cdot p_u \cdot C^u + \xi_1 \cdot p_d \cdot C^d + 2 \cdot \xi_2 \cdot \min\{C_r, C^d + p_e \cdot C^r\}$ and evaluates to

$$\frac{p_u \cdot C^u + p_d \cdot C^d + (1 - 0.9 \cdot p_d) \cdot \min\left\{C_r, C^d + 0.9 \cdot C^r\right\}}{3 - 1.8 \cdot p_d}$$

$$\implies \frac{230 - 137 \cdot p_d}{150 - 90 \cdot p_d} \text{ for any } p_d \in [0, 1] \tag{5.30}$$

The function in Eq. (5.30) is monotonically increasing on the interval $[0, 1]$, which implies that the minimum value of the expected average execution time is attained for $p_d = 0$. Therefore, in every *nominal state* q the mapping is given by $\alpha(q) = u$, i.e., to always instantiate an unreliable instance next. Moreover, for each *critical state* q we always select $\alpha(q) = d + r$, since $C_r > C_d + p_e \cdot C_r$ in the provided example. In summary, we obtain the following mapping strategy:

$$\alpha(q) = \begin{cases} d + r & \text{if } q \in \{101, 110\} \\ u & \text{if } q \in \{*11\} \end{cases} \tag{5.31}$$

which results in a minimal expected average execution time of 1.533. However, if we decrease the error probability to 1%, i.e., $p_e = 0.01$, the expected average execution time becomes

$$\frac{1150 - 766 \cdot p_d}{750 - 495 \cdot p_d} \text{ for any } p_d \in [0, 1] \tag{5.32}$$

In contrast to Eq. (5.30), Eq. (5.32) is monotonically decreasing on the interval $[0, 1]$ and thus the minimum expected average execution time is attained when $p_d = 1$. This results in the altered strategy to select $\alpha(q) = d$ for any *nominal* state, but remains the same for the *critical state* since $C_r > C_d + p_e \cdot C_r$.

5.5 REINFORCEMENT LEARNING BASED APPROACH

In the previous approach, it was assumed that the error probability for each task is known, and does not change significantly over time. In cases that the error probability for each task is unknown, the approach proposed in Section 5.4.4 is inapplicable. To that end, we propose an adaptive regulator, which is based on *reinforcement learning* (RL) to optimize the job mode selection adaptively during runtime.

reinforcement learning

At first in this section, a short overview of RL techniques is given. Afterwards, it is examined and explained, how the optimal job mode selection policy subjected to (m, k)-error constraints, is formulated as an RL-solvable problem. Moreover, a barrier function, which assures the task's compliance to its (m, k)-constraint is discussed.

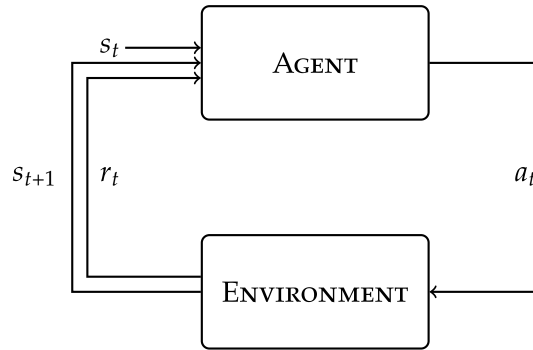

Figure 5.6: A schematic of a Markov Decision Process (MDP) which consists of an *agent* and the *environment*. The *environment* E denotes the state space $s_t \in S$, which is a representation of the environment at time t. In a markov process, the next state s_{t+1} depends solely on the current state s_t. Reinforcement learning is a framework to train an *agent* to interact with the environment by means of *actions*, which influence the state transitions.

5.5.0.1 *Overview of Reinforcement Learning*

markov decision process
environment
agent
action space
reward function

Reinforcement learning is a machine learning paradigm which can be reformulated as a *Markov Decision Process* (MDP), which is illustrated in Figure 5.6. The *environment* E is defined as a state space wit an iteration-dependent state $s_t \in S$, representing the environment in the t-th iteration. The *agent* implements the learned policy, by selection of an action $a_t \in A$ from a set of possible actions, called the *action space*. Each action induced state transition is evaluated with a reward r_t according to a *reward function* $\mathbb{H}$. Therefore, the reinforcement learning problem is defined by $\mathbb{P} : S \times A \times S \mapsto \mathbb{R}$ which represents the probability $\mathbb{P}(s_{t+1}|s_t, a_t)$, and $\mathbb{H} : S \times A \times S \mapsto \mathbb{R}$ denotes that the reward for the system state s_{t+1} given the prior state s_t and the action a_t. According to the markov property, the next state s_{t+1} only depends on the current state s_t and current action a_t, and is conditionally independent to all previous states and actions.

learned policy

The objective of the reinforcement learning procedure is to learn a *policy* π such that an action is proposed given a current state, i.e., a state-based regulator. A learned policy π can be deterministic or stochastic, i.e.;

- a deterministic policy $\pi : S \mapsto A$ is a unique mapping from state to action
- and a uniform stochastic policy $\pi : S \times A \mapsto A$ defines the possible policy, using a probability distribution of actions, i.e., $\pi(a_t|s_t) = \mathbb{P}(a = a_t \mid s = s_t)$, subject to $\sum_{a_t \in A} \pi(a_t|s_t) = 1$.

cumulative future reward
discount rate

The learned policy is evaluated by the *cumulative future reward*, i.e., $W_t = \sum_{i=t}^{\infty} r_i$. To consider that future states are less predictable and thus the future reward is often less valuable in the learning process than the present reward, a *discount rate* $\gamma \in (0, 1]$ is included, i.e., $W_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$.

action value function

To estimate the expected future reward, a so-called state value function is applied, which is defined as $V_\pi(s_t) = \mathbb{E}_A[Q_\pi(s_t, A)]$ and yields the expected cumulative reward starting in state s_t and by application of policy π. Similarly, an *action value function* for a policy π is defined as $Q_\pi(s_t, a_t) = \mathbb{E}[W_t|s = s_t, a = a_t]$,

which indicates the quality of the action a_t given the state s_t.

Value-based Reinforcement Learning. In *value-based reinforcement learning*, an action value function for a policy π is defined as $Q_\pi(s_t, a_t) = \mathbb{E}[W_t | s = s_t, a = a_t]$. The optimal action value function is defined as $Q^*(s_t, a_t) = \max_\pi Q_\pi(s_t, a_t)$, where $Q^*(s_t, a_t)$ indicates the quality of the action a_t under state s_t, which is independent from the policy π. The best action to select in state s_t is given by $a_t^* = \arg\max_a Q^*(s_t, a)$, but $Q^*(s, a)$ is unknown for the agent. Therefore, a *deep Q-Network* (DQN) is trained to approximate $Q^*(s, a)$, which is also known as value-based reinforcement learning. The DQN is denoted as $Q(s, a, \omega)$, where the s is the input state, a is the given action and ω are the parameters of the DQN. At each step t, all the possible actions are evaluated by the DQN, and the agent takes the action with the highest expected cumulative reward, i.e., $a_t^* = \arg\max_a Q^*(s_t, a)$. Afterwards, the state transitions from s_t to s_{t+1} and the procedure is repeated until the end of the process. The DQN can be trained using the *temporal difference (TD) learning* [Tes95], which adjusts the prediction of the value function (model's estimate) after each iteration, i.e., $Q(s_t, a_t, \omega) \approx r_t + \gamma Q(s_{t+1}, a_{t+1}, \omega)$. TD learning is more efficient than traditional *monte carlo methods*, where the estimates are adjusted only when the final outcome is known.

value-based reinforcement learning

deep Q-Network

temporal difference learning

monte carlo methods

Policy-based Reinforcement Learning. The state value function is defined as $V_\pi(s_t) = \mathbb{E}_A[Q_\pi(s_t, A)]$, which equals to $\sum_{a \in A} \pi(a|s_t) \cdot Q_\pi(s_t, a)$ when actions are finite and discrete. $V_\pi(s)$ defines the expected cumulative reward from state s_t by applying policy π. To achieve a higher expected cumulative reward, the policy network $\pi(a|s; \theta)$ is formulated for approximating $\pi(a|s)$, where θ is a parameter of the neural network. Hence, the state value function can be reformulated as $V_\pi(s_t; \theta) = \sum_{a \in A} \pi(a|s_t; \theta) \cdot Q_\pi(s_t, a)$. The objective is to train the (deep) neural network to map the current state to the best probabilistic action to take, i.e., the action which yields the the highest expected state value. The *policy gradient ascent approach* [SKM00] can be applied to train the policy network by maximizing $\mathbb{E}_S[V(S; \theta)]$.

objective is to train the (deep) neural network to map the current state to the best probabilistic action to take

policy gradient ascent

Actor-Critic Optimization. Besides the aforementioned value-based and policy-based approach, the actor-critic method can also be applied to train a reinforcement learning model. The actor-critic method uses the policy network (actor) to approximate $\pi(a|s)$, and the value network (critic) to approximate $Q_\pi(s, a)$ simultaneously. Therefore, $V_\pi(s_t) = \sum_{a \in A} \pi(a|s_t) \cdot Q_\pi(s_t, a)$
$\approx \sum_{a \in A} \pi(a|s_t; \theta) \cdot Q_\pi(s_t, a; \omega)$. The policy network $\pi(a|s_t; \theta)$ is trained to increase the state value $V_\pi(s_t; \theta, \omega)$, and the value network $Q(s_t, a_t, \omega)$ is trained to estimate the expected cumulative reward more precisely.

Monte Carlo Methods-based approaches. Monte Carlo (MC) methods are a broad class of algorithms, which rely on repeated random sampling to obtain an estimate. With regards to reinforcement learning, MC methods are applied to estimate the real value of a state by averaging the obtained results from sampling complete episodes for several times, where an episode denotes all states in between the current state and some terminal state by following the current policy. For a given initial state, the estimated state value is utilized to update the action value function $Q_\pi(s_t, a_t)$ and afterwards, an $\epsilon - greedy$ algorithm can be

J_1	J_2	J_3	J_4	J_5
1	0	1	0	1
1	0	1	1	2
C^d	C^u	C^d	C^d	C^r
C^d	C^u	C^d	C^d	C^r
s_t^1	s_t^2	s_t^3	s_t^4	s_t^5

J_2	J_3	J_4	J_5	J_6
0	1	0	1	1
0	1	1	2	1
C^u	C^d	C^d	C^r	C^d
C^u	C^d	C^d	C^r	$C^d + C^r$
s_{t+1}^1	s_{t+1}^2	s_{t+1}^3	s_{t+1}^4	s_{t+1}^5

Figure 5.7: Exemplary *environment* state transition from s_t to s_{t+1} for $(3, 5)$ constraints, where the j-th column vector refers to the status of the $(5 - (j - 1))$-th latest executed job for $j \in \{1, \ldots, 5\}$.

applied to optimize the policy $\pi(a|s)$. The above steps are repeat until $Q_\pi(s_t, a_t)$ converges.

5.5.0.2 *Reinformcent Learning based Regulator Problem Formulation*

For each task in the task set, an agent is trained, which acts independently from the other agents, i.e., this approach is not to be mistaken with the multi-agents reinforcement learning problem. More precisely, each agent learns a policy, based on its own observations, i.e., the correctness of the job executions of that task, only. In the following, we examine the job mode selection policy for a single task, which can be similarly applied for the other tasks in the task set.

environment *state represents the execution status for a task's jobs*

The *action space A* for each task is encoded as 0 for the *unreliable* job mode, 1 for the *detected* job mode, and 2 for the *reliable* job mode. To state the reinforcement learning based regulator problem, the *environment's* state must be defined. The *environment* state represents the execution status for a task's jobs, e.g., the status history of the last ℓ jobs $s_t = [s_t^1, s_t^2, \ldots, s_t^\ell]$, where the execution status s_t^j for some $j \in \{1, \ldots, \ell\}$ is a 4-ary vector with the following attributes:

- *Correctness*: A binary variable, which indicates the correctness of the corresponding job. In accordance to Definition 5.2, an erroneous executed job is indicated by a 0 and a correct execution is indicated by a 1.
- *Execution Mode*: The execution mode of a job, i.e., 0 denotes the *unreliable* mode, 1 denotes the *detected* mode, and 2 denotes the *reliable* mode.
- *Predicted Worst-Case Execution Time*: The predicted worst-case execution time of a job is given by the worst-case execution time of the predicted job execution mode to satisfy the (m, k) constraint, i.e., either C^u, C^d, or C^r.
- *Effective Worst-Case Execution time*: In contrast, the effective worst-case execution time of a job denotes the cumulative required worst-case execution time to finish a job – as required to satisfy the (m, k) constraint – in the schedule during runtime. That is, the effective worst-case execution time of a job may be $C^d + C^r$, in the case that a job is forced to be finish correctly according to the (m, k) constraint, and the predicted *detected* job mode has detected an error requiring to immediately execute a *reliable* job mode. Please note that for both, the *unreliable* and *reliable* mode, the predicted and effective worst-case execution time are identical.

In order to be able to assure (m, k) constraints, at least k consecutive jobs must be collected in the state vector, but may contain more for additional information.

at least k consecutive jobs must be collected in the state vector

Example. An example is shown in Figure 5.7 for a task with $(m = 3, k = 5)$ constraint and the number of considered states is set to $\ell = 5$. In the transition from state s_t to s_{t+1}, the *detected* job mode is selected, i.e., the action $a_t = 1$ is chosen for the next to be released job – which is the 6-th job J^6 in this example. Therefore, the execution mode and predicted worst-case execution time of job J^6 are determined as 1 and C^d, respectively. By inspection of the $(m = 3, k = 5)$ constraint in the first row vector, the job J^6 has to be executed correctly in order to satisfy the $(3, 5)$ constraints, and is therefore indicated by a 1. However, during the actual execution, J^6 was detected to be erroneous and thus an additional *reliable* job mode has to be executed immediately thereafter. In consequence, the effective worst-case execution time of J^6 in state s_{t+1} is found to be $C^d + C^r$. In the transition from state s_t to s_{t+1}, the oldest job is displaced. Please note that different environment construction approaches can also be applied such as a three dimensional tensor which records several latest two dimensional matrices such that more information can be recorded without displacement.

different environment construction approaches can also be applied such as a three dimensional tensor

5.5.0.3 *Deep Q-Network Reinforcement Learning Policy*

The presented reinforcement learning based approach to train a job mode selection policy is in general not limited to any specific learning policy. That is, any learning approach which supports discrete action spaces is applicable. Here, the deep Q-network (DQN) agent is used, where the *Boltzmann Q-Policy* is applied to estimate the Q value of each action. We constructed the DQN agent as a 10-layer neural network, which contains 1 input layer, 1 activation layer, 1 flatten layer, 6 fully connected layers, and 1 output layer.

Boltzmann Q-Policy

5.5.0.4 *Barrier Function*

To achieve the objective of minimizing the average execution time for each task in spite of being subjected to the (m, k)-constraint, the reward function is inversely proportional to a job's execution time and must be checked by a *barrier function* which ensures the compliant mapping strategy properties stated in Lemma 5.4.

barrier function

Hence, the barrier function determines, whether the current state is a critical or nominal state, to modify the reward function. That is, if the current state $s_t \in S_{nom}$ then the reward function is evaluated without further considerations. If however the current state $s_t \in S_{crt}$, then the next job execution must be error-free and the barrier function only admits the *detected+reliable* mode to be chosen. In addition, an extremely large negative reward value is fed back to the agent. In consequence, the barrier function is able to guarantee the satisfaction of (m, k) constraints and lets the agent learn not to select the *unreliable* mode, when being in a critical state.

Although the proposed agent is model free, the barrier function implicitly enforces the generation of the R-pattern in the worst case. The worst-case execution pattern of the RL-based approach is the same as the R-pattern adopted in [CBC+16; KS95], which consists of $k - m$ jobs that are executed in *detected*

mode, followed by m consecutive executions in the *detected+reliable* job mode – irrespective of the actual observed errors.

By construction of the barrier function, it can be seen that the worst-case generated sequence of job modes of our RL-based approach occurs if the agent always decides to execute a *detected* mode job and all jobs are erroneous. Due to the enforcement of the barrier function, this can happen for at most $k - m$ jobs and must be followed by k *detected+reliable* jobs. As an immediate consequence of the above considerations, it follows that if the task set is feasibly schedulable according to the schedulability test stated in Lemma 1 in [CBC+16] under the presumption of the static R-Pattern then it remains feasible using the reinforcement learning based job mode selection policies.

5.6 EVALUATION

To evaluate the effectiveness of our proposed approaches, we numerically simulate the task system and compare the performance of the proposed mapping strategy when p_e is known and the RL-based approach when p_e is unknown with the state of the art over a wide range of different configurations. The adopted hardware platform was a cache-coherent SMP, consisting of one 64-bit Intel i7-8700k processors running at 3.7 GHz, with 32 GB of main memory. Overall, the following approaches are evaluated, namely:

- The optimal policy presented in Section 5.4.4 (OPT).
- The RL-based approach presented in Section 5.5 (RL) where the *environment* is constructed with the help of OpenAI Gym [BCP+16] and the implementation of the RL agent is based on the Keras-rl package [Pla16] and TensorFlow [Aba+15].
- The adaptive approach (ADP) [CBC+16] in which the reliable job mode executions are postponed as much as possible.
- The static approach (STA) [NQ06], in which m jobs are executed in the *reliable* mode and the consecutive $(k - m)$ jobs in the *unreliable* mode.

Single-Task Evaluation. We conducted evaluations for one single task with different experimental settings, e.g., (m, k) constraints and error probabilities. With respect to the (m, k) constraint, the number of correct jobs m was selected from the set $m \in \{2, 4, 6, 8\}$ and the window length k was set to 10, and the error probability was given as $p_e \in \{0.05, 0.15, 0.3\}$. We set $C^d = 1.5 \cdot C^u$ and $C^r = 3.5 \cdot C^u$ to emulate the software-based error detection and error recovery induced overheads. Each task released 10.000 jobs for one iteration, and 100 iterations were performed.

In Figure 5.8, the results for one single tasks are illustrated for varying m and error probabilities p_e. The y-axis represents the normalized average execution time for jobs of one task, with respect to the static approach *STA*, i.e., the lower the better. In general, the *OPT* approach outperforms all the other approaches in all the evaluated cases.

In general, the OPT *approach outperforms all the other approaches in all the evaluated cases*

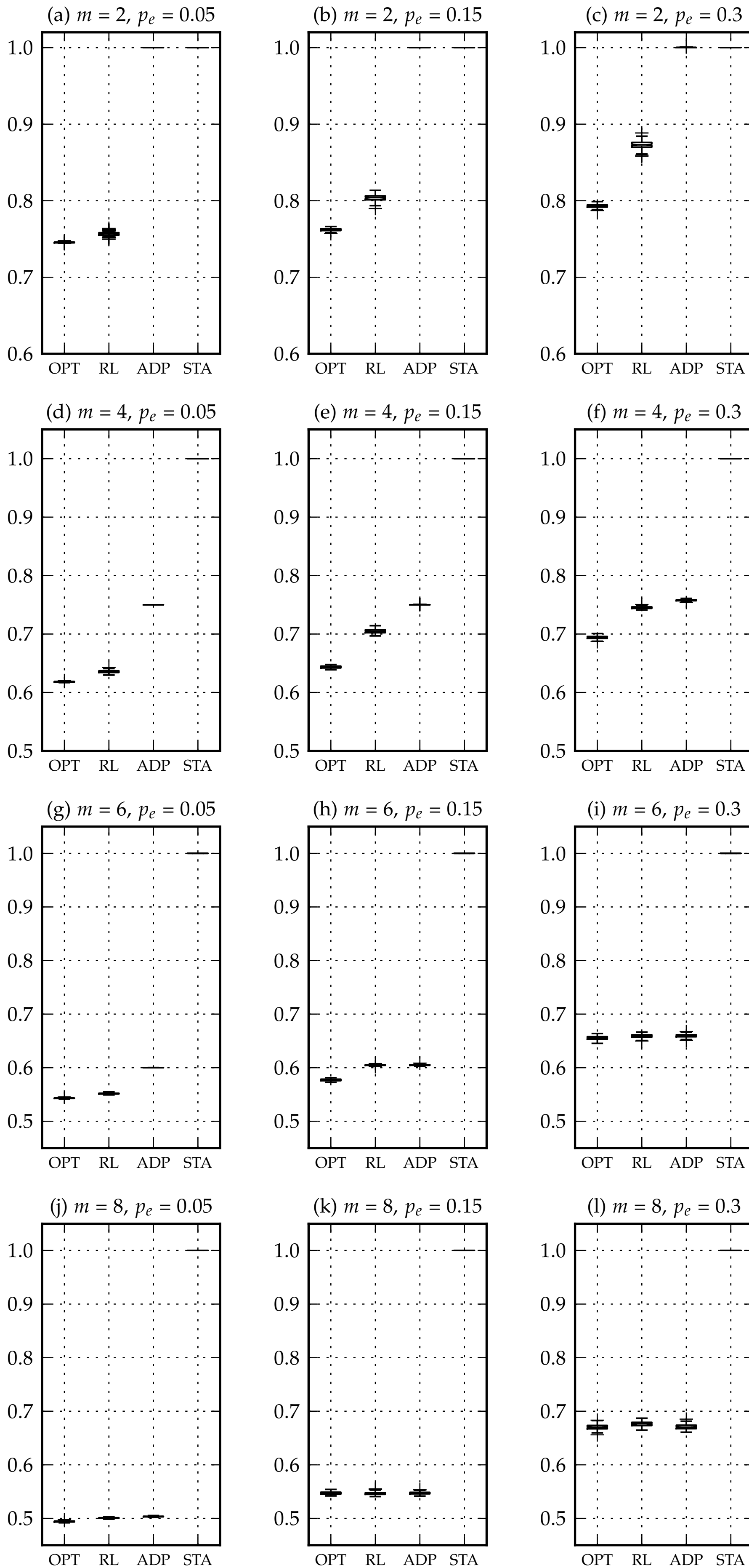

Figure 5.8: Normalized average execution time of each approach with respect to the *STA* approach with $m \in \{2, 4, 6, 8\}$ and $k = 10$, and $p_e \in \{0.05, 0.15, 0.3\}$ for a single task. The worst-case execution times are set to $C^d = 1.5 \times C^u$ and $C^r = 3.5 \times C^u$.

In particular, when the error probability p_e or the ratio m/k is relatively low, e.g., in Figure 5.8 (a)-(h), (j), and (k), the *OPT* approach outperform other approaches significantly. When both, error probability, and the ratio m/k increase, e.g., in Figure 5.8 (i) and (l), the options to select the execution modes become more limited, which result in negligible difference of the *OPT* approach, *RL* approach, and *ADP* approach. The *RL* approach also dominates in most of the evaluated cases of *ADP* and *STA* approaches, e.g., in Figure 5.8 (a)-(g) and (j), but it always performs worse than the *OPT* approach (without knowing the error probability in advance).

