# Analyses and Optimizations of Timing-Constrained Embedded Systems Considering Resource Synchronization and Machine Learning Approaches

**Dissertation**

zur Erlangung des Grades eines

D o k t o r s   d e r   I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Junjie Shi

Dortmund

2023

|  |  |
|---|---|
| Tag der mündlichen Prüfung: | 20. November 2023 |
| Dekan / Dekanin: | Prof. Dr. Gernot Fink |
| Gutachter / Gutachterinnen: | Prof. Dr. Jian-Jia Chen |
|  | Prof. Dr. Alessandro Biondi (Scuola Superiore Sant'Anna) |

# Acknowledgments

Embarking on this academic journey was both a challenge and a transformative experience. Along the way, I have been fortunate to have had the support, guidance, and camaraderie of many extraordinary individuals, and I wish to take a moment to express my heartfelt gratitude to all of them.

First and foremost, I extend my deepest appreciation to my supervisor, Prof. Dr. Jian-Jia Chen. Our shared journey from the days of my Master's research to this culmination in my PhD has been nothing short of enriching. Your unwavering guidance, encouragement, and patience have been the driving force behind my research. Your expertise, critical insights, and belief in my capabilities have continually propelled me forward, even during the most challenging times.

I am sincerely grateful to the members of my defence committee, Prof. Dr. Alessandro Biondi, Prof. Dr.-Ing. Peter Ulbrich, and Prof. Dr. Falk Howar. Your thorough reviews, constructive feedback, and discussions have enriched my work and broadened my academic horizons.

To my colleagues at DAES group, namely Georg von der Brüggen, Ching-Chi Lin, Mario Günzel, Christian Hakert, Nils Hölscher, Daniel Kuhse, Vahidreza Moghaddas, Noura Sleibi, Tristan Seidl, Harun Teper, Niklas Ueter, Zahra Valipour, Mikail Yayla, Claudia Graute, and Lars Dröge, thank you for creating a stimulating and supportive environment. Sharing ideas, coffee breaks, and moments of camaraderie with you all has been an integral part of this journey. A special mention to Kuan-Hsun Chen, who has *semi-supervised* me since my Master thesis, providing invaluable help and suggestions throughout my PhD. I extend my gratitude to Georg von der Brüggen and Niklas Ueter for their unwavering support and collaboration on our papers, as well as their invaluable feedback for my dissertation. I thank Mario Günzel and Nils Hölscher for the joy they brought to our shared spaces, and Christian Hakert for being an ever-helpful guide.

I cannot express enough gratitude to my family. To my parents, grandparents, uncles ,and aunts, your unwavering love, belief, and support have been the bedrock upon which I have built my aspirations. Your sacrifices, patience, and encouragement have been my guiding stars, ensuring I never lost sight of my goals, no matter how distant they seemed.

Lastly, to my dear friends, especially Feifei Zheng, thank you for being my sounding board and my confidants. Your optimism, timely pep talks, and unending faith have been my source of energy.

In essence, this dissertation is not just a reflection of my academic endeavor, but a testament to the wonderful people who believed in me, pushed me, and stood by me. I am eternally grateful.

# Abstract

Nowadays, embedded systems have become ubiquitous, powering a vast array of applications from consumer electronics to industrial automation. Concurrently, statistical and machine learning algorithms are being increasingly adopted across various application domains, such as medical diagnosis, autonomous driving, and environmental analysis, offering sophisticated data analysis and decision-making capabilities. As the demand for intelligent and time-sensitive applications continues to surge, accompanied by growing concerns regarding data privacy, the deployment of machine learning models on embedded devices has emerged as an indispensable requirement. However, this integration introduces both significant opportunities for performance enhancement and complex challenges in deployment optimization.

On the one hand, deploying machine learning models on embedded systems with limited computational capacity, power budgets, and stringent timing requirements necessitates additional adjustments to ensure optimal performance and meet the imposed timing constraints. On the other hand, the inherent capabilities of machine learning, such as self-adaptation during runtime, prove invaluable in addressing challenges encountered in embedded systems, aiding in optimization and decision-making processes.

This dissertation introduces two primary modifications for the analyses and optimizations of timing-constrained embedded systems. For one thing, it addresses the relatively long access times required for shared resources of machine learning tasks. For another, it considers the limited communication resources and data privacy concerns in distributed embedded systems when deploying machine learning models. Additionally, this work provides a use case that employs a machine learning method to tackle challenges specific to embedded systems.

Firstly, to meet timing requirements, we design a resource synchronization protocol that bounds the worst-case response time of tasks, addressing the long access times for shared resources, such as GPUs used in machine learning tasks. We also support the designed protocol on two real-time operating systems (RTOSes), namely LITMUS$^{RT}$ and RTEMS. Additionally, we propose a formal verification framework to ensure the correctness of implementations under the assumption that all implemented protocols are based on a correct RTOS.

Secondly, considering the resource-constrained distributed embedded systems that prohibit raw data sharing due to privacy concerns, we optimize the deployment of a machine learning model, specifically employing model-based optimization. We propose two distinct strategies: a) treating the entire system as a black box, with all end nodes collaborating to enhance prediction accuracy and statistical stability, and b) enabling parallel processing by having all end nodes work collectively to improve timing efficiency.

Thirdly, we provide a use case that employs reinforcement learning to minimize the average execution time of an embedded system under $(m, k)$ soft error constraints. Here, each task can operate in one of three distinct modes, each with specific correctness assumptions and execution times. An RL-agent is trained to dynamically select the optimal execution mode for the next job of a task, aiming to minimize its average execution time, particularly in scenarios with variable or unpredictable error rates.

By addressing these key aspects, this dissertation contributes to the analysis and optimization of timing-constrained embedded systems, considering resource synchronization and machine learning models to enable improved performance and efficiency in real-time applications with stringent constraints.

# List of 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:

[CBS+18] J.-J. Chen, G. von der Brüggen, J. Shi, and N. Ueter. “Dependency Graph Approach for Multiprocessor Real-Time Synchronization”. In: *39th IEEE Real-Time Systems Symposium, RTSS*. 2018, pp. 434–446.

[CSB+22] J.-J. Chen, J. Shi, G. von der Brüggen, and N. Ueter. “Scheduling of Real-Time Tasks With Multiple Critical Sections in Multiprocessor Systems”. In: *IEEE Trans. Computers* 71.1 (2022), pp. 146–160.

[SBR+21] J. Shi, J. Bian, J. Richter, K.-H. Chen, J. Rahnenführer, H. Xiong, and J.-J. Chen. “MODES: model-based optimization on distributed embedded systems”. In: *Mach. Learn.* 110.6 (2021), pp. 1527–1547.

[SEC+22] J. Shi, C.-C. von Egidy, K.-H. Chen, and J.-J. Chen. “Formal Verification of Resource Synchronization Protocol Implementations: A Case Study in RTEMS”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.11 (2022), pp. 4157–4168.

[SPM+22] J. Shi, J. D. T. Pham, M. Münch, J. V. Hafemeister, J.-J. Chen, and K.-H. Chen. “Supporting Multiprocessor Resource Synchronization Protocols in RTEMS”. In: *16th annual workshop on Operating Systems Platforms for Embedded Real-Time applications, OSPERT*. 2022. eprint: 2104.06366.

[SUB+19a] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Multiprocessor Synchronization of Periodic Real-Time Tasks Using Dependency Graphs”. In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2019, pp. 279–292.

[SUB+19b] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Partitioned Scheduling for Dependency Graphs in Multiprocessor Real-Time Systems”. In: *25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*. 2019, pp. 1–12.

[SUB+21] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Graph-Based Optimizations for Multiprocessor Nested Resource Sharing”. In: *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*. 2021, pp. 129–138.

[SUC+23] J. Shi, N. Ueter, J.-J. Chen, and K.-H. Chen. “Average Task Execution Time Minimization under (m, k) Soft Error Constraint”. In: *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2023, pp. 1–13.

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:

[DLB+18] Z. Dong, C. Liu, S. Bateni, K.-H. Chen, J.-J. Chen, G. von der Brüggen, and J. Shi. “Shared-Resource-Centric Limited Preemptive Scheduling: A Comprehensive Study of Suspension-Based Partitioning Approaches”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. Ed. by R. Pellizzoni. IEEE Computer Society, 2018, pp. 164–176. DOI: 10.1109/RTAS.2018.00026. URL: https://doi.org/10.1109/RTAS.2018.00026.

[FSC+21] R. Freymann, J. Shi, J.-J. Chen, and K.-H. Chen. “Renovation of EdgeCloudSim: An Efficient Discrete-Event Approach”. In: *6th International Conference on Fog and Mobile Edge Computing, FMEC*. IEEE, 2021, pp. 1–8. DOI: 10.1109/FMEC54266.2021.9732572. URL: https://doi.org/10.1109/FMEC54266.2021.9732572.

[LGS+23] C.-C. Lin, M. Günzel, J. Shi, T. T. Seidl, K.-H. Chen, and J.-J. Chen. “Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies”. In: *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. IEEE, 2023, pp. 161–173. DOI: 10.1109/RTAS58335.2023.00020. URL: https://doi.org/10.1109/RTAS58335.2023.00020.

[LSU+23] C.-C. Lin, J. Shi, N. Ueter, M. Günzel, J. Reineke, and J.-J. Chen. “Type-Aware Federated Scheduling for Typed DAG Tasks on Heterogeneous Multicore Platforms”. In: *IEEE Trans. Computers* 72.5 (2023), pp. 1286–1300. DOI: 10.1109/TC.2022.3202748. URL: https://doi.org/10.1109/TC.2022.3202748.

[RSC+20] J. Richter, J. Shi, J.-J. Chen, J. Rahnenführer, and M. Lang. “Model-based optimization with concept drifts”. In: *GECCO ’20: Genetic and Evolutionary Computation Conference*. Ed. by C. A. C. Coello. ACM, 2020, pp. 877–885. DOI: 10.1145/3377930.3390175. URL: https://doi.org/10.1145/3377930.3390175.

[SCZ+17] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings. “Implementation and Evaluation of Multiprocessor Resource Synchronization Protocol (MrsP) on LITMUSRT”. In: *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. 2017.

# Contents

Chapter 1

# Introduction

## Contents

Embedded systems have progressively emerged as a pivotal component in contemporary technological paradigms, underpinning an extensive spectrum of applications, extending from consumer electronics to industrial automation. Their ubiquity and evolving capabilities have revolutionized how we interact with technology in daily life. Concurrently, there has been a significant surge in the adoption of statistical and machine learning algorithms across various fields, including autonomous driving, industrial automation, and environmental analysis. This convergence of advanced algorithms with embedded systems not only offers sophisticated data analysis and decision-making capabilities but also highlights a critical shift towards deploying machine learning models on embedded edge devices. However, while this integration promises substantial performance enhancements, it also brings to the fore challenges in terms of data privacy and latency, necessitating the optimization of machine learning deployments on these platforms.

Optimizing these deployments, in turn, introduces a set of new challenges. Embedded systems, characterized by their limited computational resources, stringent power constraints, and rigorous timing requirements, require robust and innovative solutions to ensure optimal performance for the deployment of machine learning models. These requirements call for the development of novel methodologies and protocols to effectively manage and balance these constraints. Concurrently, the inherent capabilities of machine learning, such as dealing with complex and non-linear systems, continuous learning, handling uncertainty, and self-adaptation during runtime, prove invaluable in addressing challenges encountered in embedded systems, such as enhancing optimization and decision-making in industrial automation.

## 1.1 Overview of Explored Domains

In the subsequent sections, several critical aspects related to the analysis and optimization of timing-constrained embedded systems are presented. The focus is on resource synchronization and the deployment of machine learning models. Additionally, the potential applications of machine learning techniques in overcoming challenges specific to embedded systems are investigated.

### 1.1.1 Multiprocessor Resource Synchronization

In time critical embedded systems, also known as hard real-time systems, the correctness depends not only on the logical result of computation, but also on the time at which the results are produced. For such systems, a deadline miss can result in catastrophic consequences. Hard real-time systems can, for instance, be found in aviation control systems, medical equipment like pacemakers, nuclear power plant, and some automotive safety systems. To ensure safe operations of such embedded systems, the satisfaction of real-time requirements must be verified, for instance, by determining worst-case response time.

In concurrent multi-task systems, tasks may request the same shared resources, e.g., files, memory cells, buses, or external accelerators like graphical processing units (GPUs). In order to prevent race condition or data corruptions, the accesses to shared resources are mutually exclusive. The piece of code that accesses to shared resources is typically called *critical section*, which can be protected by using *binary semaphores* or *mutex locks*. Therefore, at any point in time no two task instances are in their critical sections that access the same shared recourse. If aborting or restarting a critical section is not allowed, due to mutual exclusion, a higher-priority job may have to be blocked until a lower-priority job unlocks the requested shared resource that was already locked earlier, a so-called *priority inversion*. Additionally, in nested resource sharing, where tasks can request multiple resources simultaneously, a deadlock can occur due to the *hold-and-wait* mechanism. For example, task $\tau_1$ locks resource 1 and waits for resource 2, while $\tau_2$ locks resource 2 and waits for resource 1. This circular wait causes the *deadlock*.

Resource synchronization protocols have been developed to ensure the timeliness while accessing shared resources, by preventing deadlocks and bounding the priority inversion. The study of such protocols for uni-processor systems can be traced back to the priority inheritance protocol (PIP) and priority ceiling protocol (PCP) by Sha et al. [SRL90] in 1990 and the stack resource policy (SRP) by Baker [Bak91] in

1991. The Immediate PCP (ICPP), a variant of PCP, has been implemented in Ada (called Ceiling locking) and POSIX (called Priority Protect Protocol).

Given the increasing demand for computational capacity and challenges associated with heat dissipation, multicore and multiprocessor systems are widely adopted as standard commercial embedded platforms. Towards this, several multiprocessor resource synchronization protocols also have been proposed, such as the Distributed PCP (DPCP) [RSL88], the Multiprocessor PCP (MPCP) [Raj90], the Multiprocessor SRP (MSRP) [GLN01], and the Multiprocessor resource sharing Protocol (MrsP) [BW13].

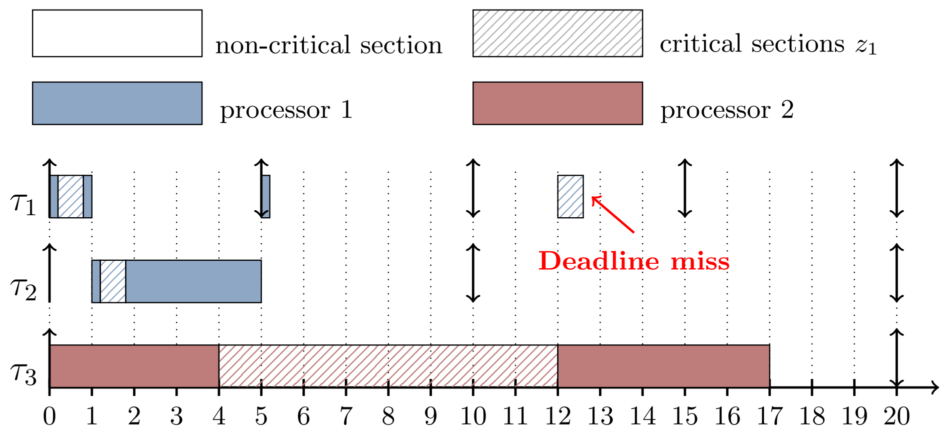
The performance of these protocols highly depends on several aspects:

1. the partitioning and prioritization strategy for tasks;
2. the methodologies employed for sharing resources, i.e., locally or globally; and
3. the decision as to whether a blocked job/task should spin or suspend itself.

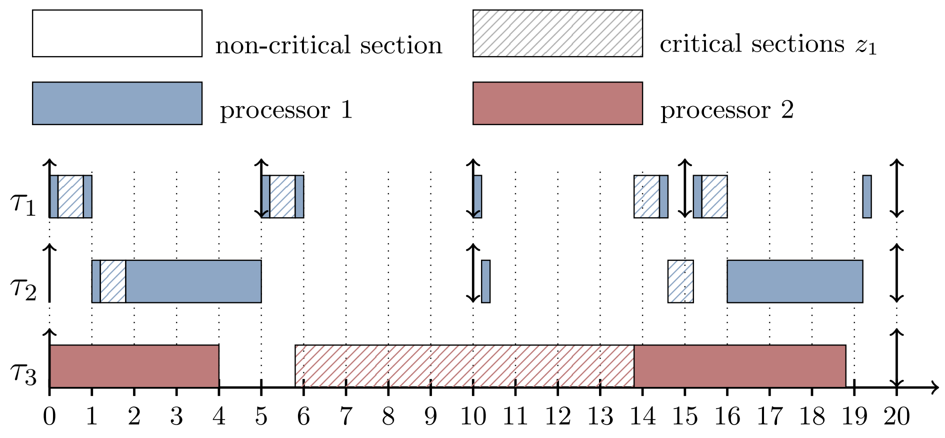
Although multiprocessor resource synchronizations have attracted extensive research in the past decades, and an array of protocols have been proffered in the literature, certain fundamental queries pertaining to the sharing of resources via locking mechanisms in multiprocessor systems remain unanswered:

- *What is the fundamental difficulty?*
- *What is the performance gap of partitioned, semi-partitioned, and global scheduling?*
- *Is it always beneficial to prioritize critical sections?*

Additionally, the protocols that currently exist are predominantly designed for traditional embedded systems where critical sections for resource occupancy are relatively short, such as for memory cell usage or shared bus access. However, in the context of machine learning tasks, these critical sections may be substantially elongated, such as instances where GPUs are utilized for computation. Although concurrent accesses to a GPU are permitted by default and can produce logically correct results, the internal scheduling within the GPU is unpredictable. Given that the worst-case response time for each task executed on a shared GPU is not bounded, in certain real-time applications, access to the GPU must be mutually exclusive. In the current landscape, the suite of multiprocessor resource synchronization protocols adhere to a work-conserving paradigm for critical sections. That is, an available processor will immediately execute any ready critical section from its corresponding ready queue. While this paradigm is generally efficient, it does not always guarantee optimal performance due to the mutually exclusive nature of critical section executions, especially when the lengths of critical sections exceed the periods of another task in the system that requires access to the same shared resource. A counter example of this can be found in Figure 1.1. We consider three tasks that request the same shared resource, i.e., $z_1$, in their critical sections, and two processors are available for execution of tasks. Figure 1.1 presents two multiprocessor partitioned schedules, which execute task $\tau_1$ and $\tau_2$ on processor 1, and task $\tau_3$ on processor 2. The schedule in Figure 1.1a is work-conserving with respect to the critical sections, i.e., the critical section of $\tau_3$ is executed immediately once it is ready, which leads to a deadline miss. However, the schedule in Figure 1.1b is non- work-conserving with respect to the critical sections, where all tasks meet their deadlines. That is, postponing the execution of critical section from $\tau_3$ until $\tau_1$ has finished the execution of its critical section.



**(a)** A work-conserving schedule.



**(b)** A non-work-conserving schedule.

**Figure 1.1:** Work-conserving multiprocessor synchronization versus non-work- conserving multiprocessor synchronization when the length of a critical section is longer than the period of another task.

Therefore, this paradigm shift presents new challenges in resource synchronization protocol design, necessitating a comprehensive reassessment and refinement of existing methodologies.

### 1.1.2 Implementation and Verification on Real-Time Systems

While the aforementioned protocols offer timing guarantees by bounding the worst-case response time of tasks, most of them operate on the assumption that the overheads invoked by their implementation are negligible. This assumption, however, necessitates critical examination. The performance of different protocols is highly contingent on their settings, such as the local or remote execution of critical sections, multiprocessor scheduling paradigm, and tasks' waiting semantics. For instance, under a suspension-based synchronization protocol, tasks waiting to access a shared resource are suspended, if the requested resource is not available, i.e., locked by another task. Although this strategy frees the processor for other ready tasks, thereby optimizing processor utilization, it simultaneously inflates context switch overhead due to additional enqueue and dequeue operations required with each suspension. Conversely, under a spin-based synchronization protocol, the task retains its privilege on the processor and waits by spinning until it can access the requested resource and execute its critical section, which is more efficient in real implementations when the critical sections are relatively short [BA07].

In contrast, only a few of protocols have been officially supported in real-time operating systems (RTOSes), such as the Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUS$^{\text{RT}}$) [CLB+06; Bra11], and Real-Time Executive for Multiprocessor Systems (RTEMS) [The21]. LITMUS$^{\text{RT}}$is an experimental platform based on the Linux kernel, primarily for academic purposes, e.g., validation of new proposed scheduling or synchronization algorithms, but not for practical applications. Brandenburg et al. [BA08] implemented DPCP, MPCP, and FMLP, Catellani et al. [CBH+15] implemented MrsP, and Shi et al. [SCZ+17] solidate the implementation of MrsP. Alternatively, RTEMS is an open-source real-time operating system which is popular for industrial applications. RTEMS has been widely used in many fields, e.g., space flight, medical, and networking. However, in RTEMS, only a few of resource synchronization protocols have been officially supported in the upstream repository, i.e., ICPP for uni-processor systems, and MrsP implemented by Catellani et al. [CBH+15] for multiprocessor systems. Therefore, we believe there is significant benefit in offering comprehensive support for LITMUS$^{\text{RT}}$ and RTEMS with resource synchronization protocols for related research. Subsequently, system designers can better understand the performance of resource synchronization protocols, leading to discussions on implementation optimizations.

During the design phase, a protocol is formally delineated as a set of rules, built upon abstracted system models and operating under specific assumptions within the operating system (OS). However, when transitioning from design to actual implementation, these abstractions and assumptions may not always be applicable. For instance, some OSes might not allow tasks with identical priorities. In such cases, the inherited ceiling priority, like that in ICPP, must be adjusted by setting diverse ceiling priorities and excluding these from the standard priorities designated for other tasks. Furthermore, components like the helper mechanism in

MrsP or the busy waiting in a FIFO-queue require appropriate data structures and operations to be integrated within the target OS. Any adaptations, driven by the OS's practical constraints, can deviate from the original specifications, potentially leading to unforeseen consequences.

Therefore, in addition to the exploration and design of resource synchronization protocols, careful implementation in real-time operating systems, coupled with a formal verification process, is pivotal. Implementing and verifying the protocol in tandem ensures that it operates as intended under real-world conditions. This method provides robust, reliable timing guarantees and minimizes the risk of unexpected outcomes due to deviations from the original specifications.

### 1.1.3 Deployment of Machine Learning Models on Distributed Embedded Systems

Besides the consideration of timeliness, the performance of machine learning models is of great importance. Recognizing that a singular, resource-constrained embedded device may not achieve optimal results, many applications now leverage the collective capabilities of multiple devices. As the focus shifts from individual devices to a more expansive view encompassing networked systems, distributed embedded systems and edge computing systems have gained prominence. These systems are increasingly utilized to execute various machine learning tasks to enhance the functionalities of embedded systems. Their advantages include high flexibility, scalability, and low energy consumption in real-world applications. These systems prove particularly useful in applications such as modern intelligent transportation systems. Traditional road condition monitoring relied on cloud computing networks to analyze data gathered from vehicles and infrastructure. This approach often resulted in significant computational overhead and marked network latency. To address these challenges, Rasheed et al. [RZH20] introduced the edge computing framework based on the vehicle-to-everything (V2X) concept, bringing about a notable enhancement in the efficacy of road monitoring systems. Nodes within this framework, including vehicles, surpass the capabilities of conventional sensors that were solely designed for data collection. These advanced nodes come equipped not just with a sensor module for environmental monitoring and data acquisition but also with a processing module, where all the data processing is performed at edge nodes. More advanced implementations in intelligent transportation now efficiently conduct lightweight machine learning operations using locally-sourced data, further enriched by information from neighboring nodes. This approach enables real-time predictions about road conditions and optimizes path planning. Such advancements represent significant progress in the realms of distributed and edge computing.

Machine learning algorithms are typically characterized by a high degree of parameterizability, with their performance being sensitive to hyper-parameter configurations. For instance, the well-established Multi-Layer Perceptron (MLP) [GD98] displays significant variance in prediction accuracy with different hyper-parameter settings for the same task. These hyper-parameters, e.g., the number of layers, the number of neurons per layer, the type of activation functions, and the learning strategies, all require precise configuration before deploying a machine learning model in a real-world platform.

Hyper-parameter tuning is essential for optimal predictive performance. However, it can be resource-intensive, especially as data size or search space grows. Over recent decades, numerous hyper-parameter tuning algorithms have been developed and scrutinized. Modern techniques such as Model-Based Optimization (MBO) employ a cycle of model fitting and utilization to determine promising configurations to explore. A specific strategy within this realm is Bayesian optimization [JSW98], which addresses the challenge of expensive optimization by fitting a Gaussian process regression to approximate predictive performance relative to the hyper-parameters.

Typically, such hyper-parameter tuning necessitates a dedicated machine learning model to be trained and evaluated on centralized data to yield a performance estimate. However, the traditional centralized design for this process becomes less efficient and sub-optimal in cases where centralized data is not accessible, for instance, due to privacy concerns or in distributed settings.

In distributed setups, transferring data through low bandwidth connections and merging all sub-datasets to one central node can lead to significant communication resource consumption, large overheads, and reduced time for tuning. The central node may also be burdened by redundant data from overlapping sensing areas. Furthermore, data collection and storage may be hindered by privacy concerns or limited storage capacity at the central node. Additionally, the performance of machine learning algorithms is usually sensitive to the adopted hardware platforms. One study [KSZ17] showed that implementation details, frameworks, and programming languages as well as the related software libraries have a high impact on the run-time performance of unsupervised learning methods. Particularly, for run-time considerations, it has been shown that caching behavior impacts the performance of implemented algorithms even more than algorithmic differences [NK06]. For example, the run-time of a random forest in [BCC+18] is optimized for different platforms using different settings due to the different hardware designs, e.g., cache size. Consequently, if tuning aims at algorithm acceleration, a setting optimized for a central node may not be optimal for the dedicated distributed embedded systems with distinct hardware architectures.

An alternate solution is individual hyper-parameter tuning at each node using its local data. However, this approach brings its own set of challenges. Due to each node's limited storage and sensing area, only a limited size of data can be collected and stored. One main challenge of hyper-parameter tuning on distributed embedded systems is leveraging the decentralized sub-datasets to derive a unified hyper-parameter setting that is universally applicable across all nodes within the system. This necessitates a novel method targeting three key objectives: improving prediction accuracy, enhancing statistical stability, and boosting runtime efficiency.

### 1.1.4 Machine Learning for Error-Tolerant Embedded Systems

In addition to optimizing the deployment of machine learning models on embedded systems, machine learning algorithms are increasingly seen as a viable solution to challenges specific to embedded systems. For example, industrial automation often faces transient states due to ever-changing environments. In such situations, traditional static solutions may falter.

To illustrate, certain safety-critical embedded systems for industrial applications, are regularly subjected to transient faults due to environmental influences such as cosmic radiation and electromagnetic interference [Bau05]. The susceptibility to these factors is accentuated by the densely integrated modern systems-on-chips, resulting in non-negligible transient fault-rates. These transient faults might lead to soft errors with potential catastrophic outcomes, error-handling strategies must be incorporated during the design phase. Various software-based techniques, such as explicit output comparison (EOC) [GGB13], control flow checking using software signatures [OSM02], and redundant multithreading [CBC18], have been widely accepted due to their flexibility in balancing error protection with additional runtime.

In practical scenarios, certain safety-critical applications have been observed to tolerate a limited number of errors by temporarily downgrading the quality of service (QoS), without catastrophic consequences, as long as error tolerance constraints are satisfied. For instance, robotic applications can still successfully finish their tasks under a limited number of errors [CBC+16; YCC18] This observation has led to the notion of $(m, k)$ robustness constraints, where a task must complete at least $m$ correct jobs out of any $k$ consecutive jobs. While the original concept of $(m, k)$ constraints was formulated for limited deadline misses, it is equally applicable to define acceptable levels of soft errors.

The predominant methods employed to enforce $(m, k)$ constraints, highly rely on static decisions. These decisions involve using appropriate fault-tolerance techniques, such as the deeply red pattern (R-pattern) [KS95] or the evenly distributed pattern (E-pattern) [QH00], to ensure reliable job execution. To improve the adaptivity of static pattern based techniques, Chen et al. [CBC+16] proposed to track the current resilience during runtime and to adapt the patterns accordingly. This pattern-based scheduler delays the resource-intensive and time-costly reliable executions until the last moment by monitoring the number of future jobs that can be faulty without violating constraints. While this method introduces some adaptivity, it can result in pessimistic resource usage as it does not consider the actual error probability.

In the literature of fault-tolerant systems, a common objective is to minimize overall system utilization and, consequently, energy consumption [CBC+16; NQ06; NZ20]. This is due to the significant difference in power consumption between busy and idle processor states [CK07]. In the busy state, the consumed power can be divided into *static* and *dynamic* power consumption in contrast to the idle state, in which (ideally) only static power is consumed. Moreover, keeping the processor in a busy state for a sustained amount of time leads to a temperature increase, which in turn results in higher energy consumption for cooling and increases the static leakage power consumption, due to the super linear relationship between temperature and static leakage power [SLD+03; LDS+07]. Reducing the average energy consumption can directly be linked to reducing the average utilization.

However, in practical applications, the environment of computational environments constantly shifts due to factors like hardware conditions and environmental temperature. Current solutions that rely on static patterns [CBC+16; NQ06; NZ20] may find it difficult to adapt to such scenarios, as they lack the adaptability needed to optimize performance in the face of these unpredictable changes.

Although static execution patterns remain prevalent for ensuring $(m, k)$ constraints, the need for adaptivity has become more evident. This is especially pertinent when accounting for the dynamic nature of soft error probability during system execution. The overarching objective remains the minimization of system utilization, leading to optimal energy consumption. To navigate these challenges, it is necessary for a novel approach based on probabilistic state transitions, specifically for known error probabilities. Such a solution would utilize the deliberate deployment of reliable executions, which are often resource-intensive. Additionally, the potential integration of machine-learning methodologies, especially in the context of error probabilities that shift dynamically, has the makings of an advanced, adaptive, and streamlined solution. Ideally, this approach would have the essential capacity to discern and dynamically adjust to evolving environmental nuances.

## 1.2 Contribution of this Dissertation

This dissertation primarily focuses on analyzing and optimizing the timing-constrained embedded systems by considering resource synchronization and machine learning approaches. It also introduces an application of a machine learning approach, i.e., based on reinforcement learning, to tackle the challenges encountered in embedded systems. The primary domains investigated and contributed to in this dissertation include: a) the design, implementation, and formal verification of resource synchronization protocols intended to bound the worst-case response time of a set of tasks with relatively long critical sections in a non-work conserving manner, e.g., machine learning tasks with requests of GPUs; b) the enhancement of deploying machine learning models on resource-constrained distributed embedded systems by utilizing model-based optimization for hyper-parameter tuning; c) a novel application that employs reinforcement learning to minimize the average execution time of an embedded system while operating under $(m, k)$ soft error constraints. This section provides an overview of the challenges being addressed and the novel approaches proposed to tackle them. Furthermore, the contributions of this dissertation are explicitly delineated.

### 1.2.1 Multiprocessor Resource Synchronization

To address the challenges of the traditional work-conserving schedule for short critical sections and the unanswered questions associated with it, we begin with the fundamental setting of frame-based real-time task systems. In these systems, all tasks share identical periods and release their jobs simultaneously. These tasks are scheduled on $M$ homogeneous processors. Our model assumes that each critical section is non-nested and is protected by either a binary semaphore or a single mutex lock. In our work, *Dependency Graph Approach for Multiprocessor Real-Time Synchronization* [CBS+18], we propose a dependency graph approach (DGA) for multiprocessor synchronization. This approach consists of two steps:

- First, construct a dependency graph as a Directed Acyclic Graph (DAG) that determines the execution order of the critical sections guarded by a single binary semaphore.
- Next, apply multiprocessor partitioned, semi-partitioned, or global scheduling algorithms, ensuring they follow the execution order defined by the constructed dependency graph.

Because of the initial step, the resulting schedule may not always be work-conserving. This situation arises when a critical section is ready to be executed but one of its predecessors in the dependency graph is not finished yet. Building on this, we further refine our approach to accommodate more complex scenarios. In our work *Scheduling of Real-Time Tasks With Multiple Critical Sections in Multiprocessor Systems* [CSB+22], we extend DGA by allowing for tasks with several critical sections. In *Graph-Based Optimizations for Multiprocessor Nested Resource Sharing* [SUB+21], nested resource accesses are handled. Additionally, we extend our method beyond frame-based task systems to include standard periodic task systems with list scheduling in *Multiprocessor Synchronization of Periodic Real-Time Tasks Using Dependency Graphs* [SUB+19a], incorporating both partitioned scheduling paradigms in *Partitioned Scheduling for Dependency Graphs in Multiprocessor Real-Time Systems* [SUB+19b].

**Contribution 1: Dependency Graph Approach for Multiprocessor Resource Synchronization**

- For frame-based task systems with non-nested resource accesses:
  - We show that finding a schedule of the tasks to meet the given common deadline is $\mathcal{NP}$-hard in the strong sense *regardless of the number of processors $M$ in the system.* The $\mathcal{NP}$-hardness holds under any scheduling paradigm, showing that preemption or migration does not reduce the complexity.
  - When each task only contains one critical section, we prove a lower bound on the approximation ratio for minimizing makespan under different

scheduling paradigms, i.e., at least $2 - \frac{2}{M} + \frac{1}{M^2}$ under any scheduling paradigm and $2 - \frac{1}{M}$ under partitioned or semi-partitioned scheduling.

  - We demonstrate the adaptation of uni-processor non-preemptive scheduling to construct DAGs for tasks with only one critical section, resulting in polynomial-time algorithms with different approximation bounds, e.g., with an approximation ratio of $2 + \varepsilon - \frac{1+\varepsilon}{M}$ for any $\varepsilon > 0$ under semi-partitioned scheduling strategies.
  - When each task contains multiple critical sections, we find a correlation between the dependency graph in the DGA and the classical job shop scheduling problem. We present a polynomial-time reduction from the classical *job shop scheduling problem*, which is $\mathcal{NP}$-hard in the strong sense [LR79]. We establish approximation bounds for minimizing the makespan based on the approximation bounds of job-shop algorithms.

- We extend DGA to synchronize (multiple) nested resource accesses per task for frame-based real-time task systems by reducing the dependency graph construction to constraint programming, where the two fundamental problems, namely *deadlocks* and *transitive blocking chains*, are solved.
- We extend the DGA from frame-based task systems to periodic task systems by unrolling the jobs of each task and creating dependency graphs at the job level for periodic tasks. Consequently, jobs that access the same shared resource over a single hyper-period must follow the generated accessing order.
- To schedule a set of dependency graphs for the given task set on $M$ processors, we proposed two scheduling algorithms: a) LIST-EDF: which combines list scheduling with an earliest-deadline-first (EDF) heuristic; and b) Partitioned-EDF: this incorporates two partitioning algorithms based on federated scheduling and a worst-fit heuristic.

### 1.2.2 Implementation and Formal Verification of Resource Synchronization Protocols

While state-of-the-art resource synchronization protocols, like ROP and the newly proposed DGA, demonstrate advantages in timing guarantees by bounding tasks' worst-case response time, they often assume that implementation overheads are negligible. This assumption can be misleading, as the real-world performance of a protocol within an RTOS depends heavily on various settings such as the local or remote execution of critical sections, multiprocessor scheduling paradigms, and tasks' waiting semantics. This means that the performance of different protocols is closely tied to their implementation.

Furthermore, protocol design typically relies on operating system-level assumptions, such as abstracted notations of tasks, scheduling policies, and queues. Additionally, specific details of the operating system or hardware are often disregarded. These abstractions and assumptions may not always be valid when implementing a protocol, and any necessary adaptations due to constraints imposed by the OS may lead to discrepancies from the original specifications. Such discrepancies can result in unexpected consequences.

To address these complexities, we validate our implementation of the newly proposed DGA for multiprocessor resource synchronization on two RTOSes, namely LITMUS$^{RT}$ and RTEMS. Moreover, in our work *Formal Verification of Resource Synchronization Protocol Implementations: A Case Study in RTEMS* [SEC+22], we introduce a formal verification framework for synchronization protocols implemented in an RTOS, working under the assumption that the underlying functionalities from the RTOS, e.g., interrupt management and scheduling infrastructure, are correct.

**Contribution 2: Implementation and Formal Verification of Resource Synchronization Protocols**

- We enhance open-source development by scrutinizing the SMP support in LITMUS$^{RT}$ and RTEMS, identifying potential pitfalls in their implementation. Moreover, our detailed prototype implementation of the dependency graph approach in these RTOSes offers overheads comparable to existing multiprocessor synchronization protocols.
- For the verification of a protocol's implementation in targeted OSes, we define the responsibilities of involved primitives and operations, advocating for deductive verification to ensure the approach's consistency and integrity.
- We introduce a framework for formally verifying properties intrinsic to synchronization protocol implementations within an OS, bolstering system reliability and robustness.
- We present two case studies that verify ICPP and MrsP in official RTEMS, uncovering long-stayed mismatches and suggesting potential remedies. Additionally, we validate the DGA implementation in RTEMS to demonstrate its broad applicability.

### 1.2.3 Optimizing the Deployment of Machine Learning on Distributed Embedded Systems

Distributed embedded systems, encompassing edge computing, have become prominent platforms for running machine learning algorithms due to their high flexibility, mobility, scalability, and energy efficiency in real-world scenarios [BXF+18; GDO+12; LAB+11]. However, within such distributed contexts, the local tuning of hyperparameters of deployed machine learning models presents significant challenges.

One common challenge is that merging data from all nodes is often impractical and inefficient due to constraints such as low bandwidth connections or limited central node storage. Privacy concerns can further complicate or even render this approach impossible. Additionally, the presence of overlapping sensing areas among distributed nodes can result in redundant data, which adds further complexity on the central node.

A potential solution lies in allowing each node to independently conduct hyper-parameter tuning based on local data. However, due to the limited storage and detection area of each node, the tuning process in each node can be operated on only a fraction of the entire dataset collected in the area. Independent tuning of hyper-parameters using these local sub-datasets leads to inconsistent performance of machine learning algorithms due to the restricted size of the training data.

To tackle these prevalent issues of hyper-parameter tuning in distributed embedded systems, we present *MODES* in our work *MODES: model-based optimization on distributed embedded systems* [SBR+21]. This strategy emphasizes the utilization of decentralized sub-datasets to derive a cohesive hyper-parameter configuration for the whole system. With *MODES*, hyper-parameters are tuned both **locally** and **efficiently**. Each node is viewed as a distinct black box, running an individual model based on its data. Meanwhile, the entire distributed system is envisioned as a more extensive black box, with the objective to enhance its overall performance, considering aspects like prediction accuracy, stability, and operational efficiency.

**Contribution 3: Model-based Optimization on Distributed Embedded Systems**

- We introduce the *MODES* framework, designed to apply Model-Based Optimization (MBO) to resource-constrained distributed embedded systems. The frame work not only enhances the tuning process by identifying optimal hyper-parameters efficiently, but also bolsters the generalizability of the resulting hyper-parameter configuration. In addition, *MODES* significantly mitigates data communication costs by transmitting only hyper-parameter settings and performance values such as prediction accuracy.
- Further, we delineate *MODES* into two distinct optimization modes: a) In the Black-box mode (*MODES*-B), the entire system is treated as a single black box. It jointly optimizes the hyper-parameters of individual models, taking into account the specific weights assigned to different nodes. b) Conversely, the Individual mode (*MODES*-I) considers all models as copies of the same black box, allowing for efficient parallel optimization in a distributed setting. As a flexible framework, *MODES* can be tailored to a broad spectrum of applications with minimal adjustments and switches between modes.

### 1.2.4 Reinforcement Learning for Average Task Execution Time Minimization under $(m, k)$ Soft Error Constraint

In safety-critical systems, a prevalent strategy to moderate resource demands and mitigate the impact of errors is the implementation of $(m, k)$-constraints. These constraints stipulate that at least $m$ out of any $k$ consecutive jobs must be error-free. Given the $(m, k)$ soft error constraint and the inherent uncertainty of soft error probabilities, the development of an error-tolerant scheduler becomes essential. This scheduler should regard each sequence of $k$ consecutive jobs for a task as a state. State transitions occur by incorporating the correctness of a new job while displacing the oldest one in the sequence. As execution errors occur probabilistically, the state transitions from any given state are inherently stochastic. To effectively manage these transitions, the scheduler should be modeled on a Markov Chain, incorporating state transition probabilities that reflect the error rates. Utilizing either explicit knowledge or estimates of these soft error probabilities, the scheduler aims to ensure $(m, k)$-compliance for each task, while adhering to real-time constraints. By focusing on minimizing the expected average execution time for tasks, the scheduler indirectly contributes to reducing the overall expected system utilization and energy consumption.

When each task's error probability remains constant at runtime, we propose an optimal method based on Markov Chain optimization. However, in scenarios with dynamic or unknown error probabilities, traditional static methods may be inefficient or inadequate. In contrast, reinforcement learning (RL) provides an adaptive solution. The inherent strength of RL lies in its capacity to iteratively learn and update strategies during runtime, which makes it especially suitable for managing dynamic error probabilities and variable system conditions. Therefore, where conventional static approaches might exhibit limitations, RL offers a responsive and robust alternative by continuously adapting to the prevailing environment.

In our work *Average Task Execution Time Minimization under* $(m, k)$ *Soft Error Constraint* [SUC+23], we assume that each task has three distinct execution modes: reliable, detected, and unreliable, each with varying execution times and correctness assumptions. We introduce a reinforcement-based approach, an adaptive algorithm that leverages either explicit knowledge or estimations of soft error probabilities to assure $(m, k)$-compliance for each task under real-time constraints. By actively selecting the execution mode for the next job, considering the current state with respect to the corresponding $(m, k)$ constraint and estimated error rate, this approach aims to minimize the expected execution time for each task. This minimization leads to a reduction in the anticipated overall system utilization and, consequently, the energy consumption of the system.

**Contribution 4: Reinforcement Learning for Average Task Execution Time Minimization under $(m, k)$ Soft Error Constraint**

- Our primary contribution lies in the utilization of finite automata for the enforcement of $(m, k)$ constraints. By formulating all $(m, k)$ compliant states of a task as a minimal automaton, we ensure transitions occur only between compliant states. In conjunction with the soft error probability and a stochastic transition system, we derive a Markov chain model.
- Assuming that a stationary error-probability can be precisely estimated, we develop an optimization algorithm using the Markov chain model mentioned above. This algorithm determines the stochastic parameters for the job selection strategy, aiming to minimize the expected execution time, enhancing efficiency.
- Additionally, we introduce a reinforcement learning (RL)-based strategy for job mode selection when soft error probabilities are not known. The RL method involves the formulation of a task's execution information into an RL-recognizable environment, and we explore the intricacies of the barrier function and learning policy to ensure effectiveness and practicality.

## 1.3 Organization of the Dissertation

This dissertation is structured as follows:

- Chapter 2 introduces the system model and experimental platforms utilized throughout the study.
- Chapter 3 explores the background and related work, illuminating the current state-of-the-art of the topics under investigation.
- Chapter 4 details our first contribution, the design of the dependency graph approach for multiprocessor real-time synchronization. It commences with basic scenarios, such as frame-based task systems with singular, non-nested resource requests. The chapter then explores multiple critical sections per task and nested resource accesses. Furthermore, extensions for standard periodic task systems are also elucidated. The chapter also discusses several algorithms to schedule the generated dependency graphs, augmented by illustrative examples.
- Chapter 5 details our second contribution: the implementation and formal verification of resource synchronization protocols on RTOSes. It begins with a deep dive into the DGA's implementation on both LITMUS$^{\mathrm{RT}}$ and RTEMS. This is followed by the introduction of a formal verification framework for implemented protocols on RTOSes, accompanied by three case studies for the implemented ICPP, MrsP, and DGA on RTEMS.
- Chapter 6 unveils our third contribution: optimizing the deployment of machine learning models on distributed embedded systems. We introduce *MODES*, a novel technique employing MBO to refine the hyper-parameter settings of such models. This chapter presents the two modes of *MODES*: *MODES*-B

optimizes prediction accuracy and statistical stability, and *MODES*-I focuses on hyper-parameter tuning efficiency.

- Chapter 7 demonstrates our fourth contribution, applying machine learning to address embedded system challenges. We transform the task execution time minimization problem under the $(m, k)$ constraints into a Markov chain optimization problem. A reinforcement learning-based approach is then presented, offering an adaptive method for selecting task execution modes based on error rate estimations.
- Lastly, Chapter 8 summarizes key results and current limitations in this dissertation, and discusses opportunities for potential future research.

## 1.4 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. The following overview lists the contribution on the results presented in the individual chapters:

- Chapter 4 explores the dependency graph approach for multiprocessor resource synchronization.
  - In the foundational work on resource synchronization for frame-based task systems with a single critical section per task, presented at RTSS 2018 [CBS+18], Prof. Dr. Jian-Jia Chen contributed the initial design, formulation of theorems, and proofs. As a co-author, I contributed to refining the design and conducted the implementation and evaluation.
  - The extended study, which addresses tasks with multiple critical sections, was published in the IEEE Transactions on Computers 2022 [CSB+22]. I collaborated with my supervisor, Prof. Dr. Jian-Jia Chen. While he conceptualized the main idea and provided guidance on the formulation of theorems and proofs, I was deeply involved in the protocol design and managed both the implementation and evaluation.
  - The further extension, focusing on nested resource accesses, was published at RTCSA 2021 [SUB+21]. I was the principal author, leading the development of concepts, theorems, and evaluations.
  - The work that introduced extensions for periodic task systems, complemented by the LIST-EDF scheduling algorithm, was published at RTAS 2019 [SUB+19a]. I was the principal author, contributing to the design, theorems, and evaluations.
  - The investigation into partitioned scheduling algorithms was published at RTCSA 2019 [SUB+19b]. I was the principal author, leading the design of concepts, algorithms, and evaluations.

- Chapter 5 details the DGA implementation on LITMUS$^{RT}$ and RTEMS, and presents the formal verification framework. Various DGA implementations on LITMUS$^{RT}$ were published in RTSS 2018, RTAS 2019, and IEEE Transactions on Computers 2022 [CBS+18; SUB+19a; CSB+22]. In these publications, I was either co-author or principal author, being responsible for all aspects of implementation. The initial DGA implementation on RTEMS originated from a student thesis by Jan Duy Thien Pham, based on a preceding version of RTEMS. I updated and adapted it to be compatible with the latest RTEMS 5.1 release with my student Surya Subramanian. The formal verification framework was published in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 2022 [SEC+22]. This framework began as a Master's thesis under my guidance by Christoph-Cordt von Egidy. We later refined and expanded this work into a comprehensive publication. Together with my student, I laid out the foundation for the verification framework's design. Christoph-Cordt von Egidy handled the case studies for ICPP and MrsP. Subsequently, I conducted the formal verification for DGA on RTEMS with my student Surya Subramanian.
- Chapter 6 presents *MODES*, a framework developed for hyper-parameter tuning in distributed embedded systems. This framework was published at ECML 2021 [SBR+21]. As the principal author, I was responsible for the design, implementation, and evaluation of the framework. My collaborator, Jiang Bian, assisted with the conceptual design and supplied the machine learning models and datasets for our evaluations.
- Chapter 7 addresses the challenge of minimizing average execution time while satisfying the $(m, k)$ soft error constraints. This work was published at RTAS 2023 [SUC+23]. As the principal author, I developed both the optimal strategy for fixed and static error probabilities and a reinforcement learning-based method for addressing unknown or unstable error probabilities. Furthermore, I conducted the complete implementation and evaluation of both methodologies. Niklas Ueter was primarily responsible for formalizing the studied problem, establishing the terminology, and delivering proofs for all theorems. These components are included in this dissertation to ensure its completeness.

Chapter 2

# System Model and Experimental Platforms

## Contents

This chapter delineates the foundational concepts and practical tools pivotal to this dissertation, encompassing the system model, task models, notations, and experimental platforms. First, we introduce the system model that forms the basis of the research conducted in this dissertation. We present the task models, highlighting time-critical tasks pertinent to resource synchronization scenarios, directed acyclic graph (DAG) task models, and safety-critical tasks governed by the constraints of the $(m, k)$ soft error paradigm. Subsequently, we explore the foundational theories of scheduling, review the common algorithms that are frequently applied in both real-time and embedded systems, and introduce two salient performance metrics for the scheduling algorithms under consideration. Moreover, we discuss the fault and error model for safety critical systems, which is crucial for understanding the mechanisms of fault tolerance in the studied systems. Finally, we detail the experimental platforms and tools that have been adopted for the evaluation of the concepts and methods presented throughout the dissertation. These platforms and tools facilitate both the empirical validation of our methodologies and the formal verification of the system implementations.

## 2.1 System Model

In this dissertation, we start by considering the system model of single-device embedded systems with timing requirements. The single device embedded system may consist of either a uni-processor or homogeneous multiprocessors, along with additional shared resources, such as Graphics Processing Units (GPUs). We assume that each device has limited storage and can only store a certain amount of data.

Several identical single device embedded systems can also be connected, either through wired or wireless network connections. These connected devices contribute to a distributed embedded system, also referred to as a *cluster*. This broader structure emphasizes a distributed environment suitable for deploying machine learning models.

In the distributed scenario, each device is treated as a node. We assume that data collected by different nodes are (at least partially) distinct and can be viewed as subsets of a complete dataset. Additionally, connections among nodes are constrained by low bandwidth, permitting only the transfer of a tiny amount of data such as hyper-parameter settings and performance results (e.g., accuracy of prediction).

It is important to note that each task can only be deployed on one of the devices, and migrations among devices are not allowed. This constraint shapes the scheduling and coordination strategies explored within the context of the dissertation.

In each device, we consider a set $\mathbf{T}$ of $N$ recurrent tasks, i.e., $\mathbf{T} = \{\tau_1, \ldots, \tau_N\}$, to be scheduled on $M$ homogeneous, i.e., identical, processors from one physical embedded device, where $M, N \geq 1$. Due to the different studied scenarios, tasks inside $\mathbf{T}$ are defined separately in the following subsections.

### 2.1.1 Tasks with Resource Synchronization

Under the von-Neumann programming model, shared resources, e.g., shared files, data structures, and memory cells, require mutually exclusive accesses to prevent race conditions. A protected code segment that has to access a shared resource is called a *critical section*. The mutually exclusive executions of critical sections have to be protected by applying synchronization (*binary semaphores*) or locking (*mutex locks*) mechanisms. In the resource synchronization scenario, we assume all tasks can have multiple critical sections and may access several of the $Z$ shared resources, i.e., $\mathbf{Z} = \{z_1, z_2, \ldots, z_Z\}$. For brevity in notation, the resource id can sometimes be represented by $z$ directly. For example, resource $z_1$ can also be denoted as resource 1. Each task $\tau_i$ has $\eta_i$ *computation segments*, in which each computational segment can be either a critical section or a non-critical section. Each task is described by $\tau_i = (\mathbf{\Theta}_i, C_i, T_i, D_i)$, where:

- $\mathbf{\Theta}_i$ is the set of all the computation segments in task $\tau_i$.
- $C_i$ is the total worst-case execution time (WCET) of task $\tau_i$, i.e., including all the computation segments in the task.

- $T_i$ is the period of $\tau_i$. That is, if an instance of $\tau_i$ is released at time $t$, the subsequent instance is released exactly at time $t + T_i$.
- $D_i$ is the relative deadline of $\tau_i$, i.e., a job of $\tau_i$ released at time $t$ must finish its execution no later than its absolute deadline $t + D_i$. We consider constrained-deadline task systems, i.e., $\forall \tau_i \in \mathbf{T},\ D_i \leq T_i$.

The number of computation segments in a task is denoted by $\eta_i$, where $\eta_i = |\mathbf{\Theta}_i|$. For the $j$-th segment of task $\tau_i$, denoted by $\theta_{i,j} = (C_{i,j}, \lambda_{i,j}, \sigma_{i,j})$:

- $C_{i,j} > 0$ is the WCET of computation segment $\theta_{i,j}$ with $C_i = \sum_{j=1}^{\eta_i} C_{i,j}$.
- $\lambda_{i,j}$ indicates whether the corresponding segment is either a non-critical section or a critical section. If $\theta_{i,j}$ is a critical section, $\lambda_{i,j}$ is 1; otherwise, $\lambda_{i,j}$ is 0.
- If $\theta_{i,j}$ is a non-critical section, then both $\theta_{i,j-1}$ and $\theta_{i,j+1}$ must be critical sections (if they exist). In other words, $\theta_{i,j}$ and $\theta_{i,j+1}$ cannot be both non-critical sections.
- If $\theta_{i,j}$ is a critical section, it starts with the lock of a mutex lock (or *wait* for a binary semaphore), denoted by $\sigma_{i,j}$, and ends with the unlock of the same mutex lock (or *signal* to the same binary semaphore). If a critical section requests nested shared resources, $\sigma_{i,j}$ represents the set of requested resources.

Furthermore, we make the following assumptions:

- A periodic task periodically releases an infinite number of instances (also called jobs). Each task $\tau_i$ releases its first job at time $\phi_i$ and releases the subsequent jobs *strictly periodically* with a given period $T_i$. We assume that $\phi_i = 0$ for every $\tau_i \in \mathbf{T}$, i.e., all the tasks in $\mathbf{T}$ release their first jobs at time 0. We denote the $\ell$-th job of $\tau_i$ as $J_i^\ell$, and the $\ell$-th sub-job of $\theta_{i,j}$ as $J_{i,j}^\ell$.
- A job cannot be executed in parallel, i.e., the sub-jobs in a job must be sequentially executed.
- The execution of the critical sections guarded by a mutex lock (or one binary semaphore) must be sequentially executed. Thus, if two computation segments share the same mutex lock, they must be executed one after another.
- There are a total of $Z$ shared resources protected by mutex locks or binary semaphores.

Utilization is commonly used to analyze the workload and computational demand of both individual tasks and the entire system. The utilization of a task $\tau_i$ is defined as $U_i = \frac{C_i}{T_i}$. Similarly, the utilization of a specific computational segment $\theta_{i,j}$ within task $\tau_i$ is defined as $U_{i,j} = \frac{C_{i,j}}{T_i}$. The total utilization of the entire task set is represented by $\sum_{i=1}^{N} U_i$.

In this dissertation, we explore three primary scenarios related to the resource access pattern within each task. Specifically, we consider:

1. The **OCS** task model, where each task has exactly one critical section with non-nested resource access.

2. The **MCS** task model, where each task can have multiple critical sections with non-nested resource accesses.
3. The **Nested-MCS** task model, where each task can have multiple critical sections with nested resource accesses.

In the following, we discuss each of these task models in detail. When each task contains only one non-nested critical section, we define the **OCS** task model:

**Definition 1. The OCS task model:** Each task comprises three computational segments. The middle segment represents a non-nested critical section and is safeguarded by a binary semaphore.

Figure 2.1a depicts an example of the OCS task, where $\theta_{i,2}$ represents the critical section that accesses the shared resource $z_1$.

For tasks that might have multiple non-nested critical sections, we define the **MCS** task model:

**Definition 2. The MCS task model:** Each task consists of several computational segments, with at least one being a non-nested critical section. Each of these critical sections is individually protected by a binary semaphore.

Figure 2.1b illustrates an example of the MCS task with five computational segments. Of these segments, two are critical sections that independently access shared resources $z_1$ and $z_2$.

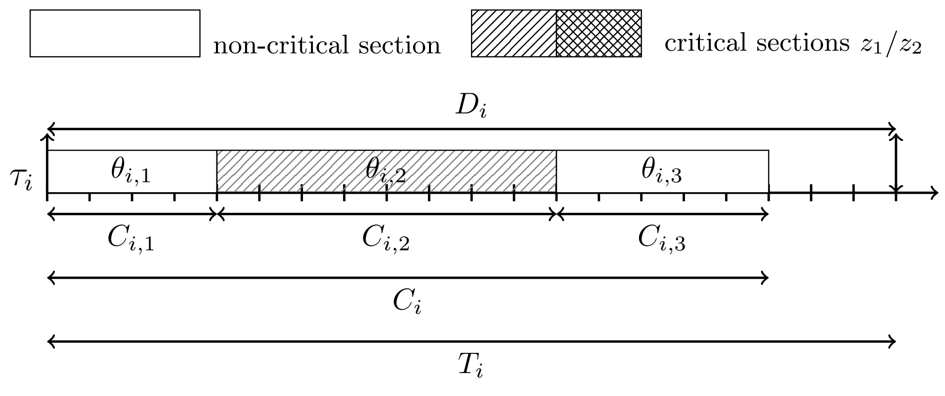
We extend the **MCS** task to the nested version, i.e., **Nested-MCS** task. This model permits nested resource sharing within each critical section, thereby facilitating the representation of more complex synchronization patterns within the task structure.

**Definition 3. The Nested-MCS task model:** Each task comprises multiple computational segments, with at least one being a nested critical section requesting multiple shared resources. Every such nested critical section is simultaneously protected by all its associated binary semaphores.

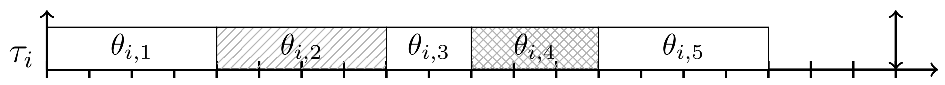
In this dissertation, we mainly consider the *all-at-once* nested locking pattern:

**Definition 4. All-at-once Locking:** A critical section is described as being locked *all-at-once* if it can only be accessed after all its requested mutex locks are successfully granted simultaneously. If not, a job remains blocked until granted access to all the requested resources. Upon completion of a critical section, all its associated mutex locks are released.

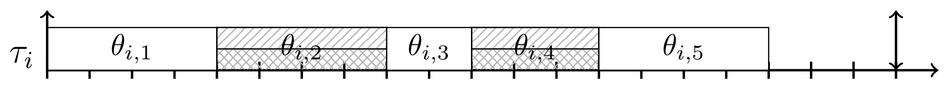
Please note that any nested locking can be transferred to a *all-at-once* locking by granting all the requested shared resources simultaneously before the execution of the critical section. An example is shown in Figure 2.1c, where each critical section concurrently lock shared resources $z_1$ and $z_2$.



**(a)** An example for OCS task model.



**(b)** An example for MCS task model.



**(c)** An example for Nested MCS task model with all-at-once locking.

**Figure 2.1:** Examples for different task models.

For all task models illustrated in Figure 2.1, two consecutive non-critical sections are not allowed, but two consecutive critical sections are permitted. Therefore, the WCET for any non-critical section may be 0.

We consider two types of task systems, namely:

- **Frame-based** task systems: All tasks release their jobs at the same time and have the same period and relative deadline, i.e., $\forall i, j;\ T_i = T_j \wedge D_i = D_j$. Hence, the analysis can be restricted to one job of each task.
- **Periodic** task systems (with synchronous release): All tasks release their first job at time 0. While subsequent jobs are released periodically, different tasks may have different periods and relative deadlines. The **hyper-period** of the task set $\mathbf{T}$ is defined as the least common multiple (LCM) of the periods of the tasks in $\mathbf{T}$.

### 2.1.2 Directed Acyclic Graph Task Model

Due to the dependencies and execution order for some computational segments, their corresponding jobs can be formulated as directed acyclic graphs (DAGs). In this section, we introduce the definition of DAG tasks, explore some of their essential properties, and explain how they are utilized within the context of this dissertation.

A DAG is represented by $G = (V, E)$, where:

- $V$ is the set of vertices or nodes in the graph. Each vertex represents a distinct task or job within the system.
- $E$ is the set of directed edges that connect the vertices. Each edge $(u, v)$ represents a dependency, meaning that the task corresponding to vertex $u$ must be finished before the task corresponding to vertex $v$ can start.

The vertices and edges together define the structure of the DAG, capturing the dependencies and execution order of tasks or jobs within the system.

When integrated with the task model in Section 2.1.1, a sub-job, i.e., a critical or a non-critical section, is a vertex in $V$. The edges in $E$ describe the precedents constraints and execution sequence of these jobs.

The resulting DAG task has the period that equals to the *hyper-period* of all the involved (sub-)jobs. Each node(sub-job) can have its own deadline according to the corresponding task and DAG structure. Further details can be found in Section 4.7.

Each DAG task has the following properties:

**Definition 5. Volume:** The volume of the DAG $G$ represents the graph's total execution time, defined as $vol(G) = \sum_{v_i \in V} len(v_i)$.

**Definition 6. Path:** In a directed-acyclic graph $G$, a path $\Delta$ is a sequence of sub-jobs $v_{i_1} \prec v_{i_2} \prec \ldots \prec v_{i_k}$ for $v_{i_j} \in V$ such that each sub-job in the sequence is an immediate successor of the previous sub-job based on precedence constraints, where $pre(v_{i_1}) = \emptyset$ and $suc(v_{i_k}) = \emptyset$.

**Definition 7. Length:** The length of a path is determined by $len(\Delta) = \sum_{v_i \in \Delta} len(v_i)$ with the length of a sub-job representing its execution time.

Based on the definition of a path,we can more formally define the critical path as the longest path in a DAG, as presented in the following definition:

**Definition 8.** In a task dependency graph $G$, the **critical path** represents one of its longest paths. The critical path length of $G$ is denoted by $len(G*) = \max\{len(\Delta) \mid \Delta\ \textit{is a path in } G\}$. The length of critical path remains independent of the number of utilized processors, i.e., $M$.

### 2.1.3 Tasks with $(m, k)$ Robustness Constraints

In fault-tolerant systems, applications can be functionally correct as long as tasks follow the specified $(m, k)$ robustness constraints.Specifically, a task must ensure that at least $m$ jobs out of any $k$ consecutive jobs are executed correctly. To minimize the average system utilization, each task can operate in one of three execution modes, each characterized by distinct WCETs and expected levels of correctness. We model each task $\tau_i$ as a tuple $(\mathbf{C}_i, D_i, T_i, m_i, k_i)$ where:

- $\mathbf{C}_i = \{C_i^u, C_i^d, C_i^r\}$ is a set of WCETs representing the different WCET demands corresponding to the task's execution modes: *unreliable* mode, *detected* mode, and *reliable* mode. Throughout this work, we assume that $C_i^u < C_i^d < C_i^r$ holds due to the additional overheads for the *detected* and *reliable* modes.
- $D_i$ is the relative deadline of $\tau_i$.
- $T_i$ is the period of $\tau_i$.
- Each task is subject to an $(m_i, k_i)$ soft error constraint, i.e., a task $\tau_i$ is required to have at least $m_i$ jobs out of any $k_i$ consecutive jobs to be correctly executed, where $0 < m_i \leq k_i$.

For this scenario, we assume sporadic task systems, where the release interval between two jobs from task $\tau_i$ is at least $T_i$. At each release, task can decide the execution mode of the next job. With regard to the varied execution modes, we assume that software-based fault tolerance techniques are employed to detect and recover fault-induced soft errors. Within this system, tasks are permitted to instantiate jobs in one of three primary modes: *reliable*, *detected*, or *unreliable*. Additionally, a composite mode, termed *detected + reliable*, acts as an immediate compensation within the same release window. For instance, a detected mode instance can be promptly followed by a reliable execution mode instance. This approach allows for immediate reaction to potential errors, ensuring a higher degree of reliability and responsiveness in the system with respect to the given $(m, k)$ constraint. The distinctive execution modes in our system have specific implications in addressing soft errors and their associated overheads:

- *Unreliable* Mode: In this mode, no additional implementation effort is required, keeping overheads low. However, the task is not able to determine whether an error has occurred during the execution of the job. To maintain compliance with the $(m, k)$ constraints, the system must, by default, assume that a soft error has occurred, even though it may not actually have.
- *Detected* Mode: This mode involves applying specific techniques to verify the correctness of the executed job. Examples of such techniques include error detection through special encoding of data or control flow checking. Unlike the unreliable mode, the detected mode allows the system to observe whether an error has occurred during execution. However, no further action is applied if an error is detected.
- *Reliable* Mode: Within this mode, the system must ensure that no soft error manifests, necessitating both the detection of errors and subsequent recovery. Several redundant copies of the job may be executed in parallel to ensure high reliability of the final result. This process adds complexity and leads

to a longer execution time compared to the other modes, but ensures the correctness of the output.

Furthermore, it is important to note that the system under consideration does not allow job skipping as a method to potentially enhance the Quality of Service (QoS). This restriction further emphasizes the need for careful consideration of the selected execution mode, balancing reliability and execution times to meet the specific demands of the application and operating environment.

We assume the use of an arbitrary preemptive scheduling algorithm, which schedules the task set $\mathbf{T}$ and guarantees the temporal requirements such as strict deadline compliance in case of hard real-time systems. The worst-case job mode sequence that can be generated by our approach is identical to the R-Pattern in which the first $k - m$ instances are executed in *detected* mode and the remaining $m$ instances are successively executed in the *reliable* mode. Therefore, any hard real-time schedulability analyses adopting the R-Pattern can be used. Please note, the fault-tolerant systems with $(m, k)$ robustness constraints are not strictly limited to hard real-time task systems. Our primary focus is generating guaranteed $(m, k)$-compliant schedules for each task, aiming to reduce the expected execution time.

## 2.2 Real-Time Scheduling

In this section, we begin by introducing the foundational principles of scheduling theory, offering a structured classification of various scheduling challenges. Subsequently, we detail several scheduling algorithms that are widely applied in the domain of real-time systems. To conclude, we introduce two prevalent metrics used to evaluate the performance of these scheduling algorithms.

Please note that the scheduling theories discussed in this section are only applied to the resource synchronization scenario, as discussed in Chapter 4 and Chapter 5. The detailed scheduling algorithms for fault-tolerant systems with $(m, k)$ robustness constraints are out of the scope of this dissertation.

### 2.2.1 Classifications of Scheduling Problems

In scheduling theory, a scheduling problem is described by a triplet $\text{Field}_1|\text{Field}_2|\text{Field}_3$.

- $\text{Field}_1$: describes the machine environment and contains exactly one entry. The commonly considered machine environments are as follows:
  - 1: uni-processor system
  - $P$: homogeneous multiprocessor system, where each job can be executed on any of the available processors.
  - $F$: flow shop, where each job has to be executed on each one of the available processors and all jobs have to follow the same route.
  - $J$: job shop, where each job has its own predetermined route to follow.

- Field$_2$: specifies the processing characteristics and constraints, that may include multiple entries. In this dissertation, we consider the following characteristics and constraints:
  - $r_j$: release time constraint, indicating that the job cannot starts its execution before its release time $r$.
  - $q_j$: delivery time constraint, indicating that a job requires an amount of time $q_j$ to deliver the result (final product) to the customer after finishing its execution on a machine.
  - *prmp*: preemptive execution, which implies that the scheduler is allowed to interrupt the execution of a job at any point in time and put a different job on the machine instead.
  - *prec*: precedence constraints, which require that one or more (sub-)jobs may have to be completed before another (sub-)job is allowed to start.
- Field$_3$: presents the objective to be optimized. Examples of possible objective functions to be minimized are as follows:
  - $C_{max}$: the makespan, defined as the total length of time required to complete a set of jobs from the start of the first job to the completion of the last one. A minimum makespan usually indicates efficient scheduling and effective resource utilization.
  - $L_{max}$: the maximum lateness measures the worst violation of the deadlines. This measure provides insight into the worst-case performance regarding deadline adherence.

For example, the scheduling problem $1|r_j|L_{\max}$ deals with a uni-processor system, in which the input is a set of jobs with different release times and different absolute deadlines, and the objective is to derive a non-preemptive schedule that minimizes the maximum lateness. As another example, the scheduling problem $P||C_{\max}$ deals with a homogeneous multiprocessor system, where the input consists of a set of jobs with identical release times. The objective here is to derive a *partitioned* schedule that minimizes the makespan. The scheduling problem $P|prec|C_{\max}$ is an extension of $P||C_{\max}$ by further considering the precedence constraints of the jobs. The scheduling problem $P|prec, prmp|C_{\max}$ further allows preemption.