In the case that the error probability is relatively low, e.g., in Figure 5.8 (a), (d), (g) and (j), or both error probability and the ratio m/k are relatively high, e.g., in Figure 5.8 (i) and (l), the difference between *OPT* and *RL* is minor. For a given (m, k) constraint, when the error probability increases, e.g., rows of Figure 5.8, or for a given error probability, the number of tolerable error jobs becomes less (m increases with a constant k), e.g., columns of Figure 5.8, we can observe that the achievable benefit from the *RL* approach significantly decreases. This is because the agent tends to execute the *unreliable* modes first to maximize the cumulative reward. However, such an intention in fact also reduces the resilience, so that the upcoming jobs executing in the *detected* modes often have to execute the *reliable* mode immediately when a error is detected.

Multi-Task System Evaluation. We conducted experiments for multitask systems on a multiprocessor system using synthetically generated task sets and simulated schedules. We considered 100 task sets, each of which contained 40 tasks, which were scheduled on 4 processors. The total utilization for each task set is set to $U_{\mathbb{T}} \in [20\%, 200\%]$ with 20% step increases, considering the *reliable* mode worst-case execution time for each task. For each task, the utilization was generated by applying the Dirichlet-Rescale (DRS) algorithm [GBD20] and each task's utilization was capped at 50%. The task periods T_i were uniformly drawn from the set $T_i \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ used in automotive systems [KZH15]. Subsequently, the worst-case execution time of the *reliable* mode for each task was calculated as $C_i^r = U_i \cdot T_i$, and C_i^u and C_i^d are calculated as $C^d = 1.5 \cdot C^u$ and $C^r = 3.5 \cdot C^u$. Each task is subject to (m, k) constraints, where $m \in \{2, 4, 6, 8\}$ is uniformly chosen and $k = 10$ for each task. For each error probability $p_e \in \{0.05, 0.15, 0.3\}$, an experiment was conducted, and we assumed that each task has the same error probability in an experiment.

We considered the largest-utilization-first worst-fit partitioning algorithm to partition all tasks to the 4 processors. That is, the tasks are assigned on the processor with the temporarily lowest utilization allocated to it. The schedule was simulated for the duration of the hyper-period, which was 10.000 time units, and obtain the average utilization of the system as the time average of the cumulative execution times during the hyper-period.

Due to the similarity of the results, we opt to show only the task system with total utilization of 50% per processor, i.e., 200% total system utilization, which is shown in Figure 5.9. In general, the results show that, the *OPT* approach and *RL* approach, both decrease the utilization for multitask systems in all the evaluated cases. The proposed *OPT* approach can save 11.7%, 9.56%, and 6.31%

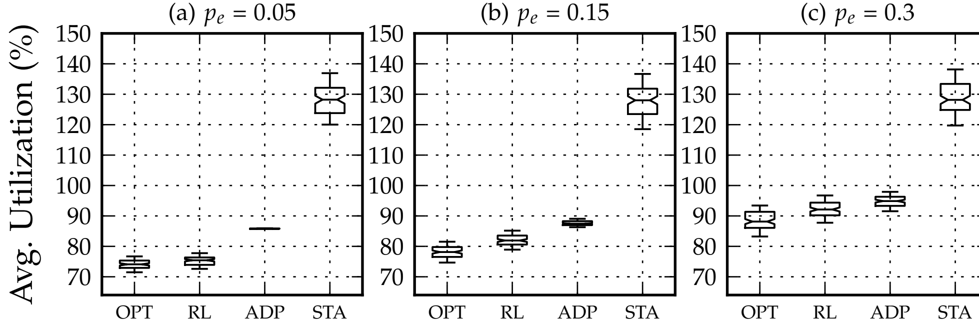

Figure 5.9: Measured time-averaged utilization of 100 task sets (of 40 tasks each) on simulated scheduled on 4 processors using partitioned scheduling. The parameters, $m \in \{2, 4, 6, 8\}$ and $k = 10$, and $p_e \in \{0.05, 0.15, 0.3\}$ and the worst-case execution times are set to $C^d = 1.5 \times C^u$ and $C^r = 3.5 \times C^u$.

in comparison to the state-of-the-art in [CBC+16], and save 54.1%, 49.42%, and 40.08% in comparison with the static approach in [NQ06] with different error probabilities respectively. In particular, the *OPT* approach and *RL* approach benefit from very low error probability, as evident in Figure 5.9 (a).

Overheads. The *OPT* approach only requires an offline-generated look-up table, to select a job mode for the current system state. The computational overhead for the look-up table generation, depends on the (m, k) constraints, e.g., $(m = 2, k = 10)$ takes 10 seconds, $(m = 4, k = 10)$ takes 20.7 hours, $(m = 6, k = 10)$ takes 7.8 hours, and $(m = 8, k = 10)$ takes 0.5 seconds on average. During runtime, the overhead due to the look-up are negligible.

To evaluate the overhead of training and mode selection, we deployed our RL-base approach on both Intel desktop and Nvidia Jetson AGX Xavier (32G) board. On the Intel desktop, the training process for each task with one configuration took 450 seconds on average. The training process is repeated for 20 times, and the trained DQN with the highest reward is selected, and the overhead for each task to select the execution mode for next job is 300 microseconds. On the Nvidia Jetson AGX Xavier board, two power modes with different power budgets are evaluated, i.e., a) default mode with 15W power budget with 4 processors running at 2188 MHz and b) MAXN mode without power budget limitation with 8 processors running at 2265.6 MHz.

The overhead for training and online execution mode selection of RL-based approach only depends on the structure of DQN, regardless of given (m, k) constraints. In the default mode, the training process took 19.7 minutes, and the execution mode selection took 1.06 milliseconds on average. In the MAXN mode, the training process took 15 minutes, and execution mode selection took 1 milliseconds on average. We observed that, the number of processors increasing from default to MAXN does not reduce the training time proportionally, and none of the processor is full-loaded in our evaluation. In the inference phase, i.e., selecting execution mode, the MAXN mode slightly outperforms the default mode, due to the minor boost of single core frequency.

A look-up table implementation can also be utilized, because the state space in our application, i.e., the minimal legal space S^*, is finite. That is, the trained

DQN network can be converted to a table, which shows the mapping between states and corresponding probabilities of different execution modes. In that case, the runtime overhead for selecting execution modes of tasks is negligible.

For systems with unknown probabilities, as one safe policy, the *ADP* proposed in [CBC+16] can be first applied to estimate a safe probability for the current scenario. If the overhead is acceptable, the proposed automata-based approach can derive the selection policy. However, to optimize the policy, recalculating for each scenario is rather expensive. The *RL* approach could still be more effective in this case.

5.7 CONCLUSION

In this chapter, fault-tolerance as a supplementary reliability aspect of real-time systems is examined in spite of dynamic external causes of fault. To assure that an acceptable quality-of-service (QoS), i.e., fault-tolerance, can be achieved, upper-bound on consecutive erroneous job executions and guaranteed m error-free executions out of k consecutive job executions are studied. Using various job variants which trade off increased execution time demand with increased error protection, a state-based policy selection strategy is proposed. The policy guarantees that all reachable states comply with the QoS constraints whilst minimizing the expected system utilization and assuring hard real-time compliance of the task system. The state-based policy further allows the usage of machine learning techniques such as reinforcement learning which are able to provide hard guarantees. The proposed approaches demonstrate significant decreased system utilization compared to the state of the art in the evaluations and reasonable system overhead for both the analytic and reinforcement learning based approach.

6 MAXIMAL SENSOR DATA TIME-STAMP DIFFERENCE

Cyber-physical systems can be described as complex multiple-input multiple-output systems, which consist of multiple sensors, processing sub systems, and actuators; all of which, conjointly implement the system function. Typically, the system is decomposed into independent sensor-, actuator-, and processing tasks. Each of these tasks, potentially operates at a different activation rate, and each of the tasks can be mapped to, and be executed individually on a heterogeneous system architecture. To add to that complexity, the respective tasks have data precedence constraints, e.g., sensors produce data, which is then read by processing tasks. To assure proper system function and quality-of-service, the overall application must comply with several temporal constraints, which are derived from the concrete cyber-physical system's design specifications. These are mapped into corresponding surrogate constraints, e.g., the maximal time-stamp difference of sensor data as perceived at sensor fusion tasks, maximum reaction time, or maximum data age.

To assure proper system function and quality-of-service, the overall application must comply with several temporal constraints

In this chapter, we analyze the maximal time-stamp difference of multiple sensor data flows, which are used for sensor fusion in a common task. Unbounded, and uncertain maximal time-stamp differences of all sensor data flows, which are used to estimate a system state, may lead to completely-off state predictions. That is, contrary to assumption, the sensor data – used for sensor fusion – represents (parts of) the system state at different points in time. Despite the fact that the imposed temporal constraints are more complex than a per-task relative deadline, we will show in this chapter how these constraints can be modularly analyzed on a heterogeneous execution architecture and non-global synchronization, only presuming that each task complies with its relative deadline constraint.

Unbounded, and uncertain maximal time-stamp differences of all sensor data flows, which are used to estimate a system state, may lead to completely-off state predictions

The remainder of this chapter is organized as follows. In the following Section 6.1 *Motivation*, a thorough motivation for the studied problem is given by example. Then, in Section 6.2 *Related Work*, the related work is presented. In Section 6.3 *Application Model*, the application and task model is formally defined and explained. Based on that, the maximal time-stamp difference analysis for sensor data flows in processing graphs is presented, in Section 6.4 *Maximal Sensor Data Time-Stamp Difference Analysis*. In Section 6.5 *Discussion & Extensions*, the results in this chapter and extensions are discussed, and lastly concluded in Section 6.6 *Conclusion*.

6.1 MOTIVATION

Cyber-physical systems, which are complex multiple-input multiple-output systems, are for instance robotics applications, and autonomous driving systems. The

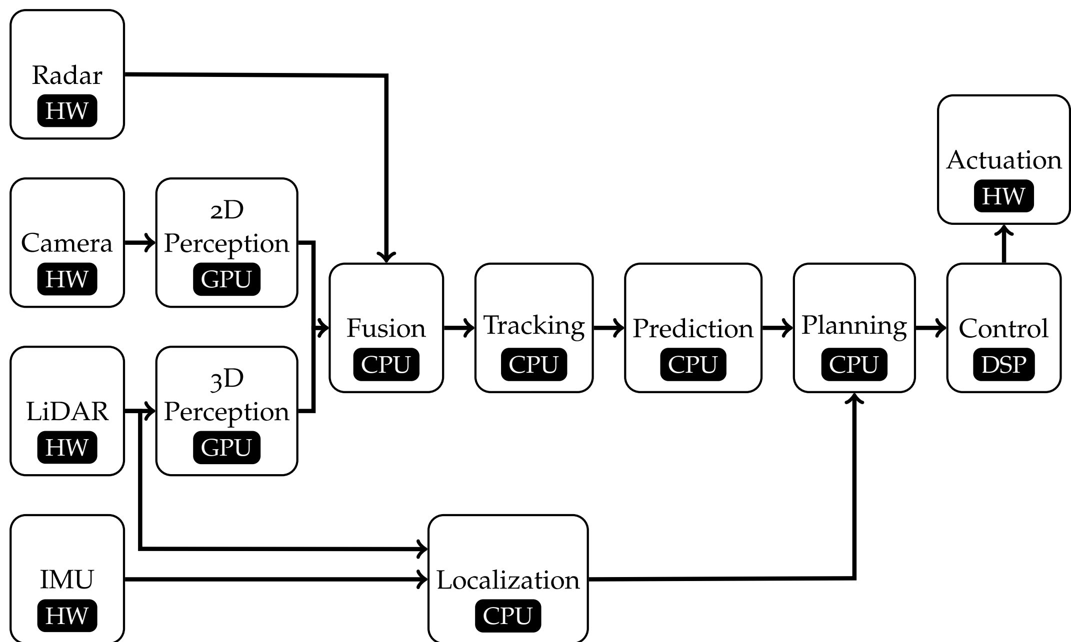

Figure 6.1: An exemplary realistic real-time system architecture used to implement autonomous driving systems by the company *Perceptin*. The vertices in the processing graph denote functional modules such as sensor data pre-processing, perception, tracking, trajectory planning, and control. The directed edges in the processing graph denote the data dependencies between the modules, e.g., the data produced by the *localization* module is used by the *planning* module in its computation.

architecture of an exemplary autonomous driving system, is shown in Figure 6.1, in a so called *processing graph*. The vertices in the processing graph describe a function, e.g., sensor data *preprocessing*, *perception*, *object tracking*, *trajectory planning*, and *control*. The directed edges in the processing graph, denote the data dependencies between vertices; the data, produced by the *localization*, and *prediction* vertex, is used by the *planning* vertex in its computation.

processing graph

The relevant parts of the physical environment are measured by various sensor systems, which are for instance, a *radar*, a *stereo camera*, a *LiDAR*, and an *inertia measurement unit* (IMU), in the provided example. The sensor data is processed through various algorithms and filters, and is used to accurately estimate the system state. To reject perturbations, noise, and partial sensor failures; sensor fusion algorithms are employed to combine information, and data, from multiple information sources to obtain a more comprehensive and de-noised estimate of the system state.

sensor data is processed through various algorithms and filters, and is used to accurately estimate the system state

There are many different sources of noise and perturbations in sensory data acquisition, such as inherent measurement noise or physical limitations of the sensor system. However, we focus on the temporal misalignment of the sensory data, which is due to the independent scheduling of the processing and filtering algorithms. There are three temporal constraints, which are important for sensor fusion algorithms, namely, the *maximal sensor data propagation latency*, *minimal sensor data propagation latency*, and the deduced *maximal sensor data time-stamp difference*. Maximal and minimal sensor data propagation latency, refers to the

we focus on the temporal misalignment of the sensory data, which is due to the independent scheduling

maximal (and minimal, respectively) time difference of the absolute sensor data generation time, and the absolute time, when the data is read in the sensor fusion vertex. The *maximal sensor data time-stamp difference* – at a sensor fusion vertex – denotes the maximal difference of any two maximal-, and minimal sensor data propagation latencies, for any two different sensors, leading to that sensor fusion vertex. The *maximal sensor data time-stamp difference*, should be in the magnitude of the sampling time, presumed in the sensor fusion algorithm. That is because, these sensor fusion algorithms require that the used sensory data refers to measurements of the physical environment at *the same time* or at least with a *bounded time difference*.

early sensor fusion
intermediate sensor fusion
late sensor fusion

Sensor fusion can be classified into *early*, *intermediate*, and *late fusion*, depending on the processing stage (after sample generation) in which the data is fused. That is, in *early fusion*, all sensory data sources are fused immediately after sample collection; In *intermediate fusion* only the calculated, i.e., preprocessed, features of the data are fused during processing; and in *late fusion*, all sensory data is fully preprocessed before being fused. An example of *late fusion*, with regards to the example in Figure 6.1, is given by the camera and LiDAR sensor data, which is processed in the 2*D* and 3*D*-perception vertices, and then fused in the fusion vertex. The vertices are mapped to different processing units, on a heterogeneous architecture, e.g, the 2*D*- and 3*D*-perception algorithms are being executed on graphics-processing units (GPU), whereas, *tracking*, *prediction*, and *planning* are executed on the central-processing units (CPUs). Interference caused by co-executing workloads on the processing units, and communication overheads all contribute to the high variability of maximal- and minimal sensor data propagation latency, and thus non-trivial maximal sensor data time-stamp differences.

In this chapter, we propose the following contributions to address the motivated problems. We propose the processing graph as a problem formalization, and a subsequent mapping of the vertices into *loosely coupled* semi-sporadic tasks with data precedence constraints. The global scheduling problem of the heterogeneous system, is analyzed on the basis of local per-processing unit, preemptive or non-preemptive scheduling algorithms. Based on the formal description, and modular problem decomposition, we provide analyses to calculate the *maximal sensor data time-stamp difference* at all sensor fusion vertices. Moreover, we propose an abstract *precedence property*, which allows to reduce the pessimism in our analyses, and can be extended to other task models. This property is proven for non-preemptive rigid gang, and stationary rigid gang scheduling.

6.2 RELATED WORK

maximum reaction time
maximum data age

In the literature, the concepts of *maximum reaction time* and *maximum data age* are used to address end-to-end latencies of processing chains in the automotive domain. The maximum reaction time refers to the longest latency of the first response (reaction) of a system to a corresponding stimulus (cause), e.g., the latency from a camera sample to an object detection, or to the final actuation. The maximum data age denotes the largest time interval between the sampling time

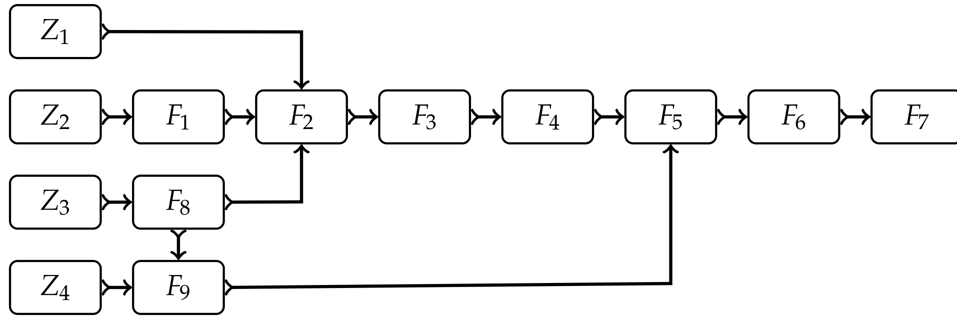

Figure 6.2: A processing graph formally describes the data communication of the application. The source vertex in a processing graph (i.e., vertices which do not have any predecessors) represent sensor vertices which produce data either in a time-triggered or event-triggered manner. All sensors produce time-stamped data with respect to the sensor's local clocks. The processing vertices in the graph execute periodically on the latest available data and produce an output when the execution is finished. Each processing vertex is assigned to a processing unit, e.g., a CPU, GPU, or DSP on which it is executed.

of a data token by a task until the last point in time when the system produces an output related to that data token. Such timing constraints have been researched intensively in the context of cause-effect chains in AUTOSAR compliant automotive systems, e.g., [DBC+19; WBP+13; FRN+09; RMD+10; GCU+21; BDM+16]. More precisely, upper bounds for the maximum reaction time have been developed by Davare et al. in [DZN+07] and Kloda et al. in [KBS18] which both focus their work on periodic task sets with synchronous release patterns with preemptive scheduling algorithms. Becker et al. in [BDM+16] provide schedule independent lower and upper-bounds for end-to-end latencies on the job-level. Schliecker et al. in [SE09] propose a recursive approach to calculate lower and upper-bounds for reaction-time and data age. Dürr et al. in [DBC+19] consider sporadic task systems on asynchronous distributed systems and provide upper-bounds for maximum reaction time and maximum data age. The result was later improved by Günzel et al. in [GCU+21] by simulating the schedule and obtaining exact results. Whilst strongly related to data age, the problem and analysis of maximal sensor data time-stamp difference has not yet been reported in the literature. Moreover, most published results in this context only consider either preemptive or non-preemptive scheduling policies in their end-to-end latency analyses, but never a combination thereof in a heterogeneous architecture with independent hardware units, and scheduling policies.

6.3 APPLICATION MODEL

processing graph

To abstract from the concrete example provided in the motivation, we formalize the problem as a *processing graph* [1], which is illustrated in Figure 6.2 and defined hereinafter. A processing vertex in the processing graph is implemented as an individual task, which is activated either periodically by a *timer-trigger* or sporadically, i.e., *event-triggered*.

1 The term processing graph is originally proposed in the RTSS 2021 Industry Challenge.

Definition 6.1 (Processing Graph)**.** *A processing graph $G = (V, E)$ is a directed acyclic graph (DAG) where $V = \{Z\} \cup \{F\}$ denotes the set of* sensor vertices *Z and* processing vertices *F, and E denotes the path of data propagation (signal/data flow) along the vertices. That is, if $e = (v_i, v_j) \in E$ then data is propagated from v_i to v_j. While processing vertices may produce and/or consume data, sensor vertices only produce data.*

The sensor vertices in the processing graph produce sensory data according to its internal clocks periodically, i.e., it is not synchronized with the processing units. The processing vertices implement a certain function such as sensor data *preprocessing*, *perception*, *tracking*, *trajectory planning*, or *control*. The directed edges represent the data flow between the vertices, that is, the data produced by a vertex is used by adjacent vertices in its computation.

processing vertices implement a certain function

The tasks, which implement the processing vertices are executed on a heterogeneous system comprised of M processing units, which we refer to as $P_1, \ldots, P_M$. We assume that the graph representation of the application has the same granularity as the tasks, which are actually executed on the platform. That is, each vertex is implemented by exactly one task. Hence, we use those terms interchangeably and write task $\tau_i \in V$ instead of v_i for notational convenience.

Definition 6.2 (Task Model)**.** *An implicit-deadline semi-sporadic task $\tau_i \in V$ is defined by the tuple $(C_i^{min}, C_i^{max}, T_i^{min}, T_i^{max})$, where $C_i^{max} \leq T_i^{min}$. Each task τ_i releases an infinite sequence of task instances (called jobs) J_i^ℓ with execution time $C_i^{\min} \leq C_i^\ell \leq C_i^{max}$, i.e., C_i^{min} is the best-case and C_i^{max} the worst-case execution time.*

The arrival-time of the ℓ-th job of τ_i is denoted as a_i^ℓ and the minimal inter-arrival time between two consecutive jobs of τ_i is constrained by minimal and maximal inter-arrival times, i.e., $a_i^\ell + T_i^{min} \leq a_i^{\ell+1} \leq a_i^\ell + T_i^{max}$. We assume implicit deadlines, i.e., each job must be finished before the next job of the same task could be released. That is, a job released at a_i^ℓ must finish before $a_i^\ell + T_i^{min}$.

The tasks $\tau_i \in V$ are mapped to appropriate processing units, due to the different computational demands and characteristics of the tasks. Henceforth, we assume that the tasks implementing processing vertices can be mapped to a subset of the processing units, like CPUs, GPUs, and DSPs, while tasks related to sensor vertices are mapped to dedicated sensor hardware.

Definition 6.3 (Mapping)**.** *Each task $\tau_i \in F$ is statically mapped to a processing unit formally described by the mapping $\sigma : F \mapsto \{P_1, P_2, \ldots, P_M\}$,*

$$\sigma(\tau_i) = P_j \text{ if task } \tau_i \text{ is mapped to processing unit } P_j \tag{6.1}$$

We assume that all processing units support and use a task-level fixed priority work-conserving scheduling algorithm. That is, every job J_i^ℓ of a task τ_i has the same priority and the priority is fixed throughout the system's lifetime. However, the scheduling algorithm may be preemptive or non-preemptive for different processing units, which may be either by design choice or enforced by the technical properties of the respective processing units. For example, non-preemptive scheduling may be used for tasks assigned to GPUs and preemptive scheduling may be used for tasks assigned to CPUs.

the scheduling algorithm may be preemptive or non-preemptive for different processing units

Definition 6.4 (Sensor Task)**.** *The raw sensor data generation is modeled by a semi-sporadic task τ_i with $C_i^{min} = C_i^{max} \to 0$ which is exclusively allocated to be executed on the sensor hardware, which implies that the response time $R_i = C_i^{min} = C_i^{max} \to 0$. We assume that the the time-stamp of the sensor data equals to the finishing time of the modeled sensor data generation task.*

The purpose of this definition is that sensor tasks can be considered in the same manner as processing tasks in the analysis framework.

Schedule. The concrete schedule, which is generated for each of the processing units P_j for $j \in \{1, \ldots, M\}$ is denoted as S_j and depends on;

1. The specific task-level fixed-priority scheduling policy used for processing unit P_j;
2. And the concrete arrival sequence of jobs generated by the tasks assigned to processing unit P_j.

Due to the fact, that we consider implicit-deadline task sets and a task-level fixed-priority policy, there is always at most one job of each task active – assuming no deadline miss. Therefore, we define a simplified schedule function $S_j : \mathbb{R} \mapsto V \cup \{\perp\}$ for $j \in \{1, \ldots, M\}$ which maps to the task instead of the job. Implicitly, we refer to the currently active job of that task.

Definition 6.5 (Schedule)**.** *For a given processing unit P_j for $j \in \{1, \ldots, M\}$, a schedule $S_j : \mathbb{R} \mapsto V \cup \{\perp\}$ is defined as*

$$S_j(t) = \begin{cases} \tau_i & \text{if a job of task } \tau_i \text{ is executing on } P_j \text{ at time } t \\ \perp & \text{if the processing unit } P_j \text{ is idle at time } t \end{cases} \tag{6.2}$$

For ease of notation, we refer to the times when jobs are released, starting to execute, and finish to execute in relation to the concrete schedule S_j while dropping the relation to the underlying arrival sequence. To be precise, we use $a(S_j, J_{i,\ell})$ to denote the arrival-time of the ℓ-th job of τ_i in the arrival sequence, $s(S_j, J_{i,\ell})$ to denote the first time that job $J_{i,\ell}$ starts execution in the schedule S_j, and $f(S_j, J_{i,\ell})$ to denote the finishing time. A schedule S_j is feasible if all jobs meet their deadline, that is $f(S_j, J_{i,\ell}) \leq a(S_j, J_{i,\ell}) + T_i^{min}$ for all tasks $\tau_i \in V$.

The temporal behaviour of the overall system is then defined by the set of schedules of each of the processing units, which we refer to as system schedule and is used in the forthcoming analyses.

Definition 6.6 (System Schedule)**.** *The system schedule S consists of the schedules of all processing units P_j for $j \in \{1, \ldots, M\}$, i.e. $S(t) = (S_1(t), S_2(t), \ldots, S_M(t))$.*

The data dependencies, which is the data flow between tasks, is described by the edges E in the processing graph G. For example, in Figure 6.2 the processing vertex F_2 needs data from the processing vertices F_1 and F_8 and from the sensor vertex Z_1. We say a vertex v_i *explicitly receives* data from v_j, if $(v_i, v_j) \in E$, and that v_i *implicitly receives* data from v_j if there is a path from v_i to v_j in E, but $(v_i, v_j) \notin E$. For example, in Figure 6.2, the processing vertex F_8 explicitly reads from sensor Z_3, since no intermediate processing is involved, while processing

explicit receiving

implicit receiving

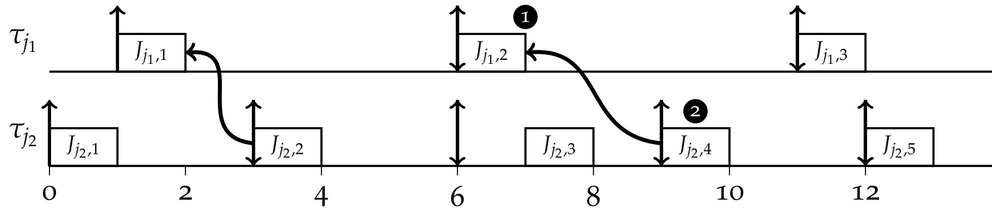

Figure 6.3: Exemplary schedule for two tasks in a processing chain $E_j = \langle \tau_{j_1}, \tau_{j_2} \rangle$. A processing chain instance for the 4-th job of task τ_{j_2} is for example given by $\langle J_{j_1,2}, J_{j_2,4} \rangle$ and the propagation latency is $9 - 6 = 3$ time units.

vertex F_2 implicitly reads from sensor Z_2, since the data from Z_2 is processed by F_1 before it can be read by F_2.

Task Communication. The task communication is based on buffers, where each task, which reads or writes data, is associated with input and output ports which can buffer exactly one data token of a specified type at any time. Whenever a new data token is written to a port, the old data token is replaced. A job reads the current data token of each of its input ports when it starts execution and writes the output data tokens to the buffers of its output ports at the end of its execution, i.e., *read-on-start* and *write-on-finish*. We assume zero-time communication (i.e., writing the data token into the buffer and reading from it happens immediately) and that each buffer is written by exactly one task. If this assumption is too optimistic, communication overheads can be either modeled by constant delays or by introducing additional communication tasks which represent the communication using, for example, network-on-chips, CAN Bus, or others with the same theory framework as presented in this chapter.

read-on-start
write-on-finish

communication overheads can be either modeled by constant delays or by introducing additional communication tasks

6.4 MAXIMAL SENSOR DATA TIME-STAMP DIFFERENCE ANALYSIS

To assure the functional correctness of the application, it is important to ensure that the signals, which are used for state estimation, represent the physical system state roughly at the same time. Therefore, the generation time (time-stamps) of all explicit and implicit sensor samples which are processed by a common vertex, namely a *fusion vertex*, must differ at most by a specified maximal time-duration. Intuitively, the idea is to calculate upper- and lower bounds of the data propagation latency – which starts at the time of sensor sample generation and ends at the time of sensor data reading at the fusion vertex – for all possible signal paths from sensor vertices to the fusion vertex of interest.

Definition 6.7 (Processing Chain). *The ordered set $E_j := \langle \tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_N} \rangle$ denotes a processing chain in the processing graph, such that $\tau_{j_{i+1}}$ reads the data token produced by τ_{j_i} for all $i \in \{1, \ldots, N-1\}$ and N is the number of processing vertices in that chain. This immediate data dependency is illustrated by the directed edges in the application's processing graph, as shown in Figure 6.2. The indexing j_k denotes the task index of the k-th task in the processing chain E_j, e.g., $j_k = z$ implies that $\tau_z \in V$ is the k-th task in the processing chain E_j.*

The processing chains describe the data and signal flow from the chain's source, i.e., sensor vertex, to the final processing vertex in the processing chain. The received data at the fusion vertex does not have to be the raw sensor data, generated by the sensor vertex, but may represent a refined, filtered, or processed artifact of the raw sensory data. That raw sensory data, and therefore the latency between the start of processing of the last processing vertex in the processing chain and the generation time of the raw sensor data is of interest in an analysis. In order to analyze the data propagation latency, the reading and writing times for each processing vertex must firstly be formalized, and secondly concrete instances of processing chains must be examined.

concrete instances of processing chains must be examined

Definition 6.8 (Processing Chain Instance). *Let $E_j = \langle \tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_N} \rangle$ denote a processing chain and let S be a feasible system schedule for all the tasks in E_j. Furthermore let $i'_N \in \mathbb{N}$ denote the job of the final task τ_{j_N} of interest. That is job J_{j_N,i'_N} reads the available data when it starts execution. A processing chain instance – with respect to the data which is read by J_{j_N,i'_N} – consists of an ordered set of job instances $\langle J_{j_1,i'_1}, J_{j_2,i'_2}, \ldots, J_{j_N,i'_N} \rangle$ such that for all $\ell \in \{1, \ldots, N-1\}$*

$$i'_{N-\ell} := \max\{k \in \mathbb{N} \mid write(S, J_{j_{N-\ell},k}) \leq read(S, J_{j_{N-\ell+1},i'_{N-\ell+1}})\} \quad (6.3)$$

holds where read denotes the times of the reading of the data and write in the schedule.

As an example, Figure 6.3 illustrates a schedule of two semi-sporadic processing vertices τ_{j_1} and τ_{j_2} with progressing chain $E_j = \langle \tau_{j_1}, \tau_{j_2} \rangle$, i.e., jobs of τ_{j_1} generate data which is read by jobs of τ_{j_2}. For job $J_{j_2,4}$, we identify $J_{j_1,2}$ ❶ as the job which wrote the data read by $J_{j_2,4}$ ❷, as it is the job of τ_{j_1} with the largest index which has a writing time not later than $J_{j_1,2}$'s reading time. Hence, the processing chain instance based on the job $J_{j_2,4}$ is $\langle J_{j_1,2}, J_{j_2,4} \rangle$. We emphasize that a propagating processing chain instance is dependent on the observed job of the last task (sink processing vertex) in a concrete global schedule S. For a specific propagation processing chain instance $E'_j = \langle J_{j_1,i'_1}, J_{j_2,i'_2}, \ldots, J_{j_N,i'_N} \rangle$ based on a specific job instance of the last task J_{j_N,i'_N}, we formally define the sensor data propagation latency, that is, the temporal latency from reading sensor data in J_{j_1,i'_1} until J_{j_N,i'_N} starts processing it, as follows.

a propagating processing chain instance is dependent on the observed job of the last task (sink processing vertex) in a concrete global schedule

Definition 6.9 (Sensor Data Propagation Latency). *Given a processing chain instance $E'_j := \langle J_{j_1,i'_1}, J_{j_2,i'_2}, \ldots, J_{j_N,i'_N} \rangle$ on the basis of a concrete system schedule S. Then the sensor data propagation latency of E'_j is defined as*

$$read(S, J_{j_N,i'_N}) - write(S, J_{j_1,i'_1}) \quad (6.4)$$

where $write(S, J_{j_1,i'_1})$ denotes the absolute time of the generated sensor data.

An example for the definition of sensor data propagation latency for a concrete schedule S is shown in Figure 6.3, where each job reads data when it starts execution and writes data when it finishes execution. In the processing chain instances $\langle J_{j_1,1}, J_{j_2,2} \rangle$ the sensor data propagation latency is $3 - 2 = 1$ and $9 - 7 = 2$ for $\langle J_{j_1,2}, J_{j_2,4} \rangle$, respectively.

The remainder of the analysis is structured as follows. At first in Section 6.4.1 and Section 6.4.2, a minimal and maximal sensor data propagation latency for

any processing chain, using preemptive and non-preemptive scheduling policies, is derived. In Section 6.4.3, the minimal an maximal sensor data propagation latencies for all processing chains which are joined in a common fusion vertex are used to deduce the maximal sensor data time-stamp difference.

6.4.1 MINIMAL SENSOR DATA PROPAGATION LATENCY

Henceforth, we analyze the minimal sensor data propagation latency of any possible processing chain instance E'_j based on a feasible preemptive or non-preemptive task-level fixed-priority system schedule S. We emphasize that the schedule S allows for jobs of some tasks of a processing chain to be scheduled preemptively and others to be scheduled non-preemptively on their respective processing units, which is to be considered in the analysis.

In a first step, we establish the following lemma which helps to prove a lower-bound for the sensor data propagation latency.

Lemma 6.1. *Let a processing chain* $E_j = \langle \tau_{j_1}, \tau_{j_2}, \dots \tau_{j_N} \rangle$ *and a mapping* σ *for all task* $\tau_i \in V$*. Then for any processing chain instance* $E'_j = \langle J_{j_1,i'_1}, J_{j_2,i'_2}, \dots, J_{j_N,i'_N} \rangle$ *in a concrete feasible preemptive or non-preemptive schedule* S

$$f(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) - s(S, J_{j_{N-\ell},i'_{N-\ell}}) \geq C^{min}_{j_{N-\ell}} \tag{6.5}$$

holds true for all ℓ *in* $\{1, \dots, N-1\}$.

Proof. Each job's finishing time is never earlier than its starting time plus its minimal execution time, i.e., for all ℓ in $\{0, \dots, N-1\}$,

$$s(S, J_{j_{N-\ell},i'_{N-\ell}}) + C^{min}_{j_{N-\ell}} \leq f(S, J_{j_{N-\ell},i'_{N-\ell}}) \tag{6.6}$$

must hold in S. Secondly, by the construction property of processing chain instances E'_j and the communication policy *to write on finish* and *to read at the start*, we know that

$$i'_{N-\ell} := \max\{k \in \mathbb{N} \mid f(S, J_{j_{N-\ell},k}) \leq s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}})\} \tag{6.7}$$

Therefore, for all ℓ in $\{1, \dots, N-1\}$ – irrespective of a specific schedule S – it follows that

$$f(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq f(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \tag{6.8}$$

By the combining Eq. (6.6) and Eq. (6.8) we get

$$C^{min}_{j_{N-\ell}} \leq f(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) - s(S, J_{j_{N-\ell},i'_{N-\ell}}) \tag{6.9}$$

which concludes the lemma. □

Based on Lemma 6.1, a lower-bound for any sensor data propagation latency and thus minimal sensor data propagation latency of a given processing chain E_j can be established.

Theorem 6.2. *The sensor data propagation latency of a processing chain $E_j = \langle \tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_N} \rangle$ is at least $\sum_{\ell=2}^{N-2} C_{j_\ell}^{min}$.*

Proof. Based on Eq. (6.7), we can lower-bound the sensor data propagation latency of E_j by:

$$s(S, J_{j_N, i'_N}) - f(S, J_{j_1, i'_1}) \geq f(S, J_{j_{N-1}, i'_{N-1}}) - s(S, J_{j_2, i'_2}) \tag{6.10}$$

$$= \sum_{\ell=2}^{N-2} f(S, J_{j_{\ell+1}, i'_{\ell+1}}) - s(S, J_{j_\ell, i'_\ell}) \tag{6.11}$$

$$\text{(by Lemma 6.1)} \geq \sum_{\ell=2}^{N-2} C_{j_\ell}^{min} \tag{6.12}$$

for any processing chain instance irrespective of the concrete schedule S. □

6.4.2 MAXIMAL SENSOR DATA PROPAGATION LATENCY

Conversely, we also analyze the maximal sensor data propagation latency of a given processing chain for any feasible preemptive or non-preemptive task-level fixed-priority schedule S and a given mapping σ.

Lemma 6.3. *If for a given chain instance $E'_j = \langle J_{j_1, i'_1}, J_{j_2, i'_2}, \ldots, J_{j_N, i'_N} \rangle$ a job $J_{j_{N-\ell}, i'_{N-\ell}}$ for some $\ell \in \{0, \ldots, N-1\}$ is scheduled* ***non-preemptively*** *in the schedule S, then*

$$s(S, J_{j_{N-\ell}, i'_{N-\ell}}) - a(S, J_{j_{N-\ell}, i'_{N-\ell}}) \leq R_{j_{N-\ell}} - C_{j_{N-\ell}}^{max} \tag{6.13}$$

Proof. In any feasible non-preemptive schedule S

$$f(S, J_{j_{N-\ell}, i'_{N-\ell}}) \leq a(S, J_{j_{N-\ell}, i'_{N-\ell}}) + R_{j_{N-\ell}}$$

for all ℓ in $\{0, \ldots, N-1\}$. Moreover, in a non-preemptive schedule all jobs run to completion once they started. Hence,

$$s(S, J_{j_{N-\ell}, i'_{N-\ell}}) + C_{j_{N-\ell}}^{i'_{N-\ell}} = f(S, J_{j_{N-\ell}, i'_{N-\ell}})$$

where $C_{j_{N-\ell}}^{i'_{N-\ell}}$ is the execution time of the job. We know S is feasible, therefore

$$s(S, J_{j_{K_j-\ell}, i'_{K_j-\ell}}) + C_{j_{N-\ell}}^{i'_{N-\ell}} \leq a(S, J_{j_{K_j-\ell}, i'_{K_j-\ell}}) + R_{j_{K_j-\ell}} \tag{6.14}$$

Since $C_{j_{N-\ell}}^{i'_{N-\ell}} = C_{j_{N-\ell}}^{max}$ is a feasible execution-time, we conclude that

$$s(S, J_{j_{K_j-\ell}, i'_{K_j-\ell}}) + C_{j_{N-\ell}}^{max} \leq a(S, J_{j_{K_j-\ell}, i'_{K_j-\ell}}) + R_{j_{K_j-\ell}} \tag{6.15}$$

in any feasible non-preemptive schedule S. □

In turn, if the analyzed job is scheduled preemptively in the schedule S, we have to resort to the following larger upper-bound.

Lemma 6.4. *If for a given processing chain instance $E'_j = \langle J_{j_1,i'_1}, J_{j_2,i'_2}, \ldots, J_{j_N,i'_N} \rangle$ a job $J_{j_{N-\ell},i'_{N-\ell}}$ for some $\ell \in \{0, \ldots, N-1\}$ is scheduled* ***preemptively*** *in the schedule S, then*

$$s(S, J_{j_{N-\ell},i'_{N-\ell}}) - a(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq R_{j_{N-\ell}} - C^{min}_{j_{N-\ell}} \tag{6.16}$$

Proof. In any feasible preemptive schedule S, for job $J_{j_{N-\ell},i'_{N-\ell}}$ we have that

$$f(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq a(S, J_{j_{N-\ell},i'_{N-\ell}}) + R_{j_{N-\ell}} \tag{6.17}$$

for each ℓ in $\{0, \ldots, N-1\}$. Moreover, we know that

$$f(S, J_{j_{N-\ell},i'_{N-\ell}}) \geq s(S, J_{j_{N-\ell},i'_{N-\ell}}) + C^{min}_{j_{N-\ell}} \tag{6.18}$$

Hence, we have

$$s(S, J_{j_{N-\ell},i'_{N-\ell}}) - a(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq R_{j_{N-\ell}} - C^{min}_{j_{N-\ell}}$$

by combining Eq. (6.17) and (6.18). □

For the final Theorem 6.8, we use a telescope sum construction on the basis of the arrival time difference of any two subsequent jobs in a processing chain instance. In the following, we prove an upper-bound for such an arrival time difference for any feasible schedule S.

Lemma 6.5. *For a given mapped processing chain E_j any two subsequent jobs $J_{j_{N-\ell},i'_{N-\ell}}$ and $J_{j_{N-\ell+1},i'_{N-\ell+1}}$ for some $\ell \in \{1, \ldots, N-1\}$ in an instance $E'_j = \langle J_{j_1,i'_1}, J_{j_2,i'_2}, \ldots, J_{j_N,i'_N} \rangle$ satisfy*

$$a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) - a(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq T^{max}_{j_{N-\ell}} + R_{j_{N-\ell}} \tag{6.19}$$

for any feasible schedule S

Proof. This lemma is based on [DBC+19] and is restated here for completeness. By the construction principle of processing chain instances, we know that for all processing chain jobs the following property

$$f(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) < f(S, J_{j_{N-\ell},i'_{N-\ell}+1}) \tag{6.20}$$

is satisfied. Since, if that was not the case, i.e., $s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \geq f(S, J_{j_{N-\ell},i'_{N-\ell}+1})$ then the next job $J_{j_{N-\ell},i'_{N-\ell}+1}$ would be in the processing chain instance instead of $J_{j_{N-\ell},i'_{N-\ell}}$.

For the remainder of this proof, we distinguish two cases.

In the first case let $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq f(S, J_{j_{N-\ell},i'_{N-\ell}})$, we then obtain

$$a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq f(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq a(S, J_{j_{N-\ell},i'_{N-\ell}}) + R_{j_{N-\ell}} \tag{6.21}$$

In the second case, let in turn $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) > f(S, J_{j_{N-\ell},i'_{N-\ell}})$. By the property stated in Eq. (6.20), we have that

$$a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) < f(S, J_{j_{N-\ell},i'_{N-\ell}+1}) \tag{6.22}$$

Since the worst-case response time is bounded, we have that

$$f(S, J_{j_{N-\ell}, i'_{N-\ell}+1}) \leq a(S, J_{j_{N-\ell}, i'_{N-\ell}+1}) + R_{j_{N-\ell}} \leq a(S, J_{j_{N-\ell}, i'_{N-\ell}}) + T^{max}_{j_{N-\ell}} + R_{j_{N-\ell}} \quad (6.23)$$

and in consequence, we have that

$$a(S, J_{j_{N-\ell+1}, i'_{N-\ell+1}}) - a(S, J_{j_{N-\ell}, i'_{N-\ell}}) \leq T^{max}_{j_{N-\ell}} + R_{j_{N-\ell}} \quad (6.24)$$

Putting both cases together, we have that

$$a(S, J_{j_{N-\ell+1}, i'_{N-\ell+1}}) - a(S, J_{j_{N-\ell}, i'_{N-\ell}}) \leq T^{max}_{j_{N-\ell}} + R_{j_{N-\ell}} \quad (6.25)$$

which concludes the proof. □

Lemma 6.5 can be further improved if the precedence behaviour of two subsequent tasks in a processing chain are considered, which depends on the concrete task to processing unit mapping and task-level fixed-priorities.

Definition 6.10 (Precedence). *Let τ_j, τ_k denote two tasks then we say τ_j precedes τ_k if and only if for any task instances J_j and J_k in a given schedule S, the property*

$$a(S, J_j) \leq a(S, J_k) \implies f(S, J_j) \leq s(S, J_k) \quad (6.26)$$

holds.

As will be shown, this property is satisfied for tasks which are subjected to the same processing unit and the same scheduler.

Corollary 6.6. *Let τ_j and τ_k denote two semi-sporadic tasks then if $\sigma(\tau_j) = \sigma(\tau_k)$ and τ_j has a higher task-level fixed priority than task τ_k then τ_j precedes τ_k according to Definition 6.10 in any preemptive or non-preemptive schedule S.*

Proof. Let by assumption $a(S, J_j) \leq a(S, J_k)$, which implies that during $[a(S, J_j), f(S, J_j))$ J_j is pending, i.e., released but not yet finished, and during $[a(S, J_k), f(S, J_k))$ job J_k is pending. For the proof, two cases are considered, namely;

- If the intervals do not intersect then $f(S, J_j) < a(S, J_k) \leq s(S, J_k)$ and nothing is left to show.
- In the other case let both intervals intersect and we consider the interval $[a(S, J_k), a(S, J_k) + t)$ that both jobs are pending for all $t > 0$. If S is a preemptive or non-preemptive schedule then if J_j has higher-priority than J_k and $\sigma(\tau_j) = \sigma(\tau_k)$ then either $s(S, J_j) \in [a(S, J_j), a(S, J_k))$ or $s(S, J_j) \in [a(S, J_k), a(S, J_k) + t)$.

  In either case $s(S, J_j) < s(S, J_k)$ and due to the higher-priority and mutual exclusive execution, we have that $f(S, J_j) < s(S, J_k)$.

□

Using the above precedence property of jobs of two adjacent tasks in a processing chain, the following lemma can be stated.

Lemma 6.7. *Let σ a mapping of V and E_j a processing chain then for any feasible schedule S and any concrete instance $E'_j = \langle J_{j1,i'_1}, J_{j2,i'_2}, \ldots, J_{jN,i'_N}\rangle$ the following implication holds for each $\ell \in \{1, \ldots, N-1\}$:*

$$J_{j_{N-\ell},i'_{N-\ell}} \prec J_{j_{N-\ell+1},i'_{N-\ell+1}} \implies a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) - a(S, J_{j_{N-\ell},i'_{N-\ell}}) \leq T^{max}_{j_{N-\ell}} \tag{6.27}$$

Proof. Let a processing chain instance $E'_j = \langle J_{j1,i'_1}, J_{j2,i'_2}, \ldots, J_{jN,i'_N}\rangle$ and only consider the case that $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) > f(S, J_{j_{N-\ell},i'_{N-\ell}})$, since the other case is already proved in Theorem 6.5.