Note that in classical scheduling theory, preemption in parallel machines implies the possibility of job migration from one machine to another. However, this is not necessarily the case in real-time systems. For instance, under preemptive partitioned scheduling, a job can be preempted and resumed later on the same processor without migration. Therefore, the scheduling problem $P|prec, prmp|C_{\max}$ allows job preemption and migration, i.e., preemptive global scheduling.

Please note that the machine environments, processing characteristics, constraints, and objectives in these three fields are not exhaustively listed in the above examples. We only present those scenarios that are studied in this dissertation.

### 2.2.2 Scheduling Algorithms

To schedule real-time tasks on multiprocessor platforms, there have been three widely adopted paradigms:

- The **partitioned** scheduling approach statically assigns tasks to specific processors. In this paradigm, each task is bound to a specific processor and consistently executes on it. This can lead to optimized processor-specific execution but might suffer from potential imbalance across processors.
- The **global** scheduling approach allows a task to migrate from one processor to another at any time. While this offers greater flexibility and potential for improved load balancing across processors, it may introduce overhead from task migration and complicate task synchronization.
- The **semi-partitioned** scheduling approach is a hybrid, drawing elements from both the partitioned and global strategies. It determines whether to statically divide a task into sub-tasks and how to assign each task or sub-task to processors. This approach can adapt to varying system conditions, potentially offering a balance between the predictability of partitioned scheduling and the flexibility of global scheduling.

A more comprehensive survey of multiprocessor scheduling in real-time systems can be found in [DB11].

#### Schedule in the Sub-job's Perspective

As defined in Section 2.1.1, the set of computational segments is defined as $\mathbf{\Theta} = \{\theta_{i,j} \mid \tau_i \in \mathbf{T}, j = 1, 2, \ldots, \eta_i\}$. A schedule for $\mathbf{T}$ is a function $\rho : \mathbb{R} \times M \to \mathbf{\Theta} \cup \{\bot\}$, where $\rho(t, m) = \theta_{i,j}$ denotes that the sub-job of $\theta_{i,j}$ is executed at time $t$ on processor $m$, and $\rho(t, m) = \bot$ denotes that processor $m$ is idle at time $t$. Since a job has to be sequentially executed, at any time point $t \geq 0$, only a sub-job of $\tau_i$ can be executed on one of the $M$ processors, i.e., if $\rho(t, m)$ is $\theta_{i,j}$, then $\rho(t, m') \neq \theta_{i,k}$ for any $j, k \leq \eta_i$ and $m' \neq m$. Moreover, since the sub-jobs of a job must be executed sequentially, $\theta_{i,k}$ cannot be executed before $\theta_{i,j}$ finishes for any $j < k \leq \eta_i$, i.e., if $\rho(t, m)$ is $\theta_{i,j}$ for some $t, m, i, j$, then $\rho(t', m) \neq \theta_{i,k}$ for any $t' \leq t$ and any $k > j$. Critical sections under the protection of a singular mutex lock must be sequentially executed. That is, if $\lambda_{i,j}$ is 1, $\lambda_{k,\ell}$ is 1, and $\sigma_{i,j} = \sigma_{k,\ell}$ then a schedule must guarantee $\rho(t, m') \neq \theta_{k,\ell}$ for any $t \geq 0$ and $m \neq m'$ when $\rho(t, m)$ is $\theta_{i,j}$.

In the resource synchronization scenario under consideration, we primarily focus on either *frame-based* task systems or *periodic* task systems. The schedule established in one *hyper-period* is consistently replicated in subsequent hyper-periods. For clarity and succinctness in our exposition, we initiate the discussion with *frame-based* task systems. Within a single frame, we only consider schedules that can meet the execution demand of all the computation segments. Let $R$ be the finishing time of the schedule. In this case, $\sum_{m=1}^{M} \int_0^R [\rho(t, m) = \theta_{i,j}] dt$ must be equal to $C_{i,j}$, where $[P]$ is the Iverson bracket, i.e., $[P]$ is 1 when the condition $P$ holds, otherwise $[P]$ is

0. Note that the integration is used in this dissertation only as a symbolic notation to represent the summation over time. The earliest moment when all sub-jobs finish their computation segments in the schedule (under all the constraints defined above) is called the *makespan* of the schedule, commonly denoted as $C_{\max}$ in scheduling theory, i.e., $C_{\max}$ of schedule $\rho$ is:

$$\text{min. } R \quad \text{s. t. } \sum_{m=1}^{M} \int_{0}^{R} [\rho(t,m) = \theta_{i,j}] dt = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta}$$

From the sub-job's perspective, a schedule is *non-preemptive* if a sub-job cannot be preempted. That is, there is only one interval where $\rho(t,m) = \theta_{i,j}$ occurs on a single processor $m$ for every sub-job $\theta_{i,j}$ in $\mathbf{\Theta}$. From the sub-job's perspective, a schedule is *preemptive* if a sub-job can be preempted, i.e., multiple intervals with $\rho(t,m) = \theta_{i,j}$ for any task $\theta_{i,j}$ in $\mathbf{\Theta}$ on processor $m$ are permissible. A critical section $\theta_{i,j}$ in a preemptive schedule can be preempted by non-critical sections or other critical sections that are not protected by mutex lock $\sigma_{i,j}$.

In a *partitioned* schedule, every sub-job of a given job is executed on a single processor, meaning there exists a processor $m$ where $\rho(t,m) = \theta_{i,j}$ for $t \geq 0$ and $j = 1, 2, \ldots, \eta_i$ for every task $\tau_i$ in $\mathbf{T}$. Under a *global* schedule, the execution of a sub-job on any of the $M$ processors can occur at an arbitrary time. That is, it is possible that $\rho(t,m) = \theta_{i,j}$ and $\rho(t',m') = \theta_{i,j}$ for $m \neq m'$ and $t \neq t'$. In a *semi-partitioned* schedule, a sub-job has to be executed only on one processor.

Both partitioned and semi-partitioned schedules can be preemptive or non-preemptive from the sub-job's perspective. By the aforementioned definition, a global schedule is always a preemptive schedule from the sub-job's perspective.

For a given schedule, we define:

**Definition 9. Feasible Schedule:** A schedule is considered feasible if it meets all previously specified non-overlapping constraints and ensures that no task misses its provided deadline (when a deadline is specified).

We assume that each computation segment or sub-task executes for its exact WCET during all releases, where early completion is prohibited. As a result, the generated schedule for one hyper-period remains static and is repeated periodically To this end, an exact schedulability test involves evaluating a specific schedule using the algorithms detailed in Section 4.7, spanning one hyper-period, to ascertain if any deadlines are missed. Since the schedule is static and repeated periodically, there is no dynamics that can lead to the multiprocessor anomalies pointed out by Graham [Gra69].

Please note that the schedule definition presented in this section pertains exclusively to Chapters 4, 5, and 6. Chapter 7, which explores the application of machine learning to tackle challenges in embedded systems, operates under different scheduling assumptions that are detailed in Section 2.1.3.

### 2.2.3 Performance Metrics for Scheduling Algorithms

In scheduling algorithms, the efficacy of an algorithm is determined both by its ability to produce a feasible schedule and by how closely this schedule approaches the theoretical optimal solution. Since many scheduling problems are $\mathcal{NP}$-hard in the strong sense, polynomial-time approximation algorithms are often used. Two primary metrics used in this context are the *approximation ratio* and *speedup factor*. Both measures provide insights into the performance of heuristic algorithms relative to optimal ones, the approximation ratio focuses on solution quality, whereas the speedup factor emphasizes computational speed and resource requirements. Please note, while both speedup factors and approximation ratios offer theoretical bounds on the worst-case performance of heuristic scheduling algorithms, they may not always reflect real-world performance benchmarks. Specifically, certain heuristic scheduling algorithms, in practical scenarios, may outperform others that ostensibly have superior approximation ratios and/or speed factors.

In this subsection, we discuss the definitions, implications, and significance of these two metrics in evaluating scheduling algorithms.

#### Approximation Ratio

The approximation ratio serves as a metric in evaluating the relative performance of heuristic algorithms against optimal solutions, particularly when achieving optimality is computationally challenging or unattainable. It measures how closely a heuristic solution approaches the optimal outcome. Formally, for an algorithm $\mathcal{A}$ addressing the makespan minimization problem, its *approximation ratio* $\alpha \geq 1$ implies that for any task set $\mathbf{T}$ scheduled on $M$ processors, the makespan produced by the algorithm will never be worse than $\alpha \cdot C^*_{\max}$, where $C^*_{\max}$ is the optimal makespan. The closer the approximation ratio is to 1, the closer the studied algorithm is to the optimal solution, indicating higher efficiency. This ratio offers a worst-case bound on the algorithm's deviation from the optimal solution. Please note, the approximation ratio does not specify the proximity of a heuristic algorithm to the optimal outcome for a particular task set.

#### Speedup Factor

The *Speedup Factor* [KP00; PST+97] of a scheduling algorithm is widely applied to describe the approximation quality of a schedulability test with respect to an optimal scheduling algorithms. Specifically, it determines the multiplicative factor, i.e., $\alpha \geq 1$, by which the system's speed must be elevated, to ensure that the scheduling algorithm $\mathcal{A}$ always produces a feasible schedule that meets deadlines, by assuming a feasible schedule is attainable by the optimal scheduling algorithm at the original speed.

Please note that the speedup factor predominantly provides insights into the worst-case scenarios and does not necessarily encapsulate the overall performance nuances. In practical applications, two algorithms with similar speedup factors could perform significantly differently. Such intricacies, especially how speedup factors influence algorithm design and might inadvertently lead to compromised performance, have been studied and discussed by Chen et al. [CBH+17].

## 2.3 Fault and Error Model

Safety-critical embedded systems are often subjected to transient faults due to the harsh environment and/or high density of logic circuits or computing components. Transient faults can lead to soft errors that cause incorrect results calculated by the affected executed jobs. For instance, soft errors can result in incorrect reading of a bit value in a digital memory or signal line without causing permanent damage or defect to the underlying hardware. Such a soft error can be corrected by rewriting the correct data to the affected memory location or by refreshing the signal.

In this dissertation, we assume that the probability that an executed job is affected by transient faults, which then results in at least one soft error, is given by a stationary probability $p_e$. Subsequently, the soft error probabilities for a sequence of jobs of the same task is an independent stochastic process. We assume that soft errors can occur at any time during a job's execution, but such an error-affected job is assumed to halt after executing for at most its worst-case execution time *even in the unreliable mode* by means of, e.g., watchdog timers. Moreover, the error detection in both the *reliable* and *detected* mode is certain in the sense that an incurred soft error is detected with the same (very high) probability as other system reliability dependent parameter guarantees. In either the *detected* mode or the *reliable* mode, errors are detected at the end of a job's execution by using sanity or consistency checks [Pra86]. 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, in the simultaneously and redundantly threaded processors with recovery approach (SRTR) [VPC02], the register values of all redundant threads are compared and are only committed if the register values of all threads agree. Otherwise, the threads are re-executed *for at most a specified number times*, i.e., *rep* times, until either the register values of all threads agree or the maximum number of re-executions is reached. Hence, the probability that a correct result is produced in the reliable mode by the SRTR approach after at most *rep* re-executions is given by $1 - (p_e)^{rep}$ under the assumption that soft error probabilities of the re-executions are independent and that each soft error results in a disagreement of the compared to register values. The maximum number of replications *rep* must be determined by the system designer depending on the required confidence and how hardened the considered system must be. The overhead for detection and recovery is henceforth integrated into the WCETs of the corresponding jobs' execution modes.

## 2.4 Experimental Platforms

This section details the experimental platforms that underpin the validation and analysis presented in this dissertation. The platforms comprise specialized components tailored to meet the research requirements, including an event-driven multiprocessor schedule simulator for resource synchronization, various real-time operating systems, specific hardware platforms, and the applied tool for formal verification. Collectively, these components establish a comprehensive environment suitable for both simulations and empirical evaluations. Subsequent subsections provide a detailed description of each component.

### 2.4.1 Event-Driven Multiprocessor Schedule Simulator for Resource Synchronization

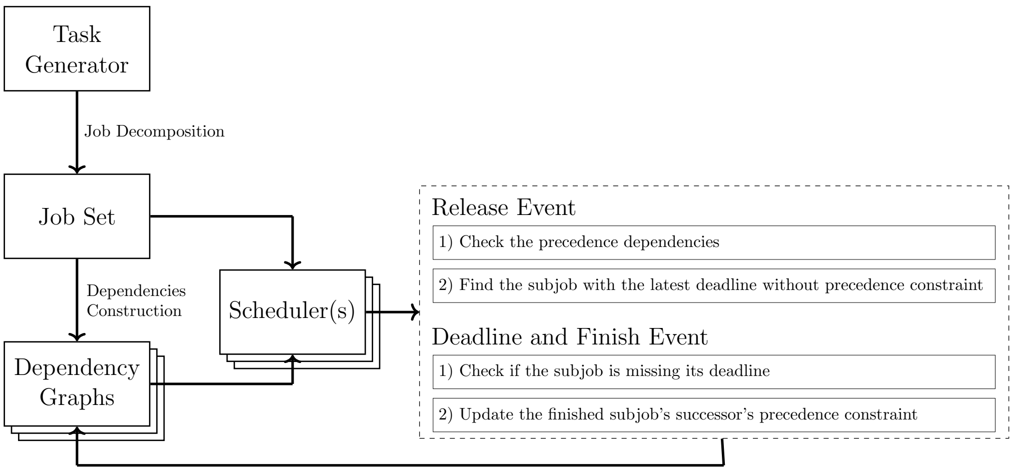

Task Generator
Job Decomposition
Job Set
Dependencies Construction
Scheduler(s)
Dependency Graphs
Release Event
1) Check the precedence dependencies
2) Find the subjob with the latest deadline without precedence constraint
Deadline and Finish Event
1) Check if the subjob is missing its deadline
2) Update the finished subjob's successor's precedence constraint

**Figure 2.2:** Overview of the Event-Driven Simulator: Using Dependency Graph Approach for Multiprocessor Resource Synchronization.

We developed an event-driven multiprocessor scheduling simulator that leverages a dependency graph approach for resource synchronization, implemented in Python 3.8. Figure 2.2 provides an overview of the simulator. The simulator is designed to generate customized task sets and to simulate detailed scheduling by applying the dependency graph approach for resource synchronization within a hyper-period. The working flow of the simulator is delineated below:

1. *Task Generation:* In the **Task Generator** module, we employ the Dirichlet-Rescale (DRS)[GBD20; GBD] method to determine the utilization for each task set based on the overall system utilization and the specified number of tasks. The Worst-Case Execution Time (WCET) of each segment is then computed as $C_{i,j} = U_{i,j} \times T_i$, where $U_{i,j}$ is the utilization allocated across all computational segments using DRS.

2. *Job Unrolling:* This stage involves the unrolling of jobs and sub-jobs for each task within the corresponding hyper-period, followed by their aggregation into the **Job Set**.
3. *Dependencies Construction:* Using the dependency graph approach, we define the execution order of critical sections protected by the same mutex lock. This ordered structure, combined with the inner execution order of sub-jobs within a job, forms the **Dependency Graphs**. Within the simulator, each sub-job instance has two attributes: `predecessors` and `successors`. These attributes are utilized to register the sub-job ID(s) of their respective predecessors and successors.
4. *Event-driven Scheduling:* The scheduler operates on the constructed dependencies, release time, execution time, and deadline of jobs as input. The simulator supports both (semi-)partitioned and global scheduling. In (semi-)partitioned scenarios, it can deploy multiple schedulers. Two types of events are triggered for each (sub-)job under scheduling: release event and deadline/finish event.
    - In the release event, precedence constraints of the released sub-job are assessed. If none are found, i.e., `predecessors=Null`, it is added to the ready queue, with the sub-job having the earliest deadline selected for execution on the available processor. Preemptive scheduling allows sub-jobs with earlier deadlines to preempt currently executing sub-jobs.
    - In the deadline/finish event, the current sub-job's adherence to its deadline is assessed. When a sub-job completes successfully without violating its deadline, the precedence constraints for any of its successors are updated. Specifically, it removes its own ID from the `predecessors` array of all its successors.

As we use the simulator to evaluate the performance of the dependency graph approach with time-critical tasks, the simulation for a particular task set will halt immediately if a sub-job misses its deadline, rendering the task set infeasible.

### 2.4.2 Real-Time Operating Systems

To study the applicability of our newly proposed approaches, we examine two widely used operating systems in real-time research domain: LITMUS$^{\mathrm{RT}}$ and RTEMS. Each system offers a comprehensive feature set, facilitating exploration of multiple aspects of real-time scheduling and synchronization.

#### LITMUS$^{\mathrm{RT}}$

The Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUS$^{\mathrm{RT}}$) is a specialized real-time extension of the Linux kernel. It focuses on multiprocessor real-time scheduling and synchronization and provides essential abstractions and interfaces within the kernel to simplify the development and prototyping of multiprocessor real-time scheduling algorithms and synchronization protocols.

Key features of LITMUS$^{RT}$ include:

- Support for both the periodic and sporadic task model, enabling a flexible scheduling paradigm.
- Modular scheduler plugins and reservation-based scheduling for various scheduling requirements.
- Support for clustered, (semi-)partitioned, and global schedulers to accommodate different system architectures.
- Integration of feather tracing tools to streamline the assessment of overheads in the deployed scheduling algorithms and resource synchronization protocols.

LITMUS$^{RT}$ often serves as a proof-of-concept platform rather than a real RTOS, demonstrating the practicality of implementing multiprocessor schedulers and synchronization protocols on modern hardware. For this dissertation, we employ the most recent release, version 2017.1, derived from the Linux kernel 4.9.30.

#### RTEMS

The Real-Time Executive for Multiprocessor Systems (RTEMS) is an adaptable open-source RTOS, aligning with open standard application programming interfaces (APIs). It supports various processor architectures, including ARM, PowerPC, x86, SPARC, RISC-V, and MIPS. Moreover, RTEMS offers an extensive array of board support packages (BSPs), enhancing its versatility.

RTEMS is renowned for its extensive adoption in real-world applications across varied domains, ranging from space flight and medical devices to networking and diverse embedded systems. RTEMS stands out for its adaptability, robustness, and wide compatibility with numerous hardware platforms.

In this dissertation, we employ the most recent release of RTEMS, version 5.1, to investigate its aptitude in the implementation of our proposed methodologies.

### 2.4.3 Hardware Platforms

In this dissertation, we study various scenarios, necessitating the consideration of diverse hardware platforms. We employ varied devices to investigate multiprocessor systems, distributed embedded systems, and GPU-equipped embedded systems.

#### Multiprocessor Systems

To deploy LITMUS$^{RT}$ and RTEMS on multiprocessor systems, we utilize two specific devices:

- **x86 Architecture (LITMUS$^{RT}$):** A cache-coherent Symmetric Multi-Processing (SMP) system consisting of two 64-bit Intel Xeon Processor E5-2650Lv4 running at 1.7 GHz, with 35 MB cache and 64 GB of main memory.

- **PowerPC Architecture (RTEMS):** An NXP QorIQ T4240 reference design board, the same as used in [CBH+15]. It features 6 GB DDR3 memory with a 1866 MT/s data rate, 128 MB NOR flash (16-bit), 2 GB SLC NAND flash, and a T4240 processor with 24 virtual cores (12 physical cores) running at 1.67 GHz.

### Distributed Embedded Systems

To examine the applicability in distributed embedded systems, we consider:

- **Emulation Platform:** Constructed from a computing server to emulate several embedded nodes and a GPU server for deploying optimization algorithms. The computing server comprises two AMD 3990X processors and 256 GB main memory. The GPU server consists of an Intel i7-8700K processor, two Nvidia GTX1080 GPUs, and 32 GB main memory, exclusively running the optimization algorithm, i.e., the MBO.
- **Physical Platform:** A distributed embedded system made of four ODROID-N2 boards [Har19]. Each board integrates a quad-core ARM Cortex-A73 CPU, a dual-core Cortex-A53 CPU, and 32GB storage, with DDR4 RAM running at 1320Mhz at 1.2 volts for low power consumption. Four nodes are connected via wired local network.

### Embedded Systems with GPU

Finally, for deploying relatively complex machine learning models, we consider:

- **Nvidia Jetson AGX Xavier (32G) Board:** This board houses an octa-core ARMv8.2 64-bit CPU, 32 GB LPDDR4 main memory, and 32 GB eMMC for storage. It includes a 512-core NVIDIA Volta GPU with tensor cores, delivering up to 32 TOPS of accelerated computing capability. Various power modes provide different computational capabilities to meet specific application requirements. In this dissertation, two power modes are evaluated: a) default mode with a 15W power budget using 4 processors at 2188 MHz, and b) MAXN mode without power budget limitation using 8 processors at 2265.6 MHz. Detailed configurations are documented in [NVI21].

These multifaceted hardware platforms provide a rich and flexible environment to explore the performance, efficiency and applicability of our proposed approaches with different scenarios.

### 2.4.4 Formal Verification Tool

Formal verification uses mathematical concepts and rigorous techniques to validate the correctness of underlying algorithms. Considering the intricate constraints that an RTOS must adhere to and the ensuing complexity of its source code, we employ

Frama-C [KKP+15; FRA23] version 22.0. to verify the protocols implemented in an RTOS, specifically focusing on RTEMS in this dissertation.

Frama-C, an open-source suite designed specifically for C and C++ code analysis, functions as a framework, utilizing a variety of plugins for detailed evaluation of C programs. Central to Frama-C is its kernel, which provides essential services. These include converting the analyzed program into a standardized format known as the Abstract Syntax Tree (AST) and offering tools for AST navigation [BBB+21]. The tool's parsing structure is anchored in the C Intermediate Language (CIL)[NMR+02], with Frama-C extending CIL's features by endorsing annotations based on ACSL contracts[KKP+15]. These contracts will be further detailed in Chapter 5. Frama-C transforms C source code into a particular subset defined by CIL. Throughout this transformation phase, the tool introduces localized code modifications to streamline the analysis, all the while preserving the original semantics of the source program [CCK+21]. Due to our distinct requirements, we employed the weakest precondition plugin [BBC+21] within Frama-C to authenticate specific code properties. An extensive Frama-C tutorial in [BKL18] demonstrated numerous plugins.

Chapter 3

# Background and Related Work

## Contents

This dissertation focuses on the challenges of optimizing the deployment of machine learning models on embedded systems that have specific timing requirements as well as the applications of machine learning to address challenges in embedded systems. To provide a comprehensive scope of our investigation, we explore the broad range of interconnected disciplines that encompass real-time systems, embedded systems, machine learning, and their intersecting domains. The goals of this chapter are: a) to present the foundational theories and outcomes, b) to demonstrate state-of-the-art researches, and c) to identify areas with potential for further research.

Section 3.1 sets the stage by introducing resource synchronization protocols across various scenarios, such as uni-processor systems, multiprocessor systems, and nested resource accesses. Following this, Section 3.2 focuses on the practicality of these protocols, providing an in-depth examination of their implementations and associated formal verification methodologies. Considering the context of distributed embedded systems, Section 3.3 provides an explanation of distributed machine learning. Alongside this, we present an overview of control robustness and soft-error compensation in embedded systems in Section 3.4. Finally, Section 3.5 gathers several illustrative use cases where machine learning is applied to schedule real-time tasks. By exploring these various aspects, this chapter aims to build a comprehensive understanding of the current landscape and potential future directions.

## 3.1 Resource Synchronization on Real-Time Systems

To ensure data consistency and prevent race conditions, tasks are mandated to execute mutually exclusively when accessing a shared resource. That is, once a task obtains a shared resource, other tasks are forbidden from accessing the same resource simultaneously.

Several mechanisms have been developed to enforce this mutual exclusion. Traditional mechanisms, such as binary semaphores or mutex locks, protect segments known as critical sections. Conversely, contemporary methods such as transactional memory (TM) or lock-free/wait-free algorithms leverage retry loops for accessing shared resources. Whereas the former guarantees execution correctness once inside a critical section, the latter promises correctness upon successful object updates or transaction commitments. This dissertation predominantly addresses methods based on critical sections. However, executing critical sections mutually exclusively introduces potential issues, including:

- Unbounded priority inversion: a scenario where a high-priority task is blocked by lower-priority tasks.
- Deadlock: a situation that arises when nested shared resources are requested.

Over the years, in efforts to bound blocking times arising from priority inversion and to avert deadlocks, numerous resource synchronization protocols have been devised and rigorously analyzed. In this section, we delineate the most widely-recognized of these protocols, with a special focus on both uni-processor and multiprocessor configurations. We also explore tailored adaptations specifically crafted for multiprocessor systems, notably partitioning approaches. Furthermore, we highlight research that studies nested resource synchronization challenges. To conclude this section, we present investigations into the complexity of navigating resource synchronization issues within varied system environments.

### 3.1.1 Uni-processor Resource Synchronization

In the realm of uni-processor real-time systems, several seminal protocols have been proposed to handle resource synchronization. The Priority Inheritance Protocol (PIP), introduced by Sha et al. [SRL90], allows a task to temporarily inherit a higher priority from another task awaiting access to the same shared resource. Extending upon this concept, the Priority Ceiling Protocol (PCP) [SRL90] was proposed to prevent deadlocks. Under PCP, every binary semaphore or mutex lock in the system is assigned a *priority ceiling*, which is the highest priority among all the tasks that may access this shared resource. A task can only acquire a binary semaphore or mutex lock if its priority is higher than the priority ceilings of all binary semaphores or mutex locks presently engaged by other tasks. In instances where a task is unable to seize the shared resource, it opts for self-suspension, and its priority is temporarily raised to the priority ceiling of the desired binary semaphore or mutex lock. While waiting, the task is added into the corresponding wait queue for the desired shared resource. When the shared resource is released by another task, the highest-priority task in the wait queue becomes eligible to acquire the binary semaphore or mutex lock and start its execution.

Emerging as a variant of PCP is the Immediate Ceiling Priority Protocol (ICPP) [BW09]. In ICPP, a task acquiring a binary semaphore or mutex lock instantaneously elevates its priority to the priority ceiling of the accessed semaphore or lock. Notably, the ICPP, has been implemented in Ada (called Ceiling locking) and POSIX (called Priority Protect Protocol). Furthermore, Baker et al. introduced the Stack Resource Policy (SRP) [Bak91], designed specifically for the EDF scheduling policy.

### 3.1.2 Multi-processor Resource Synchronization

In multiprocessor real-time systems, many resource synchronization protocols extend well-known uni-processor protocols such as the PIP, PCP, and SRP. Rajkummar et al. [RSL88] proposed Distributed-PCP (DPCP), where each resource is assigned on a processor statically, called the resources synchronization processor. To execute a critical section, a task is migrated to a dedicated synchronization processor. There, it follows the uni-processor PCP for executing critical sections. DPCP applies *semi-partitioned* scheduling. Extending this concept, the Multiprocessor PCP (MPCP) [Raj90] permits tasks to run their critical sections locally. In order to minimize the usage of stack memory in real-time systems, Gai et al. [GLN01] proposed Multiprocessor SRP. Both MPCP and MSRP apply *partitioned* scheduling. The Flexible Multiprocessor Locking Protocol (FMLP), introduced by Block et al. [BLB+07], classifies resources into two categories: long and short. For short resources, critical sections are executed in a non-preemptable manner and tasks are spinning on their processors while waiting for resources. For long resources, tasks suspend themselves into a FIFO queue while waiting. FMLP is also the first protocol that supports both *global* and *partitioned* scheduling. Easwaran and Brandenburg [EA09] introduced Parallel PCP (P-PCP), considering global fixed priority preemptive multiprocessor systems. The $O(m)$ Locking Protocol (OMLP), proposed by Brandenburg and Anderson [BA10], guarantees a maximum of $O(m)$ pi-blocking for any task set while supporting both *global* and *partitioned* scheduling. Burns et al. [BW13] proposed the Multiprocessor resource sharing Protocol (MrsP), that allows tasks help other tasks during spinning cycles, and *(semi-)partitioned* scheduling is applied.

Since the performance of these protocols highly depends on how the tasks are partitioned, several partitioning algorithms were developed, e.g., by Lakshmanan et al. [LNR09] and Nemati et al. [NNB10] for MPCP, by Wieder and Brandenburg [WB13a] for MSRP, by Hsiu et al. [HLK11], Huang et. al [HYC16], and von der Brüggen et al. [BCH+17] for DPCP.

### 3.1.3 Nested Resource Synchronization

All these protocols can support nested resource sharing by employing a coarse-grained group lock. However, only a few of these protocols support nested resource sharing in a fine-grained manner. The first protocol that supports nested resource sharing is DPCP, since uni-processor PCP is applied on synchronization processors. Once nested resources are assigned on the same processor, the nested resource sharing is supported by uni-processor PCP by default. Chen et al. [CTB94] developed MDPCP for periodic task systems by carefully defining the inter-processor ceilings. Besides, the Multiprocessor BandWidth Inheritance protocol (M-BWI) [FLC10; FLC12] and MrsP [BW13; GZB+17] allow nested resource accesses without deadlocks if all the resources or mutex locks are accessed according to a specified total order. The Real-time Nested Locking Protocols family (RNLP) [WA12; WA14; JWA15; NAA18; NAA19] encompasses various variants of supporting nested resource sharing by addressing: a) different waiting mechanisms, i.e., suspension or spinning, b) different progress mechanisms, i.e., priority boosting [LNR09; BA10], priority inheritance [SRL90], and priority donation [BA11], and c) how pi-blocking is analyzed. However most of them do not handle the *transitive blocking chain problem*. That is, the traditional First-In-First-Out (FIFO) method of accessing resources can inadvertently block an entire chain of requests, even if several of them have no conflict with the requested resources. Only C-RNLP [JWA15] breaks the transitive blocking chains for nested write requests by applying a *cutting ahead* mechanism, where the lengths of critical sections are taken into consideration for lock the and unlock logic. Dynamic group locks (DGLs), where all resources in the corresponding group that the nested request belongs to are requested simultaneously when starting a critical section, also breaks the *hold-and-wait* condition [WA13]. Moreover, a fine-grained blocking bound for nested non-preemptive FIFO spin locks under P-FP scheduling is presented in [BBW16]. The analysis is based on a graph abstraction that reflects all possible resource conflicts and transitive delays. As the state-of-the-art, the newly proposed Concurrency Group Locking Protocol (CGLP) by Nemitz et al. [NAG+21] supports lock nesting using *group* locking. In addition, concurrency groups are utilized to break transitive blocking, where a concurrency group is a group of lock requests that can safely execute together.

### 3.1.4 Complexity Results

As real-time systems evolve in both intricacy and scale, understanding the computational complexity associated with resource synchronization becomes paramount.

**Table 3.1:** The complexity results that are known and discussed in this dissertation.

| **Complexity Results** | **Studied Problem** |
|---|---|
| Theorem 3 | $M \geq N + 1$, any scheduling paradigm |
| Theorem 9 | $M \geq Z$, semi-partitioned scheduling paradigm |
| Theorem 10 and [LR79] | $M \geq N > Z = 2$, partitioned scheduling paradigm |
| Theorem 10 and [LR79] | $M \geq N > Z = 3$, unit execution time, partitioned scheduling paradigm |
| Theorem 10 and [SS95] | $M \geq N = 3, Z = 3$, partitioned scheduling paradigm (with multiple visits to a mutex lock per job) |
| Theorem 11 and [GJ79] | $M \geq N > Z = 3$, partitioned scheduling paradigm (with flow-shop compatible access patterns) |
| Theorem 12 and [LR79] | $M = Z = 2$, semi-partitioned scheduling paradigm |
| Theorem 13 and [LR79] | $Z = M = 3$, unit execution time, semi-partitioned scheduling paradigm |
| Theorem 14 and [SS95] | $N = Z = M = 3$, (semi-)partitioned scheduling paradigm |
| Theorem 15 and [GJ79] | $Z = M = 3$, semi-partitioned scheduling paradigm (with flow shop access patterns) |
| Theorem 16 and [YHL04] | $Z = 1, \eta_i \geq 3, M \geq N$, unit execution time, any scheduling paradigm |

Different system environments present unique inherent challenges in navigating resource synchronization, each introducing its unique set of constraints and requirements. This section provides investigations for the complexity of tackling these resource synchronization issues with varied system landscapes. Detailed complexity results can be found in Table 3.1.

## 3.2 Implementations and Formal Verification

Numerous resource synchronization protocols have been explored in theory, but only a few of them have seen practical implementation in real-world RTOSes. While these protocols are theoretically sound and robust, the critical step of formal verification bridges potential gaps between theory and practical implementations. In this section, we focus on the protocols implemented in prominent RTOSes, specifically

RTEMS and LITMUS$^{RT}$. Additionally, we discuss formal verification methodologies, particularly those designed for real-time systems and programs.

### 3.2.1 Implemented Resource Synchronization Protocols

In the context of protocol implementations in real-time operating systems:

- RTEMS: in its mainstream release 5.1, supports ICPP for uni-processor systems and MrsP [CBH+15] for multiprocessor systems. We supported MPCP, DPCP, and FMLP in [SPM+22] for the earlier release of RTEMS 4.12.
- LITMUS$^{RT}$ provides support for a variety of protocols through its modular scheduler plugins, including MPCP, DPCP, FMLP [Bra11], and MrsP [CBH+15; SCZ+17].
- ERIKA Enterprise [Evi21]: currently accommodates only the ICPP within its uni-processor system configuration.

Zhao and Wellings [ZW17] identified challenges arising from MrsP's integration into operating systems, such as Linux, that use a "push-and-pull" task migration approach. Specifically, allowing resource-holding tasks to migrate can lead to implementation issues and run-time anomalies. They proposed a strategy to counteract these unintended migrations, and its effectiveness was affirmed through benchmark tests on a reference implementation within LITMUS$^{RT}$. This evolution of MrsP underscores that its implementations across varied real-time operating systems, like LITMUS$^{RT}$ [SCZ+17; CBH+15] and RTEMS [CBH+15], might correspond to various adaptations or iterations of the protocol.