Step 1. Under the assumption that $\tau_{j_{N-\ell}}$ precedes $\tau_{j_{N-\ell+1}}$ according to Definition 6.10 for some $\ell \in \{0, \ldots, N-1\}$, it follows that

$$a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq a(S, J_{j_{N-\ell},i'_{N-\ell}+1}) \tag{6.28}$$

which we hereinafter prove by contradiction. Assume for contradiction that Eq. (6.28) does not hold, i.e., $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) > a(S, J_{j_{N-\ell},i'_{N-\ell}+1})$. Then it must be that $s(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) > f(S, J_{j_{N-\ell},i'_{N-\ell}+1})$, due to the precedence property, which in turn contradicts the assumption that $J_{j_{N-\ell},i'_{N-\ell}}$ is a job in the analyzed processing chain instance.

Step 2. Given the prior Step 1, we have that $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq a(S, J_{j_{N-\ell},i'_{N-\ell}+1})$, which implies $a(S, J_{j_{N-\ell+1},i'_{N-\ell+1}}) \leq a(S, J_{j_{N-\ell},i'_{N-\ell}}) + T^{max}_{j_{N-\ell}}$, due to the maximal interarrival time constraints in the semi-sporadic task model. □

Based on these lemmas, the maximal sensor data propagation latency can be bounded.

Theorem 6.8 (Maximal Sensor Data Propagation Latency)**.** *The maximal sensor data propagation latency of a processing chain E_j is upper bounded by Eq. (6.29) if the jobs of task τ_{j_N} are scheduled* ***non-preemptively*** *and upper bounded by Eq. (6.30) if the jobs of task τ_{j_N} are scheduled* ***preemptively****.*

$$R_{j_N} - (C^{max}_{j_N} - C^{min}_{j_1}) + \sum_{\ell=1}^{N-1} T^{max}_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} \tag{6.29}$$

$$R_{j_N} - (C^{min}_{j_N} - C^{min}_{j_1}) + \sum_{\ell=1}^{N-1} T^{max}_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} \tag{6.30}$$

Proof. The length of a processing chain instance is given by the absolute time difference of the starting time of the job in the last task (sink processing vertex) and the finishing time of the job of the first task (source processing vertex), i.e.,

$$s(S, J_{j_N,i'_N}) - f(S, J_{j_1,i'_1}) \leq s(S, J_{j_N,i'_N}) - (a(S, J_{j_1,i'_1}) + C^{min}_{j_1}) \tag{6.31}$$

If τ_{j_N} is scheduled non-preemptively then we know by Lemma 6.3 that

$$Eq. (6.31) \leq a(S, J_{j_N, i'_N}) + R_{j_N} - C^{max}_{j_N} - (a(S, J_{j_1, i'_1}) + C^{min}_{j_1}) \quad (6.32)$$

$$= R_{j_N} - (C^{max}_{j_N} + C^{min}_{j_1}) + \sum_{\ell=1}^{N-1} a(S, J_{j_{\ell+1}, i'_{\ell+1}}) - a(S, J_{j_\ell, i'_\ell}) \quad (6.33)$$

$$(By\ Lemma.\ 6.5) \leq R_{j_N} - (C^{max}_{j_N} + C^{min}_{j_1}) + \sum_{\ell=1}^{N-1} T^{max}_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} \quad (6.34)$$

If τ_{j_N} is scheduled preemptively, applying Lemma 6.4 with similar arguments yields

$$\leq R_{j_N} - (C^{min}_{j_N} + C^{min}_{j_1}) + \sum_{\ell=1}^{N-1} T_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} \quad (6.35)$$

which concludes the proof. □

6.4.3 MAXIMAL SENSOR DATA TIME-STAMP DIFFERENCE

In this section, we use the analyses for the minimal and maximal sensor data propagation latency to devise an analysis for the maximal sensor data time-stamp difference at each fusion vertex in the processing graph.

Definition 6.11 (Partial Processing Chain)**.** *A partial processing chain E_j^k of a processing chain $E_j = \langle \tau_{j_1}, \ldots, \tau_{j_{N_j}} \rangle$ is the consecutive sub-sequence of processing vertices in E_j, which starts in τ_{j_1} and ends in τ_{j_k} if $k \leq N_j$, and E_j otherwise.*

Theorem 6.9 (Maximal Sensor Data Time-Stamp Difference)**.** *Let E_j and E_k denote two processing chains in the processing graph G and let E_j^u, E_k^v denote two partial processing chains such that $\tau' := \tau_{j_u} = \tau_{k_v}$ and $\tau_{j_1} \neq \tau_{k_1}$, i.e., both partial processing chains have different sensor processing vertices at the source and a common fusion vertex. Then the maximal sensor data time-stamp difference, of two sensor data samples received by any job of τ', is at most*

$$T_{j_1} + R_{j_{N_j}} - C^{max}_{j_{N_j}} + \sum_{\ell=2}^{N_j-1} T_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} - \sum_{\ell=2}^{N_k-2} C^{min}_{k_\ell} \quad (6.36)$$

if $\tau_{j_{N_j}}$ is scheduled non-preemptively and

$$T_{j_1} + R_{j_{N_j}} - C^{min}_{j_{N_j}} + \sum_{\ell=2}^{N_j-1} T_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} - \sum_{\ell=2}^{N_k-2} C^{min}_{k_\ell} \quad (6.37)$$

if $\tau_{j_{N_j}}$ is scheduled preemptively.

Proof. Let $E_j = \langle \tau_{j_1}, \ldots, \tau_{j_{N_j}} \rangle$ denote a processing chain and τ_{j_1} represents the sensor such that $a(S, J_{j_1, i'_1}) = s(S, J_{j_1, i'_1}) = f(S, J_{j_1, i'_1})$ holds. With reference to the

prior analyses, we can calculate the minimal sensor data propagation latency as follows:

$$s(S, J_{j_{N_j}, i'_{N_j}}) - f(S, J_{j_1, i'_1}) \geq f(S, J_{j_{N_j-1}, i'_{N_j-1}}) - s(S, J_{j_1, i'_1}) = \sum_{\ell=1}^{N_j-2} C_{j_\ell}^{min} \tag{6.38}$$

and the maximal sensor data propagation latency is $s(S, J_{j_{N_j}, i'_{N_j}}) - f(S, J_{j_1, i'_1})$ in the case that $\tau_{j_{N_j}}$ is scheduled non-preemptively as follows:

$$\leq a(S, J_{j_{N_j}, i'_{N_j}}) + R_{j_{N_j}} + C_{j_{N_j}}^{max} - a(S, J_{j_1, i'_1}) \tag{6.39}$$

$$\leq R_{j_{N_j}} + C_{j_{N_j}}^{max} + T_{j_1}^{max} + \sum_{\ell=2}^{N_j-1} T_{j_\ell}^{max} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} \tag{6.40}$$

The absolute difference of the maximal and minimal sensor data propagation latency of two chains E_j and E_k is at most

$$R_{j_{N_j}} + C_{j_{N_j}}^{max} + T_{j_1} + \sum_{\ell=2}^{N_j-1} T_{j_\ell} + \begin{cases} R_{j_\ell} & \text{if } \tau_{j_\ell} \not\prec \tau_{j_{\ell+1}} \\ 0 & \text{otherwise} \end{cases} - \sum_{\ell=1}^{N_k-2} C_{k_\ell}^{min} \tag{6.41}$$

□

Algorithm 8 Maximal Sensor Data Time-Stamp Difference

Require: Processing graph G, mapping σ, scheduling policy of processing units, task-level fixed-priority for $\tau_i \in V$

1: **for each** processing unit $P_k \in \{P_1, \ldots, P_m\}$ **do**
2:     **for each** τ_i which is assigned to processing unit P_k **do**
3:         **if** P_k uses non-preemptive scheduling **then**
4:             $R_i \leftarrow$ time-demand analysis for non-preemptive task sets; ▷ Determine the worst-case response time R_i of τ_i on $\sigma(\tau_i)$.
5:         **else if** P_k uses preemptive scheduling **then**
6:             $R_i \leftarrow$ time-demand analysis for preemptive task sets;
7: **for each** fusion task $\tau_i \in V$ **do**
8:     Find the set of all partial processing chains according to Definition 6.11 in G which end in fusion task τ_i;
9:     Let $\rho(\tau_i) \leftarrow \left\{E_j^k \in G \mid k = i\right\}$ denote the set of all partial processing chains in which task $\tau_i \in V$ occurs and in which all other tasks after τ_i are removed from the chains;
10:     $\Delta_i \leftarrow$ Calculate the maximal sensor data time-stamp difference at τ_i using Theorem 6.9;
return $\Delta_1, \ldots, \Delta_n$; ▷ Maximal sensor data propagation latency jitter for all sensor fusion tasks in G.

On the basis of Theorem 6.9, we propose the following Algorithm 8 to compute the maximum sensor data time-stamp difference for each fusion task in a given processing graph G, task to processing unit mapping σ, and task-level fixed-priorities for all tasks. The algorithm consists of a worst-case response time analysis stage for the given processing graph and the scheduling of the tasks on

the processing units from Line 1 to Line 6 and a subsequent maximal sensor data time-stamp difference analysis from line 7 to Line 10 on the basis of the results of the first stage. To be more precise for each task τ_i, the sub task set

$$\tau_i \cup \{\tau \in V \mid \sigma(\tau) = \sigma(\tau_i) \ \wedge \Pi(\tau) \geq \Pi(\tau_i)\} \tag{6.42}$$

task-level fixed-priorities Π are assumed to be determined by the system designer

is subjected to a worst-case response-time analysis of task τ_i. The task-level fixed-priorities Π are assumed to be determined by the system designer before analysis. If the processing unit $\sigma(\tau_i)$ is a processing unit which employs non-preemptive scheduling such as for instance a GPU, then the respective non-preemptive fixed-priority scheduling analysis for sporadic constrained-deadline tasks is used. Conversely, if the processing unit $\sigma(\tau_i)$ uses preemptive scheduling then an appropriate preemptive analysis variant is used. Let $\rho(\tau_i) := \left\{E_j^k \in G | \ k = i\right\}$ denote the set of all partial processing chains in which task $\tau_i \in V$ occurs and in which all other tasks after τ_i are removed from the chains, then the difference in sensor data propagation latency Δ_i as observed by a job of task τ_i is at most

$$\Delta_i \leq \max\left\{E_j^u \neq E_k^v \in \rho(\tau_i) \mid \text{Theorem 6.9}\right\} \tag{6.43}$$

We recall, that the motivation of the proposed analysis – to determine the maximal sensor data time-stamp difference at any fusion task – was to be modular, and to be robust, with respect to execution time uncertainty, globally asynchronous processing units and sensors. The proposed analyses provide that; however at the cost of analysis precision, i.e., the proposed analyses are not exact.

6.5 DISCUSSION & EXTENSIONS

The modular design and abstract definition of *precedence* in Definition 6.10, allows to extend the provided approach to other task models and scheduling algorithms, such as the rigid gang task scheduling; if appropriate conditions to achieve task precedence can be derived.

fixed-priority non-preemptive rigid gang scheduling

In the case of *fixed-priority non-preemptive rigid gang scheduling*, which may be a suitable task model for the execution on GPUs, the following condition can be derived, which implies the *precedence* property of two rigid gang tasks, required to improve the analysis. In non-preemptive fixed-priority rigid gang scheduling, a task τ_i is eligible to start execution at time t, if there are E_i processors idle at time t, and τ_i is the highest-priority task competing for those E_i processors.

Lemma 6.10 (Non-Preemptive Rigid Gang Precedence). *If for any two sporadic gang tasks τ_i and τ_k all the following conditions hold under fixed-priority non-preemptive rigid gang scheduling*

- $(E_i \leq E_k) \wedge (E_i + E_k > M)$
- $\Pi(\tau_i) > \Pi(\tau_k)$

then task τ_i precedes τ_k.

Proof. Let J_i^* denote a job of τ_i and J_k^* denote a job of task τ_k such that $a(S, J_i^*) \leq a(S, J_k^*)$. If the intervals $[a(S, J_i^*), f(S, J_i^*))$ and $[a(S, J_k^*), f(S, J_k^*))$ have no intersection then clearly, $f(S, J_i^*) \leq s(S, J_k^*)$ holds and thus we only inspect the time interval in which both jobs are pending and none has started execution. By the stated assumptions, J_i^* has higher priority than J_k^* and $E_i \leq E_k$, i.e., the non-execution of J_i^* implies the non-execution of J_k^*. Moreover, by the assumption that $E_i + E_k > M$ the execution is mutually exclusive and in consequence $f(S, J_i^*) \leq s(S, J_k^*)$. □

In the case of *fixed-priority preemptive stationary rigid gang scheduling*, which may be used for parallel tasks on multiprocessors, the following condition implies the *precedence* property of two rigid gang tasks.

fixed-priority preemptive stationary rigid gang scheduling

Corollary 6.11 (Preemptive Stationary Rigid Gang Precedence)**.** *If for two gang tasks τ_i and τ_k all of the following conditions hold under fixed-priority stationary gang scheduling then τ_i* precedes *τ_k if:*

- *The stationary gang assignments satisfy $A_i \subseteq A_k$;*
- *And τ_i has a higher priority than τ_k,*

where the stationary gang assignments $A_i, A_k \in \mathcal{P}(\{P_1, \ldots, P_M\})$ are gang to processor assignments as described in Chapter 3.

As proved in Chapter 3, when analyzing the response-time of task τ_k, the higher-priority tasks may have self-suspension like behaviour. However if the stationary gang assignments are such that $A_i \subseteq A_k$ then τ_i has no self-suspension behaviour with respect to task τ_k. Thus the precedence behaviour is proved similar to Corollary 6.6.

6.6 CONCLUSION

With regards to the dissertation hypothesis, in this chapter, we proposed analyses to determine the maximal sensor data time-stamp difference at any fusion task in a given processing graph. The analyses are modular and robust with respect to parameter uncertainty. Starting from a formal definition of the studied systems a processing graph which describes the data flow among independent tasks, we propose a modular analysis, for any task which combines different sources of sensory information to provide an enhanced system state. Most notably, we make no assumptions on synchronous clocks on different processing units, and allow different and independent scheduling algorithms, for each different processing unit.

analyses are modular and robust with respect to parameter uncertainty

7 CONCLUSIONS AND OUTLOOK

The stated hypothesis of this dissertation is that either the parameter uncertainty and hardware peculiarities of modern architectures must be considered in the scheduling algorithm design and the formal schedulability analyses, or the predictability of the hardware must be increased.

In addition, the system should be composed of modular components, which are exposed to another and the real-time operating system by formal contracts. More precisely, each component is designed such that the formal analyses only depend on formal properties – promised by the formal contracts – which can be observed and enforced in the system.

This chapter first recapitulates the contributions of this dissertation in Section 7.1 *Summary of the Contributions*. In Section 7.2 *Examination of the Dissertation Hypothesis*, it is examined whether these contributions support the dissertation hypothesis. Afterwards, an outlook at possible future work is given in Section 7.3 *Future Work*. In Section 7.4 *Final Remarks and Outlook*, the dissertation is concluded with some final remarks and an outlook is given.

7.1 SUMMARY OF THE CONTRIBUTIONS

The contributions of this dissertation are summarized according to the chapters where they are detailed.

7.1.1 TIMING PREDICTABLE PROTOCOLS

In Chapter 3, we examine the design of protocols (and scheduling algorithms), which increase predictability and allow for safe worst-case response-time analyses. We propose a family of simultaneous progression switching protocols for real-time NoC arbitration, which is described by the *all-or-nothing* property and provides increased predictability at the cost of decreased average case performance. On the basis of the all-or-nothing property, we created a novel gang scheduling algorithm called *stationary rigid gang* scheduling, in which gangs are statically assigned to a specific sub set of processors, thus eliminating task migration. Both approaches share a common underlying modular analysis concept and framework, which is based on a formal reduction to suspension-aware uniprocessor scheduling theory.

In the first contribution, a novel rigid gang scheduling algorithm called *stationary rigid gang scheduling* is proposed for sporadic real-time gang tasks with constrained deadlines. Several sufficient schedulability analyses are proposed for task-level fixed-priority scheduling algorithms. A special class of assignment algorithms – based on the concept of *consecutive stationary gang assignment* – is

shown to admit a parametric speed-up factor bound with respect to an optimal scheduling algorithm. It is shown that *consecutive stationary gang assignments* yield beneficial theoretical properties, which can be used to upper-bound the worst-case interference suffered by any task according to the ratio of the gang sizes of two tasks. The algorithm is compared to the state-of-the-art schedulability analysis for global EDF by Dong and Liu [DL17] using synthetically generated sporadic real-time task systems with implicit deadlines. The evaluation results for implicit deadlines show that our algorithm outperforms the algorithm by Dong and Liu [DL17] and the evaluations for constrained-deadline task systems demonstrate reasonable levels of schedulable task sets.

In the second contribution, the fundamental difficulty of using scheduling theory based analyses for pipelined flit-based transmissions is formally discussed. Hereinafter, a novel timing predictable architecture and design of a two-dimensional NoC system, which is suitable for real-time multicore systems is presented. To achieve timing predictability by design, a family of less flexible switching protocols called simultaneous progression switching protocols is proposed. In this protocol, all links that are used by a flow transmit one flit of this flow (if it exists) simultaneously or none of them transmits any flit of this flow. Based on this simultaneous progression property, we reduced the schedulability of the NoC to the discrete time uniprocessor self-suspension scheduling problem. By construction, any minimal and non-minimal route is deadlock-free and therefore the path diversity can be better utilized in order to distribute the load over the links. An implementation, including router design, and arbitration algorithm is provided and evaluated with synthetically generated data. The evaluations hint to scalability issues in our proposed implementation for larger networks. However, for smaller to medium sized NoCs, our proposal suggests to be a beneficial first step towards the design and analysis of timing-predictable switching protocols of NoCs, which allow for safe response-time analyses.

7.1.2 HIERARCHICAL PARALLEL DAG SCHEDULING

In Chapter 4, hierarchical scheduling of parallel DAG tasks is studied. In the first contribution, the probabilistic conditional parallel DAG task model is proposed to express structural uncertainty during execution. A hierarchical scheduling algorithm for the analysis of the probabilistic conditional parallel DAG task model is proposed, and design rules are devised that provide guarantees such as bounded tardiness and probabilistic upper bounds for k-consecutive deadline misses. The approach is sustainable, because any early completions – due to scheduling or dynamic conditional DAG structures – are handled by the reservation system and the parametric abstraction of the workload model.

In the second contribution, the *parallel path progression* concept is proposed, which allows to consider the parallel progression of multiple paths in the DAG during execution. This property is implemented using a sub task level fixed-priority policy and a preemptive fixed-priority list-scheduling algorithm. A polynomial-time algorithm is provided, which finds a path collection that covers all vertices in the DAG and hence either yields an optimal response-time or a

parametric approximation algorithm for an optimal response-time (for a given number of processors). The concept is extended to two hierarchical scheduling algorithms, namely a sporadic arbitrary-deadline gang reservation system and a sporadic arbitrary-deadline ordinary reservation system. The hierarchical scheduling algorithm can be applied to sporadic arbitrary-deadline DAG tasks, which may be executed concurrently with tasks described by a different task model, e.g., ordinary sequential tasks. For both reservation systems, we provide response-time analyses and algorithms to generate and provision feasible reservation systems. The approach is evaluated using synthetically generated DAG task sets. The evaluations demonstrate that the approach advances the state of the art in high-parallelism scenarios and show that the performance of the approach is between the start-of-the-art and federated scheduling in more sequential scenarios. Moreover, a stricter *path-monotonic progression* concept is proposed, which allows to design suspension-aware reservation systems mitigating active-idling of the reservation systems.

7.1.3 REGULATOR-BASED ADAPTIVITY

In Chapter 5, fault-tolerance as a supplementary reliability aspect of real-time systems – in spite of dynamic external causes of fault – is examined. To assure that an acceptable quality-of-service (QoS), i.e., fault-tolerance, can be achieved, upper-bounds on consecutive erroneous job executions, and guaranteed m error-free executions out of k consecutive job executions are studied. Using job variants, which trade off increased execution time demand with increased error protection, a state-based policy selection strategy is proposed. The policy guarantees that all reachable states comply with the QoS constraints, whilst minimizing the expected system utilization and assuring hard real-time compliance of the task system. The state-based policy selection allows the usage of machine learning techniques – which infer error information during operation – to provide hard guarantees. Extensive numerical evaluations have shown that the proposed approaches require significantly decreased system utilization compared to the state of the art. The learning- and runtime overheads of the proposed approaches, are shown to be reasonably small in the evaluations.

7.1.4 MAXIMAL SENSOR DATA TIME-STAMP DIFFERENCE

In Chapter 6, modular analyses for the *maximal time-stamp differences* of sensor data are presented. The results show that in spite of the complex heterogeneous architecture and globally asynchronous processing units, the maximal time-stamp difference of any two sensor data samples can be upper bounded at the time of sensor fusion. More precisely, given a multi-rate task set, and under the assumption that each task is schedulable on its respective processing units according to readily available task-level fixed-priority worst-case response-time analyses for non-preemptive or preemptive scheduling algorithms, an algorithm is proposed to calculate the *maximal time-stamp differences* for all signals that are merged at a sensor fusion task. In addition, based on an abstract *precedence property*, which

depends on the task model and scheduling algorithm, the presented analyses can be further refined. In particular, task precedence properties are presented for non-preemptive rigid gang scheduling and preemptive stationary rigid gang scheduling.

7.2 EXAMINATION OF THE DISSERTATION HYPOTHESIS

The question remains whether the contributions support the hypothesis:

The exploration of timing predictable protocols, task models and scheduling algorithms in Chapter 3 showed that, by imposing additional constraints, safe worst-case response-time analyses can be provided even for challenging network-on-chip arbitration problems. The additional imposed *all-or-nothing* property also simplified router design, and routing protocol constraints, which improves worst-case centric predictability.

Moreover, the stationary rigid gang scheduling algorithm was proposed, which is more restrictive than traditional rigid gang scheduling by imposing execution restrictions of gangs to only execute on assigned processors. It was shown that the schedulability problem can be reduced to the uniprocessor suspension-aware task-level fixed-priority scheduling problem by virtue of the properties of the novel scheduling algorithm. The stationary rigid gang scheduling algorithm can also be used in conjunction with the hierarchical parallel DAG scheduling approaches presented in Chapter 4, which further attests to the modularity. Final evidence of whether stationary rigid gang scheduling improves the predictability in a real-time operating system implementation remains to be shown. It is however evident that task contexts do not have to be migrated, and the certainty of which processors are used for execution allows for more precise shared resource contention coordination and analyses.

The contributions of Chapter 4 are hierarchical, i.e., reservation based scheduling algorithms, which decompose the DAG scheduling problem into two modular scheduling problems. Namely, the reservation system is exposed as a standard task model, e.g., the (ordinary) sporadic arbitrary-deadline task, sporadic rigid gang task, or sporadic suspension-aware task, for which any readily available state-of-the art scheduling algorithm and analysis can be used. Secondly, the problem of reservation provisioning and scheduling of the DAG task on the premise of the promised service, is analyzed and algorithms are provided. The presented hierarchical scheduling algorithms, allow uncertainty, e.g., structural uncertainty of a conditional DAG task. This model considers conditional branching decisions, and supports some tardy jobs to finish without causing unpredictable cascading effects into system. The provided hierarchical scheduling algorithms are sustainable with respect to early completion, and uncertain worst-case execution time estimates can be contained within a reservation. By virtue of our *parallel path progression* intra-task prioritization, massively parallel architectures can be almost optimally exploited, using reservation systems.

In Chapter 5, fault-tolerance as a supplementary reliability aspect of real-time systems is examined. The results show how optimization- and machine learning

techniques can be used in spite of dynamic external causes of fault to adaptively improve average case performance and still provide hard QoS guarantees.

Lastly, in Chapter 6, it was shown how a complex problem can be decomposed into unrelated tasks using standard task models. On the basis of a mapping of tasks to processing units – which can support preemptive or non-preemptive scheduling, and do not have to be globally synchronized – analyses and algorithms for the *maximal sensor data time-stamp difference* are provided. The analysis is modular, since all formal presumptions, which must be met, are abstractly stated, e.g., the *precedence* property. This property can be provided for many task models and scheduling algorithms.

7.3 FUTURE WORK

Resulting from the observations and results in Chapter 3 regarding the rigid stationary gang scheduling algorithm, it would be most interesting to come up with an efficient implementation in a real-time operating system and to evaluate the benefits or problems in the real system. From a theoretical perspective, other stationary gang assignment algorithms, e.g., optimization-based approaches, may be studied for improved schedulability. At last, an extension from task-level fixed-priority to job-level fixed-priority is a direction for future work. With regards, to the proposed simultaneous progression protocol and concept implementation, further considerations and FPGA prototypes are required to fully examine the limits and potentials for improvement.

Regarding, the hierarchical scheduling algorithms presented in Chapter 4, the most interesting directions of future work are to examine if further and other intra-task prioritization can provide more interesting properties. Another interesting question is if any of the observations and techniques can be expanded to heterogeneous systems with *typed* or *grouped* DAG tasks. That is, if sub tasks are tied to specific processors. With respect to probabilistic DAG task scheduling, further work is required to improve the response-time of a tardy job like is possible in federated scheduling [LAG+14], but with less resource requirements. Moreover, resource reclamation and suspension-aware reservation system design is a interesting and challenging direction of future study.

The automata-based regulator approach in Chapter 5 allows adaptivity, and the safe utilization of unpredictable and not explainable machine learning techniques for optimization, while providing hard guarantees. The visionary goal is to include the schedulability problem in the state space representation, which however is computationally intractable in the general case. While the per-task deterministic finite automata (DFA) can be easily composed using the potency automata approach to a per-task-set DFA, the resulting connection to time and the schedule is unclear. In future work, special cases, such as non-preemptive scheduling for periodic task sets – for which a state space representation of the scheduling problem has been proposed in [RNN22] – can be a starting point for extension.

The maximal sensor data time-stamp difference analysis in Chapter 6, can potentially be improved if an improved best-case response time analysis is devised,

and incorporated. Further, it would be interesting to practically evaluate if an increased analysis precision, by assuming more formal properties and requirements, is worth the loss of model robustness and modularity.

7.4 FINAL REMARKS AND OUTLOOK

This dissertation examined the design and formal verification problem of modern and complex real-time systems. In particular, modular scheduling algorithms for fine-grained parallel task models on multicore architectures were studied, which are robust towards parameter uncertainty. Moreover, external disturbances, e.g., radiation induced errors, unsound worst-case execution time estimates, or environmental dependent control flows are considered in this dissertation.

Most likely, future industry practice will be to use measurement based pWCET estimates, even at the cost of soundness, and subsequently the need to react to this uncertainty with adaptivity. The challenge is in providing hard guarantees and allow for adaptivity at the same time. It should be noted that uncertainty poses a significant challenge to *worst-case state* centric real-time system verification and design with respect to model- and analysis fidelity, if not considered properly. Most importantly, temporal and spatial isolation is a key requirement to prevent uncertainty induced effects from cascading into the system.

While admittedly, the overall research problem of modular, safe, robust and adaptive real-time systems remains challenging, with many open questions left, the results in this dissertation provide promising results and directions to be further pursued.

BIBLIOGRAPHY

[Aba+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[Abd17] A.B. Abdallah. *Advanced Multicore Systems-On-Chip: Architecture, On-Chip Network, Design*. Springer Singapore, 2017. ISBN: 9789811060922.

[AJE+15] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul. "Wormhole networks properties and their use for optimizing worst case delay analysis of many-cores." In: *Proc. of the 10th IEEE International Symposium on Industrial Embedded Systems*. 2015, pp. 1–10. DOI: 10.1109/SIES.2015.7185041.

[ABD+13] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. "Impact of Resource Sharing on Performance and Performance Prediction: A Survey." In: *CONCUR*. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013, pp. 25–43.

[APC+17] Jaume Abella, Maria Padilla, Joan del Castillo, and Francisco J. Cazorla. "Measurement-Based Worst-Case Execution Time Estimation Using the Coefficient of Variation." In: *ACM Trans. Design Autom. Electr. Syst.* 22.4 (2017), 72:1–72:29.

[ANN+22] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. "A comprehensive survey of industry practice in real-time systems." In: *Real Time Syst.* 58.3 (2022), pp. 358–398.

[AA05] Tarek A. AlEnawy and Hakan Aydin. "Energy-Constrained Scheduling for Weakly-Hard Real-Time Systems." In: *RTSS*. IEEE Computer Society, 2005, pp. 376–385.

[AY19] Waqar Ali and Heechul Yun. "RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems." In: *RTAS*. IEEE, 2019, pp. 143–155.

[AFM+96] Martin Helmut Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. "Cache Behavior Prediction by Abstract Interpretation." In: *SAS*. Vol. 1145. Lecture Notes in Computer Science. Springer, 1996, pp. 52–66.

[AM11] Sebastian Altmeyer and Claire Maiza. "Cache-related preemption delay via useful cache blocks: Survey and redefinition." In: *J. Syst. Archit.* 57.7 (2011), pp. 707–719.

[AS00] James H. Anderson and Anand Srinivasan. "Pfair scheduling: beyond periodic task systems." In: *RTCSA*. IEEE Computer Society, 2000, pp. 297–306.

[ASO+20] Luis Fernando Arcaro, Karila Palma Silva, Rômulo Silva de Oliveira, and Luís Almeida. "Reliability Test based on a Binomial Experiment for Probabilistic Worst-Case Execution Times." In: *RTSS*. IEEE, 2020, pp. 51–62.

[ABN22] Federico Aromolo, Alessandro Biondi, and Geoffrey Nelissen. "Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling." In: *ECRTS*. Vol. 231. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 13:1–13:18.

[AEF+14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. "Building Timing Predictable Embedded Systems." In: *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014), 82:1–82:37. ISSN: 1539-9087. DOI: 10.1145/2560033. URL: http://doi.acm.org/10.1145/2560033.

[ASE12] Philip Axer, Maurice Sebastian, and Rolf Ernst. "Probabilistic response time bound for CAN messages with arbitrary deadlines." In: *DATE*. IEEE, 2012, pp. 1114–1117.

[AES+16] Hamdi Ayed, Jérôme Ermont, Jean-luc Scharbarg, and Christian Fraboul. "Towards a unified approach for worst-case analysis of Tilera-like and Kalray-like NoC architectures." In: *World Conf. on Factory Communication Systems, WiP Session*. IEEE. Aveiro, Portugal, 2016.

[Bae09] Jean-Loup Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. 1st. USA: Cambridge University Press, 2009.

[Bak03] Theodore P. Baker. "Multiprocessor EDF and Deadline Monotonic Schedulability Analysis." In: *IEEE Real-Time Systems Symposium*. 2003, pp. 120–129. DOI: 10.1109/REAL.2003.1253260. URL: http://dx.doi.org/10.1109/REAL.2003.1253260.

[Bar21] Sanjoy K. Baruah. "Feasibility Analysis of Conditional DAG Tasks is co-NPNP-Hard." In: *RTNS*. ACM, 2021, pp. 165–172.

[BBM15] Sanjoy K. Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. "The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks." In: *ECRTS*. IEEE Computer Society, 2015, pp. 222–231.

[BBM+12] Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. "A Generalized Parallel Task Model for Recurrent Real-time Processes." In: *RTSS*. IEEE Computer Society, 2012, pp. 63–72.

[BCG+99] Sanjoy K. Baruah, Deji Chen, Sergey Gorinsky, and Aloysius K. Mok. "Generalized Multiframe Tasks." In: *Real-Time Systems* 17.1 (1999), pp. 5–22. DOI: 10.1023/A:1008030427220.

[BF07] Sanjoy K. Baruah and Nathan Fisher. "Global Deadline-Monotonic Scheduling of Arbitrary-Deadline Sporadic Task Systems." In: *OPODIS*. Vol. 4878. Lecture Notes in Computer Science. Springer, 2007, pp. 204–216.

[BF08] Sanjoy K. Baruah and Nathan Fisher. "Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems." In: *ICDCN*. Vol. 4904. Lecture Notes in Computer Science. Springer, 2008, pp. 215–226.

[BF05] Sanjoy K. Baruah and Nathan Fisher. "The Partitioned Multiprocessor Scheduling of Sporadic Task Systems." In: *RTSS*. IEEE Computer Society, 2005, pp. 321–329.

[BL04] Sanjoy K. Baruah and Giuseppe Lipari. "A Multiprocessor Implementation of the Total Bandwidth Server." In: *IPDPS*. IEEE Computer Society, 2004.

[BM21] Sanjoy K. Baruah and Alberto Marchetti-Spaccamela. "Feasibility Analysis of Conditional DAG Tasks." In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 12:1–12:17.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. "Preemptively scheduling hard-real-time sporadic tasks on one processor." In: *proceedings Real-Time Systems Symposium (RTSS)*. 1990, pp. 182–190. DOI: 10.1109/REAL.1990.128746.

[Bar15a] Sanjoy Baruah. "Federated Scheduling of Sporadic DAG Task Systems." In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. 2015, pp. 179–186. DOI: 10.1109/IPDPS.2015.33. URL: http://dx.doi.org/10.1109/IPDPS.2015.33.

[Bar07] Sanjoy Baruah. "Techniques for Multiprocessor Global Schedulability Analysis." In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium*. 2007, pp. 119–128.

[Bar15b] Sanjoy Baruah. "The federated scheduling of constrained-deadline sporadic DAG task systems." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE*. 2015, pp. 1323–1328. URL: http://dl.acm.org/citation.cfm?id=2757121.

[Bar15c] Sanjoy Baruah. "The federated scheduling of systems of conditional sporadic DAG tasks." In: *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*. 2015.

[BBA11] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. "Is Semi-Partitioned Scheduling Practical?" In: *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*. ECRTS '11. 2011, pp. 125–135.

[Bau05] R.C. Baumann. "Radiation-induced soft errors in advanced semiconductor technologies." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449.

[BDM+16] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. "Synthesizing Job-Level Dependencies for Automotive Multi-rate Effect Chains." In: *RTCSA*. IEEE Computer Society, 2016, pp. 159–169.

[BCM19] Slim Ben-Amor, Liliana Cucu-Grosjean, and Dorin Maxim. "Worst-case response time analysis for partitioned fixed-priority DAG tasks on identical processors." In: *ETFA*. IEEE, 2019, pp. 1423–1426.

[BCM+20a] Slim Ben-Amor, Liliana Cucu-Grosjean, Mehdi Mezouak, and Yves Sorel. "Probabilistic Schedulability Analysis for Precedence Constrained Tasks on Partitioned Multi-core." In: *ETFA*. IEEE, 2020, pp. 345–352.

[BCM+20b] Slim Ben-Amor, Liliana Cucu-Grosjean, Mehdi Mezouak, and Yves Sorel. "Probabilistic Schedulability Analysis for Real-time Tasks with Precedence Constraints on Partitioned Multi-core." In: *ISORC*. IEEE, 2020, pp. 142–143.

[BMC16] Slim Ben-Amor, Dorin Maxim, and Liliana Cucu-Grosjean. "Schedulability analysis of dependent probabilistic real-time tasks." In: *RTNS*. ACM, 2016, pp. 99–107.

[BBL01] Guillem Bernat, Alan Burns, and Albert Llamosí. "Weakly Hard Real-Time Systems." In: *IEEE Trans. Computers* 50.4 (2001), pp. 308–321. DOI: 10.1109/12.919277. URL: https://doi.org/10.1109/12.919277.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan M. Petters. "pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems." In: *WCET*. Vol. MDH-MRTC-116/2003-1-SE. Department of Computer Science and Engineering, Mälardalen University Västerås, Sweden, 2003, pp. 21–38.

[BC07] Marko Bertogna and Michele Cirinei. "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms." In: *Real-Time Systems Symposium (RTSS)*. 2007, pp. 149–160. DOI: 10.1109/RTSS.2007.31. URL: http://dx.doi.org/10.1109/RTSS.2007.31.

[BB06a] Adam Betts and Guillem Bernat. "Tree-Based WCET Analysis on Instrumentation Point Graphs." In: *ISORC*. IEEE Computer Society, 2006, pp. 558–565.

[BB05] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests." In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

[BS18] Alessandro Biondi and Youcheng Sun. "On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling." In: *Real Time Syst.* 54.3 (2018), pp. 515–536.

[BDOD+94] J. Błażewicz, P. Dell' Olmo, M. Drozdowski, and M.G. Speranza. "Corrigendum to: Scheduling multiprocessor tasks on three dedicated processors." In: *Inf. Process. Lett.* 49.5 (1994), pp. 269–270.

[BAH+15] Konstantinos Bletsas, Neil Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. *Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions*. Tech. rep. CISTER-TR-150713. CISTER, 2015.

[BMSS+13] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. "Feasibility Analysis in the Sporadic DAG Task Model." In: *ECRTS*. 2013, pp. 225–233.

[BMS+13] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. "Feasibility Analysis in the Sporadic DAG Task Model." In: *ECRTS*. IEEE Computer Society, 2013, pp. 225–233.

[BC94] Rajendra V. Boppana and Suresh Chalasani. "Fault-tolerant routing with non-adaptive wormhole algorithms in mesh networks." In: *SC*. IEEE Computer Society, 1994, pp. 693–702.

[Bou98] Jean-Yves Le Boudec. "Application of Network Calculus to Guaranteed Service Networks." In: *IEEE Trans. Inf. Theory* 44.3 (1998), pp. 1087–1096.

[BDG+18] Marc Boyer, Benoît Dupont de Dinechin, Amaury Graillat, and Lionel Havet. "Computing Routes and Delay Bounds for the Network-on-Chip of the Kalray MPPA2 Processor." In: *ERTS^2 2018A*. 2018.

[Bra11a] Björn B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems." PhD thesis. The University of North Carolina at Chapel Hill, 2011.

[Bra11b] Björn B. Brandenburg. "Scheduling and locking in multiprocessor real-time operating systems." PhD thesis. University of North Carolina, Chapel Hill, USA, 2011.

[BG16] Björn B. Brandenburg and Mahircan Gul. "Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations." In: *RTSS*. IEEE Computer Society, 2016, pp. 99–110.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[BCH+16] Georg von der Brüggen, Kuan-Hsun Chen, Wen-Hung Huang, and Jian-Jia Chen. "Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments." In: *RTSS*. IEEE Computer Society, 2016, pp. 303–314.

[BPC+18] Georg von der Brüggen, Nico Piatkowski, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. "Efficiently Approximating the Probability of Deadline Misses in Real-Time Systems." In: *ECRTS*. Vol. 106. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 6:1–6:22.

[BUC+17] Georg von der Brüggen, Niklas Ueter, Jian-Jia Chen, and Matthias Freier. "Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems." In: *RTNS*. ACM, 2017, pp. 108–117.

[BS15] Tobias Bund and Frank Slomka. "Sensitivity Analysis of Dropped Samples for Performance-Oriented Controller Design." In: *ISORC*. IEEE Computer Society, 2015, pp. 244–251.

[BE00] Alan Burns and Stewart Edgar. "Predicting computation time for advanced processor architectures." In: *12th Euromicro Conference on Real-Time Systems (ECRTS 2000), 19-21 June 2000, Stockholm, Sweden, Proceedings*. IEEE Computer Society, 2000, pp. 89–96.

[BIS+20] Alan Burns, Leandro Soares Indrusiak, N. Smirnov, and J. Harrison. "A Novel Flow Control Mechanism to Avoid Multi-Point Progressive Blocking in Hard Real-Time Priority-Preemptive NoCs." In: *RTAS*. IEEE, 2020, pp. 137–147.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Real-Time Systems Series. Springer, 2011. ISBN: 978-1-4614-0675-4.

[But05] Giorgio C. Buttazzo. "Rate Monotonic vs. EDF: Judgment Day." In: *Real Time Syst.* 29.1 (2005), pp. 5–26.

[BB06b] Giorgio C. Buttazzo and Enrico Bini. "Optimal Dimensioning of a Constant Bandwidth Server." In: *RTSS*. IEEE Computer Society, 2006, pp. 169–177.

[BBW10] Giorgio C. Buttazzo, Enrico Bini, and Yifan Wu. "Partitioning Parallel Applications on Multiprocessor Reservations." In: *ECRTS*. IEEE Computer Society, 2010, pp. 24–33.

[BBW11] Giorgio C. Buttazzo, Enrico Bini, and Yifan Wu. "Partitioning Real-Time Applications Over Multicore Reservations." In: *IEEE Trans. Ind. Informatics* 7.2 (2011), pp. 302–315.

[BG06] Giorgio C. Buttazzo and Paolo Gai. "Efficient EDF Implementation for Small Embedded Systems." In: 2006.

[BLA98] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. "Elastic Task Model for Adaptive Rate Control." In: *RTSS*. IEEE Computer Society, 1998, pp. 286–295.

[BLA+05] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Plenum Publishing Co., 2005. ISBN: 0387237011.

[CA08] John M. Calandrino and James H. Anderson. "Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study." In: *ECRTS*. IEEE Computer Society, 2008, pp. 299–308.

[CAB+17] Daniel Casini, Luca Abeni, Alessandro Biondi, Tommaso Cucinotta, and Giorgio C. Buttazzo. "Constant bandwidth servers with constrained deadlines." In: *RTNS*. ACM, 2017, pp. 68–77.

[CBN+20] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. "A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling." In: *RTAS*. IEEE, 2020, pp. 239–252.

[CBN+18a] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. "Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures." In: *RTSS*. IEEE Computer Society, 2018, pp. 312–324.

[CBN+18b] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio C. Buttazzo. "Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions." In: *RTSS*. IEEE Computer Society, 2018, pp. 421–433.

[CB94] Suresh Chalasani and Rajendra V. Boppana. "Adaptive fault-tolerant wormhole routing algorithms with low virtual channel requirements." In: *ISPAN*. IEEE Computer Society, 1994, pp. 214–221.

[Che] Jian-Jia Chen. "Fundamentals of Real-Time Systems." In: URL: https://daes.cs.tu-dortmund.de/storages/daes-cs/r/publications/chen-rtsbook-V0-2023.pdf.

[Che16] Jian-Jia Chen. "Partitioned Multiprocessor Fixed-Priority Scheduling of Sporadic Real-Time Tasks." In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2016, pp. 251–261.

[CA14] Jian-Jia Chen and Kunal Agrawal. *Capacity Augmentation Bounds for Parallel DAG Tasks under G-EDF and G-RM*. Tech. rep. 845. Faculty for Informatik at TU Dortmund, 2014.

[CBH+17a] Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Robert I. Davis. "On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling." In: *ECRTS*. Vol. 76. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 9:1–9:25.

[CBH+17b] Jian-Jia Chen, Georg von der Brüggen, Wen-Hung Huang, and Cong Liu. "State of the art for scheduling and analyzing self-suspending sporadic real-time tasks." In: *RTCSA*. IEEE Computer Society, 2017, pp. 1–10.

[CBU18] Jian-Jia Chen, Georg von der Brüggen, and Niklas Ueter. "Push Forward: Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems." In: *ECRTS*. Vol. 106. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 8:1–8:24.

[CC11] Jian-Jia Chen and Samarjit Chakraborty. "Resource Augmentation Bounds for Approximate Demand Bound Functions." In: *RTSS*. IEEE Computer Society, 2011, pp. 272–281.

[CC13] Jian-Jia Chen and Samarjit Chakraborty. "Resource augmentation for uniprocessor and multiprocessor partitioned scheduling of sporadic real-time tasks." In: *Real-Time Systems* 49.4 (2013), pp. 475–516.

[CHH+19] Jian-Jia Chen, Tobias Hahn, Ruben Hoeksma, Nicole Megow, and Georg von der Brüggen. "Scheduling Self-Suspending Tasks: New and Old Results." In: *ECRTS*. Vol. 133. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 16:1–16:23.

[CHL15a] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. "k2Q: A Quadratic-Form Response Time and Schedulability Analysis Framework for Utilization-Based Analysis." In: *CoRR* (2015).

[CHL15b] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. "k2U: A General Framework from k-Point Effective Schedulability Analysis to Utilization-Based Tests." In: *Real-Time Systems Symposium (RTSS)*. 2015, pp. 107–118.

[CNH16] Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. "A Unifying Response Time Analysis Framework for Dynamic Self-Suspending Tasks." In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2016, pp. 61–71.

[CNH+19] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. "Many suspensions, many problems: a review of self-suspending tasks in real-time systems." In: *Real Time Syst.* 55.1 (2019), pp. 144–207.

[CBC+16] Kuan-Hsun Chen, Björn Bönninghoff, Jian-Jia Chen, and Peter Marwedel. "Compensate or ignore? meeting control robustness requirements through adaptive soft-error handling." In: *LCTES*. ACM, 2016, pp. 82–91.

[CBC18a] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. "Analysis of Deadline Miss Rates for Uniprocessor Fixed-Priority Scheduling." In: *RTCSA*. IEEE Computer Society, 2018, pp. 168–178.

[CBC18b] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. "Reliability Optimization on Multi-Core Systems with Multi-Tasking and Redundant Multi-Threading." In: *IEEE Trans. Computers* 67.4 (2018), pp. 484–497.

[CGJ+22] Kuan-Hsun Chen, Mario Günzel, Boguslaw Jablkowski, Markus Buschhoff, and Jian-Jia Chen. "Unikernel-Based Real-Time Virtualization Under Deferrable Servers: Analysis and Realization." In: *ECRTS*. Vol. 231. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 6:1–6:22.

[CLJ+19] Peng Chen, Weichen Liu, Xu Jiang, Qingqiang He, and Nan Guan. "Timing-Anomaly Free Dynamic Scheduling of Conditional DAG Tasks on Multi-Core Systems." In: *ACM Trans. Embed. Comput. Syst.* 18.5s (2019), 91:1–91:19.

[CKZ19] Hyunjong Choi, Hyoseung Kim, and Qi Zhu. "Job-Class-Level Fixed Priority Scheduling of Weakly-Hard Real-Time Systems." In: *RTAS*. IEEE, 2019, pp. 241–253.

[CLP+13] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. "Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms." In: *ECRTS*. IEEE Computer Society, 2013, pp. 25–34.

[CB02] Antoine Colin and Guillem Bernat. "Scope-Tree: A Program Representation for Symbolic Worst-Case Execution Time Analysis." In: *ECRTS*. IEEE Computer Society, 2002, p. 50.