### 3.2.2 Formal Verification Tools

Formal verification tools, especially those adept at handling C implementations, have grown in prominence due to their applicability across various domains. In this section, a few notable contributions are introduced:

- Frama-C: This tool has found extensive applications across multiple sectors. For instance, Efremov et al. [EMK18] innovated a deductive verification method for Linux kernel functions. They developed a new plugin to address the incompatibilities between Frama-C and some specific kernel constructs.
- Verifier for Concurrent C (VCC): Cohen et al. introduced VCC, another tool for deductive verification, as detailed in [CDH+09]. It ensures program correctness by monitoring the ownership of non-volatile data across all possible concurrent thread executions. Notably, VCC has been employed in the partial verification of Microsoft's Hyper-V Hypervisor [LS09] and a compact, exemplar Hypervisor [AHP+10].
- Prusti: The Prusti project [AMP+19] extends similar capabilities to languages like Rust. Prusti emphasizes function contracts akin to those explored in this work, underscoring the tool's adaptability.

- Software Analysis Workbench (SAW): In the domain of security, Chudnov et al. demonstrated how SAW can be used for the continuous verification of Amazon's open-source TLS implementation [CCC+18]. This tool, optimized for minimal developer intervention, automatically updates and verifies proof conditions as source code evolves. It is versatile, supporting C, Java, and Cryptol, with the latter tailored for cryptographic algorithm specifications. Due to its reliance on bounded symbolic execution, SAW proves especially useful for programs with finite loops.
- Coq [BC04] offers a formal language environment to draft mathematical definitions, executable algorithms, and theorems. It is complemented by a semi-interactive interface for developing machine-checked proofs.

### 3.2.3 Formal Verification of Operating Systems

Numerous approaches focusing only on the formal verification of operating systems have been developed and evaluated. One approach emphasizes the design and implementation of operating system kernels with comprehensive formal verification from inception, rather than retroactively verifying existing kernels. A concrete example is *seL4*, which is presented by Klein et al. [KEH+09]. The microkernel is verified via refinement steps from the abstract specification represented by a Haskell prototype over the executable specification in Isabelle/HOL to a manually optimized C version. Every layer below the verified source code of the microkernel, from the compiler to the hardware, is assumed correct and not target of the verification. Gu et al. [GSC+16] presented an architecture for concurrent operating system kernels consisting of multiple layers. The code of each system layer is verified with Coq [BC04]. As a demonstration of the architecture, the kernel *mC2* was developed for multiprocessor x86 computers, supporting fine-grained locking, threads with suspension and serving as a hypervisor.

Gadia et al. introduced an approach specifically targeting RTEMS [GAB16]. In order to verify the implementation of the Priority Inheritance Protocol (PIP) with a software model checker, it was remodeled in Java along with the relevant associated scheduling mechanisms. PIP- and race-condition-related safety properties were included as assertions and the resulting model was investigated with *Java Pathfinder*. During the evaluation, an implementation error related to nested resource sharing was confirmed and fixed. Additionally, Almatary et al. [AAB15] proposed an approach to reduce the kernel calls when implementing ICPP in POSIX, where the implementation is verified by using model checking. Vanhems et al. [VRN+22] provided a formally verified implementation of an EDF scheduler tailored for arbitrary job sequences, with proofs crafted in the Coq proof assistant.

Recently, Nicole et al. [NLB+21] proposed an automated method to verify two key properties of an operating system: the absence of runtime errors (ARTE) and privilege escalation (APE). The verification target is represented by the binary executable. An abstract interpreter was developed to process the executable and determine all possibly reachable states of registers and memory. Based on this, ARTE and APE can be verified automatically. Compared to verifying source code, this approach is highly specific to the instruction set architecture for which the image was built, and the verification is restricted to critical, but fixed, low-level properties.

### 3.2.4 Formal Verification of Real-Time Programs

Several studies have focused on verifying specific real-time programs that employ locks. Chaki et al. [CGS13] presented an approach for verifying the safety and deadlock freedom of programs utilizing PIP locks. Their approach is based on sequentialization. That is, a periodic program is converted into an equivalent (non-deterministic) sequential program at first. Afterwards, a model checker is applied for verifying the correctness. Furthermore, Suresh et al. [SPD+22] introduced a technique to statically identify data races in periodic real-time programs, specifically when using locks on uni-processor systems. Their approach is based on a small set of rules that exploit the priority, periodicity, locking, and timing information of tasks in the program. Their focus was on the verification of distinct programs employing locks, particularly those with multiple concrete tasks and resources.

### 3.2.5 Formal Verification of Schedulability Analysis

The verification of mathematical correctness has attracted more attention in the recent years. Pascal et al. [FGM+19] introduced CertiCAN using the Coq proof assistant, aimed at the formal certification of CAN analysis outcomes. Serving as a pivotal foundation for formally verified schedulability analysis, Prosa [CSB16], constructed with Coq, is widely applied for mechanical proof verification. Building on this foundation, Pascal et al. [FLM+18] provided a generic proof for the Typical Worst-Case Analysis (TWCA) designed specifically for weakly-hard real-time uni-processor systems, with the entire formalization embedded in Coq, complemented by the Prosa library. In parallel, Bozhko et al. [BB20] took efforts to formalize the long-standing busy-window principle applicable to uni-processor systems. In their work, the schedulability analyses were formally formed and verified by using Coq and Prosa framework. Meanwhile, Roux et al. [RQB22] linked the response time analysis (RTA) and network calculus (NC), that applied Prosa and NCCoq [RBR19] respectively. Furthermore, Maida et al. [MBB22] presented the foundational response time analysis through POET, which spawns human-inspectable proofs ensuring temporal correctness. POET was built on Prosa, and diversified the spectrum of scheduling paradigms. In a similar context, Bedarkar et al. [BVB+22] provided a case study, that corroborated a novel response time analysis for sporadic tasks under FIFO scheduling using Coq. Finally, Guo et al. [GRT21] delineated an approach that used Prosa's certified schedulability analysis to formally verify CertiKOS schedules based on periodic task architectures and preemptive scheduling.

Collectively, these studies highlight the critical role of schedulability analysis verification in ensuring the reliability of real-time systems.

## 3.3 Distributed Machine Learning

The performance of machine learning algorithms is significantly influenced by their hyper-parameters. The tuning process in resource-constrained distributed embedded systems is complicated by factors such as privacy concerns and limited computational capacity. In this section, we present various hyper-parameter tuning algorithms, followed by a succinct overview of both *Model-Parallelism* and *Federated Learning*.

### 3.3.1 Hyper-parameter Tuning Algorithms

The most direct and easy to implement tuning algorithm is grid search [LBO+12] which discretizes the hyper-parameter search space and exhaustively evaluates all possible combinations in a Cartesian grid to find the setting with the best performance. Another variation is random search [BB12], which randomly samples hyper-parameter settings from the search space. Both methods, however, do not leverage information from previous evaluations, leading to potential computational inefficiencies. In contrast, Sequential Model-Based Optimization (SMBO) [JSW98] takes advantage of the previous search trajectory. Multiple benchmarks [HHL13; BRB+17; BNG+18] highlight MBO's superiority over grid and random search, and even over evolutionary approaches. In the classical approach, Gaussian process regression, also called Kriging, is used as its regression model [SLA12]. In specific scenarios with hierarchical search spaces, tree-based surrogates like Tree-structured Parzen Estimator (TPE) [BYC13] or random forests [HHL11] have shown advantages. Also, Bayesian Neural Networks (BNN) [Gra11] can serve as a surrogate. In this method, a probability distribution for each network weight is established to provide a variance around the prediction. However its training process is very time-consuming. Several extensions are proposed to speed up the BNN, e.g., sample multiple sub-networks from a network trained with Dropout [SHK+14; GG16].

In order to extend MBO with parallel evaluations, various techniques have been developed to propose and evaluate multiple points in each iteration. Ginsbourger et. al. [GLC10] proposed several approaches based on imputing the results of currently running experiments. Hutter et. al. [HHL12] proposed the qUCB, which uses the Gaussian process upper confidence bound (GP-UCB). By optimizing the GP-UCB with different weights for the uncertainty, we obtain a set of proposals, i.e., q denotes the number of obtained proposals. Recently, Rebolledo et. al. [CRE+20] introduced a parallelized Bayesian optimization that maintains low evaluation counts for efficiency. By performing parallel evaluations, this approach not only reduces wall-clock time but also outperforms state-of-the-art parallel CMA-ES techniques [HO01], even in high-dimensional scenarios like the 20-dimensional Sharp Ridge function. To account for heterogeneous run-times of different proposals, asynchronous parallel strategies [JRG+12] with scheduling methods [RKB+16; KSL+19] have been developed.

### 3.3.2 Model-Parallelism and Federated Learning

Due to the increasing demands of distributed data collection, storage, and processing as well as the concerns about preserving privacy in many applications, federated learning [KMY+16; LXG+19] has become one of the popular computing paradigms, where a machine learning model is trained across multiple decentralized edge devices or servers with their local data. In most federated computing platforms, "no raw data sharing" is an important requirement, where a machine learning algorithm should be trained using all data stored in all the distributed machines but without any cross-machine raw data sharing. In particular, the aforementioned hyper-parameter tuning algorithms can be accelerated by federated learning and typically be divided into two types: *Data-Parallelism* [Bae11] and *Model-Parallelism* [XHD+15]. In each embedded system or node, the Data-Parallelism approach begins by training the model on local data. Afterwards, a global model is obtained via model-averaging [CH+08]. The aggregated model is considered as the trained model based on the overall data (from multiple nodes). Due to the construction of Data-Parallelism, parallel computing method can be easily applied. The *Model-Parallelism* requires multiple nodes to learn a shared prediction model collaboratively. This approach necessitates either synchronous or asynchronous parameter updates across all nodes, incurring additional overheads. In several applications, updating these parameters introduces significant challenges.

Both of the aforementioned methods ensure that the training data remains localized to their respective nodes. Compared with the Data-Parallelism, the Model-Parallelism usually can achieve better performance, as it globally optimizes the performance of the model [XHD+15]. One of the most popular branch of Neural Architecture Search (NAS) that employs Model-Parallelism is federated NAS [GSD20; HAA20; ZJ22], which is designed to autonomously search for global and tailored models suited for non-IID data. To further preserve privacy, differentially-private FNAS [SZY+20], which adds random noise to the gradients of architecture variables, has been designed for a higher level of privacy protection. These algorithms mainly focus on federated learning solutions for NAS with computationally expensive method(s) (e.g., reinforcement learning-based surrogate method) and powerful GPUs (e.g., RTX 2080Ti in [HAA20]).

## 3.4 Control Robustness and Soft-Error Compensation

In control theory, controllers are designed to tolerate erroneous input signals and maintain control system functionalities amid uncertain environments. To address this, several techniques have been proposed [Ram99; KGC+12] or dropped signal samples [HSJ08; BS15; GDD19]. If a sample input contains an error, the sample may be discarded, and a subsequent control decision can be determined using previous inputs to maintain the loop [Ram99; HSJ08; BS15].

Another series of fault tolerance techniques rely on the $(m, k)$ models, which is originally developed to ensure a limited number of deadline misses in firm real-time systems, also known as weakly-hard real-time systems [BBL01], where a task has to meet at least $m$ deadlines, or can miss at most $m$ deadlines, in any of $k$ consecutive jobs[1]. While the original study [BBL01] utilized $\binom{n}{m}$ to describe the any $n$ of $m$, most of the following works utilize the $(m, k)$ to describe the weakly-hard constraints [CKZ19; HQE20; SKT20; VPM22]. Afterwards, such $(m, k)$ models have been widely adopted in the fault tolerance domain to define the robustness constraints of control systems. That is, a task must have at least $m$ functionally correct instances out of any $k$ consecutive instances. Such a requirement can ensure that a control system is still feasible only if it can satisfy the corresponding $(m, k)$ robustness constraint [CBC+16; YCC18]. To adhere to a given $(m, k)$ constraint, several static patterns are widely applied for different purposes, i.e., deep red pattern (R-pattern) [KS95], evenly distributed pattern (E-pattern) [Ram99], and reverse E-pattern [QH00]. Besides the static pattern based approaches, Chen et al. proposed an adaptive approach in [CBC+16]. Such an approach tries to minimize the overall execution time of a task by postponing the execution of reliable mode. Liang et al. [LWJ+20] developed a novel method and an optimization algorithm to analyze and improve control stability and system schedulability under deadline misses, faults, and the application of two different fault-tolerance techniques, where redundant execution is applied of EOC [GGB13] techniques and re-execution are performed in case of a soft error.

In recent years, numerous methodologies have been advanced to reduce the overheads associated with fault tolerance techniques. Nikiema et al. proposed a new mechanism with near-zero and bounded timing overhead in [NKT+23], to circumvents the faults as soon as they occur, to significantly reduce the WCET estimations in systems with hardware faults. In safety-critical systems, achieving high mean-times-to-failures (MTTFs) while minimizing the overheads of re-execution or replication is paramount. Addressing this challenge, Matovic et al. introduced the Consensual Resilient Control (CRC) approach in [MGL+23]. The CRC method transforms stateful controllers into instances that can be recovered in a stateless manner, effectively eliminating cold-start effects and permitting control tasks to rejuvenate during each control cycle. Miedema et al. proposed a new approach for applying fault-tolerance technique in weakly hard resource constrained real-time applications, namely strategy switching in [MG22]. Strategy switching aims at minimizing the effective unmitigated fault-rate by switching which tasks are to be run under a fault-tolerance scheme at runtime. Their approach does not require bounding the number of faults for a given number of consecutive iterations.

[1]Although nonconsecutive situations are considered as well in the original work, this situation is not considered in this work and hence omitted here.

In addition, to maintain the control quality and ensure the schedulability, skipping certain control computation is considered. AlEnawy et al. [AA05] presented an online speed adjustment algorithm to exploit the slack time of skipped and completed jobs in order to minimize the number of dynamic failures (in terms of (m,k)-firm deadline constraints) while remaining within the energy budget. von der Brüggen et al. [BCH+16] determined if the system with dynamic real-time guarantee can provide full timing guarantees or limited timing guarantees without any online adaptation after a fault occurred. Wang et al. [WHK+21] presented a cross-layer approach to improve system adaptability by allowing proactive skipping of task executions. Huang et al. [HXW+20] introduced an online intermittent-control framework that integrates formal verification, 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 main constraint is the control safety rather than schedulability and $(m, k)$ robustness.

## 3.5 Machine Learning for Real-Time Scheduling

In recent years, machine learning (ML) has attracted significant interests in both academic and industrial areas. However, only a few of studies have applied ML-based approaches in (real-time) embedded systems domain due to the stringent requirements, e.g., timing guarantee, security, and power consumption. Bo et al. [BQL+21] proposed a deep RL-based scheduler for multiprocessor real-time systems, modeling the real-time scheduling process as a multi-agent cooperative game. In [LBW+21], a ML-based approach for priority assignment was proposed. To meet the complex time-critical requirements, Dole et al. [DGK+21] advocated for using Duration Calculus (DC) to frame learning objectives in model-free RL for stochastic real-time systems. In order to improve the efficiency of generating Clock Constraint Specification Language (CCSL) specifications, Hu et al. [HDZ+21] combined the merits of both RL and deductive techniques in logical reasoning for efficient co-synthesis of CCSL specifications. Recently, Xu et al. [XKH+23] introduced a RL strategy aimed at minimizing overall path latency for all scheduled runnables in automotive systems. This approach ensures adherence to other essential constraints, including schedulability, load balancing, and data contention control. The Logical Execution Time paradigm is employed to schedule all runnables, facilitating time-deterministic communication on multi-core processors.

Chapter 4

# Dependency Graph Approach for Multiprocessor Real-Time Synchronization

## Contents

## 4.1 Overview

In this chapter, we explore the complex domain of multiprocessor resource synchronization. Although much research has been dedicated to multiprocessor resource synchronization protocols, a considerable gap still exists between theoretical results and their real-world applications.

**Challenges in current approaches:** Several challenges persist in bridging this gap. Most existing synchronization protocols focus on sporadic task systems, which, while theoretically sound, introduces significant pessimism. To be more precise, although the minimum inter-arrival time between consecutive job releases has been determined in sporadic task systems, the actual releases can be irregular and unpredictable. For bounding the worst-case response time, schedulability analysis often assumes the worst-case scenario, ensuring all deadlines are met. However, the majority of tasks in practice are periodic, creating a discrepancy between theoretical constructs and real-world scenarios. In a recent empirical survey by Akesson et al. [ANN+20], based on 120 responses, 82% of the systems contained periodic activation patterns, while over 60 included aperiodic activations. Interestingly, 22% of the respondents specified that their systems exclusively employed either periodic or time-triggered activations, eschewing sporadic or aperiodic tasks, in order to achieve highly predictable behaviors. In contrast, a mere 4% and 2% of participants reported systems with solely sporadic and aperiodic activations, respectively, devoid of any time-triggered or periodic tasks. Therefore, having good solutions for periodic activations is useful for industrial practice.

Moreover, with the widespread integration of machine learning tasks, the lengths of critical sections have substantially increased, e.g., access to accelerators. Traditional work-conserving-based resource synchronization methods may no longer be suitable, necessitating novel approaches.

**This dissertation:** To tackle these challenges, our study concentrates on the fundamental difficulties and complexities inherent in resource synchronization. We also explore the performance disparities between different scheduling algorithms and devise strategic scheduling for critical sections to enhance performance.

We initiate our exploration with a frame-based task system: a scenario where all tasks share identical periods and deadlines, with each task having a single non-nested critical section. In this simplified setting, the schedulability optimization problem can be reduced to a makespan minimization problem, offering a valuable starting point. Recognizing that this basic setting is often too simplistic for real-world applications, we subsequently expand our analysis by relaxing constraints. We explore scenarios allowing multiple critical sections per task, nested resource sharing, and periodic task systems with varying periods. These explorations provide a bridge between the theoretical models and practical applications, forging new pathways for effective synchronization in modern multiprocessor systems.

We commence with Section 4.2, where we define the multiprocessor resource synchronization problem, considering two distinct scenarios, i.e., one critical section per task and multiple critical sections per task. Next, in Section 4.3, we introduce the dependency graph approach tailored for a frame-based task system with one critical section per task. This section includes a detailed overview of the approach's two steps, its complexity, a lower bound of the approximation ratio, and the techniques for constructing dependency graphs. Following this, Section 4.4 extends the approach to handle multiple critical sections per task, establishing a connection with the job shop scheduling problem. In Section 4.5, we propose the group execution constraint to facilitate nested resource synchronization within the dependency graph framework. We then explore the transition from the frame-based task system to periodic task systems in Section 4.6, addressing the added complexities and proposed solutions. To enable effective scheduling of the generated dependency graphs, Section 4.7 presents several algorithms, including semi-partitioned, LIST-EDF, and partitioned scheduling approaches. Finally, Section 4.8 provides comprehensive evaluations, illustrating the performance of the proposed methods across various scenarios.

Through this systematic analysis, we aim to contribute a nuanced understanding of the multiprocessor resource synchronization problem and deliver innovative solutions that bridge the gap between theoretical insights and practical applicability.

## 4.2 Resource Synchronization Problem

To start our study of the multiprocessor resource synchronization problem, we first concentrate on a fundamental and straightforward configuration. In this configuration, all tasks share the same period and deadline, and simultaneously release their jobs, creating what is known as a *frame-based real-time task system*. These tasks are then scheduled on $M$ homogeneous processors, implying that all processors are identical. Within this context, we consider two distinct resource access patterns: a) the OCS task model, where each task contains exactly one non-nested critical section; and b) the MCS task model, where each task may have multiple non-nested critical sections. The consideration of these two task models allows us to understand the fundamental challenges and dynamics of the problem. This simplified environment serves as a foundational basis for subsequent, more intricate investigations.

### 4.2.1 Single Critical Section in Each Task

We denote the multiprocessor synchronization problem for OCS task systems as *MS-OCS*. In scheduling theory, a primary objective is to minimize the **makespan** of a task set, defined as the time interval between the first job's arrival and the last job's completion. For frame-based real-time task systems, tasks have identical periods and deadlines. Each task releases its initial job at time 0 and one job at the start of every subsequent period. Consequently, if the released jobs from each task execute exactly with the corresponding WCET, the schedule in the first period is repeated in the following periods. If the makespan of the task set within one period does not exceed the relative deadline, all tasks in the task set can feasibly meet their identical deadline. That is, the task set can be feasibly scheduled under the given number of processors and the applied scheduling algorithm. We state the makespan problem for *MS-OCS* that is studied here as follows:

**Definition 10. The *MS-OCS* Makespan Problem:** Given $M$ identical homogeneous processors and $N$ tasks following the OCS task model arriving at time 0, the objective is to minimize the makespan.

Alternatively, we can additionally investigate the **bin packing** version of the problem, i.e., minimizing the number of allocated processors to meet a common deadline $D$.

**Definition 11. The *MS-OCS* Bin Packing Problem:** Given $M$ identical homogeneous processors and $N$ tasks following the OCS task model arriving at time 0, the objective is to find a schedule to meet the common deadline $D$ with the minimum number of allocated processors.

Essentially, the decision versions of the makespan and the bin packing problems are identical:

**Definition 12. The *MS-OCS* Schedulability Problem:** Given $M$ identical homogeneous processors and $N$ tasks following the OCS task model arriving at time 0, the objective is to find a schedule to meet the common deadline $D$ by using the $M$ homogeneous processors.

We note that an algorithm with an approximation ratio $\alpha$ for the makespan problem in Definition 10 also has a speedup factor $\alpha$ for the schedulability problem in Definition 11. That is, an algorithm $\mathcal{A}$ for the bin packing problem (i.e., Definition 11) has an approximation ratio $\alpha \geq 1$, if given any task set $\mathbf{T}$, it finds a schedule of $\mathbf{T}$ on $\alpha M^*$ processors to meet the common deadline, where $M^*$ is the minimum (optimal) number of processors required to feasibly schedule $\mathbf{T}$.

### 4.2.2 Multiple Critical Sections in Each Task

We define the problem of multiprocessor synchronization in the context of MCS task systems as *MS-MCS*. The *MS-MCS* problem can be transferred to the following two general problems:

**Definition 13. The *MS-MCS* Makespan Problem:** Given $M$ identical homogeneous processors and $N$ tasks following the MCS task model arriving at time 0, the objective is to find a schedule that minimizes the makespan.

A feasible schedule of the *MS-MCS* makespan problem is a schedule that satisfies all aforementioned non-overlapping constraints stated in Section 2.2.2. For an input instance of the *MS-MCS* makespan problem, an optimal solution is a schedule with the shortest makespan compared to all feasible schedules of that instance. An algorithm $\mathcal{A}$ for the *MS-MCS* makespan problem has an *approximation ratio* $\alpha \geq 1$, if given any task set $\mathbf{T}$ and $M$ processors, the resulting makespan is at most $\alpha \cdot C^*_{\max}$, where $C^*_{\max}$ is the optimal makespan.

**Definition 14. The *MS-MCS* Schedulability Problem:** Given $M$ identical homogeneous processors and $N$ tasks following the MCS task model arriving at time 0, the objective is to find a feasible schedule that meets the common deadline $D$ on the given $M$ homogeneous processors.

As in Definition 9, a feasible schedule of the *MS-MCS* schedulability problem is a schedule that has a makespan no more than the common deadline $D$ and satisfies all the non-overlapping constraints in Section 2.2.2. The *MS-MCS* schedulability problem is a decision problem, in which for a given $D$ and a given algorithm either the algorithm produces a feasible schedule that meets the deadlines, or it does not yield a feasible schedule.

## 4.3 Dependency Graph Approach

To solve the multiprocessor synchronization problem, we begin with a simpler setting where all tasks follow the OCS task model and share the same period and deadline. When the OCS task model is applied, each task contains three computational segments, i.e., $\theta_{i,1},\ \theta_{i,2},\ and\ \theta_{i,3}$, where $\theta_{i,2}$ is the critical section that is protected by a binary semaphore. We propose a *Dependency Graph Approach* with two steps to handle the makespan problem in Definition 10:

- In the first step, we construct a directed graph $G = (V, E)$, as depicted in Figure 4.1. A computational segment (i.e., a critical or a non-critical section) is a vertex in $V$ and the edges in $E$ describe the precedence constraints of these computational segments. The computational segment $\theta_{i,1}$ is a predecessor of the computational segment $\theta_{i,2}$, and $\theta_{i,2}$ is a predecessor of the computational segment $\theta_{i,3}$. If two computational segments of $\tau_i$ and $\tau_j$ share the same binary

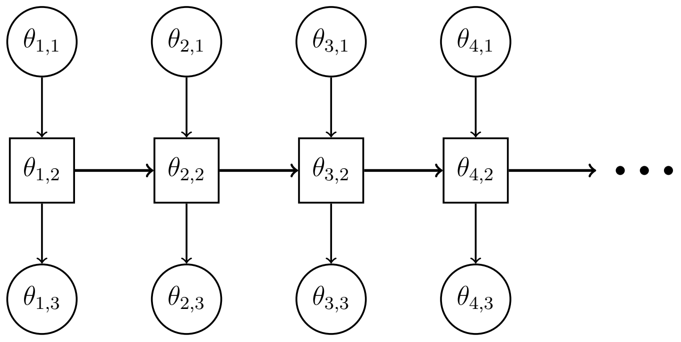


**Figure 4.1:** A task dependency graph for a task set with one binary semaphore, where a circle represents the non-critical section and a rectangle represents the critical section.

semaphore, i.e., $\sigma(\tau_i) = \sigma(\tau_j)$, then either the computational segment $\theta_{i,2}$ is the predecessor of $\theta_{j,2}$ or the computational segment $\theta_{j,2}$ is the predecessor of $\theta_{i,2}$. All the critical sections guarded by a binary semaphore form a chain in $G$, i.e., the critical sections of the binary semaphore follow a total order. Therefore, we have the following properties in set $E$:

- The two directed edges $(\theta_{i,1}, \theta_{i,2})$ and $(\theta_{i,2}, \theta_{i,3})$ are in $E$.
- Suppose that $\mathbf{T}_k$ is the set of tasks which request the same shared resource $z_k$. Then, the $|\mathbf{T}_k|$ tasks in $\mathbf{T}_k$ follow a certain total order $\pi$ such that $(\theta_{i,2}, \theta_{j,2})$ is a directed edge in $E$ when $\pi(\tau_i) = \pi(\tau_j) - 1$.

Figure 4.1 provides an example for a task dependency graph with one binary semaphore. Since there are $Z$ binary semaphores in the task set, the task dependency graph $G$ has in total $Z$ connected sub-graphs, denoted as $G_1, G_2, \ldots, G_Z$. In each connected sub-graph $G_z$, the corresponding critical sections of the tasks that request critical sections guarded by the same semaphore form a chain and have to be executed sequentially. For example, in Figure 4.1, the dependency graph forces the scheduler to execute the critical section $\theta_{1,2}$ prior to any of the other three critical sections.

- In the second step, a corresponding schedule of $G$ on $M$ processors is generated. The schedule can be based on system's restrictions or user's preferences, i.e., either preemptive or non-preemptive schedules, either global, semi-partitioned, or partitioned schedules.

In the initial design of dependency graph approach, we focus exclusively on frame-based task systems. We determine the execution order for computational segments in the first step and schedule these dependency graphs on $M$ given processors in the second step. For these systems, the scheduling is determined over a single period and this schedule is subsequently repeated for subsequent periods. Within this period, each task releases just one job. The execution sequence of these sub-jobs strictly adheres to the dependency graph established in the first step.

The scheduling of dependency graphs has been widely studied in the literature. A solution to the problem $P|prec|C_{\max}$ leads to a semi-partitioned schedule, since the dependency graph is formed by treating either a critical or a non-critical section as a sub-job. Moreover, a solution of the problem $P|prec, prmp|C_{\max}$ results in a global schedule. To derive a partitioned schedule, we can assign the sub-jobs generated by a job to be *tied* to a specific processor. That is, $P|prec, tied|C_{\max}$ targets a partitioned non-preemptive schedule and $P|prec, prmp, tied|C_{\max}$ targets a partitioned preemptive schedule. The detailed algorithms for scheduling generated dependency graphs are discussed in Section 4.7.

the primary challenge lies in constructing the dependency graph, which corresponds to the first step. We construct the dependency graph under the assumption of a consistently sufficient number of processors. Specifically, processor count is assumed as $M = N + Z$ in the first step. Subsequently, in the second step, we take into account the processor constraint as defined by the target platform.

**Definition 15.** A **feasible schedule** $S(G)$ of a task dependency graph $G$ respects the precedence constraints defined in $G$ and the specified scheduling requirement, e.g., being global/semi-partitioned/partitioned and preemptive/non-preemptive. $L(S(G))$ is the makespan of $S(G)$.

With the above definitions, we can recap the objectives of the two steps in the dependency graph approach. In the first step, we would like to construct a dependency graph $G$ to minimize $len(G)$, and in the second step, we would like to construct a schedule $S(G)$ to minimize $L(S(G))$.

**Lemma 1.** $len(G^*)$ is the lower bound of the *MS-OCS* makespan problem for task set $\mathbf{T}$ on $M$ processors.

*Proof.* This comes from the setting of the problem, i.e., each task $\tau_i$ has only one critical section guarded by one binary semaphore, and the Definition 8 of the graph $G^*$, i.e., using as many processors as possible. □

**Theorem 1.** *The optimal makespan of the* MS-OCS *makespan problem for* $\boldsymbol{T}$ *on* $M$ *processors is at least*

$$\max\left\{\sum_{\tau_i \in \boldsymbol{T}} \frac{C_{i,1} + C_{i,2} + C_{i,3}}{M}, len(G^*)\right\} \tag{4.1}$$

*where* $C_{i,1}$*,* $C_{i,2}$*, and* $C_{i,3}$ *are WCETs of the three computational segments in OCS task model, and* $G^*$ *is a dependency task graph of* $\boldsymbol{T}$ *that has the minimum critical path length.*

*Proof.* The lower bound $len(G^*)$ comes from Lemma 1 and the lower bound $\sum_{\tau_i \in \mathbf{T}} \frac{C_{i,1}+C_{i,2}+C_{i,3}}{M}$ is due to the pigeon hole principle. □

### 4.3.1 Computational Complexity

The following theorem shows that constructing $G^*$ is $\mathcal{NP}$-hard in the strong sense.

**Theorem 2.** *Constructing a dependency task graph $G^*$ that has the minimum critical path length is $\mathcal{NP}$-hard in the strong sense.*

*Proof.* This theorem is proved by a reduction from the decision version of the scheduling problem $1|r_j|L_{\max}$, i.e., uni-processor non-preemptive scheduling, in which the objective is to minimize the maximum lateness assuming that each job $J_j$ in the given job set $\mathbf{J}$ has its known processing time $p_j \geq 0$, arrival time $r_j \geq 0$, and absolute deadline $d_j$. This problem is $\mathcal{NP}$-hard in the strong sense by a reduction from the 3-Partition problem [LRB77]. Suppose that the decision version of the scheduling problem $1|r_j|L_{\max}$ is to validate whether there exists a schedule in which the finishing time of each job $J_j$ is no less than $d_j$.

Let $H$ be any positive integer greater than $\max_{j \in \mathbf{J}} d_j$. For each job $J_j$ in $\mathbf{J}$, we construct a task $\tau_j$ with OCS task model, where $C_{j,1}$ is set to $r_j$, $C_{j,3}$ is set to $H - d_j$, and $C_{j,2}$ is set to $p_j$. By the setting, $C_{j,1} \geq 0, C_{j,2} \geq 0$, and $C_{j,3} \geq 0$ for every constructed task $\tau_j$. The critical sections of all the constructed tasks are guarded by *only one* binary semaphore. Let the task set constructed above be $\mathbf{T}$. The above input task set $\mathbf{T}$ by definition is a feasible input task set for the one-critical-section task synchronization problem (*MS-OCS*).

We now prove that there is a non-preemptive uni-processor schedule for $\mathbf{J}$ in which all the jobs can meet their deadlines if and only if there is a dependency task graph $G^*$ with a critical path length less than or equal to $H$ for the constructed task set $\mathbf{T}$.

**If part**, i.e., $len(G^*) \leq H$ holds: Without loss of generality, we index the tasks in $\mathbf{T}$ so that the critical section of $\theta_{i,2}$ is the immediate predecessor of the critical section $\theta_{i+1,1}$ in $G^*$, e.g., as in Figure 4.1. Suppose that $G^*(\tau_i)$ is the sub-graph of $G^*$ that consists of only the vertices representing $\left\{\theta_{k,1}, \theta_{k,2}, \theta_{k,3} \mid k = 1, 2, \ldots, i-1\right\} \cup \{\theta_{i,1}, \theta_{i,2}\}$ and the corresponding edges. Let $f_i$ be the longest path in $G^*(\tau_i)$ that *ends at the vertex representing* $\theta_{i,1}$.

By definition, $f_1$ is $C_{1,1}+C_{1,2}$. Moreover, $f_i$ is $\max\{f_{i-1}, C_{i,1}\}+C_{i,2}$ for every task $\tau_i$ in $\mathbf{T}$. Since $len(G^*) \leq H$ and $C_{i,3} = H - d_i$, we know that $f_i + C_{i,3} \leq H \Rightarrow f_i \leq d_i$ for every task $\tau_i$ in $\mathbf{T}$.

We can now construct the uni-processor non-preemptive schedule for $\mathbf{J}$ by following the same execution order. Here, we index the jobs in $\mathbf{J}$ corresponding to $\mathbf{T}$. The finishing time of job $J_1$ is $r_1 + p_1 = C_{1,1} + C_{1,2} = f_1$. The finishing time of job $J_i$ is $\max\{f_{i-1}, r_i\} + p_i = \max\{f_{i-1}, C_{i,1}\} + C_{i,2} = f_i$.

This proves the if part.