[CP00] Antoine Colin and Isabelle Puaut. "Worst Case Execution Time Analysis for a Processor with Branch Prediction." In: *Real Time Syst.* 18.2/3 (2000), pp. 249–274.

[Con63] Melvin E. Conway. "A multiprocessor system design." In: *AFIPS Fall Joint Computing Conference*. ACM, 1963, pp. 139–146.

[CSH+12] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs." In: *ECRTS*. IEEE Computer Society, 2012, pp. 91–101.

[DA93] William J. Dally and Hiromichi Aoki. "Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels." In: *IEEE Trans. Parallel Distributed Syst.* 4.4 (1993), pp. 466–475.

[DT01] William J. Dally and Brian Towles. "Route Packets, Not Wires: On-Chip Interconnection Networks." In: *DAC*. ACM, 2001, pp. 684–689.

[DT04] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2004. ISBN: 9780080497808.

[DZN+07] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. "Period Optimization for Hard Real-time Distributed Automotive Systems." In: *Proceedings of the 44th Design Automation Conference, DAC*. 2007, pp. 278–283. DOI: 10.1145/1278480.1278553.

[Dav16] Robert I. Davis. "On the Evaluation of Schedulability Tests for Real-Time Scheduling Algorithms." In: *WATERS*. 2016.

[DB11] Robert I. Davis and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems." In: *ACM Comput. Surv.* 43.4 (2011), p. 35. DOI: 10.1145/1978802.1978814. URL: http://doi.acm.org/10.1145/1978802.1978814.

[DC19] Robert I. Davis and Liliana Cucu-Grosjean. "A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems." In: *Leibniz Trans. Embed. Syst.* 6.1 (2019), 03:1–03:60.

[DFP+14] Robert I. Davis, Timo Feld, Victor Pollex, and Frank Slomka. "Schedulability tests for tasks with Variable Rate-dependent Behaviour under fixed priority scheduling." In: *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2014, pp. 51–62. DOI: 10.1109/RTAS.2014.6925990.

[DKP+13] Robert I. Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. "Schedulability analysis for Controller Area Network (CAN) with FIFO queues priority queues and gateways." In: *Real Time Syst.* 49.1 (2013), pp. 73–116.

[DN12] Robert I. Davis and Nicolas Navet. "Traffic shaping to reduce jitter in controller area network (CAN)." In: *SIGBED Rev.* 9.4 (2012), pp. 37–40.

[Dem79] Robert W. Deming. “Independence numbers of graphs-an extension of the Koenig-Egervary theorem.” In: *Discrete Mathematics* 27.1 (1979), pp. 23–33. ISSN: 0012-365X.

[DRW98] Robert P. Dick, David L. Rhodes, and Wayne H. Wolf. “TGFF: task graphs for free.” In: *CODES*. IEEE Computer Society, 1998, pp. 97–101.

[Dil90] R. P. Dilworth. “A Decomposition Theorem for Partially Ordered Sets.” In: *The Dilworth Theorems: Selected Papers of Robert P. Dilworth*. Ed. by Kenneth P. Bogart, Ralph Freese, and Joseph P. S. Kung. Boston, MA: Birkhäuser Boston, 1990, pp. 7–12. ISBN: 978-1-4899-3558-8.

[DPS14] Giorgos Dimitrakopoulos, Anastasios Psarras, and Ioannis Seitanidis. *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461443008.

[DGA20] Son Dinh, Christopher D. Gill, and Kunal Agrawal. “Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks.” In: *RTCSA*. IEEE, 2020, pp. 1–10.

[DL22] Zheng Dong and Cong Liu. “A Utilization-based Test for Non-preemptive Gang Tasks on Multiprocessors.” In: *RTSS*. IEEE, 2022, pp. 105–117.

[DL17] Zheng Dong and Cong Liu. “Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems.” In: *IEEE Real-Time Systems Symposium, RTSS*. 2017, pp. 128–138. DOI: 10.1109/RTSS.2017.00019. URL: http://doi.ieeecomputersociety.org/10.1109/RTSS.2017.00019.

[DL18] Zheng Dong and Cong Liu. “Work-in-Progress: New Analysis Techniques for Supporting Hard Real-Time Sporadic DAG Task Systems on Multiprocessors.” In: *RTSS*. IEEE Computer Society, 2018, pp. 151–154.

[Dua94] José Duato. “A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks.” In: *ICPP (1)*. CRC Press, 1994, pp. 142–149.

[Dua93] José Duato. “A New Theory of Deadlock-free Adaptive Multicast Routing in Wormhole Networks.” In: *SPDP*. IEEE Computer Society, 1993, pp. 64–71.

[DAB+11] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures.” In: *Parallel Process. Lett.* 21.2 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151.

[DBC+19] Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. “End-to-End Timing Analysis of Sporadic Cause-Effect Chains in Distributed Systems.” In: *ACM Trans. Embed. Comput. Syst.* 18.5s (2019), 58:1–58:24.

[EB01] Stewart Edgar and Alan Burns. "Statistical Analysis of WCET for Scheduling." In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, 2-6 December 2001*. IEEE Computer Society, 2001, pp. 215–224.

[EY22] Pontus Ekberg and Wang Yi. "Complexity of Uniprocessor Scheduling Analysis." In: *Handbook of Real-Time Computing*. Springer, 2022, pp. 489–506.

[ESD10] Paul Emberson, Roger Stafford, and Robert I. Davis. "Techniques For The Synthesis Of Multiprocessor Tasksets." In: *WATERS*. 2010.

[FRN+09] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics." In: *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*. 2009.

[FR92] Dror G. Feitelson and Larry Rudolph. "Gang Scheduling Performance Benefits for Fine-Grain Synchronization." In: *J. Parallel Distributed Comput.* 16.4 (1992), pp. 306–318.

[FHW04] Christian Ferdinand, Reinhold Heckmann, and Reinhard Wilhelm. "Analyzing the Worst-Case Execution Time by Abstract Interpretation of Executable Code." In: *ASWSD*. Vol. 4147. Lecture Notes in Computer Science. Springer, 2004, pp. 1–14.

[FNN17] José Carlos Fonseca, Geoffrey Nelissen, and Vincent Nélis. "Improved response time analysis of sporadic DAG tasks for global FP scheduling." In: *RTNS*. ACM, 2017, pp. 28–37.

[GGB13] Yue Gao, Sandeep K. Gupta, and Melvin A. Breuer. "Using explicit output comparisons for fault tolerant scheduling (FTS) on modern high-performance processors." In: *DATE*. EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 927–932.

[GDD19] Sumana Ghosh, Soumyajit Dey, and Pallab Dasgupta. "Synthesizing Performance-Aware (m, k)-Firm Control Execution Patterns Under Dropped Samples." In: *VLSID*. IEEE, 2019, pp. 1–6.

[GM18] Frederic Giroudot and Ahlem Mifdaoui. "Buffer-Aware Worst-Case Timing Analysis of Wormhole NoCs Using Network Calculus." In: *Real-Time and Embedded Technology and Applications Symposium, (RTAS)*. 2018, pp. 37–48. DOI: 10.1109/RTAS.2018.00010. URL: https://doi.org/10.1109/RTAS.2018.00010.

[GN92] Christopher J. Glass and Lionel M. Ni. "The Turn Model for Adaptive Routing." In: *ISCA*. ACM, 1992, pp. 278–287.

[GN20] Yilian Ribot González and Geoffrey Nelissen. "HopliteRT*: Real-Time NoC for FPGA." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.11 (2020), pp. 3650–3661.

[GNT22] Yilian Ribot González, Geoffrey Nelissen, and Eduardo Tovar. "IP-DeN: Real-Time deflection-based NoC with in-order flits delivery." In: *RTCSA*. IEEE, 2022, pp. 160–169.

[GNT21] Yilian Ribot González, Geoffrey Nelissen, and Eduardo Tovar. "nDimNoC: Real-Time D-dimensional NoC." In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 5:1–5:22.

[GB10a] Joël Goossens and Vandy Berten. "Gang FTP scheduling of periodic and parallel rigid real-time tasks." In: *CoRR* abs/1006.2617 (2010).

[GR16] Joël Goossens and Pascal Richard. "Optimal Scheduling of Periodic Gang Tasks." In: *Leibniz Trans. Embed. Syst.* 3.1 (2016), 04:1–04:18.

[GDR05] K. Goossens, J. Dielissen, and A. Radulescu. "AEthereal network on chip: concepts, architectures, and implementations." In: *IEEE Design Test of Computers* 22.5 (2005), pp. 414–421. ISSN: 0740-7475. DOI: 10.1109/MDT.2005.99.

[GBD20] David Griffin, Iain Bate, and Robert I. Davis. "Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests." In: *RTSS*. IEEE, 2020, pp. 76–88.

[GB10b] David Griffin and Alan Burns. "Realism in Statistical Analysis of Worst Case Execution Times." In: *WCET*. Vol. 15. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 44–53.

[GS20] Geoffrey Grimmett and David Stirzaker. *Probability and random processes*. Oxford university press, 2020.

[GSY+09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. "New Response Time Bounds for Fixed Priority Multiprocessor Scheduling." In: *IEEE Real-Time Systems Symposium*. 2009, pp. 387–397.

[GYG+08] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. "New Schedulability Test Conditions for Non-preemptive Scheduling on Multiprocessor Platforms." In: *RTSS*. IEEE Computer Society, 2008, pp. 137–146.

[GBC+22] Mario Günzel, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. "EDF-Like Scheduling for Self-Suspending Real-Time Tasks." In: *RTSS*. IEEE, 2022, pp. 172–184.

[GCU+21] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. "Timing Analysis of Asynchronized Distributed Cause-Effect Chains." In: *RTAS*. IEEE, 2021, pp. 40–52.

[GUC21] Mario Günzel, Niklas Ueter, and Jian-Jia Chen. "Suspension-Aware Fixed-Priority Schedulability Test with Arbitrary Deadlines and Arrival Curves." In: *RTSS*. IEEE, 2021, pp. 418–430.

[GUC+21] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, Georg von der Brüggen, Junjie Shi, and Jian-Jia Chen. "End-To-End Processing Chain Analysis." In: *RTSS Industrial Challenge*. 2021.

[GYY+20] Zhishan Guo, Kecheng Yang, Fan Yao, and Amro Awad. "Inter-task cache interference aware partitioned real-time scheduling." In: *SAC*. ACM, 2020, pp. 218–226.

[HR95] M. Hamdaoui and P. Ramanathan. "A dynamic priority assignment technique for streams with (m, k)-firm deadlines." In: *IEEE Transactions on Computers* 44.12 (1995), pp. 1443–1451. DOI: 10.1109/12.477249.

[HQE20] Zain Alabedin Haj Hammadeh, Sophie Quinton, and Rolf Ernst. "Weakly-hard Real-time Guarantees for Earliest Deadline First Scheduling of Independent Tasks." In: *ACM Trans. Embed. Comput. Syst.* 18.6 (2020), 121:1–121:25.

[HHM09] Jeffery P. Hansen, Scott A. Hissam, and Gabriel A. Moreno. "Statistical-Based WCET Estimation and Validation." In: *WCET*. Vol. 10. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[HAZ17] Mohammad A. Haque, Hakan Aydin, and Dakai Zhu. "On Reliability Management of Energy-Aware Real-Time Systems Through Task Replication." In: *IEEE Trans. Parallel Distributed Syst.* 28.3 (2017), pp. 813–825.

[HFB+18] Tim Harde, Matthias Freier, Georg von der Brüggen, and Jian-Jia Chen. "Configurations and Optimizations of TDMA Schedules for Periodic Packet Communication on Networks on Chip." In: *International Conference on Real-Time Networks and Systems, RTNS*. 2018, pp. 202–212. DOI: 10.1145/3273905.3273928. URL: https://doi.org/10.1145/3273905.3273928.

[HO97] S. L. Hary and F. Ozguner. "Feasibility test for Real-Time Communication using Wormhole Routing." In: *IEEE Proceedings - Computers and Digital Techniques* 144.5 (1997), pp. 273–278. DOI: 10.1049/ip-cdt:19971369.

[HJG+19] Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. "Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores." In: *IEEE Trans. Parallel Distributed Syst.* 30.10 (2019), pp. 2283–2295.

[HLG21] Qingqiang He, Mingsong Lv, and Nan Guan. "Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment." In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 8:1–8:21.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. "Bounding Pipeline and Instruction Cache Performance." In: *IEEE Trans. Computers* 48.1 (1999), pp. 53–70.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[HSJ08] Erik Henriksson, Henrik Sandberg, and Karl Henrik Johansson. "Predictive compensation for communication outages in networked control systems." In: *CDC*. IEEE, 2008, pp. 2063–2068.

[HTA19] Clara Hobbs, Zelin Tong, and James H. Anderson. "Optimal soft real-time semi-partitioned scheduling made simple (and dynamic)." In: *RTNS*. ACM, 2019, pp. 112–122.

[HVV94] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman. "Complexity of scheduling multiprocessor tasks with prespecified processor allocations." In: *Discrete Appl. Math.* 55.3 (1994), pp. 259–272.

[Hop71] John Hopcroft. "An n log n algorithm for minimizing states in a finite automaton." In: *Theory of machines and computations*. Elsevier, 1971, pp. 189–196.

[HXW+20] Chao Huang, Shichao Xu, Zhilu Wang, Shuyue Lan, Wenchao Li, and Qi Zhu. "Opportunistic Intermittent Control with Safety Guarantees for Autonomous Systems." In: *DAC*. IEEE, 2020, pp. 1–6.

[HCR16] Wen-Hung Huang, Jian-Jia Chen, and Jan Reineke. "MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources." In: *Design Automation Conference, DAC*. 2016, 158:1–158:6. DOI: 10.1145/2897937.2898046. URL: http://doi.acm.org/10.1145/2897937.2898046.

[IBN16] Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolic. "Analysis of buffering effects on hard real-time priority-preemptive wormhole networks." In: *CoRR* abs/1606.02942 (2016). arXiv: 1606.02942. URL: http://arxiv.org/abs/1606.02942.

[IBN18] Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolic. "Buffer-aware bounds to multi-point progressive blocking in priority-preemptive NoCs." In: *Design, Automation & Test in Europe Conference & Exhibition, DATE*. 2018, pp. 219–224. DOI: 10.23919/DATE.2018.8342006. URL: https://doi.org/10.23919/DATE.2018.8342006.

[JNS+12] Aamer Jaleel, Hashem Hashemi Najaf-abadi, Samantika Subramaniam, Simon C. Steely Jr., and Joel S. Emer. "CRUISE: cache replacement and utility-aware scheduling." In: *ASPLOS*. ACM, 2012, pp. 249–260.

[Jet97] Morris A. Jette. "Performance Characteristics of Gang Scheduling in Multiprogrammed Environments." In: *SC*. ACM, 1997, p. 54.

[JGL+21] Xu Jiang, Nan Guan, Haochun Liang, Yue Tang, Lei Qiao, and Yi Wang. "Virtually-Federated Scheduling of Parallel Real-Time Tasks." In: *RTSS*. IEEE, 2021, pp. 482–494.

[JGL+20] Xu Jiang, Nan Guan, Xiang Long, and Han Wan. "Decomposition-Based Real-Time Scheduling of Parallel Tasks on Multicores Platforms." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.10 (2020), pp. 2319–2332.

[JGL+17] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. "Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors." In: *RTSS*. IEEE Computer Society, 2017, pp. 80–91.

[JP86] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System." In: *The Computer Journal* 29.5 (May 1986), pp. 390–395.

[JC07] Edward G. Coffman Jr. and János Csirik. "A Classification Scheme for Bin Packing Theory." In: *Acta Cybern.* 18.1 (2007), pp. 47–60.

[KI21] Muhammad Kaleem and Ismail Isnin. "A Survey on Network on Chip Routing Algorithms Criteria." In: Jan. 2021, pp. 455–466.

[KSS+16] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sorensen, Christoph Thomas Muller, Kees Goossens, and Jens Sparsø. "Argo: A Real-Time Network-on-Chip Architecture With an Efficient GALS Implementation." In: 24.2 (2016), pp. 479–492. DOI: 10.1109/TVLSI.2015.2405614. URL: https://doi.org/10.1109/TVLSI.2015.2405614.

[KP16] H. Kashif and H. Patel. "Buffer Space Allocation for Real-Time Priority-Aware Networks." In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, pp. 1–12. DOI: 10.1109/RTAS.2016.7461324.

[KGP14] Hany Kashif, Sina Gholamian, and Hiren Patel. "SLA: A Stage-level Latency Analysis for Real-time Communication in a Pipelined Resource Model." In: *IEEE Transactions on Computers* PP (2014). ISSN: 0018-9340. DOI: 10.1109/TC.2014.2315617.

[KI09] Shinpei Kato and Yutaka Ishikawa. "Gang EDF Scheduling of Parallel Task Systems." In: *IEEE Real-Time Systems Symposium, RTSS*. 2009, pp. 459–468. DOI: 10.1109/RTSS.2009.42. URL: https://doi.org/10.1109/RTSS.2009.42.

[KY08] Shinpei Kato and Nobuyuki Yamasaki. "Scheduling Aperiodic Tasks Using Total Bandwidth Server on Multiprocessors." In: *EUC (1)*. IEEE Computer Society, 2008, pp. 82–89.

[KS03] N.K. Kavaldjiev and Gerardus Johannes Maria Smit. "A Survey of Efficient On-Chip Communications for SoC." Undefined. In: *4th PROGRESS Symposium on Embedded Systems*. STW Technology Foundation, Oct. 2003, pp. 129–140. ISBN: 90-73461-37-5.

[KKH+98] Byungjae Kim, Jong Kim, Sungje Hong, and Sunggu Lee. "A real-time communication method for wormhole switching networks." In: *International Conference on Parallel Processing*. 1998, pp. 527–534. DOI: 10.1109/ICPP.1998.708526.

[KBD07] John Kim, James D. Balfour, and William J. Dally. "Flattened Butterfly Topology for On-Chip Networks." In: *MICRO*. IEEE Computer Society, 2007, pp. 172–182.

[KAN+13] Junsung Kim, Björn Andersson, Dionisio de Niz, and Ragunathan Rajkumar. "Segment-Fixed Priority Scheduling for Self-Suspending Real-Time Tasks." In: *RTSS*. IEEE Computer Society, 2013, pp. 246–257.

[KS90] David Blair Kirk and Jay K. Strosnider. "SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000." In: *RTSS*. IEEE Computer Society, 1990, pp. 322–330.

[KBS18] Tomasz Kloda, Antoine Bertout, and Yves Sorel. "Latency analysis for data chains of real-time periodic tasks." In: *ETFA*. IEEE, 2018, pp. 360–367.

[KS95] G. Koren and D. Shasha. "Skip-Over: algorithms and complexity for overloaded systems that allow skips." In: *Proceedings 16th IEEE Real-Time Systems Symposium*. 1995, pp. 110–117. DOI: 10.1109/REAL.1995.495201.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. "Real world automotive benchmark for free." In: *6th International Workshop WATERS*. 2015.

[Kub87] M. Kubale. "The complexity of scheduling independent two-processor tasks on dedicated processors." In: *Inf. Process. Lett.* 24.3 (1987), pp. 141–147.

[KGC+12] Pratyush Kumar, Dip Goswami, Samarjit Chakraborty, Anuradha Annaswamy, Kai Lampka, and Lothar Thiele. "A hybrid approach to cyber-physical systems verification." In: *DAC*. ACM, 2012, pp. 688–696.

[LKR10] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. "Scheduling Parallel Real-Time Tasks on Multi-core Processors." In: *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*. RTSS '10. 2010, pp. 259–268. ISBN: 978-0-7695-4298-0.

[Law75] Duncan H. Lawrie. "Access and Alignment of Data in an Array Processor." In: *IEEE Trans. Computers* 24.12 (1975), pp. 1145–1155.

[Lee19a] Edward A. Lee. "Freedom From Choice and the Power of Models: in Honor of Alberto Sangiovanni-Vincentelli." In: *ISPD*. ACM, 2019, p. 126.

[Lee19b] Edward A. Lee. "Modeling in engineering and science." In: *Commun. ACM* 62.1 (2019), pp. 35–36.

[Lee18] Edward A. Lee. "Models of Timed Systems." In: *FORMATS*. Vol. 11022. Lecture Notes in Computer Science. Springer, 2018, pp. 17–33.

[LS18] Edward A. Lee and Marjan Sirjani. "What Good are Models?" In: *FACS*. Vol. 11222. Lecture Notes in Computer Science. Springer, 2018, pp. 3–31.

[Lee17] Edward Ashford Lee. "Plato and the Nerd." In: *Plato and the Nerd: The Creative Partnership of Humans and Technology*. 2017, pp. i–xvi.

[LNP+13] Junghee Lee, Chrysostomos Nicopoulos, Sung Joo Park, Madhavan Swaminathan, and Jongman Kim. "Do we need wide flits in Networks-on-Chip?" In: *ISVLSI*. IEEE Computer Socity, 2013, pp. 2–7.

[LGL22] Seongtae Lee, Nan Guan, and Jinkyu Lee. "Design and Timing Guarantee for Non-Preemptive Gang Scheduling." In: *RTSS*. IEEE, 2022, pp. 132–144.

[Leh96] J. P. Lehoczky. "Real-time queueing theory." In: *17th IEEE Real-Time Systems Symposium*. 1996, pp. 186–195.

[Leh90] John P. Lehoczky. "Fixed priority scheduling of periodic task sets with arbitrary deadlines." In: *proceedings Real-Time Systems Symposium (RTSS)*. 1990, pp. 201–209. DOI: 10.1109/REAL.1990.128748.

[LSD89] John P. Lehoczky, Lui Sha, and Y. Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." In: *IEEE Real-Time Systems Symposium'89*. 1989, pp. 166–171.

[LCA+14] J. Li, J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks." In: *Euromicro Conference on Real-Time Systems*. 2014, pp. 85–96.

[LAG+14] Jing Li, Kunal Agrawal, Christopher D. Gill, and Chenyang Lu. "Federated scheduling for stochastic parallel real-time tasks." In: *RTCSA*. IEEE Computer Society, 2014, pp. 1–10.

[LAL+13] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. "Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks." In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2013, pp. 3–13. DOI: 10.1109/ECRTS.2013.12.

[LM95] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration." In: *DAC*. ACM Press, 1995, pp. 456–461.

[LWJ+20] Hengyi Liang, Zhilu Wang, Ruochen Jiao, and Qi Zhu. "Leveraging Weakly-hard Constraints for Improving System Fault Tolerance with Functional and Timing Guarantees." In: *ICCAD*. IEEE, 2020, 101:1–101:9.

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. "OS-Controlled Cache Predictability for Real-Time Systems." In: *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, 1997, pp. 213–224.

[LB17] George Lima and Iain Bate. "Valid Application of EVT in Timing Analysis by Randomising Execution Time Measurements." In: *RTAS*. IEEE Computer Society, 2017, pp. 187–198.

[LDB16] George Lima, Dario Dias, and Edna Barros. "Extreme Value Theory for Estimating Task Execution Time Bounds: A Careful Look." In: *ECRTS*. IEEE Computer Society, 2016, pp. 200–211.

[LL73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (1973), pp. 46–61.