**Only-If part**, i.e., there is a uni-processor non-preemptive schedule in which all the deadlines of the jobs in $\mathbf{J}$ are met: The proof for the **if part** can be reverted and the same arguments can be applied. Due to space limitation, details are omitted. □

**Theorem 3.** *The makespan problem with task synchronization for $\boldsymbol{T}$ on $M$ processors is $\mathcal{NP}$-hard in the strong sense even if $M$ is sufficiently large under any scheduling paradigm.*

*Proof.* This follows directly from Theorem 2. Consider $M \geq |\mathbf{T}| + 1$ processors. The if-and-only-if proof in Theorem 2 can be extended by introducing a concrete schedule that executes the two non-critical sections of task $\tau_i$ on processor $i$ and the critical section of task $\tau_i$ on processor $|\mathbf{T}| + 1$.[1] $\square$

Theorem 3 expresses the fundamental difficulty of the multiprocessor synchronization problem and shows that already a simplified version of this problem is $\mathcal{N}P$-hard in the strong sense regardless of the number of processors and the underlying scheduling paradigm. Therefore, the allowance of preemption or migration does not reduce the computational complexity. The fundamental problem is the sequencing of the critical sections, which is independent from the underlying scheduling paradigm. Therefore, no matter what flexibility the scheduling algorithm has (unless aborting and restarting a critical section is allowed), the computational complexity remains $\mathcal{N}P$-hard in the strong sense.

### 4.3.2 Remarks: Bin Packing

Although the focus of this section is the makespan problem in Definition 10 and the schedulability problem in Definition 12, we also state the following theorems to explain the difficulty of the bin packing problem in Definition 11.

**Theorem 4.** *Minimizing the number of processors for a given common deadline of $\boldsymbol{T}$ with task synchronization for $\boldsymbol{T}$ (i.e., Definition 11) is $\mathcal{NP}$-hard in the strong sense under any scheduling paradigm.*

*Proof.* As the decision problem is Definition 12, we reach the conclusion based on Theorem 3. $\square$

**Theorem 5.** *There is no polynomial-time (approximation) algorithm to minimize the number of processors for a given common deadline of $\boldsymbol{T}$ with task synchronization for $\boldsymbol{T}$ under any scheduling paradigm unless $\mathcal{P} = \mathcal{NP}$.*

*Proof.* This is based on Theorems 2 and 3. If such a polynomial-time algorithm exists, then the problem $1|r_j|L_{\max}$ is solvable in polynomial time, which implies $\mathcal{P} = \mathcal{NP}$. $\square$

### 4.3.3 Lower Bounds

The dependency graph approach requires two steps. The following theorem shows that even if both steps are optimized, the resulting schedule for the makespan problem with task synchronization is not optimal and has an asymptotic lower bound 2 of the approximation ratio.

[1] The same statement also holds for using $M = |\mathbf{T}|$ processors, but the proof is more complicated.

**Theorem 6.** *The optimal schedule on $M$ identical processors for the dependency graph $G^*$ that has the minimum critical path length is not optimal for the* MS-OCS *makespan problem and can have an approximation bound of at least*

- $2 - \frac{2}{M} + \frac{1}{M^2}$ *under any scheduling paradigm, and*
- $2 - \frac{1}{M}$ *under partitioned or semi-partitioned scheduling.*

*Proof.* We prove this theorem by providing a concrete input instance as follows:

- Suppose that $M \geq 2$ is a given integer and we have $N = M^2 - M + 1$ tasks.
- We assume a very small positive number $\delta$ and a number $Q$ which is much greater than $\delta$, i.e., $\frac{Q}{MN} \gg \delta > 0$.
- All $N$ tasks have a critical section guarded by the same binary semaphore.
- Task $\tau_1$ has $C_{1,1} = \delta, C_{1,2} = Q - \frac{Q}{M}$, and $C_{1,3} = \frac{Q}{M} + N\delta$
- Task $\tau_i$ has $C_{i,1} = \delta, C_{i,2} = \delta$, and $C_{i,3} = \frac{Q}{M}$ for $i = 2, 3, \ldots, N$.

We need to show that the optimal dependency graph of this input instance in fact leads to the specified bound. Due to the design of the task set, there are only $N$ different dependency graphs, depending on the position of $\tau_1$ in the execution order. Suppose that the critical section of task $\tau_1$ is the $j$-th critical section in the dependency graph. It can be proved that the critical path of this dependency graph is $j\delta + Q + N\delta$. We *sketch* the proof:

- The non-critical section $C_{1,3}$ must be part of the critical path since $C_{1,3} = \frac{Q}{M} + N\delta$, which is greater than any $(N-1)C_{i,2} + C_{i,3}$ for any $i = 2, 3, \ldots, N-1$.
- The longest path that ends at the vertex representing $\theta_{1,2}$ has a) one non-critical section, b) $j-1$ critical sections from $\tau_i$ for $i = 2, 3, \ldots, N$, and c) 1 critical section from task $\tau_1$. Therefore, this length is $\delta + (j-1)\delta + Q - \frac{Q}{M} = j\delta + Q - \frac{Q}{M}$.
- Combining the two scenarios, we reach the conclusion.

Therefore, the dependency graph $G^*$ that has the minimum critical path length is the one where $\tau_1$'s critical section is the first one among the $N$ critical sections. The optimal schedule of the graph $G^*$ on $M$ processors has the following properties:

- Task $\tau_1$ finishes its critical section at time $\delta + Q - \frac{Q}{M}$.
- Before time $\delta + Q - \frac{Q}{M}$, none of the second non-critical sections is executed. Therefore, the makespan of any feasible schedule $S(G^*)$ of $G^*$ on $M$ processors is defined as:

$$\begin{aligned} L(S(G^*)) &\geq \delta + Q - \frac{Q}{M} + \sum_{i=1}^{N} \frac{C_{i,2}}{M} \\ &= \delta + Q - \frac{Q}{M} + \frac{(M^2 - M + 1)\frac{Q}{M} + N\delta}{M} \\ &= \left(1 + \frac{N}{M}\right)\delta + \left(2 - \frac{2}{M} + \frac{1}{M^2}\right)Q \end{aligned}$$

- Moreover, when the scheduling policy is either semi-partitioned or partitioned scheduling, by the pigeon hole principle, at least one processor must execute $\left\lceil \frac{N}{M} \right\rceil$ of the $N$ second non-critical sections no earlier than $\delta + Q - \frac{Q}{M}$. Therefore, the makespan of a feasible semi-partitioned or partitioned schedule $S_p$ of $G^*$ on $M$ processors is

$$\begin{aligned} L(S_p(G^*)) &\geq \delta + Q - \frac{Q}{M} + \left\lceil \frac{N}{M} \right\rceil \frac{Q}{M} \\ &= \delta + Q - \frac{Q}{M} + \left\lceil M - 1 + \frac{1}{M} \right\rceil \frac{Q}{M} \\ &= \delta + Q - \frac{Q}{M} + M\frac{Q}{M} \\ &= \delta + \left(2 - \frac{1}{M}\right) Q \end{aligned}$$

We can have another feasible partitioned schedule $S^*$:

- The first non-critical section $\tau_1$ is executed on processor $M$, and the first non-critical sections of the other $N - 1$ tasks are executed on the first $M - 1$ processors based on list scheduling. All the first non-critical sections finish no later than $M\delta$. Each of the first $M - 1$ processors executes *exactly* $M$ tasks since there are $N - 1 = M(M - 1)$ tasks on these $M - 1$ processors.
- The critical sections of tasks $\tau_N, \tau_{N-1}, \ldots, \tau_1$ are executed sequentially by following the above reversed-index order on the same processor of the corresponding first non-critical sections, starting from time $M\delta$.
- At time $M\delta + N\delta$, all the second non-critical sections of $\tau_2, \ldots, \tau_N$ are eligible to be executed. We execute them in parallel on the first $M - 1$ processors by respecting the partitioned scheduling strategy. That is, each of the first $M - 1$ processors executes *exactly* $M$ tasks with $C_{i,2} = Q/M$. The makespan of these $N - 1$ tasks is $(N + M)\delta + \frac{(N-1)\frac{Q}{M}}{M-1} = (N + M)\delta + Q$.
- At time $M\delta + N\delta$, the critical section of $\tau_1$ starts its execution on processor $M$. Furthermore, at time $(N + M)\delta + Q - \frac{Q}{M}$, the second non-critical section of $\tau_1$ is executed on processor $M$ and it is finished at time $(N + M)\delta + Q + N\delta = (2N + M)\delta + Q$.
- As a result, the makespan of the above partitioned schedule $S^*$ is *exactly* $(2N + M)\delta + Q$.

Therefore, the approximation bound of the optimal task dependency graph approach is at least $\frac{L(S(G^*))}{L(S^*)}$ under any scheduling paradigm and is at least $\frac{L(S_p(G^*))}{L(S^*)}$ under partitioned or semi-partitioned scheduling paradigm. We reach the conclusion by taking $\delta \to 0$.

$\square$

### 4.3.4 Algorithms to Construct $G$

The key to successfully solving the problem is finding $G^*$. Unfortunately, as shown in Theorem 2, finding $G^*$ is $\mathcal{NP}$-hard in the strong sense. However, finding good approximations is possible. We refer to the problem of constructing $G$ as the *dependency-graph construction problem.* Here, instead of presenting new algorithms to find good approximations of $G^*$, we explain how to use the existing algorithms of the scheduling problem $1|r_j|L_{\max}$ to derive good approximations of $G^*$.

It should be first noted that the problem $1|r_j|L_{\max}$ cannot be approximated with a bounded approximation ratio because the optimal schedule may have no lateness at all and any approximation leads to an unbounded approximation ratio. However, a variant of this problem can be easily approximated. This is known as the *delivery-time* model of the problem $1|r_j|L_{\max}$. In this model, each job $J_j$ has its release time $r_j$, processing time $p_j$, and delivery time $q_j \geq 0$. The objective is to minimize the makespan $K$. Therefore, the *effective* deadline $d_j$ of job $J_j$ on the given single machine is $d_j = K - q_j$. Since $K$ is a constant, this is effectively equivalent to the case when $d_j$ is set to $-q_j$.

The delivery-time model of the problem $1|r_j|L_{\max}$ can then be effectively approximated. Moreover, our problem to construct a good dependency graph for $\mathbf{T}$ is indeed equivalent to the delivery-time model of the problem $1|r_j|L_{\max}$. To show such equivalence, Algorithm 1 presents the detailed transformation. For each shared resource $z_k$, suppose that $\mathbf{T}_k$ is the set of tasks that use $z_k$ (Line 1 in Algorithm 1). For each task set $\mathbf{T}_k$, we transform the problem to construct $G_k$ to an equivalent delivery-time model of the problem $1|r_j|L_{\max}$ (Line 3 to Line 8). We then construct the graph $G_k$ based on the derived schedule of an approximation algorithm for the delivery-time model of the problem $1|r_j|L_{\max}$.

**Theorem 7.** *An $\alpha$-approximation algorithm for the delivery-time model of the problem $1|r_j|L_{\max}$ applied in Algorithm 1 guarantees to derive a dependency graph $G$ with $len(G) \leq \alpha \times len(G^*)$.*

*Proof.* This theorem can be proved by a counterpart of the proof of Theorem 2. We will show that Algorithm 1 is in fact an L-reduction (i.e., a reduction that preserves the approximation ratio) from the input task set to the delivery-time model of the problem $1|r_j|L_{\max}$. In this L-reduction, there is no loss of the approximation ratio.

First, by definition, two tasks are considered independent if they do not share any semaphore. Moreover, since the *MS-OCS* problem assumes that a task accesses at most one binary semaphore, a task $\tau_i$ can only appear at most in one $\mathbf{T}_k$ for a certain $k$. Therefore, $len(G^*) = \max_{k=1,2,\ldots,z} len(G_k^*)$.

To show that the reduction preserves the approximation ratio, we only need to prove the one-to-one mapping. One possibility is to prove that a schedule for the input instance of the problem $1|r_j|L_{\max}$ delivers the last result at time $X$ if and only if the corresponding graph $G_k$ constructed by using Lines 9 and 10 in Algorithm 1 has a critical path length $X$. However, this direct correlation is not

**Algorithm 1** Graph Construction Algorithm

```
Input: set $\mathbf{T}$ of $N$ tasks with $Z$ shared binary semaphores;
1:  $\mathbf{T}_k \leftarrow \{\tau_i \mid \sigma(\tau_{i,1}) = z_k\}$ for $k = 1, 2, \ldots, Z$;
2:  for $k \leftarrow 1$ to $Z$ do
3:      $\mathbf{J} \leftarrow \emptyset$;
4:      for each $\tau_i \in \mathbf{T}_k$ do
5:          create a job $J_i$ with $r_i \leftarrow C_{i,1}$, $p_i \leftarrow C_{i,2}$, and $q_i \leftarrow C_{i,3}$, where $q_i$ is the delivery time;
6:          $\mathbf{J} \leftarrow \mathbf{J} \cup \{J_i\}$;
7:      end for
8:      apply an approximation algorithm to derive a non-preemptive schedule $\rho_z$ for the delivery-
        time model of the problem $1|r_j|L_{\max}$ on one machine;
9:      construct the initial dependency graph $G_k$ for $\mathbf{T}_k$, with the directed edges $(\theta_{i,1}, \theta_{i,2})$ and
        $(\theta_{i,2}, \theta_{i,3})$ for every task $\tau_i \in \mathbf{T}_k$;
10:     create a directed edge from $\theta_{i,2}$ to $\theta_{j,2}$ in $G_k$ if job $J_j$ is executed right after (but not
        necessarily consecutively to) job $J_i$ for accessing shared resource $z_k$;
11: end for
12: return $G = G_1 \cup G_2 \cup \ldots \cup G_Z$;
```

possible because a (*technically bad but possible*) schedule for the input instance of the problem $1|r_j|L_{\max}$ can be arbitrarily alerted by inserting useless delays.

Fortunately, for a given permutation to order the $|\mathbf{T}_k|$ tasks in $\mathbf{T}_k$, we can always construct a schedule for the input instance of the problem $1|r_j|L_{\max}$ by respecting the given order and their release times. Such a schedule for the input instance of the problem $1|r_j|L_{\max}$ delivers the last result at time $X$ if and only if the corresponding graph $G_k$ constructed by using Lines 9 and 10 in Algorithm 1 has a critical path length $X$. Moreover, the schedule for one such permutation is optimal for the input instance of the problem $1|r_j|L_{\max}$.

Therefore, the approximation ratio is preserved during the construction of $G_k$. According to the above discussions, $len(G_k) \le \alpha \times len(G_k^*)$. Moreover,

$$\begin{aligned} len(G) &\le \max_{k=1,2,\ldots,z} len(G_k) \\ &\le \alpha \times \max_{k=1,2,\ldots,z} len(G_k^*) = \alpha \times len(G^*) \end{aligned}$$

□

According to Theorem 7 and Algorithm 1, we can simply apply the existing algorithms of the scheduling problem $1|r_j|L_{\max}$ in the delivery-time model to derive $G^*$ by using well-studied branch-and-bound methods, see for example [Car82; MF75; NZ86], or good approximations of $G^*$, see for example [HS92; Pot80]. Here, we will summarize several polynomial-time approximation algorithms. The details can be found in [HS92].

For the delivery-time model of the scheduling problem $1|r_j|L_{\max}$, the **extended Jackson's rule** (**JKS**) is as follows: "Whenever the machine is free and one or more jobs are available for processing, schedule the available job with the largest delivery time," as explained in [HS92].

**Lemma 2.** The extended Jackson's rule (**JKS**) is a polynomial-time 2-approximation algorithm for the dependency-graph construction problem.

*Proof.* This is based on Theorem 7 and the approximation ratio of **JKS** for the problem $1|r_j|L_{\max}$, where the proof can be found in [KIM79]. □

Potts [Pot80] observed some nice properties when the extended Jackson's rule is applied. Suppose that the last delivery is due to a job $J_c$. Let $J_a$ be the earliest scheduled job so that the machine in the problem $1|r_j|L_{\max}$ is not idle between the processing of $J_a$ and $J_c$. The sequence of the jobs that are executed sequentially from $J_a$ to $J_c$ is called a *critical sequence*. By the definition of $J_a$, all jobs in the critical sequence must be released no earlier than the release time $r_a$ of job $J_a$. If the delivery time of any job in the critical sequence is not shorter than the delivery time $q_c$ of $J_c$, then it can be proved that the extended Jackson's rule is optimal for the problem $1|r_j|L_{\max}$. However, if the delivery time $q_b$ of a job $J_b$ in the critical sequence is shorter than the delivery time $q_c$ of $J_c$, the extended Jackson's rule may start a non-preemptive job $J_b$ too early. Such a job $J_b$ that appears last in the critical sequence is termed the *interference job* of the critical sequence.

Potts [Pot80] recommended an *attempt to improve the schedule by ensuring that some interference jobs are executed only after the critical job* $J_c$. This can be achieved by delaying the release time of $J_b$ from $r_b$ to $r'_b = r_c$. This procedure is repeated for at most $n$ iterations and the best schedule among the iterations is returned as the solution.

**Lemma 3.** Potts' iterative process (**Potts**) is a polynomial-time 1.5-approximation algorithm for the dependency-graph construction problem.

*Proof.* This is based on Theorem 7 and the approximation ratio of **Potts** for the problem $1|r_j|L_{\max}$. The proof for the approximation ratio has been provided by Hall and Shmoys in [HS92]. □

Hall and Shmoys [HS92] further improved the approximation ratio to 4/3. This improvement addresses a specific scenario where two jobs, $J_i$ and $J_h$, satisfy the conditions $p_i > P/3$ and $p_h > P/3$, with $P$ being defined as $\sum_{J_j} p_j$. In this context, Potts' algorithm is run for $2n$ iterations.[2]

**Lemma 4.** Algorithm **HS** is a polynomial-time 4/3-approximation algorithm for the dependency-graph construction problem.

*Proof.* This is based on Theorem 7 and the approximation ratio of **HS** for the problem $1|r_j|L_{\max}$. The proof for the approximation ratio has been provided by Hall and Shmoys in [HS92] as well. □

[2]Hall and Shmoys [HS92] further use the concept of forward and inverse problems of the input instance of $1|r_j|L_{\max}$. As they are not highly related, we omit those details.

The algorithm that has the best approximation ratio for the delivery-time model of the problem $1|r_j|L_{\max}$ is a polynomial-time approximation scheme (PTAS) developed by Hall and Shmoys [HS92].

**Lemma 5.** The dependency-graph construction problem admits a polynomial-time approximation scheme (PTAS), i.e., the approximation bound is $1 + \varepsilon$ under the assumption that $\frac{1}{\varepsilon}$ is a constant for any $\varepsilon > 0$.

### 4.3.5 Schedules for Dependency Graphs

This section introduces our heuristic algorithms for scheduling the dependency graph $G$, obtained from Algorithm 1. We aim to determine the upper bound of the makespan and the approximation ratio. Detailed scheduling methodologies can be found in Section 4.7.

We first consider the special case where the number of processors does not restrict the schedule, i.e., $M \geq N$.

**Lemma 6.** Assume a given task set $\mathbf{T}$, $M$ identical processors, and a given dependency graph $G$. The makespan of the schedule which executes task $\tau_i$ on exactly one processor $i$ as early as possible by respecting the precedence constraints defined in $G$ is $len(G)$ if $M \geq N$. By definition, this is a partitioned schedule for the given jobs which is non-preemptive with respect to the sub-jobs.

*Proof.* Since $M \geq N$, all the tasks can start their first non-critical sections at time 0. Therefore, the critical section of task $\tau_i$ arrives exactly at time $C_{i,1}$. Then, the finishing time of the critical section of task $\tau_i$ is exactly the longest path in $G$ that finishes at the vertex representing $C_{i,2}$. Therefore, the makespan of such a schedule is exactly $len(G)$. □

For the rest of this section, we focus on the other case, i.e., when $M < N$. We will heavily utilize the concept of list schedules developed by Graham [Gra69] and extensions of list scheduling to schedule the dependency graph $G$ derived from Section 4.3.4. A list schedule works as follows: Whenever a processor idles and there are sub-jobs eligible to be executed (i.e., all of their predecessors in $G$ have finished), one of the eligible sub-jobs is executed on the processor. When more eligible sub-jobs exist than idle processors, many heuristic strategies exist to decide which sub-jobs should be executed with higher priorities. Graham [Gra69] showed that list schedules can be generated in polynomial time and have an approximation ratio of $2 - \frac{1}{M}$ for the scheduling problem $P|prec|C_{\max}$.

We now explain how to use or extend list schedules to generate semi-partitioned with preemptive or non-preemptive schedules based on $G$. Since the sub-jobs of a task are scheduled individually in list scheduling, a task may migrate among different processors in the generated list schedule, i.e., resulting in a semi-partitioned schedule. However, a sub-job by default is non-preemptive in list schedules.

The following lemma is widely used in the literature for the list schedules developed by Graham [Gra69]. All the existing results of federated scheduling, e.g., [LCA+14; Bar15; Che16], for scheduling sporadic dependent tasks (that are not due to synchronizations) all implicitly or explicitly use this property.

**Lemma 7.** The makespan of a list schedule of a given task dependency graph $G$ for task set $\mathbf{T}$ on $M$ processors is at most $\frac{\sum_{\tau_i \in \mathbf{T}}(C_{i,1}+C_{i,2}+C_{i,3})-len(G)}{M} + len(G)$.

*Proof.* The original proof can be traced back to Theorem 1 by Graham [Gra69] in 1969. We omit the proof here as this is a standard procedure in the proof of list schedules for the scheduling problem $P|prec|C_{\max}$. □

**Lemma 8.** If $len(G) \leq \alpha \times len(G^*)$ for a certain $\alpha \geq 1$, the makespan of a list schedule of the task dependency graph $G$ for task set $\mathbf{T}$ on $M$ processors has an approximation bound of $1 + \alpha - \frac{\alpha}{M}$ if $M < N$.

*Proof.* Since $M < N$, the makespan of a list schedule of $G$, denoted as $L(List(G))$, is

$$\begin{aligned} & L(List(G)) \\ \overset{\text{Lemma 7}}{\leq} & \frac{(\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + C_{i,3}) - len(G)}{M} + len(G) \\ = & \frac{\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + C_{i,3}}{M} + len(G)(1 - \frac{1}{M}) \\ \overset{\text{assumption}}{\leq} & \frac{\sum_{\tau_i \in \mathbf{T}} C_{i,1} + C_{i,2} + C_{i,3}}{M} + \alpha \times len(G^*)(1 - \frac{1}{M}) \\ \overset{\text{Theorem 1}}{\leq} & (1 + \alpha - \frac{\alpha}{M})OPT \end{aligned} \tag{4.2}$$

□

We now conclude the approximation ratio.

**Theorem 8.** *When applying **JKS** ($\alpha = 2$, from Lemma 2), **Potts** ($\alpha = 1.5$, from Lemma 3), **HS** ($\alpha = 4/3$, from Lemma 4), and PTAS ($\alpha = \varepsilon$ for any $\varepsilon > 0$, from Lemma 5) to generate the task dependency graph $G$, the* MS-OCS *Makespan problem admits polynomial-time algorithms to generate a semi-partitioned schedule that has an approximation ratio of*

$$\begin{cases} \alpha & \text{if } M \geq N \\ 1 + \alpha - \frac{\alpha}{M} & \text{if } M < N \end{cases} \tag{4.3}$$

*Proof.* The case for $M < N$ is derived from Lemma 8. The case for $M \geq N$ is based on Lemma 6 and the fact that a partitioned schedule is also a semi-partitioned schedule by definition. □

The default list schedulers are non-preemptive in the sub-job level. However, it may be more efficient if the second non-critical section of a task can be preempted by a critical section. Otherwise, some processors may be busy executing second non-critical sections and a critical section has to wait. As a result, not only this critical section itself but also its successors in $G$ may be unnecessary postponed and therefore increase the makespan. Allowing such preemption in the scheduler design can be achieved as follows:

- In the algorithm, the scheduling decision is made at a time $t$ when there is a sub-job eligible or finished.
- Whenever a sub-job representing a critical section is eligible, it can be assigned to a processor that executes a second non-critical section of a job by preempting that sub-job.

The makespan of the resulting schedule remains at most $\frac{\sum_{\tau_i \in \mathbf{T}}(C_{i,1}+C_{i,2}+C_{i,3})-len(G)}{M}+len(G)$ as in Lemma 7. Therefore, the approximation ratios in Theorem 8 still hold even if preemption of the second non-critical sections is possible.

## 4.4 Extension for Tasks with Multiple Critical Sections

In this section, we explore the synchronization challenges associated with multiprocessor frame-based task systems, especially when leveraging the MCS task model, and elucidate the connections to job shop scheduling. First, we introduce two critical section access patterns. Subsequently, we explain the connection of the *MS-MCS* schedulability problem to the job and flow shop problem by showing different reductions that can be applied for demonstrating different scenarios with respect to their computational complexity. Conclusively, we detail the dependency graph approach for tasks with MCS task model, base on job shop scheduling to construct the dependency graphs.

### 4.4.1 Access Patterns for Critical Sections

For convenience, we categorize the access patterns of critical sections into two types based on the applicable algorithms:

- **Flow-Shop Compatible Access Patterns**: A task set has a pattern where *flow-shop* approaches can be applied. That is, if all tasks access each resource (in a non-nested manner) at most once and a total order $\prec$ in which tasks access the resources can be constructed over all tasks in the set. Hence, a flow-shop pattern means that $\sigma_{i,j'} \prec \sigma_{i,j}$ when $j' < j$ and $\theta_{i,j'}$ and $\theta_{i,j}$ are both critical sections. In such a case, we can assume that the mutex locks are indexed according to the specified total order set. Although the order must be always respected, a task does not need to access all the mutex locks. That is, the access pattern of the mutex locks of a task is a subset of the specified total order set.

- **Job-Shop Compatible Access Patterns** allow tasks to access shared resources multiple times and without any restriction on the order in which resources are accessed.

*Flow-shop compatible access patterns* are a very restrictive special case of the much more general *job-shop compatible access patterns*. We implicitly assume job-shop compatible access patterns if not specified differently, but examine flow-shop compatible access patterns when showing certain complexity results.

### 4.4.2 Reductions from the Jobshop/Flowshop Problem

In Section 4.3, we show that a special case of the *MS-MCS* makespan problem is $\mathcal{NP}$-hard in the strong sense when OCS task model is applied and $M$ is sufficiently large. The *MS-MCS* schedulability problem represents the decision version of the *MS-MCS* makespan problem. We therefore focus on the hardness of the decision version in Definition 14. In this dissertation, we provide reductions from the job/flow shop scheduling problems to different restricted scenarios of the *MS-MCS* schedulability problem. Such reductions are used in Section 4.4.5 for demonstrating the $\mathcal{NP}$-completeness for different scenarios.

We start from the more general scenario under the semi-partitioned scheduling paradigm.

**Theorem 9.** *Under the semi-partitioned scheduling paradigm, there is a polynomial-time reduction from an input instance of the decision version of the job shop scheduling problem $J_Z||C_{\max}$ with $Z$ shops to an input instance of the* MS-MCS *schedulability problem that has $Z$ mutex locks on $M$ processors with $M \geq Z$.*

*Proof.* The proof is based on a polynomial-time reduction from an instance of the job shop scheduling problem $J_Z||C_{\max}$ to the *MS-MCS* schedulability problem. Suppose a given input instance with $N$ jobs of the job shop scheduling problem $J_Z||C_{\max}$.

- We have $Z$ shops with non-preemptive execution.
- A job $i$ is defined by a chain of $\eta_i$ sub-jobs, denoted as $O_{i,1}, O_{i,2}, \ldots, O_{i,\eta_i}$. The processing time of $O_{i,j}$ is $C_{i,j}$.
- These $\eta_i$ operations should be executed in the specified order and $O_{i,j}$ is executed on one of the given $Z$ shops, i.e., on shop $s(O_{i,j})$, where $s(O_{i,j}) \in \{1, 2, \ldots, Z\}$.

The decision version of the job shop scheduling problem is to decide whether there is a non-preemptive schedule whose makespan is no more than a given $D$. The polynomial-time reduction to the *MS-MCS* schedulability problem is as follows:

- There are $M \geq Z$ processors.
- There are $Z$ mutex locks, indexed as $1, 2, \ldots, Z$.

- For a job $i$ of the input instance of the job shop scheduling problem, we create a task $\tau_i$, which is composed of $\eta_i$ computation segments. The execution time of $\theta_{i,j}$ is the same as the processing time of the operation $O_{i,j}$. The mutex lock $\sigma_{i,j}$ used by $\theta_{i,j}$ has the same id as is used by shop $s(O_{i,j})$.
- The deadline of the tasks is $D$ and the period is $T = D$.

We denote the above input instance for the job shop scheduling problem as $I$ (the *MS-MCS* schedulability problem as $I'$, respectively). We show that there exists a feasible schedule $\rho$ for $I$ (in the job shop scheduling problem) if and only if there exists a feasible schedule $\rho'$ for $I'$ (in the *MS-MCS* schedulability problem).[3]

**Only-if part**: Suppose $\rho$ is a feasible schedule for $I$, i.e.,

$$\left(\sum_{m=1}^{Z}\int_0^D [\rho(t,m) = O_{i,j}]dt\right) = C_{i,j}, \forall O_{i,j} \tag{4.4}$$

and $\rho(t,m) \neq O_{i,j}$ for any $t$ and $m$ if $s(O_{i,j}) \neq m$. Since the execution on shops is non-preemptive, if two operations $O_{i,j}$ and $O_{k,\ell}$ are supposed to be executed on a shop $z$, they are executed sequentially in $\rho$. As a result, without any conflict, for $0 \le t \le D$, we can set

$$\rho'(t,m) = \begin{cases} \perp & \text{if } \rho(t,m) = \perp \\ \theta_{i,j} & \text{if } \rho(t,m) = O_{i,j} \end{cases} \tag{4.5}$$

In the schedule $\rho'$, critical sections guarded by the mutex lock $z$ are executed sequentially on the $z$-th processor. Therefore,

$$\left(\sum_{m=1}^{Z}\int_0^D [\rho'(t,m) = \theta_{i,j}]dt\right) = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta} \tag{4.6}$$

and all the constraints of a feasible schedule for $I'$ are met. Such a schedule is a semi-partitioned and non-preemptive schedule (from the sub-job's perspective), which is also a global preemptive schedule (from the job's perspective).

**If part**: Suppose that $\rho'$ is a feasible schedule for $I'$, i.e.,

$$\sum_{m=1}^{M}\int_0^D [\rho'(t,m) = \theta_{i,j}]dt = C_{i,j}, \forall \theta_{i,j} \in \mathbf{\Theta} \tag{4.7}$$

and the schedule $\rho'$ executes any two critical sections $\theta_{i,j}$ and $\theta_{k,\ell}$ with $\sigma_{i,j} = \sigma_{k,\ell} = z$ sequentially. Therefore, for a mutex lock $z \in \{1, 2, \dots, Z\}$, the critical sections guarded by $z$ must be sequentially executed. As a result, without any conflict, for $0 \le t \le D$, we can set

$$\rho(t,z) = \begin{cases} O_{i,j} & \text{if } \exists m \text{ with } \rho'(t,m) = \theta_{i,j} \text{ and } \sigma_{i,j} = z \\ \perp & \text{otherwise} \end{cases} \tag{4.8}$$

[3]Although we do not formally define the schedule function of the job shop scheduling problem, we believe that the context is clear enough by replacing the use of the computation segments with the operations.

However, since we do not put any constraint on the feasible schedule $\rho'$, it is possible that the execution of $O_{i,j}$ on shop $z$ is not continuous. Suppose that $a_{i,j}$ ($f_{i,j}$, respectively) is the first (last, respectively) time instant when $O_{i,j}$ is executed on shop $z$ in $\rho$. Since the schedule $\rho'$ executes any two critical sections $\theta_{i,j}$ and $\theta_{k,\ell}$ sequentially when $\sigma_{i,j} = \sigma_{k,\ell} = z$, we know that for any $t$ between $a_{i,j}$ and $f_{i,j}$ either $\rho(t,z) = O_{i,j}$ or $\rho(t,z) = \bot$. Therefore, we can simply set $\rho(t,z)$ to $O_{i,j}$ for any $t$ in the time interval $[a_{i,j}, a_{i,j} + C_{i,j})$ and set $\rho(t,z)$ to $\bot$ for any $t$ in $[a_{i,j} + C_{i,j}, f_{i,j})$. The resulting schedule $\rho$ executes all the operations non-preemptively on the corresponding shops. Therefore, all the scheduling constraints of the job shop scheduling problem are met and

$$\left(\sum_{m=1}^{Z}\int_0^D [\rho(t,m) = O_{i,j}]dt\right) = C_{i,j}, \forall O_{i,j} \tag{4.9}$$

We note that there is no specific constraint of scheduling imposed by the schedule $\rho'$. □

The proof of Theorem 9 is not valid for the more restrictive partitioned scheduling paradigm, i.e., all the computation segments of a task must be executed on the same processor, since the constructed schedule $\rho'$ in the proof of the only-if part is not a partitioned schedule. Interestingly, if we use an abundant number of processors, i.e., $M \geq N$, then the reduction in Theorem 9 holds for the partitioned scheduling paradigm as well.

**Theorem 10.** *Under the partitioned scheduling paradigm, there is a polynomial-time reduction which reduces from an input instance of the decision version of the job shop scheduling problem $J_Z||C_{\max}$ with $Z$ shops to an input instance of the* MS-MCS *schedulability problem that has $N$ tasks and $Z$ mutex locks on $M$ processors with $M \geq N \geq Z$.*

*Proof.* The proof is identical to the proof of Theorem 9 by ensuring that $\rho'$ constructed in the only-if part in the proof of Theorem 9 can be converted to a partitioned schedule. Instead of applying Equation (4.5), since $M \geq N$, without any conflict, for $0 \leq t \leq D$ and $i = 1, 2, \ldots, N$, we can set

$$\rho'(t,i) = \begin{cases} \bot & \text{if } \nexists m \text{ with } \rho(t,m) = O_{i,j} \\ \theta_{i,j} & \text{if } \exists m \text{ with } \rho(t,m) = O_{i,j} \end{cases} \tag{4.10}$$

Since all computation segments of $\tau_i$ are executed on processor $i$, the schedule $\rho'$ is a partitioned schedule. All the remaining analysis follows the proof of Theorem 9. □

**Theorem 11.** *There is a polynomial-time reduction which reduces from an input instance of the decision version of the flow shop scheduling problem $F_Z||C_{\max}$ with $Z$ flow shops to an input instance of the* MS-MCS *schedulability problem that has $Z$ mutex locks with a flow-shop compatible access pattern. The conditions in Theorems 9 and 10 for different scheduling paradigms with respect to constraint of $M$ remain the same.*

*Proof.* The proof is identical to the proofs of Theorems 9 and 10. The additional condition is to access the $Z$ mutex locks by following the index, starting from 1. □

The above theorems show that the computational complexity of the *MS-MCS* schedulability problem is almost independent from the number of processors (i.e., adding processors may not be helpful) and the underlying scheduling paradigm. The fundamental problem is the sequencing of the critical sections.