[LDS+07] Yongpan Liu, Robert P. Dick, Li Shang, and Huazhong Yang. "Accurate temperature-dependent integrated circuit leakage power estimation is easy." In: *DATE*. EDA Consortium, San Jose, CA, USA, 2007, pp. 1526–1531.

[LJS05] Zhonghai Lu, A. Jantsch, and I. Sander. "Feasibility analysis of messages for on-chip networks using wormhole routing." In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. Vol. 2. 2005, pp. 960–964. DOI: 10.1109/ASPDAC.2005.1466499.

[LGR+16] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. “A Survey on Static Cache Analysis for Real-Time Systems.” In: *Leibniz Trans. Embed. Syst.* 3.1 (2016), 05:1–05:48.

[MHM+20] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. “Control-System Stability Under Consecutive Deadline Misses Constraints.” In: *ECRTS*. Vol. 165. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 21:1–21:24.

[MBN+14] Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luís Miguel Pinho. “Response-Time Analysis of Synchronous Parallel Tasks in Multiprocessor Systems.” In: *RTNS*. ACM, 2014, p. 3.

[MMS+20] Alberto Marchetti-Spaccamela, Nicole Megow, Jens Schlöter, Martin Skutella, and Leen Stougie. “On the Complexity of Conditional DAG Scheduling in Multiprocessor Systems.” In: *IPDPS*. IEEE, 2020, pp. 1061–1070.

[MNP22] Filip Markovic, Thomas Nolte, and Alessandro Vittorio Papadopoulos. “Analytical Approximations in Probabilistic Analysis of Real-Time Systems.” In: *RTSS*. IEEE, 2022, pp. 158–171.

[MPN21] Filip Markovic, Alessandro Vittorio Papadopoulos, and Thomas Nolte. “On the Convolution Efficiency for Probabilistic Analysis of Real-Time Systems.” In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 16:1–16:22.

[MBB+15] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. “Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems.” In: *ECRTS*. IEEE Computer Society, 2015, pp. 211–221.

[MSB+17] Alessandra Melani, Maria A. Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones, and Giorgio C. Buttazzo. “A static scheduling approach to enable safety-critical OpenMP applications.” In: *ASP-DAC*. IEEE, 2017, pp. 659–665.

[MNT+04] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. “Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip.” In: *Design, Automation and Test in Europe Conference (DATE)*. 2004, pp. 890–895. DOI: 10.1109/DATE.2004.1269001. URL: https://doi.org/10.1109/DATE.2004.1269001.

[MR99] Mark Moir and Srikanth Ramamurthy. “Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources.” In: *RTSS*. IEEE Computer Society, 1999, pp. 294–303.

[Mok83] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. Tech. rep. Cambridge, MA, USA, 1983.

[MC97] Aloysius K. Mok and Deji Chen. “A Multiframe Model for Real-Time Tasks.” In: *IEEE Trans. Software Eng.* (1997), pp. 635–645. DOI: 10.1109/32.637146.

[MT19] David Monniaux and Valentin Touzeau. "On the Complexity of Cache Analysis for Different Replacement Policies." In: *J. ACM* 66.6 (2019), 41:1–41:22.

[MCM+04] Fernando Gehm Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. "HERMES: an infrastructure for low area overhead packet-switching networks on chip." In: *Integr.* 38.1 (2004), pp. 69–93.

[Mue95] Frank Mueller. "Compiler Support for Software-Based Cache Partitioning." In: *Workshop on Languages, Compilers, & Tools for Real-Time Systems*. ACM, 1995, pp. 125–133.

[Mut94] M. W. Mutka. "Using rate monotonic scheduling technology for real-time communications in a wormhole network." In: *Second Workshop on Parallel and Distributed Real-Time Systems*. 1994, pp. 194–199. DOI: 10.1109/WPDRTS.1994.365629.

[NF16] Mitra Nasri and Gerhard Fohler. "Non-work-conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions." In: *ECRTS*. IEEE Computer Society, 2016, pp. 165–175.

[NNB19] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. "Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling." In: *ECRTS*. Vol. 133. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 21:1–21:23.

[NIN22] Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri. "Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks." In: *ECRTS*. Vol. 231. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 12:1–12:22.

[NWF78] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. "An analysis of approximations for maximizing submodular set functions—I." In: *Mathematical Programming* 14 (1978), pp. 265–294.

[NHE19] Borislav Nikolic, Robin Hofmann, and Rolf Ernst. "Slot-Based Transmission Protocol for Real-Time NoCs - SBT-NoC." In: *31st Euromicro Conference on Real-Time Systems, ECRTS*. 2019, 26:1–26:22. DOI: 10.4230/LIPIcs.ECRTS.2019.26. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2019.26.

[NIP16] Borislav Nikolic, Leandro Soares Indrusiak, and Stefan M. Petters. "A Tighter Real-Time Communication Analysis for Wormhole-Switched Priority-Preemptive NoCs." In: *CoRR* abs/1605.07888 (2016). arXiv: 1605.07888. URL: http://arxiv.org/abs/1605.07888.

[NTI+19] Borislav Nikolić, Sebastian Tobuschat, Leandro Soares Indrusiak, Rolf Ernst, and Alan Burns. "Real-time analysis of priority-preemptive NoCs with arbitrary buffer sizes and router delays." In: *Real-Time Systems* 55.1 (2019), pp. 63–105. ISSN: 1573-1383. DOI: 10.1007/s11241-018-9312-0. URL: https://doi.org/10.1007/s11241-018-9312-0.

[NQ06] Linwei Niu and Gang Quan. "Energy minimization for real-time systems with (m, k)-guarantee." In: *IEEE Trans. Very Large Scale Integr. Syst.* 14.7 (2006), pp. 717–729.

[NZ20] Linwei Niu and Dakai Zhu. "Reliable and Energy-Aware Fixed-Priority (m, k)-Deadlines Enforcement with Standby-Sparing." In: *DATE*. IEEE, 2020, pp. 424–429.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Control-flow checking by software signatures." In: *IEEE Trans. Reliab.* 51.1 (2002), pp. 111–122.

[PFA+16] Luigi Palopoli, Daniele Fontanelli, Luca Abeni, and Bernardo Villalba Frias. "An Analytical Solution for Probabilistic Guarantees of Reservation Based Soft Real-Time Systems." In: *IEEE Trans. Parallel Distributed Syst.* 27.3 (2016), pp. 640–653.

[PQC+09] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. "An Analyzable Memory Controller for Hard Real-Time CMPs." In: *IEEE Embed. Syst. Lett.* 1.4 (2009), pp. 86–90.

[Pat16] Risat Mahmud Pathan. "Design of an efficient ready queue for earliest-deadline-first (EDF) scheduler." In: *DATE*. IEEE, 2016, pp. 293–296.

[PK08] Christian Paukovits and Hermann Kopetz. "Concepts of Switching in the Time-Triggered Network-on-Chip." In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (2008), pp. 120–129.

[PMM+19] Paolo Pazzaglia, Claudio Mandrioli, Martina Maggio, and Anton Cervin. "DMAC: Deadline-Miss-Aware Control." In: *ECRTS*. Vol. 133. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 1:1–1:24.

[PST+02] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. "Optimal Time-Critical Scheduling via Resource Augmentation." In: *Algorithmica* 32.2 (2002), pp. 163–200.

[PNY+15] Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. "P-SOCRATES: A parallel software framework for time-critical many-core systems." In: *Microprocess. Microsystems* 39.8 (2015), pp. 1190–1203.

[Pla16] Matthias Plappert. *keras-rl*. https://github.com/keras-rl/keras-rl. 2016.

[Pra86] Dhiraj K. Pradhan, ed. *Fault-Tolerant Computing: Theory and Techniques; Vol. 1*. USA: Prentice-Hall, Inc., 1986. ISBN: 013308230X.

[PTV+07] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[QLD09] Yue Qian, Zhonghai Lu, and Wenhua Dou. "Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip." In: *International Symposium on Networks-on-Chip*. 2009, pp. 44–53.

[QH00] Gang Quan and Xiaobo Sharon Hu. "Enhanced Fixed-Priority Scheduling with (m, k)-Firm Guarantee." In: *RTSS*. IEEE Computer Society, 2000, pp. 79–88.

[QP06] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches." In: *MICRO*. IEEE Computer Society, 2006, pp. 423–432.

[RGG+12] Petar Radojkovic, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments." In: *ACM Trans. Archit. Code Optim.* 8.4 (2012), 34:1–34:25.

[RMD+10] A. C. Rajeev, Swarup Mohalik, Manoj G. Dixit, Devesh B. Chokshi, and S. Ramesh. "Schedulability and end-to-end latency in distributed ECU networks: formal modeling and precise estimation." In: *EMSOFT*. ACM, 2010, pp. 129–138.

[Raj91] R. Rajkumar. *Dealing with Suspending Periodic Tasks.* Tech. rep. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/period-enforcer.ps`. IBM T. J. Watson Research Center, 1991.

[Ram99] Parameswaran Ramanathan. "Overload Management in Real-Time Control Applications Using (m, k)-Firm Guarantee." In: *IEEE Trans. Parallel Distributed Syst.* 10.6 (1999), pp. 549–559.

[RE15] Eberle A. Rambo and Rolf Ernst. "Worst-case communication time analysis of networks-on-chip with shared virtual channels." In: *Design, Automation & Test in Europe Conference, (DATE)*. 2015, pp. 537–542. URL: `http://dl.acm.org/citation.cfm?id=2755874`.

[RNN22] Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. "Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks." In: *RTAS*. IEEE, 2022, pp. 121–132.

[RSF20] Federico Reghenzani, Luca Santinelli, and William Fornaciari. "Dealing with Uncertainty in pWCET Estimations." In: *ACM Trans. Embed. Comput. Syst.* 19.5 (2020), 33:1–33:23.

[RGB+07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. "Timing predictability of cache replacement policies." In: *Real Time Syst.* 37.2 (2007), pp. 99–122.

[RLP+11] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. "PRET DRAM controller: bank privatization for predictability and temporal isolation." In: *CODES+ISSS*. ACM, 2011, pp. 99–108.

[RS09] Jan Reineke and Rathijit Sen. “Sound and Efficient WCET Analysis in the Presence of Timing Anomalies.” In: *WCET*. Vol. 10. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[RGK17] Pascal Richard, Joël Goossens, and Shinpei Kato. “Comments on "Gang EDF Schedulability Analysis".” In: *CoRR* http://arxiv.org/abs/1705.05798 (2017). arXiv: 1705.05798. URL: http://arxiv.org/abs/1705.05798.

[RP17] Benjamin Rouxel and Isabelle Puaut. “STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation & Real-Time Scheduling.” In: *WCET*. Vol. 57. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 1:1–1:12.

[SAL+11] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. “Multi-core Real-Time Scheduling for Generalized Parallel Task Models.” In: *RTSS*. IEEE Computer Society, 2011, pp. 217–226.

[SMD+14] Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart. “On the Sustainability of the Extreme Value Theory for WCET Estimation.” In: *WCET*. Vol. 39. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 21–30.

[SYM+11] Luca Santinelli, Patrick Meumeu Yomsi, Dorin Maxim, and Liliana Cucu-Grosjean. “A component-based framework for modeling and analyzing probabilistic real-time systems.” In: *ETFA*. IEEE, 2011, pp. 1–8.

[SE09] Simon Schliecker and Rolf Ernst. “A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems.” In: *CODES+ISSS*. ACM, 2009, pp. 433–442.

[Sch] Martin Schoeberl. “A Time-Triggered Network-on-Chip.” In: *FPL 2007, International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27-29 August 2007*, pp. 377–382. DOI: 10.1109/FPL.2007.4380675. URL: https://doi.org/10.1109/FPL.2007.4380675.

[SAA+15] Martin Schoeberl et al. “T-CREST: Time-predictable multi-core architecture for embedded systems.” In: *J. Syst. Archit.* 61.9 (2015), pp. 449–471.

[SMV+15] Maria A. Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quiñones. “Timing characterization of OpenMP4 tasking model.” In: *CASES*. IEEE, 2015, pp. 157–166.

[SRQ18] Maria A. Serrano, Sara Royuela, and Eduardo Quiñones. “Towards an OpenMP Specification for Critical Real-Time Systems.” In: *IWOMP*. Vol. 11128. Lecture Notes in Computer Science. Springer, 2018, pp. 143–159.

[SUC+23] Junjie Shi, Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. "Average Task Execution Time Minimization under (m,k) Soft-Error Constraint." In: *RTAS*. IEEE, 2023, p. 12.

[SB08] Zheng Shi and Alan Burns. "Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching." In: *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip (NoCS)*. 2008, pp. 161–170. DOI: 10.1109/NoCS.2008.4492735. URL: http://dl.acm.org/citation.cfm?id=1397757.1397996.

[SKT20] Mahmoud Shirazi, Mehdi Kargahi, and Lothar Thiele. "Performance maximization of energy-variable self-powered (m, k)-firm real-time systems." In: *Real Time Syst.* 56.1 (2020), pp. 64–111.

[Sho10] Michael Short. "Improved Task Management Techniques for Enforcing EDF Scheduling on Recurring Tasks." In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2010, pp. 56–65.

[SKM00] Satinder Singh, Michael J. Kearns, and Yishay Mansour. "Nash Convergence of Gradient Dynamics in General-Sum Games." In: *UAI*. Morgan Kaufmann, 2000, pp. 541–548.

[SP19] Muhammad Refaat Soliman and Rodolfo Pellizzoni. "PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling." In: *ECRTS*. Vol. 133. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 4:1–4:23.

[SS95] Yuri N. Sotskov and Natalia V. Shakhlevich. "NP-hardness of Shop-scheduling Problems with Three Jobs." In: *Discret. Appl. Math.* 59.3 (1995), pp. 237–266.

[SA00] Friedhelm Stappert and Peter Altenbernd. "Complete worst-case execution time analysis of straight-line hard real-time programs." In: *J. Syst. Archit.* 46.4 (2000), pp. 339–355.

[SEE01] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. "Efficient longest executable path search for programs with complex flows and pipeline effects." In: *CASES*. ACM, 2001, pp. 132–140.

[SMA+12] Radu Stefan, Anca Molnos, Angelo Ambrose, and Kees Goossens. "A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-up." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '12. Dresden, Germany, 2012, pp. 1283–1288. ISBN: 978-3-9810801-8-6. URL: http://dl.acm.org/citation.cfm?id=2492708.2493025.

[SEG+11] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. "The Digraph Real-Time Task Model." In: *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*. 2011, pp. 71–80. DOI: 10.1109/RTAS.2011.15.

[SLS95] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. “The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments.” In: *IEEE Trans. Computers* 44.1 (1995), pp. 73–91.

[SLD+03] Haihua Su, Frank Liu, Anirudh Devgan, Emrah Acar, and Sani R. Nassif. “Full chip leakage estimation considering power supply and temperature variations.” In: *ISLPED*. ACM, 2003, pp. 78–83.

[SM08] Vivy Suhendra and Tulika Mitra. “Exploring locking & partitioning for predictable shared caches on multi-cores.” In: *DAC*. ACM, 2008, pp. 300–303.

[SGW+17] Jinghao Sun, Nan Guan, Yang Wang, Qingqiang He, and Wang Yi. “Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks.” In: *RTSS*. IEEE Computer Society, 2017, pp. 92–103.

[SJX+22] Jinghao Sun, Tao Jin, Yekai Xue, Liwei Zhang, Jinrong Liu, Nan Guan, and Quan Zhou. “ompTG: From OpenMP Programs to Task Graphs.” In: *J. Syst. Archit.* 126 (2022), p. 102470.

[SN18] Youcheng Sun and Marco Di Natale. “Assessing the pessimism of current multicore global fixed-priority schedulability analysis.” In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC*. ACM, 2018, pp. 575–583.

[TBE+16] Bogdan Tanasa, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. “Correlation-Aware Probabilistic Timing Analysis for the Dynamic Segment of FlexRay.” In: *ACM Trans. Embed. Comput. Syst.* 15.3 (2016), 54:1–54:31.

[TBE+13] Bogdan Tanasa, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. “Probabilistic Timing Analysis for the Dynamic Segment of FlexRay.” In: *ECRTS*. IEEE Computer Society, 2013, pp. 135–144.

[Tan09] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition*. Pearson Prentice-Hall, 2009.

[Tes95] Gerald Tesauro. “Temporal Difference Learning and TD-Gammon.” In: *Commun. ACM* 38.3 (1995), pp. 58–68.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. “Fast and Precise WCET Prediction by Separated Cache and Path Analyses.” In: *Real Time Syst.* 18.2/3 (2000), pp. 157–179.

[TE17] Sebastian Tobuschat and Rolf Ernst. “Real-time communication analysis for Networks-on-Chip with backpressure.” In: *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*. 2017, pp. 590–595. DOI: 10.23919/DATE.2017.7927055. URL: https://doi.org/10.23919/DATE.2017.7927055.

[UBC+18] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. “Reservation-Based Federated Scheduling for Parallel Real-Time Tasks.” In: *RTSS*. IEEE Computer Society, 2018, pp. 482–494.

[UCB+20] Niklas Ueter, Jian-Jia Chen, Georg von der Brüggen, Vanchinathan Venkataramani, and Tulika Mitra. "Simultaneous Progression Switching Protocols for Timing Predictable Real-Time Network-on-Chips." In: *RTCSA*. IEEE, 2020, pp. 1–10.

[UGB+21] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. "Hard Real-Time Stationary GANG-Scheduling." In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 10:1–10:19.

[UGB+23] Niklas Ueter, Mario Günzel, Georg von der Brüggen, and Jian-Jia Chen. "Parallel Path Progression DAG Scheduling." In: *IEEE Transactions on Computers* (2023), pp. 1–15. DOI: 10.1109/TC.2023.3280137.

[UGC21] Niklas Ueter, Mario Günzel, and Jian-Jia Chen. "Response-Time Analysis and Optimization for Probabilistic Conditional Parallel DAG Tasks." In: *RTSS*. IEEE, 2021, pp. 380–392.

[Uth04] Patchrawat Uthaisombut. "The Optimal Online Algorithms for Minimizing Maximum Lateness." In: *SWAT*. Vol. 3111. Lecture Notes in Computer Science. Springer, 2004, pp. 420–430.

[VQM15] Roberto Vargas, Eduardo Quiñones, and Andrea Marongiu. "OpenMP and timing predictability: a possible union?" In: *DATE*. ACM, 2015, pp. 617–620.

[VRS+16] Roberto Vargas, Sara Royuela, Maria A. Serrano, Xavier Martorell, and Eduardo Quiñones. "A lightweight OpenMP4 run-time for embedded systems." In: *ASP-DAC*. IEEE, 2016, pp. 43–49.

[VSM+22] Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, Francisco J. Cazorla, and Joan del Castillo. "Using Markov's Inequality with Power-Of-k Function for Probabilistic WCET Estimation." In: *ECRTS*. Vol. 231. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:24.

[VCM21] Nils Vreman, Anton Cervin, and Martina Maggio. "Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses." In: *ECRTS*. Vol. 196. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 15:1–15:23.

[VPM22] Nils Vreman, Richard Pates, and Martina Maggio. "WeaklyHard.jl: Scalable Analysis of Weakly-Hard Constraints." In: *RTAS*. IEEE, 2022, pp. 228–240.

[VPM+22] Nils Vreman, Paolo Pazzaglia, Victor Magron, Jie Wang, and Martina Maggio. "Stability of Linear Systems Under Extended Weakly-Hard Constraints." In: *IEEE Control. Syst. Lett.* 6 (2022), pp. 2900–2905.

[WWL00] Song Wang, Yu-Chung Wang, and Kwei-Jay Lin. "Integrating the fixed priority scheduling and the total bandwidth server for aperiodic tasks." In: *RTCSA*. IEEE Computer Society, 2000, pp. 215–222.

[WGS+17] Yang Wang, Nan Guan, Jinghao Sun, Mingsong Lv, Qingqiang He, Tianzhang He, and Wang Yi. "Benchmarking OpenMP programs for real-time scheduling." In: *RTCSA*. IEEE Computer Society, 2017, pp. 1–10.

[WHK+21] Zhilu Wang, Chao Huang, Hyoseung Kim, Wenchao Li, and Qi Zhu. "Cross-Layer Adaptation with Safety-Assured Proactive Task Job Skipping." In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021), 100:1–100:25.

[WP19] Saud Wasly and Rodolfo Pellizzoni. "Bundled Scheduling of Parallel Real-Time Tasks." In: *RTAS*. IEEE, 2019, pp. 130–142.

[Wil20] Reinhard Wilhelm. "Real time spent on real time." In: *Commun. ACM* 63.10 (2020), pp. 54–60.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem - overview of methods and survey of tools." In: *ACM Trans. Embed. Comput. Syst.* 7.3 (2008), 36:1–36:53.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 28.7 (2009), pp. 966–978. DOI: 10.1109/TCAD.2009.2013287. URL: https://doi.org/10.1109/TCAD.2009.2013287.

[WLP+12] Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. "Improving the Precision of WCET Analysis by Input Constraints and Model-Derived Flow Constraints." In: *Advances in Real-Time Systems*. Springer, 2012, pp. 123–143.

[WPG+21] Reinhard Wilhelm, Markus Pister, Gernot Gebhard, and Daniel Kästner. "Testing Implementation Soundness of a WCET Analysis Tool." In: *A Journey of Embedded and Cyber-Physical Systems*. Springer, 2021, pp. 5–17.

[WR12] Reinhard Wilhelm and Jan Reineke. "Embedded systems: Many cores - Many problems." In: *SIES*. IEEE, 2012, pp. 176–180.

[WW08] Reinhard Wilhelm and Björn Wachter. "Abstract Interpretation with Applications to Timing Validation." In: *CAV*. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 22–36.

[WBP+13] Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. "End-to-end latency computation in a multi-periodic design." In: *SAC*. ACM, 2013, pp. 1682–1687.

[XCW+20] Zheng Xiao, Liwen Chen, Bangyong Wang, Jiayi Du, and Keqin Li. "Novel fairness-aware co-scheduling for shared cache contention game on chip multiprocessors." In: *Inf. Sci.* 526 (2020), pp. 68–85.

[XLW+16] Q. Xiong, Z. Lu, F. Wu, and C. Xie. "Real-time analysis for wormhole NoC: Revisited and revised." In: *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. 2016, pp. 75–80. DOI: 10.1145/2902961.2903023.

[XWL+17] Q. Xiong, F. Wu, Z. Lu, and C. Xie. "Extending Real-Time Analysis for Wormhole NoCs." In: *IEEE Transactions on Computers* 66.9 (2017), pp. 1532–1546. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2686391.

[YCC18] Mikail Yayla, Kuan-Hsun Chen, and Jian-Jia Chen. "Fault Tolerance on Control Applications: Empirical Investigations of Impacts from Incorrect Calculations." In: *EITEC@CPSWeek*. IEEE Computer Society, 2018, pp. 17–24.

[Yun15] Heechul Yun. "Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms." In: 2015.

[ZDS09] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. "Towards practical page coloring-based multicore cache management." In: *EuroSys*. ACM, 2009, pp. 89–102.

[ZDB+20] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. "DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency." In: *RTSS*. IEEE, 2020, pp. 128–140.