### 4.4.3 The DGA Based on Jobshop/Flowshop

The extended dependency graph approach for tasks with the MCS task model also consists of two steps:

- In the first step, a directed *acyclic* graph $G = (V, E)$ is constructed. For each sub-job $\theta_{i,j}$ of task $\tau_i$ in $\mathbf{T}$, we create a vertex in $V$. The sub-job $\theta_{i,j}$ is a predecessor of $\theta_{i,j+1}$ for $j = 1, 2, \ldots, \eta_i - 1$. Suppose that $\mathbf{\Theta}^z$ is the set of computation segments whose critical sections request the shared resource $z$, i.e., $\mathbf{\Theta}^z \leftarrow \{\theta_{i,j} \mid \lambda_{i,j} = 1 \text{ and } \sigma_{i,j} = z\}$. For each $z = 1, 2, \ldots, Z$, the sub-graph of the computation segments in $\mathbf{\Theta}^z$ is a directed chain, which represents the total execution order of these computation segments.
- In the second step, we construct a schedule of $G$ on $M$ processors either globally or partitioned, either preemptive or non-preemptive.

Next, we explain the reduction process from an input instance $I^{MS}$ of the *MS-MCS* makespan problem to an input instance $I^{JS}$ of the job shop scheduling problem $J_{Z+N}||C_{\max}$.

- We create $Z + N$ shops:
  - Shop $z \in \{1, 2, \ldots, Z\}$ is exclusively used to execute critical sections guarded by mutex lock $z$. That is, only critical sections $\theta_{i,j}$ with $\lambda_{i,j} = 1$ and $\sigma_{i,j} = z$ (i.e., $\theta_{i,j} \in \mathbf{\Theta}^z$) can be executed on shop $z$.
  - Shop $Z + i$ is exclusively used to execute non-critical sections of task $\tau_i$. That is, only non-critical sections $\theta_{i,j}$ with $\lambda_{i,j} = 0$ can be executed on shop $Z + i$.
- The operation of each computation segment $\theta_{i,j}$ is transformed to the corresponding shop, and the processing time is the same as the segment's execution time, i.e., $C_{i,j}$. The optimization objective is to minimize the makespan of the generated schedule.

Suppose that $\rho^{JS}$ is a feasible job shop schedule for $I^{JS}$. Since $\rho^{JS}$ is non-preemptive, the operations on a shop are executed sequentially in $\rho^{JS}$. The construction of the dependency graph $G$ sets the precedence constraints of $\mathbf{\Theta}^z$ by following the total order of the execution of the operations on shop $z$, i.e., the shop dedicated for $\mathbf{\Theta}^z$ in $\rho^{JS}$. Once the dependency graph $G$ is constructed, a schedule $\rho^{MS}$ of the original input instance $I^{MS}$ can be generated by applying any scheduling algorithms to schedule $G$, as detailed in Section 4.7.

**Table 4.1:** Example of a frame-based task set consisting of 4 tasks and 2 shared resources. Each task comprises 5 computational segments, of which 2 are critical sections.

| Task | WCETs | | | | | Requested Resource | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $C_{i,1}$ | $C_{i,2}$ | $C_{i,3}$ | $C_{i,4}$ | $C_{i,5}$ | $\sigma(\theta_{i,1})$ | $\sigma(\theta_{i,2})$ | $\sigma(\theta_{i,3})$ | $\sigma(\theta_{i,4})$ | $\sigma(\theta_{i,5})$ |
| $\tau_1$ | 3 | 4 | 1 | 1 | 1 | $\varnothing$ | 1 | $\varnothing$ | 2 | $\varnothing$ |
| $\tau_2$ | 1 | 1 | 2 | 4 | 3 | $\varnothing$ | 1 | $\varnothing$ | 2 | $\varnothing$ |
| $\tau_3$ | 3 | 2 | 2 | 3 | 3 | $\varnothing$ | 2 | $\varnothing$ | 1 | $\varnothing$ |
| $\tau_4$ | 1 | 4 | 1 | 1 | 1 | $\varnothing$ | 2 | $\varnothing$ | 1 | $\varnothing$ |

### 4.4.4 An Example of the DGA Based on Job Shop Scheduling

To illustrate the workflow of the DGA based on job shop scheduling, we provide an example in Table 4.1. Consider a frame-based task set with 4 tasks and 2 shared resources. All tasks have the same period, e.g., $T_i = 25$. Each task is composed of five computational segments: two critical sections and three non-critical sections. These segments within a task must be executed in sequence. Each critical section accesses one of the shared resources, which is protected by mutex locks respectively.

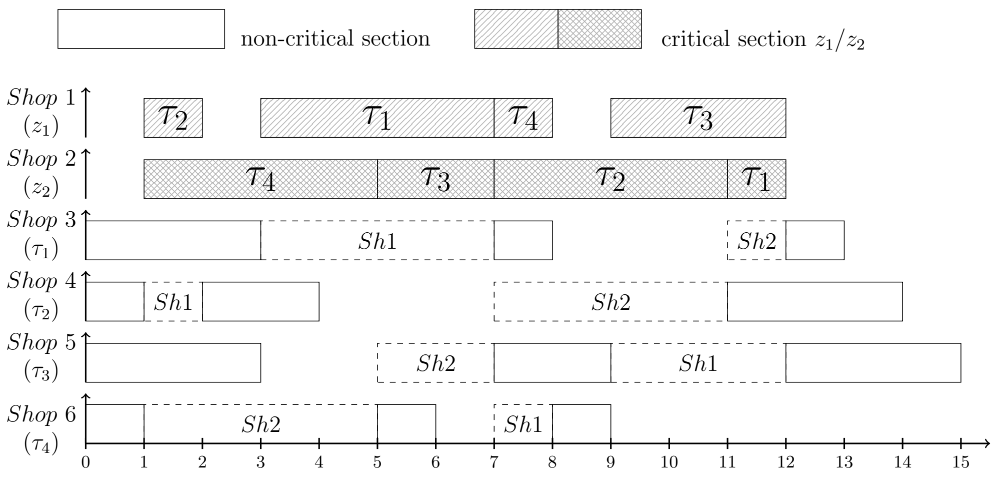
To construct a dependency graph for the task set, we apply job shop scheduling using 6 exclusively assigned shops: Shop 1 and 2 are designated for the critical sections of the two shared resources, while Shop 3 to 6 are reserved for the non-critical sections of tasks $\tau_1$ to $\tau_4$. When a task requires access to a shared resource, its execution is migrated to the appropriate shop (e.g., Shop 1 for resource 1). The input instance for $J_{Z+n}||C_{\max}$ is denoted as $I^{JS}$.

Figure 4.2a presents a job shop schedule for $I^{JS}$. The execution order for shared resources 1 and 2 in Shops 1 and 2, respectively, dictates the precedence constraints depicted in Figure 4.2b. Within this figure, rectangles denote the critical sections, while circles signify non-critical sections. The numbers inside the circles and rectangles indicate the execution times of their respective computational segments. Dashed red directed edges highlight the precedence constraints associated with mutex lock for shared resource $z_1$, whereas dotted blue directed edges represent those for mutex lock for shared resource $z_2$. Additionally, black solid directed edges delineate the internal execution order for each task. Collectively, these directed edges and shapes form the dependency graph.

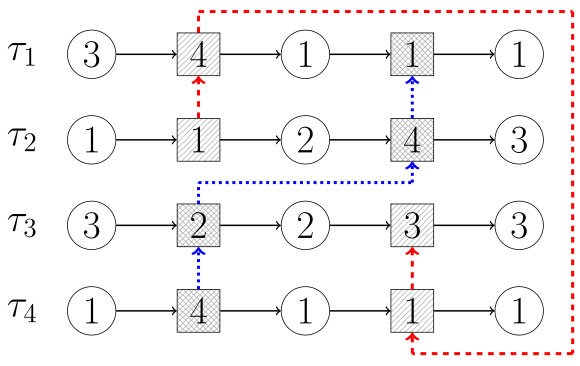
To schedule the generated dependency graph, several scheduling algorithms are discussed in Section 4.7. The detailed schedule of the generated graph by applying the LIST-EDF scheduling algorithm on two processors can be found in Appendix A.1.1.

### 4.4.5 Computational Complexity Analysis

In this subsection, we show the computational complexity of different scenarios.



**(a)** The job shop schedule (with 6 shops denoted as Sh1-6).



**(b)** Dependency graph for a task set with two binary semaphores: execution order for shared resource $z_1$ denoted by red dashed lines and $z_2$ by blue dashed lines.

**Figure 4.2:** An example of the DGA based on job shop scheduling for a framebBased task set with MCS task model.

### Computational Complexity for Small $M$

We can now determine the computational complexity of the *MS-MCS* schedulability problem when $Z \geq 2$ for small $M$. For completeness, we state the following lemma.

**Lemma 9.** The *MS-MCS* schedulability problem is in $\mathcal{NP}$.

*Proof.* Since the feasibility of a given schedule for the *MS-MCS* schedulability problem can be verified in polynomial-time, it is in $\mathcal{NP}$. □

The following four theorems are based on the reductions in Theorem 9 and Theorem 11. In general, even very special cases are $\mathcal{NP}$-complete in the strong sense.

**Theorem 12.** *Under the semi-partitioned scheduling paradigm, the* MS-MCS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = M = 2$.*

*Proof.* The job shop scheduling problem $J_2||C_{\max}$ with 2 shops is $\mathcal{NP}$-complete in the strong sense [LR79]. Together with Theorem 9, we conclude the theorem. □

The *MS-MCS* schedulability problem is also difficult when all computation segments have the same execution time.

**Theorem 13.** *Under the semi-partitioned scheduling paradigm, the* MS-MCS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = M = 3$ and $C_{i,j} = 1$ for any computation segment $\theta_{i,j}$.*

*Proof.* The job shop scheduling problem $J_3|p_{i,j} = 1|C_{\max}$ with unit execution time on 3 shops is $\mathcal{NP}$-complete in the strong sense [LR79]. Together with Theorem 9, we conclude the theorem. □

The following theorem shows that the *MS-MCS* schedulability problem is also difficult when there are just three tasks, three mutex locks, and three processors.

**Theorem 14.** *The* MS-MCS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $N = Z = M = 3$.*

*Proof.* The job shop scheduling problem $J_3|N = 3|C_{\max}$ with 3 jobs (with multiple operations) on 3 shops is $\mathcal{NP}$-complete in the strong sense [SS95]. Together with Theorem 9, we conclude the theorem for semi-partitioned scheduling paradigm.

For the partitioned scheduling paradigm, since there are exactly 3 tasks, 3 processors, and 3 mutex locks, the computational complexity remains the same, as a semi-partitioned schedule can be mapped to a partitioned schedule. □

**Theorem 15.** *Under the semi-partitioned scheduling paradigm, the* MS-MCS *schedulability problem* for flow-shop compatible access patterns *is $\mathcal{NP}$-complete in the strong sense when $Z = M = 3$.*

*Proof.* The flow shop scheduling problem $F_3||C_{\max}$ with 3 shops is $\mathcal{NP}$-complete in the strong sense [GJ79]. Together with Theorem 11, we conclude the theorem. □

### Computational Complexity When $M \geq N$

Section 4.3 shows that a special case of the *MS-MCS* makespan problem is $\mathcal{NP}$-hard in the strong sense when a task has only one critical section and $M$ is sufficiently large. The following theorem shows that the *MS-MCS* schedulability problem is $\mathcal{NP}$-complete when there are only two critical sections per task and the critical sections are with unit execution time.

**Theorem 16.** *The* MS-MCS *schedulability problem is $\mathcal{NP}$-complete in the strong sense when $Z = 1$, $\eta_i \geq 3$ for every $\tau_i \in \boldsymbol{T}$, $C_{i,j} = 1$ for every computation segment $\theta_{i,j}$ with $\lambda_{i,j} = 1$, and $M \geq N$.*

*Proof.* The problem is in $\mathcal{N}P$, since the feasibility of a given schedule can be verified in polynomial-time. Similar to the proof of Theorem 9, we show a polynomial-time reduction from the master-slave scheduling problem with unit execution time on the master [YHL04]. Assume a given input instance with $N$ jobs of the master-slave scheduling problem:

- We assume a sufficient number of slaves, but only one master that can be modeled as a uni-processor.
- A job $i$ has a chain of three sub-jobs, in which the first and third sub-jobs have to be executed on the master and the second sub-job has to be executed on a slave.
- The processing time of the first and third sub-jobs of a job $i$ is 1. The processing time of the second sub-job of a job $i$ is $O_i > 0$.

The decision version of the master-slave scheduling problem is to decide whether there is a schedule whose makespan is no more than a given target $D$, which is $\mathcal{N}P$-complete in the strong sense [YHL04]. The master-slave scheduling problem is equivalent to the uni-processor self-suspension problem with two computation segments and one suspension interval.
The polynomial-time reduction to the *MS-MCS* schedulability problem is as follows:

- There are $M \geq N$ processors.
- There is one mutex lock.
- For a job $i$ of the input instance of the master-slave scheduling problem, we create a task $\tau_i$, which is composed of three computation segments. The execution time $C_{i,1} = C_{i,3}$ and $C_{i,2} = O_i$. Computation segments $\theta_{i,1}$ and $\theta_{i,3}$ are critical sections guarded by the only mutex lock. Computation segment $\theta_{i,2}$ is a non-critical section.
- The deadline of the tasks is $D$ and the period is $T = D$.

It is not difficult to prove that a feasible schedule $\rho$ for the original input of the master-slave scheduling problem exists if and only if there exists a feasible schedule $\rho'$ for the reduced input of the *MS-MCS* schedulability problem. Details are omitted due to space limitation. □

### 4.4.6 Properties of Our Approach

We now prove the equivalence of a schedule of $I^{JS}$ and a directed acyclic graph $G$ for $I^{MS}$.

**Lemma 10.** Suppose that there is a DAG $G$ for $I^{MS}$ whose critical path length is $len(G)$. There is a job shop schedule for $I^{JS}$ whose makespan is $len(G)$.

*Proof.* This lemma is proved by constructing a job shop schedule $\rho^{JS}$ for $I^{JS}$, in which the makespan of $\rho^{JS}$ is $len(G)$. Suppose that the longest path ended at a vertex $\theta_{i,j}$ in $V$ in the directed acyclic graph $G$ is $L_{i,j}$. There are two cases to schedule $\theta_{i,j}$ in $\rho^{JS}$:

- If $\theta_{i,j}$ is a non-critical section, the schedule $\rho^{JS}$ schedules the operation on shop $i + Z$ from time $L_{i,j} - C_{i,j}$ to $L_{i,j}$.
- If $\theta_{i,j}$ is a critical section guarded by mutex lock $z$, the schedule $\rho^{JS}$ schedules the operation on shop $z$ from time $L_{i,j} - C_{i,j}$ to $L_{i,j}$.

The above schedule has a makespan of $len(G)$ by construction. The only thing that has to be proved is that the schedule is a feasible job shop schedule for $I^{JS}$.

Suppose for contradiction that the schedule $\rho^{JS}$ is not a feasible job shop schedule for $I^{JS}$. This is only possible if the schedule $\rho^{JS}$ has a conflicting decision to schedule two operations at the same time $t$ on a shop $z$. There are two cases:

1. $z$ is an exclusively reserved shop for the non-critical sections of a task. This contradicts to the definition of $G$ since the non-critical sections of task $\tau_i$ form a total order in graph $G$.
2. $z$ is a shop for the critical sections guarded by the mutex lock $z$. This contradicts to the definition of $G$ since the critical sections in $\boldsymbol{\Theta}^z$ form a total order in graph $G$.

In both cases, we reach the contradiction. Therefore, $I^{JS}$ is a feasible job shop schedule with a makespan of $len(G)$. □

**Lemma 11.** Suppose that there is a job shop schedule for $I^{JS}$ whose makespan is $\Delta$. Then, there is a directed acyclic graph $G$ for $I^{MS}$ whose critical path length is at most $\Delta$.

*Proof.* This lemma is proved by constructing a graph $G$ for $I$, in which the critical path length of $G$ is at most $\Delta$. By the definition of $G$, the sub-job $\theta_{i,j}$ is a predecessor of $\theta_{i,j+1}$ for $j = 1, 2, \ldots, \eta_i - 1$ for every task $\tau_i$. For the sub-jobs in $\boldsymbol{\Theta}^z$, we define their total order and form a chain in $G$ by following the execution order on shop $z$ in the given schedule $\rho^{JS}$ for $I^{JS}$. Such a graph $G$ must be acyclic; otherwise, the schedule $\rho^{JS}$ is not a valid job shop schedule for $I^{JS}$.

We now prove that the critical path length $len(G)$ of $G$ is no more than $\Delta$. Suppose for contradiction that $len(G) > \Delta$. This critical path of $G$ defines a total order of the execution of the computation segments in the critical path, which follows *exactly* the total order of the operations of a job and a shop in $\rho^{JS}$. Therefore, this contradicts to the fact that the makespan of schedule $\rho^{JS}$ for $I^{JS}$ is $\Delta$. □

Based on Lemmas 10 and 11, we get the following theorem:

**Theorem 17.** *An $\alpha$-approximation algorithm for the job shop scheduling problem $J_{Z+N}||C_{\max}$ can be used to construct a dependency graph $G$ with $len(G) \leq \alpha \times len(G^*)$, where $G^*$ is a dependency graph that has the shortest critical path length for the input instance $I^{MS}$ of the* MS-MCS *makespan problem.*

*Proof.* Suppose that $\Delta^*$ is the optimal makespan for $I^{JS}$. By Lemma 10, we know that $\Delta^* \leq len(G^*)$. By Lemma 11, we know that $\Delta^* \geq len(G^*)$. Therefore, $\Delta^* = len(G^*)$. Suppose that the algorithm derives a solution for $I^{JS}$ with a makespan $\Delta$. By the $\alpha$-approximation for $I^{JS}$ and Lemma 11, we know $\Delta \leq \alpha \times \Delta^*$. Therefore, by Lemma 11 and above discussions, $len(G) \leq \Delta \leq \alpha\Delta^* = \alpha \times len(G^*)$. □

**Lemma 12.** Let $G^*$ be defined as in Theorem 17. The optimal makespan for the input instance $I^{MS}$ of the *MS-MCS* makespan problem is at least

$$\max\left\{\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}, len(G^*)\right\} \tag{4.11}$$

*Proof.* The lower bound $\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}$ is due to the pigeon hole principle. The lower bound $len(G^*)$ is due to the definition with an infinite number of processors. □

**Theorem 18.** *Applying list scheduling for the dependency graph $G$ with $len(G) \leq \alpha \times len(G^*)$ results in a schedule with an approximation ratio of $\alpha + 1$ for the* MS-MCS *makespan problem under semi-partitioned scheduling, where $G^*$ is defined in Theorem 17.*

*Proof.* According to Theorem 1 and Section 4 in [Gra69], by applying list scheduling, the makespan of $I^{MS}$ for the *MS-MCS* makespan problem is at most

$$\begin{aligned} len(G) + \sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M} &\leq \alpha \times len(G^*) + \sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M} \\ &\leq (\alpha + 1) \times \max\left\{\sum_{\tau_i \in \mathbf{T}} \frac{C_i}{M}, len(G^*)\right\} \end{aligned}$$

The resulting schedule is a semi-partitioned schedule since two computation segments of a task can be executed on different processors. By Lemma 12, we conclude the theorem. □

Since the 1950s, job/flow shop scheduling problems have been extensively studied, i.e., in [LLK+93; CPW98]. Although the problems are $\mathcal{NP}$-complete in the strong sense (even for very restrictive cases), algorithms with different properties have been reported in the literature. If time complexity is not a major concern, applying constraint programming as well as mixed integer linear programming (MILP) or branch-and-bound heuristics can derive optimal solutions for the job shop scheduling problem. In such a case, based on Theorem 18, our DGA has an approximation ratio of 2 for the *MS-MCS* makespan problem.

### 4.4.7 Remarks

At first glance, it may seem impractical to reduce the *MS-MCS* makespan problem to another very challenging problem, i.e., job shop scheduling, in the first step of our DGA algorithms. However, an advantage of considering the job shop scheduling problem is that it has been extensively studied in the literature, related results can directly be applied, and commercial tools, like the Google OR-Tools [Goo23], can be utilized, as we did in our evaluation. In addition, due to Lemma 10, constructing a good dependency graph implies a good schedule for $I^{JS}$.

The last $N$ job shops, i.e., shops $Z+1, Z+2, \ldots, Z+N$, in $I^{JS}$, are created just to match the original job shop scheduling problem. From the literature of flow and job shop scheduling, we know that these additional $N$ job shops can be removed by introducing *delay* ($l_{i,j}$ in Section 2.2). If the first computation segment $\theta_{i,1}$ of task $\tau_i$ is a non-critical section, this implies a non-zero release time $r_i$ of task $\tau_i$ in $I^{JS}$.

In our Google OR-Tools implementation for solving $I^{JS}$, the no overlap constraint has to be taken into consideration for both shop and job perspectives. For each shop, it prevents jobs assigned on the same shop from overlapping in time. For each job, it prevents sub-jobs for the same job from overlapping in time. The first constraint can be achieved by applying the `AddNoOverlap` method, by default supported in Google OR-Tools, for each shop. For the second constraint, instead of creating $N+Z$ job shops, we utilize the above concept by creating only $Z$ job shops and adding proper delays between the operations. We configure the start time (denoted as $\theta_{i,j}.start$) of a computation segment based on the finish time (denoted as $\theta_{i,j}.finish$) of an earlier computation segment. For notational brevity, we assign $\theta_{i,1}.start \geq 0$ and $\theta_{i,0}.finish = 0$. For any $j \geq 2$ with $\lambda_{i,j} = 1$:

$$\begin{cases} \theta_{i,j}.start \geq \theta_{i,j-1}.finish & \text{if } \lambda_{i,j-1} \text{ is } 1 \\ \theta_{i,j}.start \geq \theta_{i,j-2}.finish + C_{i,j-1} & \text{if } \lambda_{i,j-1} \text{ is } 0 \end{cases} \tag{4.12}$$

That is, if $\theta_{i,j-1}$ is a non-critical section, the execution time $C_{i,j-1}$ is added directly to the finish time of $\theta_{i,j-2}$; otherwise $\theta_{i,j}$ is started after the finish time of $\theta_{i,j-1}$.

Hence, a proper job shop scheduling problem for $I^{JS}$ is $J_Z|r_j, l_j|C_{\max}$, i.e., scheduling of jobs with release time and delays between operations on $Z$ shops. An $\alpha$-approximation algorithm for the problem $J_Z|r_j, l_j|C_{\max}$ can be used to construct a dependency graph. This problem is not widely studied and only few results can be found in the literature.

For a task system with a *flow-shop compatible access pattern*, i.e., the $Z$ mutex locks have a predefined total order, the instance $I^{JS}$ is in fact a flow shop problem. For a special case with three computation segments per task in which the second segment is a non-critical section, and the first and the third segments are critical sections of mutex locks 1 and 2, respectively, the constructed input $I^{JS}$ is a two-stage flow shop problem with delays, i.e., $F_2|l_j|C_{\max}$. For the problem $F_2|l_j|C_{\max}$, several polynomial-time approximation algorithms are known: Karuno and Nagamochi [KN03] developed a $\frac{11}{6}$-approximation, Ageev [Age07] developed a 1.5 approximation for a special case when $\forall \tau_i : C_{i,1} = C_{i,3}$, and Zhang and van de Velde [ZV10] proposed polynomial-time approximation schemes (PTASes), i.e., $(1+\varepsilon)$-approximation for any $\varepsilon > 0$.

Specifically, Zhang and van de Velde [ZV10] presented PTASes for different settings of the job/flow shop scheduling problems. For any of such scenarios, the approximation ratio of DGA is at most $2+\varepsilon$ for any $\varepsilon > 0$, according to Theorem 18.

## 4.5 Extension for Nested Resource Synchronization

In this section, we expand the dependency graph approach to accommodate frame-based real-time task systems that employ the Nested-MCS task model, where each critical section can request multiple shared resources. We begin by detailing the two core steps of the approach: constructing the dependency graphs and scheduling the generated graphs, for adapting the Nested-MCS task model. Subsequently, we introduce the relevant properties of the extension for Nested-MCS task model. Lastly, we discuss the extended version of the standard nested locking pattern.

### 4.5.1 Step I: Dependency Graph Construction

We construct a directed acyclic graph (DAG) $G = (V, E)$. For each computation segment $\theta_{i,j}$ of all tasks $\tau_i$ in $\mathbf{T}$ a vertex is created and, to ensure the sequential execution of tasks. Since we consider nested resources, each critical section may request multiple resources. Hence, each critical section may have multiple predecessors, depending on the number of resources it requests.

When constructing a dependency graph for nested resource sharing, the process can be formulated using constraint programming. For consistency with Section 4.4, we use the term 'shop' to represent the location to which a computational segment is assigned. In this system, each shared resource is denoted by a distinct shop, represented as $z$. Furthermore, every task $\tau_i$ is allocated to a dedicated shop, denoted as $Z + i$. Therefore, in total $Z + N$ shops are created:

- Shop $z \in \mathbf{Z}$ exclusively executes critical sections guarded by mutex lock $z$, i.e., only if the critical section $\theta_{i,j}$ requests resource $z$ it is executed on shop $z$.
- Shop $Z + i$ is only used to execute non-critical sections $\theta_{i,j}$ of $\tau_i$.

We note that the shops are purely conceptual in order to generate an execution order irrespective of the actual number of processors in our studied problem.

The operation of each computation segment $\theta_{i,j}$ is expressed as a processing on the corresponding shop for the duration of the segment's execution time. To be precise, each task $\tau_i$ is assigned to the shop $Z + i$ for the execution of its non-critical section. Once a task $\tau_i$ has to access a shared resource, the execution of its critical section will be migrated to the assigned shop for the shared resource. Moreover, any feasible solution must satisfy the following four constraints:

**Constraint 1** (No-overlap Constraint)**.** No two tasks (or segments) can be executed on the same shop simultaneously. That is, for any shop at any time point, there is at most one task (or segment) executed on that shop, i.e.,

$$\begin{aligned}\forall z \in \mathbf{Z}, i \neq g : \theta_{i,j}.start \geq \theta_{g,\ell}.finish\ or \\ \theta_{g,\ell}.start \geq \theta_{i,j}.finish\end{aligned} \tag{4.13}$$

**Constraint 2** (Precedence Constraint)**.** For any two computation segments with precedence constraints, e.g., $\theta_{i,j} \prec \theta_{g,\ell}$, the starting time of $\theta_{g,\ell}$ is no earlier than the finishing time of $\theta_{i,j}$ if $\theta_{i,j} \prec \theta_{g,\ell}$, i.e.,

$$\forall z \in \mathbf{Z}' : \ \theta_{g,\ell}.start \geq \theta_{i,j}.finish \quad if \quad \theta_{i,j} \prec \theta_{g,\ell} \tag{4.14}$$

and

$$\forall z \in \mathbf{Z}' : \ \theta_{g,\ell}.start \geq 0 \quad if \quad \theta_{k,\ell} \text{ has no predecessor} \tag{4.15}$$

where $\mathbf{Z}'$ is the conflict nested resource set.

**Constraint 3** (Non-preemption Constraint)**.** On the shops, computation segments execute non-preemptively, ensuring exclusive execution of critical sections and maintaining the sequential execution order of non-critical sections for task $\tau_i$ on shop $Z+i$. That is, if a computation segment $\theta_{i,j}$ with WCET $C_{i,j}$ is scheduled at time $t_0$, the finishing time of $\theta_{i,j}$ has to be $t_0 + C_{i,j}$. Then the time interval $[t_0, t_0 + C_{i,j}]$ is appended to the corresponding shop, i.e.,

$$\forall z \in \mathbf{Z} : \ \theta_{i,j}.finish = \theta_{i,j}.start + C_{i,j} \tag{4.16}$$

**Constraint 4** (All-at-once Constraint)**.** All resources that are requested within a critical section must be assigned to their corresponding shops at the same time, execute for the same amount of time, and finish at the same time. This implies that the construction of the dependency graph transforms a nested locking scheme into an all-at-once locking. That is, the time interval $[t_0, t_0 + C_{i,j}]$ of $\theta_{i,j}$ is appended to all the shops representing for these resources that the segment requests, i.e.,

$$\forall z \in \mathbf{Z}'' : \ if \ \theta_{i,j}.start = t_0, \ then \ \theta_{i,j}.finish = t_0 + C_{i,j} \tag{4.17}$$

where the $\mathbf{Z}''$ is the set of resources that $\theta_{i,j}$ requests. This constraint is similar to gang scheduling [Ous82], i.e., multiple processors are requested simultaneously for a single task.

Consequently, we formulate the dependency graph construction problem using constraint programming. This involves generating a feasible schedule for a frame-based task set across $Z+N$ shops in accordance with Constraints 1 to 4. The optimization objective is to minimize the makespan for the generated schedule, i.e., the latest finishing time of any job on any shop. This is formulated to minimize $\max_{i,j} \theta_{i,j}.finish$. Since the generated schedule is non-preemptive, the computation segments on each shop are executed sequentially. The initial dependency graph $G$, which is given by the tasks internal precedence constraints, is refined by the execution order of the critical sections given by the order of computation segments on the respective shops.

**Table 4.2:** Example of a frame-based task set consisting of 3 tasks and 4 shared resources. Each task comprises 5 computational segments, of which 2 are critical sections for nested resource accesses.

| Task | WCETs | | | | | Requested Resource | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $C_{i,1}$ | $C_{i,2}$ | $C_{i,3}$ | $C_{i,4}$ | $C_{i,5}$ | $\sigma(\theta_{i,1})$ | $\sigma(\theta_{i,2})$ | $\sigma(\theta_{i,3})$ | $\sigma(\theta_{i,4})$ | $\sigma(\theta_{i,5})$ |
| $\tau_1$ | 1 | 2 | 2 | 2 | 4 | $\varnothing$ | $\{1, 2\}$ | $\varnothing$ | $\{2, 3\}$ | $\varnothing$ |
| $\tau_2$ | 2 | 2 | 1 | 5 | 1 | $\varnothing$ | $\{1, 3\}$ | $\varnothing$ | $\{2, 4\}$ | $\varnothing$ |
| $\tau_3$ | 4 | 3 | 4 | 3 | 6 | $\varnothing$ | $\{1, 4\}$ | $\varnothing$ | $\{3, 4\}$ | $\varnothing$ |

### 4.5.2 Step II: Dependency Graph Scheduling

In the second step, the dependency graph $G$ is scheduled on $M$ processors. This scheduling can be either global or partitioned, and either preemptive or non-preemptive. The detailed schedule algorithms are discussed in Section 4.7.

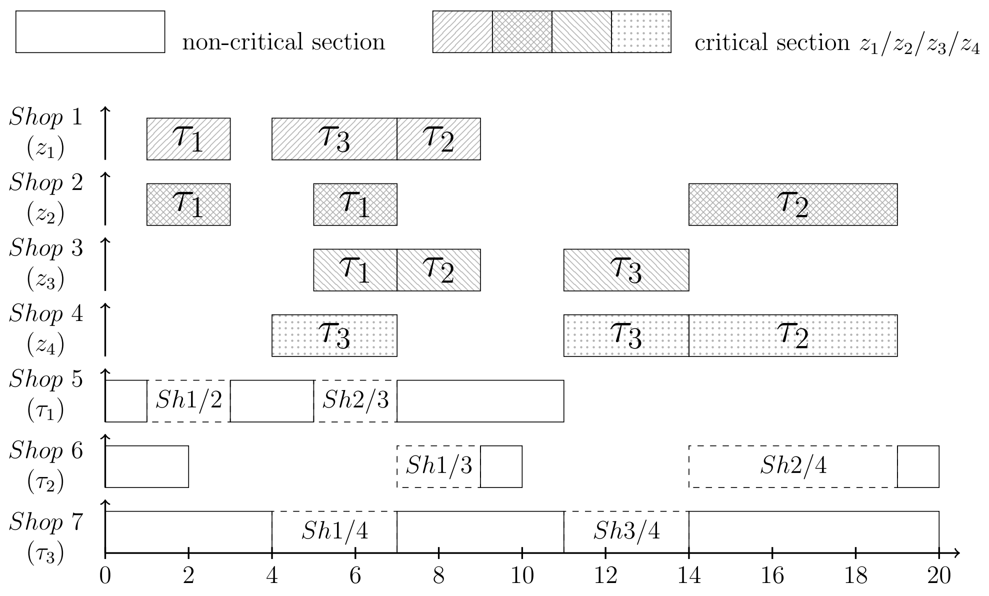
In the all-at-once locking scheme, the executions of critical sections that access at least one identical resource must be mutually exclusive. This implies that if two computation segments have overlapping resource requests, i.e., they both request at least one shared resource, their critical sections on these shared resources must execute sequentially. Critical sections from two distinct tasks can concurrently execute on separate processors if they do not request overlapping resources.

In addition, we enforce that for each job all computation segments execute for the duration of their WCETs. That is, even if a segment's actual execution time is shorter than its WCET, it must remain active on its designated processor until the WCET duration is met. Therefore, the generated schedule for one frame/hyper-period is static and repeated periodically, which avoids the multiprocessor timing anomalies described by Graham [Gra69].

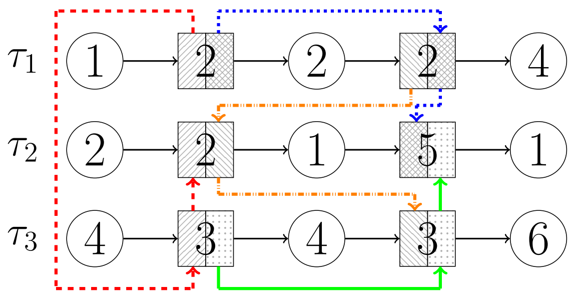
### 4.5.3 An Example of the DGA with All-at-once Locking

To demonstrate the operation of the DGA in handling nested resource accesses using all-at-once locking, we provide an example detailed in Table 4.2. This table presents a frame-based task set comprising 3 tasks and 4 shared resources. All tasks share a common period of $T = 30$ time units. Each task consists of 5 computational segments, including 2 critical sections and 3 non-critical sections. These segments are to be executed sequentially within their respective tasks. Each critical section simultaneously accesses two shared resources, each protected by its own mutex lock.

To construct a dependency graph for the task set, we use constraint programming as detailed in 4.5.1. We assign a total of 7 shops for this task set. Specifically, the first 4 shops are designated for the critical sections associated with the 4 shared resources, while shops 5 to 7 are tasked with the non-critical sections of $\tau_1$, $\tau_2$, and $\tau_3$. Whenever a task requires access to a shared resource, its execution migrates to the corresponding shop. For instance, access to resource $z_1$ directs the task to shop



**(a)** The job shop schedule (with 7 shops denoted as Sh1-7).



**(b)** A dependency graph of a task set with 4 binary semaphores.

**Figure 4.3:** An example of the DGA based on job shop scheduling for a frame-based task set with Nested-MCS task model.

1. Since the *all-at-once* locking is applied, a critical section operates on multiple shops (representing the shared resources) concurrently and does not overlap with the execution of other critical sections on the same shop.

Figure 4.3a shows an optimized schedule for the provided task set, which is only used for generating the dependency graph. The execution order of critical sections from tasks $\tau_1$, $\tau_2$, and $\tau_3$ on shops 1 through 4 determines the precedence constraints within the derived dependency graph. This is further detailed in Figure 4.3b, where dashed red arrows represent precedence constraints for mutex lock of shared resource $z_1$, dotted blue arrows for mutex lock of shared resource $z_2$, densely dashed and dotted orange arrows for mutex lock of shared resource $z_3$, and solid green arrows for mutex lock of shared resource $z_4$.

To schedule the generated dependency graph, several scheduling algorithms are discussed in Section 4.7. The detailed schedule of the generated graph by applying the LIST-EDF scheduling algorithm on two processors can be found in Appendix A.1.1.

### 4.5.4 Properties of Our Approach

In this section, we prove properties of our graph-based approach to schedule task sets with nested resource sharing. Specifically, in this section, we demonstrate that our approach ensures deadlock-free by design and avoids transitive blocking. Furthermore, we show that it has bounded approximation factor.

#### Deadlock-Free

Whenever nested resource requests are considered, the possibility of deadlocks is a major concern. A simple way to avoid deadlocks is to specify an order for all available resources, and to require that nested locks are acquired according to this determined order and therefore avoid *circular waiting*, but it is not obvious how to determine this access order. Alternatively, dynamic group locks (DGLs) [WA13], where a superset of the actual requested resources by a critical section are requested simultaneously, break the *hold-and-wait* condition. We start by introducing the required notation. We define:

$$pre(v_i) : \{v_j \in V | (v_j, v_i) \in E\} \text{ and } v_j \prec v_i \text{ if } v_j \in pre(v_i)$$
$$suc(v_i) : \{v_j \in V | (v_i, v_j) \in E\} \text{ and } v_j \succ v_i \text{ if } v_j \in suc(v_i)$$

to denote precedence constraints and paths in a given DAG.

First, we show that the dependency graph $G$ constructed in 4.5.1 has no cycles.

**Theorem 19.** *The generated dependency graph $G$ that respects the all-at-once locking constraint for nested resources in critical sections from 4.5.1 is a DAG.*

*Proof.* Before the optimization by the constraint programming, each task $\tau_i$ is given by a chain that is composed of the computation segments $\theta_{i,1} \prec \theta_{i,2} \prec \ldots \prec \theta_{i,\eta_i}$ and thus does not contain any cycles. Let any two critical sections of different tasks, e.g., $\theta_{i,j}$ and $\theta_{g,\ell}$, have conflicting nested resource requests, i.e., a subset of resources $\mathbf{Z}' \in \mathbf{Z}$ is requested in both critical sections. Assuming that any generated feasible graph $G$ must respect the constraints $1-4$, if any resource of $\theta_{i,j}$ is granted, then all resources in $\mathbf{Z}'$ are granted to that critical section as well. By the non-preemption constraint (Constraint 3) the resources are held until the completion of that critical section. In consequence, for any two critical sections that have conflicting nested resource requests, either $\theta_{i,j} \prec \theta_{g,\ell}$ or $\theta_{g,\ell} \prec \theta_{i,j}$ must hold. Since the internal order inside a task is respected by the generation as well, the generated dependency graph does not contain any cycles. $\square$

**Theorem 20.** *Any schedule on $M$ processors that respects the precedence constraints of the dependency graph $G$ as described in 4.5.1 is deadlock-free even if resources in critical sections are locked as soon as they are required, i.e., when not enforcing all-at-once locking at run-time.*

*Proof.* We disprove the possibility of *circular waiting* in any schedule that respects the precedence constraints in $G$ by contradiction. Let $S$ be a schedule that respects the precedence constraints in $G$ and at let $\tau_i$ and $\tau_j$ be in the state of *circular waiting* at some point in time. This implies that task $\tau_i$ holds at least one resource $z$ and waits for at least one other resource $z'$ which is held by task $\tau_j$, which in return waits for resource $z$ (held by task $\tau_i$). However, this means that there exist critical sections, e.g., $\theta_{j,k}$ and $\theta_{i,l}$, that are in conflict. By Theorem 19, we know that the set of conflicting critical sections is ordered, i.e., $\theta_{i,l} \prec \theta_{j,k}$ or $\theta_{j,k} \prec \theta_{i,l}$. Therefore if $S$ respects the precedence constraints then any resource from the critical section $\theta_{j,k}$ could not have been scheduled before all resources used in $\theta_{i,j}$ have finished execution. That is, the circular waiting implies a violation of constraints given by $G$, which contradicts the assumption. □

### No Transitive-Blocking

Besides being deadlock-free, the proposed approach also avoids transitive blocking.

**Theorem 21.** *Any schedule on $M$ processors that respects the precedence constraints of the dependency graph $G$ with nested resources sharing as described in 4.5.1 breaks the transitive blocking chain.*

*Proof.* Since the makespan minimization of the constraint program only generates precedence constraints for conflicting critical sections, computation segments that do not conflict can be executed in-parallel with respect to the precedence constraints. □

### Approximation Factor

In this section, we prove that our algorithm has a bounded approximation factor for any variant of list-scheduling when a dependency graph with a bounded approximation factor $\alpha$ compared to an optimal dependency graph is given. We formally define a dependency graph with approximation factor $\alpha$ as follows.

**Definition 16.** A dependency graph $G$ is an $\alpha$-approximation of a dependency graph $G'$ if for some $\alpha \geq 1$ the following constraints are satisfied:

- $vol(G) = vol(G')$
- $len(G) = \alpha \cdot len(G')$

The optimization quality is determined in relation to an optimal dependency graph, defined as a dependency graph with the minimum length.

**Theorem 22.** *LIST-EDF of an $\alpha$-approximated optimal dependency graph $G^*$ is an $(1 + (1 - \frac{1}{M}) \cdot \alpha)$ approximation algorithm for frame-based task sets that use all-at-once locking to access critical sections with nested resources.*

*Proof.* Let $G$ be the dependency graph that is $\alpha$-approximated by an optimal dependency graph $G^*$. Given the properties of list scheduling, the makespan $L(G)$ on $M$ processors is at most

$$L(G) \leq \frac{vol(G)}{M} + (1 - \frac{1}{M}) \cdot \alpha \cdot len(G^*) \tag{4.18}$$

By the fact that an optimal makespan can be no shorter than the length of the longest path and the perfectly distributed workload, we know that $L(G^*) \geq \max\{vol(G^*)/M, len(G^*)\}$. Consequently,

$$L(G) \leq L(G^*) \cdot (1 + (1 - \frac{1}{M}) \cdot \alpha)$$

□

**Corollary 1.** The Dependency Graph Approach offers a $(1 + \alpha)$-approximation for frame-based task sets that use all-at-once locking to access critical sections with nested resources.

### 4.5.5 Extension to Normal Locking Patterns

In this section, we extend the dependency graph approach to nested resource sharing with normal locking patterns, where each critical section can request each resource at most once.

**Definition 17. Normal Locking Pattern:** A critical section is said to follow a *normal locking pattern* when locking nested resources, if the critical section requests a resource only when it is needed. Hence, not all resources are necessarily locked when the execution of the critical section starts.

Within this context, we incorporate the *resource access sequence* into the computational segment definition. This modification is pertinent only to this subsection. It is defined as follows:

**Definition 18. Resource Access Sequence:** The $j$-th resource access sequence of task $\tau_i$ (related to computation segment $\theta_{i,j}$) is a finite sequence of tuples $\sigma'_{i,j} = a_1, a_2, \ldots, a_\ell$, where each $a_i \in (\mathbb{R}_+, \mathcal{P}(\mathbf{Z}))$ and $\mathcal{P}(\mathbf{Z})$ denotes the power set of $\mathbf{Z}$. The first element in each access $a_i$, i.e., $a_i^0$, denotes the execution duration while holding these resources and the second element, i.e., $a_i^1$, denotes the set of locked resources.

If resource $z$ is used in $a_k$ but not in $a_{k+1}$ then $z$ is unlocked after $a_k$ finished. If $z$ is used in $a_k$ and in $a_{k+1}$ then the lock for $z$ remains. If resource $z$ is not used in $a_k$ but in $a_{k+1}$ then $z$ is locked after $a_k$ finished.

**Definition 19. Computation Segment:** The $j$-th computation segment of a task $\tau_i$ is defined by $\theta_{i,j} = (C_{i,j}, \lambda_{i,j}, \sigma_{i,j}, \sigma'_{i,j})$, where $C_{i,j} = \sum_{a_k \in \sigma'_{i,j}} a_k^0$ and denotes the overall execution time of the resource access sequence $\sigma'_{i,j}$. For a non-critical section $\cup_{a_k \in \sigma'_{i,j}} a_k^1 = \emptyset$.

For simplicity in this subsection, $\lambda_{i,j}$ and $\sigma_{i,j}$ are omitted from $\theta_{i,j}$.

Under the *normal nested locking pattern*, a critical section may request shared resources even if it already holds other resource(s).

**Example 1.** If the $j$-th resource access sequence of $\tau_i$ is described by $\sigma'_{i,j} = ((1, \{z_1\}), (2, \{z_1, z_2\}), (1, \{z_1\}))$, the first part that only requests resource $z_1$ can be started once resource $z_1$ is locked, rather than waiting for both resource $z_1$ and $z_2$ to be locked. Please note, if resource $z_2$ is not available when the first time units of execution for resource $z_1$ has finished, the critical section will suspend itself and wait for the release of resource $z_2$. However during this waiting time, no other critical section is able to access resource $z_1$, since it is still locked by $\tau_i$.

The normal locking pattern is well-known for its potential to cause deadlocks, once two tasks request two shared resources in reversed order, since then both the *hold-and-wait* as well as the *circular waiting* condition are fulfilled. To prevent this, a new constraint is designed to replace the Constraint 4 in 4.5.1:

**Constraint 5** (Pattern-respect Constraint)**.** For any computation segment (representing a critical section) that follows the normal locking pattern, the locking pattern has to be respected. For a computation segment $\theta_{i,j}$ with $\sigma'_{i,j} = a_1, a_2, \dots a_\ell$, we have to determine stating times and finishing times for all tuples. We explain this explicitly for the first and second tuple. The starting time on shops in the first tuple is $\theta_{i,j}.start = t_0 \ \forall z \in a_1^1$; the starting time of $\theta_{i,j}$ on shops that are in the second tuple but not in the first tuple is $\theta_{i,j}.start = t_0 + a_1^0 \ \forall z \in (a_2^1 - (a_1^1 \cap a_2^1))$; and the finishing time of $\theta_{i,j}$ on shops that are occupied in the first tuple but not occupied any more in second tuple is $\theta_{i,j}.finish = t_0 + a_1^0 \ \forall z \in (a_1^1 - (a_1^1 \cap a_2^1))$. Such a calculation is applied for all the tuples in $\sigma'_{i,j}$.

**Example 2.** For a computation segment $\sigma'_{i,j} = ((1, \{1\}), (2, \{1, 2\}), (1, \{2\}))$, the pattern is respected if $\theta_{i,j}$ is scheduled on shop $z_1$ at time $t_0$ (by locking of resource 1) and scheduled on shop $z_2$ (by locking resource 2) at time $t_0 + 1$. Combined with the Non-preemption constraint, $\theta_{i,j}$ has to finish its execution at time $t_0 + 3$ on shop $z_1$ (by releasing resource 1), at time $t_0 + 4$ on shop $z_2$ (by releasing resource 2).

**Theorem 23.** *The generated dependency graph $G$ for nested resources with the normal locking pattern in critical sections that respects Constraints 1, 2, 3, and 5 is a directed acyclic graph.*

*Proof.* For each task, the computation segments are still chained, i.e, $\theta_{i,1} \prec \theta_{i,2} \prec \dots \prec \theta_{i,\eta_i}$, and hence the graph does not contain any cycles. Let any two critical sections of different tasks, e.g., $\theta_{i,j}$ and $\theta_{g,\ell}$, have conflicting nested resource requests.

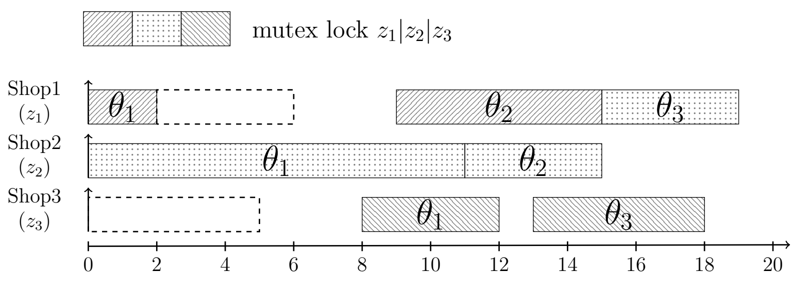


**Figure 4.4:** An example of the dependency graph that with normal locking pattern.

That is, a subset of resources $\mathbf{Z}' \in \mathcal{P}(\mathbf{Z})$ is requested in both critical sections. On any two shops, e.g., $z_a$ and $z_b$, that are requested by $\theta_{i,j}$ or $\theta_{g,\ell}$ simultaneously, the execution time (occupation time) of $\theta_{i,j}$ on two shops has an overlap, i.e., $(\theta_{i,j}^{z_a}.finish - \theta_{i,j}^{z_a}.start) \cap (\theta_{i,j}^{z_b}.finish - \theta_{i,j}^{z_b}.start) \neq \varnothing$ (the same for $\theta_{g,\ell}$), since otherwise it is not a nested resource access. The overlap on both shops for each computation segment can be treated as an *all-at-once* lock. Therefore the execution order of these two segments on both shops are unified. Combined with the non-overlap constraint and non-preemption constraint, the extra execution of each computation segment on both shops follows the same order as the overlapped parts. Therefore, no cycle is included during the generation of graph(s). □

**Example 3.** Consider a task set consisting of 3 shared resources and 3 computation segments:
$\theta_1 = (12, ((2, \{1, 2\}), (6, \{2\}), (3, \{2, 3\}), (1, \{3\})))$,
$\theta_2 = (6, ((2, \{1\}), (4, \{1, 2\})))$, and
$\theta_3 = (6, ((2, \{3\}), (3, \{1, 3\}), (1, \{1\})))$.

A feasible schedule with respect to the constraints 1, 2, 3, and 5 is shown in Figure 4.4. Due to Constraint 5, although shop $z_1$ is already free at time 2, $\theta_2$ starts its execution at time 9, in order to start its execution on shop $z_2$ at time 11 (since $\theta_2$ on shop $z_2$ has to start its execution 2 time units after the execution on shop $z_1$).

Please note, the optimal schedule in Figure 4.4 is that $\theta_2$ is schedule on the dashed slots on shop $z_1$ and $z_3$. Although $\theta_1$ requested 3 resources, it does not request resource 1 and 3 at the same time. Therefore, $\theta_1$ is not considered as accessing nested resource 1 and 3. Hence $\theta_3$ has no nested conflict with $\theta_1$, and its executions on shop $z_1$ and $z_3$ do not necessarily follow the same precedence constraints with regard to the whole computation segment.

**Algorithm 2** Dependency graph construction for periodic tasks

```
Input: Union of task graphs that share a common semaphore $\mathbf{G}_1, \mathbf{G}_2, \dots, \mathbf{G}_Z$ and the
    hyper-period of the task system $H \leftarrow LCM(\mathbf{T})$;
Output: Dependency graphs for all jobs in a hyper-period for each semaphore
    $\mathbf{G}'_1,\ \mathbf{G}'_2, \dots,\ \mathbf{G}'_Z$;
 1: for each $\mathbf{G}_z \in \{\mathbf{G}_1, \mathbf{G}_2, \dots, \mathbf{G}_Z\}$ do
 2:   unroll all jobs that are released in the hyper-period for the tasks in $\mathbf{G}_z$;
 3:   for the $\ell$-th job of task $\tau_i$ in $\mathbf{G}_s$ with $1 \le \ell \le \lceil H/T_i \rceil$ do
 4:     $r_i^\ell \leftarrow (\ell - 1) \cdot T_i + C_{i,1}$;
 5:     $p_i^\ell \leftarrow C_{j,2}$;
 6:     $d_i^\ell \leftarrow (\ell - 1) \cdot T_i + D_i - C_{i,3}$;
 7:   end for
 8:   $\mathbf{G}'_z \leftarrow$ calculate the precedence constraints for the critical sections;
 9:   for the $\ell$-th job of task $\tau_i$ in $\mathbf{G}_s$ with $1 \le \ell \le \lceil H/T_i \rceil$ do
10:     add the precedence constraints $C_{i,1}^\ell \rightarrow C_{i,2}^\ell \rightarrow C_{i,3}^\ell$ to $\mathbf{G}'_z$;
11:   end for
12: end for
13: return $\mathbf{G}'_1,\ \mathbf{G}'_2, \dots,\ \mathbf{G}'_Z$;
```

## 4.6 Extension for Periodic Task Systems

The dependency graph approach, along with its various extensions for different resource access patterns, has been designed for frame-based task systems. To broaden the applicability of this approach, we extend it to periodic tasks by unrolling all jobs within a single hyper-period, constructing corresponding job-level dependency graphs. These graphs can subsequently be utilized for scheduling within each hyper-period.

Suppose that $H$ is the length of the hyper-period, i.e., the least common multiple (LCM) of the periods of all the tasks in the system. For each task $\tau_i$, we create $H/T_i$ jobs of task $\tau_i$. We discuss the approaches of constructing dependency graphs at the job-level for different task models.

In the OCS task model, consider the $\ell$-th job $J_i^\ell$ of task $\tau_i$, the earliest time that the critical section can be executed is $r_i^\ell = (\ell - 1)T_i + C_{i,1}$ and the absolute deadline to finish the critical section must be no later than $d_i^\ell = (\ell - 1)T_i + D_i - C_{i,3}$. Furthermore, for the job's processing time represented by $p_i^\ell$, the value is $C_{i,2}$. The set of the jobs that have to be scheduled over one hyper-period $H$ after the above reduction is thus defined as $\mathbf{J} = \left\{J_i^\ell \mid 1 \le \ell \le H/T_i\right\}$.

Please note that we currently consider only the construction of the dependency graph for the critical sections and assume sufficient resources for the non-critical sections, i.e., all critical sections are executed sequentially on one processor and a dedicated processor is exclusively assigned to each task for the execution of non-critical sections. Therefore, the execution of the non-critical sections is not considered but the related WCET is used for setting up the release time and the absolute deadline.

After this reduction, we can apply any existing algorithm for non-preemptive uni-processor scheduling to construct the precedence constraints for the critical sections in $\mathbf{J}$, i.e., the execution order for the critical sections. Moreover, since the execution of a job has to follow the execution order of $\theta_{i,1}$, $\theta_{i,2}$, and $\theta_{i,3}$, we include these precedence constraints to get the complete dependency graph for binary semaphore $z$.

Algorithm 2 shows the pseudo-code of our approach and Figure 4.5 displays an example for a resulting dependency graph, where $T_1 = 5$, $T_2 = 10$, and $T_3 = 20$. When constructing the precedence constraints for $\mathbf{J}$ (Line 8 in Algorithm 2), any algorithm for non-preemptive uni-processor scheduling can be exploited. For example, when considering the literature of the real-time systems community, Precautious-RM by Nasri et al. [NK14; NBF+14] and the critical time window-based EDF scheduling policy (CW-EDF) by Nasri and Fohler [NF16] can be applied.

Alternatively, classical results for the machine scheduling problem of independent jobs under non-preemptive scheduling can be considered, e.g., the algorithms by Hall and Shmoys [HS92], Potts [Pot80], and Jackson [Jac55] in Section 4.3.4. These algorithms assume knowledge about the release times, processing times, and deadlines of all jobs that have to be considered in the schedule. Please note that the classical scheduling problem assumes a delivery time that a job needs after it finishes its execution instead of a deadline and minimizes the length of the schedule. However, the problem under study can be directly translated by setting the delivery time to $H - d_i^\ell$ or each job and verifying if the schedule's length is less than $H$. In this work, we consider Potts algorithm [Pot80] and the extended Jackson's rule [Jac55] for the construction. While the extended Jackson's rule [Jac55] is similar to non-preemptive EDF, Potts algorithm [Pot80] starts with a non-preemptive EDF schedule that is updated over a (fixed) number of iterations by defining a critical job $J_c$, i.e., a job that misses the deadline, and forcing a job with a later absolute deadline than $J_c$ that is executed before $J_c$ (due to an earlier release time than $J_c$) to execute after $J_c$. This procedure may lead to a non-work-conserving schedule.

For both the MCS and Nested MCS task models, the process is similar. A notable distinction arises when a task requests multiple critical sections, rendering the sub-graph representation for each shared resource non-viable. To address this, we commence with the job unrolling process for every task within a hyper-period, followed by two specific adjustments:

- For the $\ell$-th job of $\tau_i$, set its release time as $(\ell - 1) \cdot T_i$ and its absolute deadline as $(\ell - 1) \cdot T_i + D_i$.
- For each $j$-th sub-job of $\ell$-th job of $\tau_i$, denoted as $J_{i,j}^\ell$, set its release time as $(\ell - 1) \cdot T_i + \sum_{x=1}^{j-1} C_{i,x}$, and its absolute deadline as $(\ell - 1) \cdot T_i + D_i - \sum_{x=j+1}^{\eta_i} C_{i,x}$.

While both model extensions utilize constraint programming for dependency graph formulation, their objectives are different. In frame-based task systems, the goal is to minimize the makespan. However, in periodic task systems, this objective loses its relevance due to the lack of a direct relationship between a job's deadline and

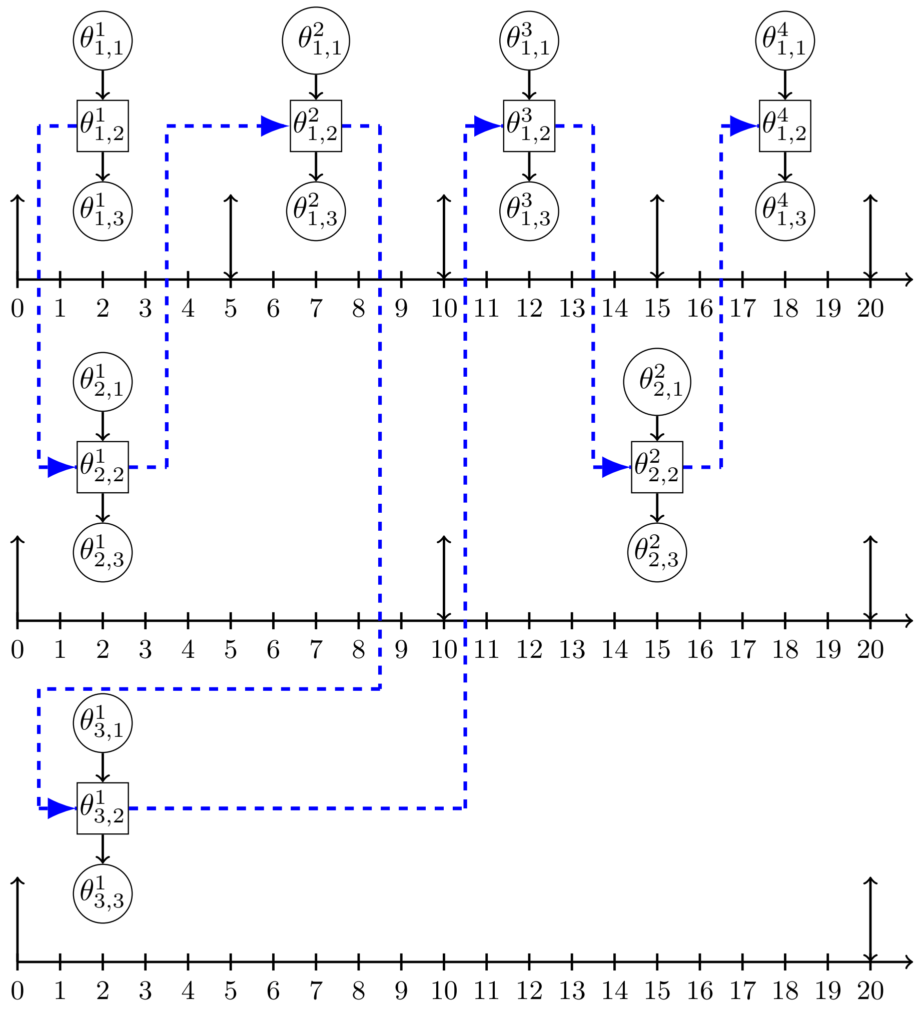


**Figure 4.5:** The dependency graph for three tasks in over one hyper-period. The precedence constraints for the critical sections are detailed by the blue dashed line.

the makespan. The primary aim for constraint programming is to minimize the maximum lateness of all jobs, defined as the difference between a job's finishing time and its deadline.

## 4.7 Algorithms to Schedule Dependency Graphs

In this section, we delineate our heuristic algorithms for scheduling the generated dependency graphs, tailored for various task models within both frame-based and periodic task systems. We consider list and partitioned scheduling using EDF, and address both preemptive and non-preemptive scheduling approaches

### 4.7.1 LIST-EDF Scheduling

We describe how to schedule the unrolled dependency graphs over the hyper-period. According to our notation, each job $J_i^\ell$ has several sub-jobs $J_{i,1}^\ell, J_{i,2}^\ell, \dots, J_{i,\eta_i}^\ell$ that represent the related computational segments $\theta_{i,1}, \theta_{i,2}, \dots, \theta_{i,\eta}$, respectively. The release time of the first sub-job $J_{i,1}^\ell$ is $(\ell-1)T_i$, and the absolute deadline of the last sub-job $J_{i,\eta_i}^\ell$ is $(\ell-1)T_i + D_i$. Regarding the release times of the rest sub-jobs, we initially set the earliest possible time the job may be released based on the WCETs of the other sub-jobs. As for the deadline of the other sub-jobs, we initially assign the latest possible time the sub-job can finish while still allowing schedulability. To be precise, the release time of $J_{i,j}^\ell$ is set to $(\ell-1)\cdot T_i + \sum_{x=1}^{j-1} C_{i,x}$, and the absolute deadline of $J_{i,j}^\ell$ is set to $(\ell-1)\cdot T_i + D_i - \sum_{x=j+1}^{\eta_i} C_{i,x}$. For example, the release time of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + C_{i,1}$ and the release time of $J_{i,3}^\ell$ is set to $(\ell-1)T_i + C_{i,1} + C_{i,2}$. The absolute deadline of $J_{i,2}^\ell$ is set to $(\ell-1)T_i + D_i - \sum_{x=3}^{\eta_i} C_{i,x}$ and the absolute time of $J_{i,1}^\ell$ is set to $(\ell-1)T_i + D_i - \sum_{x=2}^{\eta_i} C_{i,x}$.

For brevity, let $r_{i,j}^\ell$ denote the release time of the sub-job $J_{i,j}^\ell$ and $d_{i,j}^\ell$ as the absolute deadline of $J_{i,j}^\ell$. If the absolute deadline of an immediate predecessor of $J_{i,j}^\ell$, denoted as $IPre(J_{i,j}^\ell)$, is larger than $d_{i,j}^\ell$, the absolute deadline of the immediate predecessor should be reassigned to $d_{i,j}^\ell$ minus the WCET of $J_{i,j}^\ell$. This is a standard procedure for scheduling jobs subject to release dates and precedence constraints. Details can be found in [BLL+83] and an illustrative example is provided in Section 4.7.1.

For the rest of this section, we assume that the absolute deadline assignment is adjusted accordingly so that $d_{i,j}^\ell$ for the sub-job $J_{i,j}^\ell$ is always greater than the absolute deadline of $IPre(J_{i,j}^\ell)$.

After constructing the dependency graphs for a task set, the scheduling problem becomes a classical multiprocessor scheduling problem. Specifically, scheduling **G** on $M$ homogeneous (identical) processors is a special case of the classical scheduling problem $P|prec; r_j|L_{\max}$, i.e., scheduling a set of jobs with specified release times and precedence constraints on $M$ identical processors, minimizing the maximum lateness. Our goal is to avoid any deadline misses. Therefore, a schedule is feasible if $L_{\max} \le 0$, where $L_{\max}$ is defined as $\max_j\{f_j - d_j\}$.

One possible scheduling strategy is to use the List scheduling developed by Graham [Gra69] in combination with earliest-deadline-first scheduling (EDF). The List scheduling method operates as follows: Whenever a processor idles and there are sub-jobs eligible to be executed (i.e., all of their predecessors in the dependency graph have finished), one of the eligible sub-jobs is executed on the processor. If there are more available sub-jobs than processors, we prioritize the sub-jobs that have the earliest absolute deadlines, and if two sub-jobs have the same absolute

**Table 4.3:** The example task set with the (earliest possible) release times and the deadlines, where tasks request the shared resource 1, and the generated dependency graph is in Figure 4.5. The release times and deadlines of the critical sections are changed based on the order and the resulting restrictions in the dependency graph (colored red), which propagates to later releases of the second non-critical section or an earlier deadline of the first non-critical section (colored blue).

| Task | WCETs | | | Other Param. | | | Release Times | | | | Deadlines | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $C_{i,1}$ | $C_{i,2}$ | $C_{i,3}$ | $T_i$ | $D_i$ | $\sigma(\tau_i)$ | $\ell$ | $J_{i,1}^{\ell}$ | $J_{i,2}^{\ell}$ | $J_{i,3}^{\ell}$ | $J_{i,1}^{\ell}$ | $J_{i,2}^{\ell}$ | $J_{i,3}^{\ell}$ |
| $\tau_1$ | 0.2 | 0.6 | 0.2 | 5 | 5 | 1 | 1 | 0 | 0.2 | 0.8 | 4.2 | 4.8 | 5 |
| | | | | | | | 2 | 5 | 5.2 | 5.8 | 5.4 | 6 | 10 |
| | | | | | | | 3 | 10 | 13.8 | 14.4 | 14.2 | 14.8 | 15 |
| | | | | | | | 4 | 15 | 15.2 | 15.8 | 19.2 | 19.8 | 20 |
| $\tau_2$ | 0.2 | 0.6 | 3.2 | 10 | 10 | 1 | 1 | 0 | 0.8 | 1.4 | 4.8 | 5.4 | 10 |
| | | | | | | | 2 | 10 | 14.4 | 15 | 16.2 | 16.8 | 20 |
| $\tau_3$ | 4 | 8 | 5 | 20 | 20 | 1 | 1 | 0 | 5.8 | 13.8 | 6 | 15 | 20 |

**Table 4.4:** The tasks requesting the shared resource 2, with release times and deadlines.

| Task | WCETs | | | Other Param. | | | Release Times | | | | Deadlines | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $C_{i,1}$ | $C_{i,2}$ | $C_{i,3}$ | $T_i$ | $D_i$ | $\sigma(\tau_i)$ | $\ell$ | $J_{i,1}^{\ell}$ | $J_{i,2}^{\ell}$ | $J_{i,3}^{\ell}$ | $J_{i,1}^{\ell}$ | $J_{i,2}^{\ell}$ | $J_{i,3}^{\ell}$ |
| $\tau_4$ | 0.2 | 0.2 | 0.2 | 10 | 10 | 2 | 1 | 0 | 0.2 | 0.4 | 9.6 | 9.8 | 10 |
| | | | | | | | 2 | 10 | 10.2 | 10.4 | 19.6 | 19.8 | 20 |
| $\tau_5$ | 2 | 3 | 2 | 20 | 20 | 2 | 1 | 0 | 2 | 5 | 15 | 18 | 20 |

deadline, the one with the larger remaining workload has a higher priority. This scheduling algorithm is denoted as LIST-EDF.

We note that LIST-EDF in our setting is a preemptive algorithm. Whenever a new (eligible) sub-job has an earlier absolute deadline than an executing sub-job on a processor $m$, this new sub-job can preempt the one that is executing on processor $m$. Such flexibility to allow preemption does not create any problem for the mutual-exclusive constraint of the critical sections guarded by one binary semaphore $z$ because their execution order has been predefined in the dependency graph. Therefore, a critical section guarded by semaphore $z$ can only be preempted by non-critical sections or by critical sections protected by different semaphores.