[ZGD11] Haitao Zhu, Steve Goddard, and Matthew B. Dwyer. "Response Time Analysis of Hierarchical Scheduling: The Synchronized Deferrable Servers Approach." In: *RTSS*. IEEE Computer Society, 2011, pp. 239–248.

[ZCC+22] Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms." In: *Softw. Pract. Exp.* 52.5 (2022), pp. 1095–1113.

INDEX

n-step equivalent, 179
n-way associative cache, 17

acceptance ratio, 50
ack-nack flow control, 24
action space, 186
action value function, 186
adaptive routing algorithm, 25
agent, 186
algorithmic routing, 25
all-or-nothing property, 53
aperiodic activation, 35
arbitrary deadline, 36
arbitration response message, 98
arbitration state, 96
arbitration table, 95

banyan network, 20
barrier, 43
barrier function, 189
basic block, 29
benchmarks, 50
benes network, 20
bisection bandwidth, 22
blocked, 34
body flits, 21
Boltzmann Q-Policy, 189
branching decision, 115
bufferless flow control, 23
busy waiting, 44
busy window, 45
butterfly network, 20

cache affinity, 35
cache analysis, 29
cache hit, 17
cache miss, 17
cache partitioning, 18
cache-related preemption delay, 30
capacity augmentation bound, 49
capacity cache miss, 18
centralized-priority arbitration, 94
circuit switching, 23
circular waiting, 93
clear to send, 98
clock skew, 96
clustered scheduling, 41
coherency cache miss, 18
complete series of progressions, 91
compulsory cache miss, 18
conditional branch, 115
consecutive stationary gang assigments, 68
constrained deadline, 36
context switch, 33
control-flow graph, 29
control-flow joins, 30
credit based flow control, 24
critical instant, 45
critical state, 174
crossbar, 20
cumulative future reward, 186

de-multiplexer, 95
deadline-monotonic, 40
decision vertex, 115
deep Q-Network, 187
deeply red pattern, 166
deferrable server, 40
deflection based routing, 26
degree, 22
destination router, 24
detected mode, 166
deterministic finite automata, 167
deterministic routing, 25
deterministic worst-case execution time, 28
diameter, 22
dimension order routing, 25
direct & indirect networks, 21
direct interference, 88
direct mapped cache, 17
dirichlet-rescale , 50
discount rate, 186
discrete time domain, 94

dispatcher, 33
distributed routing, 25
dominance relation, 48
downstream router, 21
DP-Fair, 57
dynamic self-suspension, 36

early sensor fusion, 197
east port, 96
engineering model, 3
environment, 186
epoch, 94
equivalent states, 179
Erdős–Rényi, 50, 153
error-bound models, 166
evenly distributed pattern, 166
event-triggered, 35
execution model, 37
explicit output comparison, 166
explicit receiving, 200

fastest progression, 91
fault event, 171
feasible job collection, 37
feasibly schedulable, 38
federated scheduling, 15
firm real-time systems, 1
fixed tasks, 43
fixed-priority non-preemptive rigid gang scheduling, 210
fixed-priority preemptive stationary rigid gang scheduling, 211
flat scheduling algorithm, 40
flits, 21
flow id, 95
flow ready, 96
flows per core, 98
fluid scheduling algorithm, 40
Fork-Join, 50
from core port, 96
full-duplex, 20
fully associative cache, 17

gang scheduling, 15, 43
global rigid gang scheduling, 56
global scheduling, 41

hard real-time system, 1
header flit, 21
heterogeneous cores, 15
hierarchical scheduling, 39
homogeneous cores, 15
hop count, 22

implicit deadline, 36
implicit path enumeration, 30
implicit receiving, 200
indirect interference, 88
induced markov chain, 181
induced self-suspension behaviour, 63
integer-linear program, 75
intellectual property cores, 20
inter- and intra task analysis, 48
inter-processor interrupts, 34
intermediate sensor fusion, 197
interrupt service routine, 33
interrupts, 33
invalid- & valid complete branching decisions, 117

job collection, 37
job shop scheduling problem with fixed machines, 89
job-level fixed-priority, 40

k2U framework, 75

late sensor fusion, 197
lateness, 37
Layer-by-Layer, 50
layer-by-layer, 153
learned policy, 186
least-frequently used, 17
least-recently used, 17
length, 111
level-i busy window, 45
link width, 20
list scheduling, 106
local schedule, 62
look-up table, 95

makespan, 39
malleable, 58
malleable gang scheduling, 44
markov decision process, 186
maximum data age, 197
maximum lateness, 39
maximum reaction time, 197

may- and must analysis, 18
measurement-based probabilistic timing analyses, 31
messages, 21
migratory tasks, 43
minimal and non-minimal path routing algorithms, 25
minimal- & maximal observed execution times, 28
modable gang scheduling, 44
moldable, 58
monte carlo methods, 187
multi-point progressive blocking, 88
multiplexer, 95
must & may cache states, 30
mutual exclusion, 43

network interface, 20
nominal state, 174
non-bufferless switching, 23
non-deterministic routing, 25
non-uniform memory access, 16
non-work-conserving, 40
north port, 96

omega network, 20
on-off flow control, 24
ordinary reservation system, 147

packetizer, 21
packets, 21
parallel path progression, 107
parallel path progression prioritization, 137
parameter vectors, 50
parametric speed-up factor, 55
partitioned scheduling, 41
path diversity, 24, 55
path monotonic prioritization, 107
path monotonic progression, 107
path vector, 95
path-monotonic decomposition, 158
pending subjob, 113
per-path & per-program analysis, 31
periodic activation, 35
periodic task model, 4
phits, 21
policy gradient ascent, 187
polling server, 39
precedence constraints, 111
preemption, 34
priority-driven, 40
Probabilistic analyses for parallel DAG tasks, 109
probabilistic conditional DAG task, 105
probabilistic worst-case execution time, 28, 31
processing graph, 196, 198
proportionate-fair scheduling, 40

quasi-continuous, 2

Rand Fixed Sum, 50
random replacement, 17
rate-monotonic, 40
read-on-start, 201
ready queue, 34
ready-to-transmit, 96
redundant multi-threading, 166
regular vertex, 115
reinforcement learning, 185
reliable mode, 166
request message, 97
request state, 96
residue pC-DAG, 116
response message, 97
response phase, 96
response state, 96
response-time, 37
reward function, 186
rigid, 58
rigid gang scheduling, 44
rigid gang task model, 54
rt-gang model, 54

scheduling anomaly, 107
scientific model, 3
segment-level fixed-priority, 40
segmented self-suspension, 36
semi-stationary rigid gang scheduling, 56
simplex, 20
simultaneous progression switching protocols, 53, 55
slowest progression, 91
soft real-time systems, 1
software-based fault-tolerance, 165

source core, 96
source router, 24
source routing, 25
south port, 96
speed-up factors, 49, 78
spinning parallel reservation systems, 121
sporadic activation, 35
sporadic task model, 4
starting time, 37
stationary gang assignment, 59
stationary rigid gang scheduling, 44, 55, 56
store-and-forward switching, 23
suspension, 34
suspension-aware uniprocessor scheduling, 53
sustainability, 44
synchronization message, 96
system modes, 2
system tick, 33

tail flit, 21
tardiness, 37
tardy, 118
task-level fixed-priority, 40
temporal difference learning, 187
threads, 34
time-demand analysis, 45
time-triggered, 35
timing analysis, 27
total volume, 111
transfer function, 29

uniform heterogeneous multiprocessors, 15
uniform memory access, 16
unrelated heterogeneous multiprocessors, 15
unreliable mode, 166
upstream router, 21
UUnifast, 50
UUnifast Discard, 50

value analysis, 29
value-based reinforcement learning, 187
virtual channel, 21

weakly-hard soft error constraints, 166
west port, 96
window of interest, 46
work-conserving, 40
write-on-finish, 201

zig zag path, 97

LIST OF FIGURES

Figure 1.1 An exemplary realistic real-time system architecture, which is proposed by the company *Perceptin* in the RTSS 2021 Industry challenge, to implement an autonomous driving system. The vertices in the processing graph denote functional modules such as sensor data preprocessing, perception, tracking, trajectory planning, and control. The directed edges in the processing graph denote the data dependencies between the modules, e.g., the data produced by the *localization* module is used by the *planning* module in its computation. 2

Figure 1.2 Model hierarchy redrawn from Edward Lee: Plato and the Nerd [Lee17]. The emphasized arrows denote the focus of the modeling used in this dissertation. 3

Figure 1.3 An exemplary periodically activated single-input single-output real-time system architecture. The periodically generated sensor sample is processed by the control vertex, which is activated upon new sensor data, and writes the result to the actuator when the execution is finished. 4

Figure 2.1 A taxonomy of real-time systems as considered relevant to this dissertation, where the bold vertices accent the relevance to the contributions in this dissertation. 13

Figure 2.2 Architectural view of an exemplary multicore system-on-chip, which is redrawn from [Abd17]. 15

Figure 2.3 Architectural view of an exemplary memory and cache hierarchy. 16

Figure 2.4 Packetization of a message into packets; each of which consist of a head flit, body flits, and a tail flit. 21

Figure 2.5 Two exemplary direct regular topologies, namely a 2D-mesh on the left, and a 2D-Torus on the right. 22

Figure 2.6 Exemplary transmission of flits in a wormhole switched network-on-chip from a source core to a destination core over 3 links, which is redrawn from [Abd17]. The routers are partially represented by the input and output buffers of a single virtual channel. When the buffers are fully exhausted at the downstream router, the upstream router is stalled on transmission. 23

Figure 2.7 Symbolic representation of all execution times in a system and the corresponding terminology with regard to worst-case execution time analysis adapted and redrawn from [WEE+08]. 27

Figure 2.8 Flow chart of the standard approach to static timing analysis as used for instance by the tool AbsInt. 29

Figure 2.9 Symbolic example task states combined from RTEMS and FreeRTOS specifications. 33
Figure 2.10 Taxonomy of scheduling algorithms as relevant to this dissertation. 39
Figure 2.11 An exemplary *window of interest* for a job J_i^ℓ, released at time a_i^ℓ, and missing its deadline at time d_i^ℓ. Time t' denotes the earliest time before d_i^ℓ such that during $[t', d_i^\ell]$ the processor is continuously busy executing jobs with deadline no later than d_i^ℓ. 46
Figure 3.1 A symbolic job J_i^ℓ of a rigid gang task τ_i with gang size $E_i = 2$. Each thread (subjob) in the gang is subject to the co-scheduling constraint. 59
Figure 3.2 A cut-out of an exemplary task-level fixed-priority stationary rigid gang schedule with respect to the processors' local schedules of the three processors P_1, P_2, and P_3. In the shown example, the two tasks τ_k and τ_i are analyzed where τ_i has a higher priority than τ_k. Task τ_k is assigned to processors P_2 and P_3, and τ_i is assigned to P_1, P_2, P_3. 62
Figure 3.3 An illustration of the suspension induced behavior of task τ_i from the *perspective* of task τ_k under analysis as derived from Figure 3.2. The cause of the interference of the higher-priority task τ_i in the *schedule* is transparent to the *local schedule* of τ_k. This transparent behaviour is modeled as *self-suspension*. 62
Figure 3.4 Consecutive stationary gang assignments $A_k^0, A_k^1, A_k^2, A_k^3$ of a gang task τ_k with $E_k = 3$ on a system using 4 identical processors P_i for $i \in \{0, \ldots, 3\}$. The four distinct assignments are generated by a sliding window of size 3. 68
Figure 3.5 Enumeration of all consecutive stationary gang assignments of a task τ_k (black window) under the condition of a given consecutive stationary gang assignment of a higher-priority task (light gray window). 69
Figure 3.6 Acceptance ratio for *light* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according *Setting II*. 83
Figure 3.7 Acceptance ratio for *moderate* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*. 83
Figure 3.8 Acceptance ratio for *heavy* sporadic implicit-deadline gang task sets where the gang size of each task is chosen according to *Setting I*. 84
Figure 3.9 Acceptance ratio for *light* sporadic constrained-deadline gang task sets according to *Setting I*. The deadline is chosen randomly between $70\% - 100\%$ of the minimum inter-arrival time. 85

Figure 3.10 Acceptance ratio for *light*, *moderate*, *heavy* sporadic constrained-deadline gang task sets according to *Setting II*. The deadline is chosen randomly between $70\% - 100\%$ of the minimum inter-arrival time. 85
Figure 3.11 Progressions of a message which involves 2 cores and 2 routers, i.e., 3 links, when $C_i = 10$. The numbers associated with an edge indicate the number of buffered phits at the respective routers or cores. The red dashed path illustrates the beginning of the *fastest* progression and the blue dashed path illustrates the beginning of a *slowest* progression. 90
Figure 3.12 The arbitration and data message transmission can be done sequentially within a single data net, or interleaved using a separate arbitration net. Each cluster can transmit a flit in each cycle to the centralized arbiter core. 93
Figure 3.13 The arbitration and communication is pipelined using two distinct nets, i.e., the arbitration and the data net. 94
Figure 3.14 A possible implementation of the SP2 arbitration using phit-sized virtual channels, and crossbars. 94
Figure 3.15 Clustering and routing for each router in the *arbitration state* for the $N \times N$ $2D$-Mesh topology for even and odd dimensions. Even dimensions lead to irregular clusters, whereas odd dimensions lead to regular clusters. 97
Figure 3.16 Overhead for the arbitration algorithm in μs for varying number of flows per core (FPC), 2D-Mesh dimensions, and core frequencies from 800 MHz to 3000 MHz. 100
Figure 3.17 Overhead for the arbitration communication in μs for varying number of flows per core (FPC), 2D-Mesh dimensions, and link bandwidths. Note that the communication overhead is independent from the arbiter core frequency. 101
Figure 4.1 An exemplary directed-acyclic graph (DAG) with subtasks $v_1, v_2, \ldots, v_9$. The numbers within the vertices denote the subtasks' worst-case execution time. The arrows represent the precedence constraints indicating that the release of a subjob depends on the finishing of all incident subjobs. 111
Figure 4.2 An exemplary probabilistic conditional DAG task in which each conditional vertex (diamond) denotes that only one of its adjacent subjobs is released (with the annotated probability) during runtime. In this specific example four different DAG structures can be instanced during runtime. 114
Figure 4.3 An exemplary probabilistic conditional DAG task in which each conditional vertex (diamond) denotes that only one of its adjacent subjobs is released (with the annotated probability) during runtime. In this specific example four different DAG structures can be instanced during runtime. 115

Figure 4.4 A complete substitution of the pC-DAG shown in Figure 4.2 where the valid branching decisions $b_{2,4}$ and $b_{7,8}$ are chosen. The probability for this complete substitution G' is given by $0.4 \cdot 0.7 = 0.282$ with $C = 15$ and $L = 14$. 119

Figure 4.5 An exemplary schedule of the pC-DAG illustrated in Figure 4.2 where the decisions are taken according to Figure 4.4. The resulting DAG job with its subjobs (vertices) are executed on 2-in parallel spinning reservations with budget $E_i = 8$ each. One spinning reservation is partitioned to processor P_1 and the other is partitioned to processor P_2 where they are scheduled according to some scheduling policy. The dashed areas represents that the reservations are preempted by higher-priority reservations or are depleted and therefore do not provide service. The unused time interval from 8 to 10 in the first reservation indicates that the reservation provides the service but the DAG is not executed in the reservation due to either its precedence constraints or lack of workload. 122

Figure 4.6 Supply Bound Function $sbf(t)$ of the a reservation system with m_i in-parallel service and E_i reservation budget each. The supply-bound function denotes the minimal guaranteed service in any interval of length t for any feasible schedule. 125

Figure 4.7 Service density (E_i/D_i) with respect to the associated number of in-parallel reservations of equal probability upper bounds for a single deadline-miss of $5\%, 10\%, 20\%$. Furthermore, the tardy workload bound ρ_i is set to 10% and 20% of the task's deadline. The expected contribution of the critical-path length to the overall volume is 19%. 133

Figure 4.8 Service density (E_i/D_i) with respect to the associated number of in-parallel reservations of equal probability upper bounds for a single deadline-miss of $5\%, 10\%, 20\%$. Furthermore, the tardy workload bound ρ_i is set to 10% and 20% of the task's deadline. The expected contribution of the critical-path length to the overall volume is 56%. 134

Figure 4.9 Required reservation resources compared for synthetically generated pC-DAGs for each deadline-miss probability $5\%, 6\%, \cdots, 20\%$ compared to a hard real-time reservation system. 135

Figure 4.10 Exemplary schedule for a 2-gang reservation system with a 2-path collection of the DAG illustrated in Figure 4.1 with a deadline of 16 time units. 145

Figure 4.11 Exemplary schedule for an ordinary reservation system of the DAG illustrated in Figure 4.1 with a deadline of 16. A reservation system that consists of 4 equally sized reservations of 13.5 time units using a 3-path collection computed by nPCA. 150

Figure 4.12 Relative makespan of DAGs generated by the Erdős–Rényi method with $100-150$ vertexes and a connection probability of $45-50\%$. 155
Figure 4.13 Relative makespan of DAGs generated by the layer-by-layer method with $10-15$ layers, parallelism of $10-30$, and a connection probability of $50-60\%$. 155
Figure 4.14 Over-provisioning for Erdős–Rényi with $100-150$ vertexes, $35-40\%$ connection probability and $\alpha = [1.2, 1.5]$. 156
Figure 4.15 Over-provisioning for layer-by-layer $10-15$ layers, parallelism $10-30\%$, connection probability $50-60\%$ and $\alpha = [2, 3]$. 157
Figure 4.16 An exemplary directed-acyclic graph (DAG) with subtasks $v_1, v_2, \ldots, v_{10}$. The numbers within the vertices denote the subjob's worst-case execution time and the arrows represent the precedence constraints of the subjobs. The path-monotonic decomposition is highlighted by the border around the respective subtasks. 158
Figure 4.17 Exemplary schedule for the DAG illustrated in Figure 4.16 on five suspension-aware reservations for the path-monotonic decomposition $\{v_1, v_4, v_5, v_6\}, \{v_7, v_8\}, \{v_{10}\}, \{v_2, v_3\}, \{v_9\}$. The reservation that services the subjobs $\{v_1, v_4, v_5, v_6\}$ is assigned the lowest priority 1 and $\{v_9\}$ is assigned the highest priority 5. 160
Figure 5.1 An exemplary schedule for two tasks $\tau_i, \tau_k \in \mathbb{T}$ that are subjected to faults, which result in soft-errors if at least one fault occurs during that jobs execution. Job J_k^q is executed in the unreliable mode. 170
Figure 5.2 An exemplary k-error-automata $\mathcal{A}_k$ and the $(2, 3)$-compliant automata $\mathcal{A}_k^*$ is highlighted in bold, where the darker states are *critical states* and the lighter states are *nominal states*. 174
Figure 5.3 An exemplary k-error-automata $\mathcal{A}_k$ and the 3-consecutive error constraint compliant automata $\mathcal{A}_k^*$ is highlighted in bold, where the darker states are *critical states* and the lighter states are *nominal states*. 175
Figure 5.4 3-consecutive error constraint compliant automata $\mathcal{A}_k^*$. 176
Figure 5.5 A minimal $(2, 3)$-compliant 3-error-automata $\mathcal{A}_3^*$ as generated by Algorithm 7. 184
Figure 5.6 A schematic of a Markov Decision Process (MDP) which consists of an *agent* and the *environment*. The *environment* E denotes the state space $s_t \in S$, which is a representation of the environment at time t. In a markov process, the next state s_{t+1} depends solely on the current state s_t. Reinforcement learning is a framework to train an *agent* to interact with the environment by means of *actions*, which influence the state transitions. 186

Figure 5.7 Exemplary *environment* state transition from s_t to s_{t+1} for $(3, 5)$ constraints, where the j-th column vector refers to the status of the $(5 - (j - 1))$-th latest executed job for $j \in \{1, \ldots, 5\}$. 188

Figure 5.8 Normalized average execution time of each approach with respect to the *STA* approach with $m \in \{2, 4, 6, 8\}$ and $k = 10$, and $p_e \in \{0.05, 0.15, 0.3\}$ for a single task. The worst-case execution times are set to $C^d = 1.5 \times C^u$ and $C^r = 3.5 \times C^u$. 191

Figure 5.9 Measured time-averaged utilization of 100 task sets (of 40 tasks each) on simulated scheduled on 4 processors using partitioned scheduling. The parameters, $m \in \{2, 4, 6, 8\}$ and $k = 10$, and $p_e \in \{0.05, 0.15, 0.3\}$ and the worst-case execution times are set to $C^d = 1.5 \times C^u$ and $C^r = 3.5 \times C^u$. 193

Figure 6.1 An exemplary realistic real-time system architecture used to implement autonomous driving systems by the company *Perceptin*. The vertices in the processing graph denote functional modules such as sensor data pre-processing, perception, tracking, trajectory planning, and control. The directed edges in the processing graph denote the data dependencies between the modules, e.g., the data produced by the *localization* module is used by the *planning* module in its computation. 196

Figure 6.2 A processing graph formally describes the data communication of the application. The source vertex in a processing graph (i.e., vertices which do not have any predecessors) represent sensor vertices which produce data either in a time-triggered or event-triggered manner. All sensors produce time-stamped data with respect to the sensor's local clocks. The processing vertices in the graph execute periodically on the latest available data and produce an output when the execution is finished. Each processing vertex is assigned to a processing unit, e.g., a CPU, GPU, or DSP on which it is executed. 198

Figure 6.3 Exemplary schedule for two tasks in a processing chain $E_j = \langle \tau_{j_1}, \tau_{j_2} \rangle$. A processing chain instance for the 4-th job of task τ_{j_2} is for example given by $\langle J_{j_1,2}, J_{j_2,4} \rangle$ and the propagation latency is $9 - 6 = 3$ time units. 201

LIST OF TABLES

Table 3.1 An exemplary *arbitration table* at the core A_5 of the network in Figure 3.12. The additional color annotation is used to visually identify the three exemplary flows. 95
Table 3.2 Number of arbiter core cycles required for the arbitration algorithm in a N×N 2D-Mesh network-on-chip with 1,2,4, and 8 flows per core. 101
Table 4.1 Tabular representation of the probabilities of the parameters total volume and length for the probabilistic conditional DAG task illustrated in Figure 4.2. 117
Table 4.2 Summary of used notation in this section. 120
Table 4.3 Summary of used Notation. 140
Table 4.4 Difference of the path cover bound w compared to the minimal number of paths calculated by the greedy approach for a full cover for the layer-by-layer DAG sets. 154
Table 4.5 Difference of the path cover bound w compared to the minimal number of paths calculated by the greedy approach for a full cover for the Erdős–Rényi DAG sets. 154
Table 4.6 A possible path-monotonic prioritization of the DAG illustrated in Figure 4.1. 161

LIST OF ALGORITHMS

Figure 1 DM Stationary Rigid Gang Schedulability Analysis and Assignment. 72
Figure 2 Behavioural description of the arbitration routine in SP2 93
Figure 3 Calculation of Reservation Systems 130
Figure 4 n-Path Collection Approximation (nPCA) 142
Figure 5 Approximate Minimal Waste Gang 146
Figure 6 Minimal Service Ordinary Reservations 151
Figure 7 Generation of minimal compliant $\mathcal{A}_k^*$ 181
Figure 8 Maximal Sensor Data Time-Stamp Difference 209