### An Illustrative Example for LIST-EDF

In this subsection, we illustrate the operation of our algorithm with an example. We consider a task set consisting of the three tasks with OCS task model, as defined in Table 4.3, where all tasks request shared resource 1, and two additional tasks defined in Table 4.4 which request shared resource 2. These five tasks are scheduled on $M = 2$ processors by using LIST-EDF.

Starting with the tasks that request resource 1, the chain of critical sections is depicted by the dependency graph in Figure 4.5. This dependency graph results in an order of $J^1_{1,2}, J^1_{2,2}, J^2_{1,2}, J^1_{3,2}, J^3_{1,2}, J^2_{2,2}, J^4_{1,2}$.

From this order, we calculate the release times and deadlines for all sub-jobs' releases, as shown in Table 4.3. In terms of release times, we consider only the earliest potential release for critical sections and the second non-critical sections as defined by the dependency graph. We assume that there is no early completion of jobs. Please note that the actual release times may be later, depending on the actual schedule.

All first sub-jobs follow a strict periodic release pattern. However, subsequent sub-jobs may have adjusted release times, dependent on the earliest completion times of their predecessors. We make these adjustments sequentially, starting from time 0 and proceeding until the hyper-period's end at time $H$ (or 20 as shown in both Table 4.3 and Table 4.4). Hence, $J^1_{2,2}$ is released at time 0.8 (marked red in Table 4.3) due to the earliest possible finishing time of $J^1_{1,2}$ at 0.8. Therefore, the release of $J^1_{2,3}$ is postponed as well (marked blue here and for all other third sub-jobs that are postponed). The second sub-job of the second release of $\tau_1$ can finish no earlier than at time 5.8, hence the release of $J^1_{3,2}$ is postponed accordingly. Due to the long critical section of task $\tau_3$, the releases of $J^3_{1,2}$ and $J^2_{2,2}$ are postponed to time 13.8 and 14.4 as well.

In Table 4.3, deadlines are set in a reverse sequence, starting from the hyper-period's end and working backward. All the third sub-jobs align their deadlines with the respective period's end. Deadlines for the second sub-jobs derive from the dependency graph. For tasks like $J^4_{1,2}, J^2_{2,2}, J^3_{1,2}$, and $J^1_{3,2}$, the deadlines are directly influenced by their third sub-jobs. The extended duration of $J^1_{3,2}$, however, dictates an earlier deadline for $J^2_{1,2}$ (6 instead of 9.8). This, in turn, results in a 5.4 deadline (as opposed to 6.8) for $J^1_{2,2}$. We highlighted these adjusted deadlines in red for second sub-jobs and in blue for the first sub-jobs in Table 4.3.

For tasks $\tau_4$ and $\tau_5$, listed in Table 4.4, which access resource 2, we deduced the order as $J^1_{4,2}, J^1_{5,2}, J^2_{4,2}$. Given that neither deadlines nor release times undergo adjustments in Table 4.4, we have omitted further details.

The schedule based on global EDF is displayed in Figure 4.6 and considers the deadlines provided in Table 4.3 and Table 4.4. Execution on processor 1 is marked blue while execution on processor 2 is marked red. In addition, the access to the critical sections related to resource 1 and resource 2 is shown with different hatching patterns as detailed in Figure 4.6. We assume that if two tasks with the same deadline compete for a processor, then the sub-job with the larger remaining workload is preferred in the scheduling decision. At time 0 the sub-jobs $J^1_{1,1}$ and $J^1_{2,1}$ are scheduled since their deadlines are 4.2 and 4.8. As soon as $J^1_{2,1}$ is finished at time 0.2, $J^1_{2,2}$ cannot be scheduled since $J^1_{1,2}$ is its predecessor and has not finished

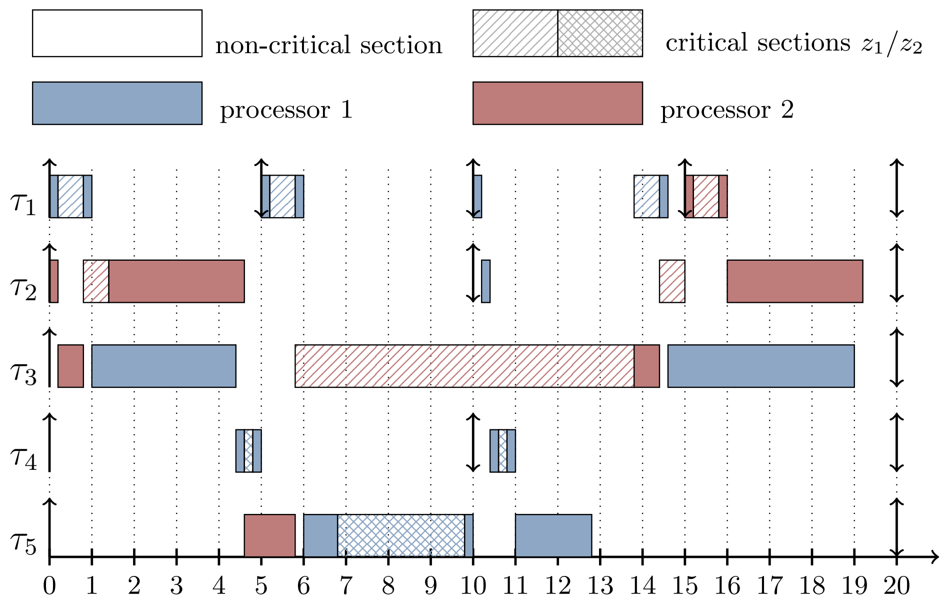


**Figure 4.6:** An Example of LIST-EDF with two shared resources.

yet. Therefore, $J^1_{3,1}$ gets the processor due to its deadline at time 6. At time 0.8, $J^1_{3,1}$ is preempted by $J^1_{2,2}$ which has a shorter deadline, namely 5.4, but $J^1_{3,1}$ is assigned to the other processor at time 1.0. Then, $J^1_{3,1}$ finishes its execution at time 4.4, and $J^1_{4,1}$ is assigned to the processor. After $J^1_{2,3}$ finishes executing at time 4.6, the non-work-conserving behavior of our method becomes evident.

In this scenario, while $J^1_{3,2}$ has an absolute deadline of 14, earlier than $J^1_{5,1}$'s deadline at 15, precedence constraints from the dependency graph dictate that $J^2_{1,2}$ executes before $J^1_{3,2}$. As a result, $J^1_{3,2}$ is not immediately scheduled, and processor 2 is given to $J^1_{5,1}$. Note that if processor 2 would be assigned to $J^1_{3,2}$ at this point in time, then $J^2_{1,2}$ would miss its deadline since $J^1_{3,2}$ would not finish its execution before time 12.6. Under the dependency graph approach this is prevented by postponing $J^1_{3,2}$ until $J^2_{1,2}$ is finished at time 6.

We point out that the release times displayed in Table 4.3 and Table 4.4 are only considered when constructing the dependency graph but do not have any impact on the scheduling of the jobs afterwards. In the actual schedule, the sub-jobs are released based on the actual finishing time of all predecessors which may be much later than the earliest possible release times considered during construction. The reason is that during construction we assume that all non-critical sections can be executed as soon as they are released without considering if sufficient processors are available to schedule all available sub-jobs. For instance, in the actual schedule in Figure 4.6, the second sub-jobs of the first job of $\tau_4$ and $\tau_5$ are released much later than at time 0.2 and 2, which are the release times in Table 4.4. However, not considering the combined workload of the non-critical sections when constructing the dependency graph allows us to construct the graphs for different resources individually and to use algorithms for uni-processor non-preemptive scheduling.

### 4.7.2 Partitioned Scheduling

In a partitioned schedule of the frame-based task set $\mathbf{T}$, all sub-jobs of a task must be executed on the same processor. Therefore, the list scheduling algorithm variant must ensure that once the first computational segment $\theta_{i,1}$ of task $\tau_i$ is executed on a processor, all subsequent computational segments of task $\tau_i$ are tied to the same processor in any generated list schedule. Specifically, the problem is termed $P|prec, tied|C_{\max}$ in Section 2.2. In this section, we describe how to schedule the unrolled dependency graphs of the periodic tasks over their hyper-period by using Partitioned Earliest Deadline First (P-EDF).

After the unrolling process, we determine the earliest possible release time and the absolute deadline for each sub-job, as described in Section 4.7.1. We generate the dependency graphs at the (sub-)job level, within a single hyper-period $H$. For the OCS task model, we employ the approaches from Section 4.3.4, for the MCS task model we use the methodology outlined in Section 4.4.3, and for the Nested-MCS task model, we refer to Section 4.5.1.

Subsequently, the scheduling problem becomes a classical multiprocessor scheduling problem. The detailed partitioned algorithms are discussed in Section 4.7.2. Once the tasks partition is given, tasks are executed on the assigned processor using EDF with the modified deadlines. That is, whenever the processor is idle and there are sub-jobs eligible to be executed, the one with the earliest deadline is executed on that processor. The P-EDF in our setting is a preemptive algorithm. Whenever a new (eligible) sub-job has an earlier absolute deadline than an executing sub-job on the corresponding processor $m$, the new sub-job can preempt the one that is executing on that processor. Such flexibility to allow preemption does not create any problem for the mutual-exclusive constraint of the critical sections guarded by one binary semaphore $z$, because their execution order has been predefined in the dependency graph, i.e., only when the critical section of the predecessor finishes its execution, the successor can release its critical section if the first non-critical section of the successor also has finished its execution. Therefore, a critical section guarded by a semaphore $z$ can only be preempted by either non-critical sections or by critical sections guarded by other semaphores.

In addition, if two or more sub-jobs have the same deadline on the assigned processor, the sub-job with the larger remaining execution time is scheduled.

#### Partitioning Algorithms for Dependency Graphs

Traditionally, tasks are partitioned in a predetermined sequence. In the OCS task model, dependency graphs are constructed for each resource. This implies that jobs inherently have precedence constraints. Therefore, it might be more practical to partition tasks collectively when they access the same resource, rather than individually. However, for MCS and Nested-MCS task models, where a task can access multiple shared resources, resource-level partitioning is infeasible.

This section introduces two partitioning strategies for dependency graphs: a) Federated Scheduling-based Partitioning [LCA+14] partitions tasks sub-graph by sub-graph and is suitable only for the OCS task model; b) Global Worst-fit Heuristic-based Partitioning, where tasks are successively partitioned, ensuring compatibility across all task models.

### Federated Based Partitioning Algorithm

Federated scheduling was proposed by Li et al. [LCA+14] in order to schedule parallel real-time task systems with internal precedence constraints that can be modeled as a directed-acyclic graph (DAG). The primary goal of this scheduling algorithm is to offer approximations that are provably close to an optimal scheduling algorithm, taking into account implementation constraints such as cache hit-rates and runtime memory accesses. Federated scheduling assigns DAGs, specifically those resulting from our dependency graph construction, that require multiple processors (termed *heavy* graphs) exclusively to those processors. Similarly, graphs that can be scheduled on a single processor, termed *light* graphs, are jointly scheduled on the remaining, non-exclusively allocated processors. After this initial partition, the actual scheduling is done by a work-conserving scheduler on the assigned processors. Our Federated scheduling heuristic for DGAs is shown in Algorithm 3.

In the first stage, all graphs are categorized into either the set of *heavy* graphs, or the set of *light* graphs. All graphs with utilizations larger than 100% are *heavy* by default. or the remaining graphs that have utilizations of 100% or less, an EDF schedule is simulated to determine if the graph is *light* or *heavy*, i.e., if it can be feasibly scheduled on a single processor. This test is necessary, even for implicit-deadline tasks with very low resource utilization, a total utilization of 100% might not be achievable on a single processor. For example, consider two tasks, $\tau_a$ and $\tau_b$. Each task comprises three computational segments with WCETs as follows: $\tau_a$ has $(0, \varepsilon, 3)$ and $\tau_b$ has $(0, 6, 0)$, where $\varepsilon > 0$ but is small. Both tasks request the same shared resource during their second computational segment. Let's define the periods and deadlines such that $T_a = D_a = 6$ for $\tau_a$, while $\tau_b$ has an arbitrarily large period and deadline. This task set has a total utilization of $(50 + 2 \cdot \varepsilon)\%$ and a resource utilization of $(2 \cdot \varepsilon)\%$. However, since a critical section cannot be preempted by another critical section for the same resource, the critical section of $\tau_b$ will eventually need to be scheduled between two of $\tau_a$'s critical sections (in two jobs). This results in a total workload of $2 \cdot (\varepsilon + 3) + 6$, which is larger then 12 and, therefore, the critical section of $\tau_b$ is not schedulable in two consecutive periods of $\tau_a$. Nevertheless, the task set can easily be scheduled by assigning both tasks to individual processors.

**Algorithm 3** Federated Based Partitioning Algorithm

```
Input: Task set $\mathbf{T}$, dependency graph $G(\mathbf{T})$, number of processors $M$, number of
    resources $Z$, the total utilization for each graph $U_z$;
 1: Initialize: Schedule $S_z \leftarrow \varnothing$ for each graph, Heavy graphs $G_H \leftarrow \varnothing$, Light graphs
    $G_L \leftarrow \varnothing$, Partition $P_z$ for each graph, Available processor $M_a \leftarrow M$;
 2: Divide the graphs from $G(\mathbf{T})$ to either $G_H$ or $G_L$
 3: Sort the tasks in each $G_z$ decreasingly w.r.t utilizations;
 4: for all $G_h \in G_H$ do
 5:   $m_h \leftarrow \lceil U_{G_h} \rceil$;
 6:   Initialize the temporary schedule $S'_h \leftarrow \varnothing$ for $G_h$;
 7:   while $m_h \leq M_a$ and $S'_h$ is unschedulable; do
 8:     Generate the $P_h$ for $G_h$ on $m_h$ using worst-fit;
 9:     Create $S'_h$ based on $P_h$ and $m_h$ using P-EDF;
10:     if $S'_h$ is unschedulable then
11:       Assign one more processor to $G_h$: $m_h \leftarrow m_h + 1$;
12:     else
13:       $M_a \leftarrow M_a - m_h$;
14:     end if
15:     if $m_h > M_a$ then
16:       Return unschedulable;
17:     end if
18:   end while
19: end for
20: for all $G_l \in G_L$ do
21:   schedule light graphs using greedy algorithm;
22: end for
23: Return task partition;
```

Within both the heavy and light groups, graphs are sorted in descending order based on their utilization. For each *heavy* graph $G_h$, we must determine the minimum number of required processors for feasible scheduling (see lines 4-14 in Algorithm 3). The initial number of processors $m_h$ is given by the ceiling of the utilization of $G_h$. The tasks in $G_h$ are partitioned on $m_h$ processors based on the individual task utilization, using the worst-fit strategy. Once the partition is generated, P-EDF is simulated to verify whether the $m_h$ processors are sufficient to feasibly schedule $G_h$. In case of an infeasible schedule, the number of processors is incremented and the above procedure is repeated until either the generated schedule is feasible or the number of allocated processors exceeds the number of available processors. After feasibly assigning a $G_h$, the number of available processors is updated, i.e., the available processors for the following graphs is set to the number of currently available processors minus the number of processor necessary to schedule $G_h$.

**Algorithm 4** Greedy Algorithm to Partition Light Graphs

```
Input: Set of light graphs $\mathbf{G_L}$ and number of remaining processors $M_a$;
1: Sort light graphs $\mathbf{G_L}$ in non-increasing order with respect to the graph's utilization;
2: Initialize: Partitions $P_1 \leftarrow \varnothing, P_2 \leftarrow \varnothing, \ldots, P_{M_a} \leftarrow \varnothing$;
3: for $i \leftarrow 1$ to $M_a$ do
4:   for each graph $G_\ell$ in $G_L$ do
5:     if $P_i \leftarrow P_i \cup G_\ell$ is not EDF schedulable then
6:       continue;
7:     else
8:       $P_i \leftarrow P_i \cup G_\ell$;
9:       $\mathbf{G_L} \leftarrow \mathbf{G_L} \setminus G_\ell$;
10:    end if
11:  end for
12: end for
13: if All graphs are partitioned, i.e., $\mathbf{G_L}$ is empty; then
14:   return Task partition;
15: else
16:   return Infeasible
17: end if
```

On the remaining processors, the greedy algorithm from Algorithm 4 is used to assign light tasks in descending order based on graph utilization. To maximize processor capacity utilization, we employ a best-fit strategy. Here, the graph in $G_L$ with the highest utilization is first assigned to a new processor. Afterwards, we traverse the remaining graphs in $G_l$ and assign them to the same processor if possible (Line 4 - 9 in Algorithm 4). Whether a graph can be assigned is determined by running EDF on the related processor. Thereafter, we remove all graphs that are assigned to the processor from $G_L$ and continue with the next processor. This process is repeated until either all light graphs are assigned to processors or no remaining processors can accommodate the leftover tasks in $G_L$.

If the graphs in both the *heavy* group and the *light* group can be scheduled feasibly, the corresponding partition is returned.

### Worst-Fit Heuristic

Additionally, Algorithm 5 introduces a worst-fit heuristic, where tasks are individually partitioned. Tasks are first sorted based on a specific strategy and then partitioned onto available processors using a worst-fit approach. That is, each task is allocated to the least utilized processor.

We propose two sorting strategies:

**Algorithm 5** Worst-Fit Based Heuristic

```
Input: Task set $\mathbf{T}$, dependency graph $G(\mathbf{T})$, number of processors $M$;
1: Initialize: Partition $P$
2: Sort all the tasks in $\mathbf{T}$ decreasingly w.r.t the utilizations;
3: for all $\tau_i \in \mathbf{T}$ do
4:     Generate the $P$ on $M$ using worst-fit;
5: end for
6: Create schedule $S$ based on $P$ and $M$ using P-EDF;
7: if $S$ is unschedulable then
8:     Sort the graphs in $G(\mathbf{T})$ decreasingly w.r.t to the $U_z$;
9:     Sort the tasks in $G_z$ decreasingly w.r.t the utilizations;
10:    for all $\tau_i \in \mathbf{T}$ do
11:        Generate the $P'$ on $M$ using worst-fit;
12:    end for
13:    Create new $S'$ based on $P'$ and $M$ using P-EDF;
14:    if $S'$ is schedulable then
15:        Return task partition
16:    else
17:        Return infeasible;
18:    end if
19: else
20:    Return task partition
21: end if
```

1. Sort all tasks in descending order based on their utilizations, irrespective of the resources they request.
2. First, sort the graphs in descending order based on their utilizations. Subsequently, sort tasks in each graph in descending order based on their utilizations.

Our heuristic employs both sorting strategies. If the partition $P$ from the first strategy proves infeasible (meaning the task set cannot be scheduled on $M$ processors using P-EDF), we turn to the second strategy, generating partition $P'$, and then apply P-EDF to validate it. The algorithm only returns 'infeasible' if neither of the aforementioned sorting strategies can produce a schedulable partition. Otherwise, the task set is schedulable and the partition is returned. Again, if a time driven schedule should be created the schedule can be returned as well.

Unlike the federated scheduling approach, in this heuristic, tasks that share the same resource can be partitioned across all available processors. In other words, all the $M$ processors might host tasks that share a common resource.

### An Example for P-EDF

To further explain the workflow, we provide an example to demonstrate how our algorithm works. We consider the same task set as stated in Section 4.7.1. From the previous discussions, we can deduce the earliest potential release times and deadlines for all sub-jobs across every task. This is illustrated in Table 4.3, which presents tasks $\tau_1$, $\tau_2$, and $\tau_3$ accessing shared resource 1. Meanwhile, Table 4.4 showcases tasks $\tau_4$ and $\tau_5$, which interact with shared resource 2. These five tasks are scheduled on $M = 2$ processors, the partition is defined by applying the worst-fit based algorithm in Section 4.7.2. As a result, $\tau_3$ and $\tau_4$ are assigned to processor 1, and $\tau_1$, $\tau_2$, and $\tau_5$ are assigned to processor 2. Afterwards, the P-EDF algorithm is applied to schedule the tasks accordingly.

The schedule based on partitioned EDF is displayed in Figure 4.7. Execution on processor 1 is marked in blue while execution on processor 2 is marked in red. In addition, the access to the critical sections related to resource 1 and resource 2 are shown with different hatching patterns as detailed in Figure 4.7. Recall that we assume that the sub-job with the larger remaining workload is preferred in the scheduling decision if two tasks with the same deadline compete for a processor.

Due to the high utilization of $\tau_3$, only two tasks, i.e., $\tau_3$ and $\tau_4$ are assigned on processor 1. The remaining three tasks, i.e., $\tau_1$, $\tau_2$, and $\tau_5$ are assigned on processor 2. Several properties of our P-EDF schedule can be observed in the example:

- Partitioned: All the jobs of one task are assigned to the same processor, so one row only has one color.
- Earliest Deadline First: At time 0 the sub-jobs $J_{1,1}^1$ and $J_{3,1}^1$ are scheduled due to their deadlines of 4.2 and 6, which are earlier than for the other sub-jobs released on the related processors.
- Preemptive schedule: At time 10 the critical section of resource 2 of task $\tau_5$ is preempted by the non-critical section of the third job of $\tau_1$.
- Larger remaining execution time first: After $J_{4,2}^2$ finished its execution, $J_{3,3}^1$ resumes on processor 1. Although $J_{3,3}^1$ has the same deadline as $J_{4,3}^2$, it has a larger remaining execution time, thus $J_{3,3}^1$ has higher priority to be executed.
- Precedence constraints: At time 5, processor 1 is idle and $J_{3,1}^1$ has finished its execution, in a work-conserving schedule, $J_{3,2}^1$ would start the execution of its critical section. However, due to the precedence constraints, $J_{3,2}^1$ cannot be executed until $J_{1,2}^2$ has finished its critical section for resource 1 at time 5.8.

This example demonstrates that our proposed approach is able to schedule a task set with relatively high total utilization, i.e., the total utilization of 186% on two processors.

Comparing the actual schedule in Figure 4.7 with the earliest release times from Table 4.3 and Table 4.4, we see clear differences in release times during dependency graph construction. For instance, in Figure 4.7's actual schedule, the second sub-jobs of the first jobs of $\tau_4$ and $\tau_5$, namely $J_{4,2}^1$ and $J_{5,2}^1$, are released later than the

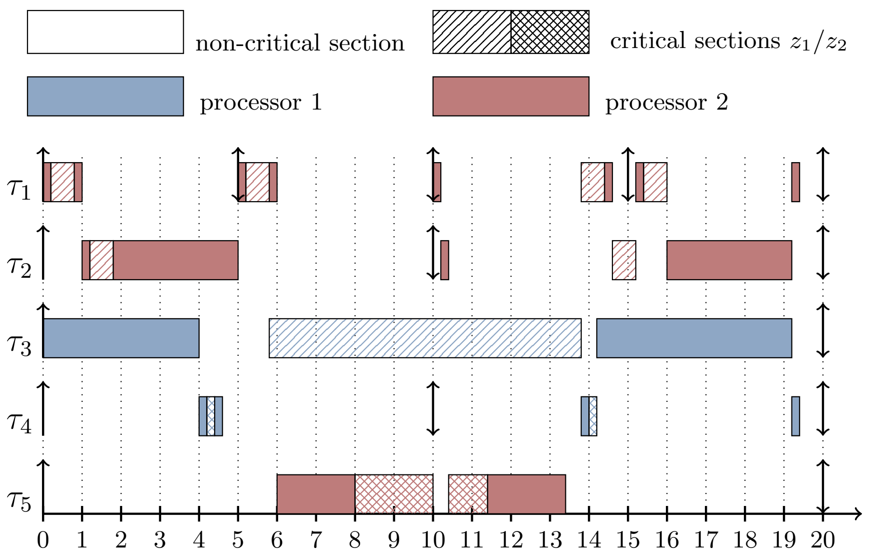


**Figure 4.7:** An example of P-EDF with two shared resources.

expected times of 0.2 and 2. These expected times are listed as the earliest possible in Table 4.4.

### 4.7.3 Timing Anomaly

Graham [Gra69] demonstrated that the list scheduling can suffer from multiprocessor timing anomalies. Specifically, the reduction of the execution time of a sub-job can lead to longer response times of other sub-jobs. We have not proved the absence of multiprocessor timing anomalies in LIST-EDF and P-EDF. Thus, any schedule derived from LIST-EDF and P-EDF must be applied *statically* or offline. One option is to apply table-driving scheduling to ensure the repetitive schedule in every hyper-period. Another is to enforce the actual execution time of each sub-job to be the same as its WCET. Since the scheduling algorithm is deterministic, the schedule is always repeated. The implementation issues for both options are discussed in Section 5.3.1.

Additionally, other options exist, such as carefully reclaiming unused time (slack) without introducing timing anomalies, as seen in [ZMC01]. However, this approach, which involves additional runtime overhead, is out of the scope of this dissertation.

## 4.8 Experimental Evaluation

In this section, we detail our numerical evaluation of the proposed approaches across various task systems, resource access patterns, and configurations. Firstly, we outline the evaluation setups and all assessed approaches, including the newly proposed

DGA and its extensions, as well as the state-of-the-art approaches for the dedicated task systems. Secondly, we focus on the makespan evaluation for OCS task model with different graph construction strategies.Thirdly, we present the results, focusing on schedulability across various task models and configurations.

### 4.8.1 Evaluation Setup

We conducted evaluations for $M = 4$, 8, and 16 processors. Based on the value of $M$, we generate randomized task sets with $10 \times M$ tasks each. The number of shared resources (binary semaphores) $Z$ was either 4, 8, or 16. The utilization of each task $\tau_i$ is denoted as $U_i = \frac{C_i}{T_i}$, hence the execution time for each task is $C_i = U_i \times T_i$. We generated synthetic task sets with total utilization level, i.e., $\sum_{\tau_i \in \mathbf{T}} U_i$, from 0 to $100\% \times M$ in steps of $5\% \times M$ by applying the DRS algorithm, enforcing that $U_i \leq 0.5$ for each task $\tau_i$.

We generated two types of task sets: frame-based and periodic. In frame-based task sets, every task has a shared period and a relative deadline, defined as $\forall \tau_i : T_i = D_i = 1$. For periodic task sets, we randomly selected task periods $T_i$ from a subset of semi-harmonic periods – specifically, $T_i \in 1, 2, 5, 10$ ms. These periods align with those often employed in automotive systems [HDK+17; KZH15; TEH+16; BUC+17]. We restricted the period range to create task sets with a high utilization of the critical sections. These are otherwise by default not schedulable, since critical sections would be longer than the smallest period.

#### Configurations for Non-nested Resource Accesses

For the OCS task model, each task $\tau_i$ accesses the shared resource once. For the MCS model, this access ranges between 2 and 5 times, i.e., $\sum \lambda_{i,j} \in 1, [2, 5]$. The total length of the critical sections $\sum_{\lambda_{i,j}=1} C_{i,j}$ is a fraction of the total execution time $C_i$ of task $\tau_i$, depended on $H \in \{[1\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$. In classical real-time systems settings, the utilization of each task's critical sections is relatively low. However, with the surge in computation demand in real-time systems, especially with applications like machine learning, adopted accelerators, such as GPUs, function as traditional shared resources. These critical sections access accelerators have high utilization. This informs our choice for the spectrum of $H$ settings. The total length of critical sections and non-critical sections are split into dedicated segments by applying DRS separately. Each critical section accesses a random shared resource from those available. For task $\tau_i$, the number of critical sections $Num_{cs}$ equals to $\sum \lambda_{i,j}$, and the number of non-critical sections $Num_{ncs} = Num_{cs} + 1$. In the end, the generated non-critical sections and critical sections are combined in pairs, and the last segment is the last non-critical section. We evaluated all resulting 27 combinations of $M$, $Z$, and $H$.

### Configurations for Nested Resource Accesses

In scenarios involving nested resource sharing, a single critical section may access multiple shared resources simultaneously. To address these nested resource accesses, we introduced two additional configurations specific to them:

- The nesting depth, denoted as $d$, can range from 2 to 4. This implies that a critical section may access up to either 2 or 4 shared resources simultaneously.
- The probability $q$ of a critical section requesting nested resource accesses is selected from the set $\{10\%, 30\%, 50\%\}$.

## 4.8.2 Configurations of DGA

For generating the dependency graph, we used the following methods:

- JKS: The extended Jackson's rule [KIM79] is only valid for OCS task systems.
- POTTS: The potts algorithm [Pot80] is only valid for OCS task systems.[4]
- JS: The method in Section 4.4 for MCS task systems, and 4.5 for nested resource accesses, with the objective to minimize the makespan for frame based task systems and minimize the maximum lateness for periodic task systems. We utilized the constraint programming approach provided in the Google OR-Tools [Goo23] to solve the job shop scheduling problem.

Our proposed scheduling algorithms are named by combining the following elements sequentially:

1. LIST-EDF/P-EDF: We scheduled the generated graph using either the LIST-EDF or the partitioned EDF (P-EDF). Both methods modify the deadlines according to the rule introduced by Baker et al. [BLL+83].
2. WF/FED: For P-EDF, we considered two partitioned algorithms: a) a *worst-fit* partitioning algorithm with respect to the task's utilization $U_i$ (Section 4.7.2); and b) a *federated*-based partitioning algorithm (Section 4.7.2).
3. P/NP: preemptive or non-preemptive for critical sections.

## 4.8.3 Other Considered Algorithms

We also compare our approach with the following protocols for non-nested resource synchronization, regarding their schedulability by applying the publicly available tool SET-MRTS in [Che18] with the same naming:

- ROP-PCP [HYC16]: The Resource Oriented Partitioned PCP binds the resources on dedicated processors and schedules tasks using semi-partitioned PCP. For OCS task systems, two variants with release enforcement are considered, i.e., the ROP under fixed-priority scheduling (ROP-FP) and the ROP under dynamic-priority scheduling (ROP-EDF) [BCH+17].

[4]We did not implement Lemma 5 due to the complexity issue. Algorithm HS in general has similar performance to POTTS.

- GS-MSRP [WB13b]: The Greedy Slacker (GS) partitioning heuristic for spin-based locking protocol MSRP [GLN01], using Audsley's Optimal Priority Assignment [Aud91] for priority assignment.
- LP-GFP-PIP: Linear programming (LP) analysis for global FP scheduling using the PIP [EA09].
- LP-PFP-DPCP [Bra13]: DPCP [RSL88] with a Worst-Fit-Decreasing (WFD) task assignment strategy [Bra13]. The analysis is based on a LP.
- LP-PFP-MPCP [Bra13]: MPCP [Raj90] with a WFD task assignment strategy as proposed in [Bra13]. The analysis is based on a LP.
- LP-GFP-FMLP [BLB+07]: FMLP [BLB+07] for global FP scheduling with a LP analysis.

For nested resource synchronization, the state-of-the-art methods are evaluated, namely the Concurrency Group Locking Protocol (CGLP) [NAG+19] and the Uniform Contention-sensitive Real-time Nested Locking Protocol (C-RNLP) [JWA15].

- CGLP-G: CGLP [NAG+19] where concurrency groups are assigned by a greedy algorithm.
- CGLP-N: CGLP [NAG+19] where concurrency groups are assigned by minimizing the number of groups.
- UC-RNLP: C-RNLP [JWA15], which grants resource access to sets of requests, and the sets are determined dynamically during the run time.
- GC-RNLP: General C-RNLP [JWA15], that grants resource access contention-sensitively on a per-request basis.

Please note that while the aforementioned methods support concurrent *read* resource accesses, we only evaluated mutually exclusive *write* resource accesses.

### 4.8.4 Evaluation Results for Makespan

We initiated our investigation into the multiprocessor resource synchronization problem by focusing on the *MS-OCS* makespan problem. Consequently, we evaluated the makespan of different dependency graph approaches along with different scheduling algorithms. For a generated task set $\mathbf{T}$ with total utilization $100\% \times M$, we calculated a lower bound $LB$ on the optimal makespan based on Equation (4.1). Since deriving $len(G^*)$ is computationally expensive, we used $\min_{\tau_i \in \mathbf{T}} C_{i,1} + \min_{\tau_i \in \mathbf{T}} C_{i,2} + \max_{k=1,\ldots,z} CriticalSum_k$ as a safe approximation for $len(G^*)$, where $CriticalSum_k$ is the summation of the lengths of the critical sections that request shared resource $z_k$. If the relative deadline of the task set is less than $LB$, the task set is not schedulable by any algorithm. We compare the performance of different algorithms according to the *acceptance ratio* by setting the relative deadline $D = T$ in the range of $[LB, 1.8LB]$. We consider both JKS and POTTS for constructing the dependency graphs. Both the semi-partitioned (as discussed in Section 4.3.5) and the partitioned scheduling algorithms were applied

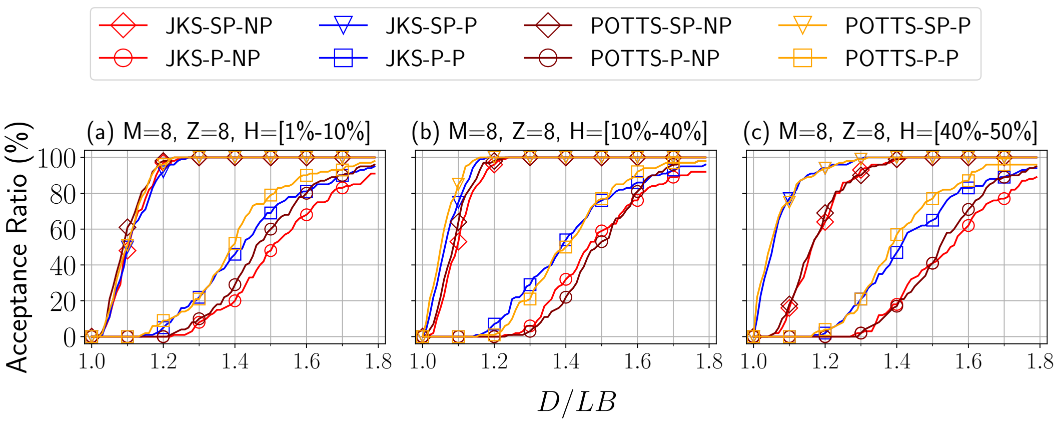


**Figure 4.8:** The makespan for frame-based OCS task systems on 8 processors with 8 shared resources, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

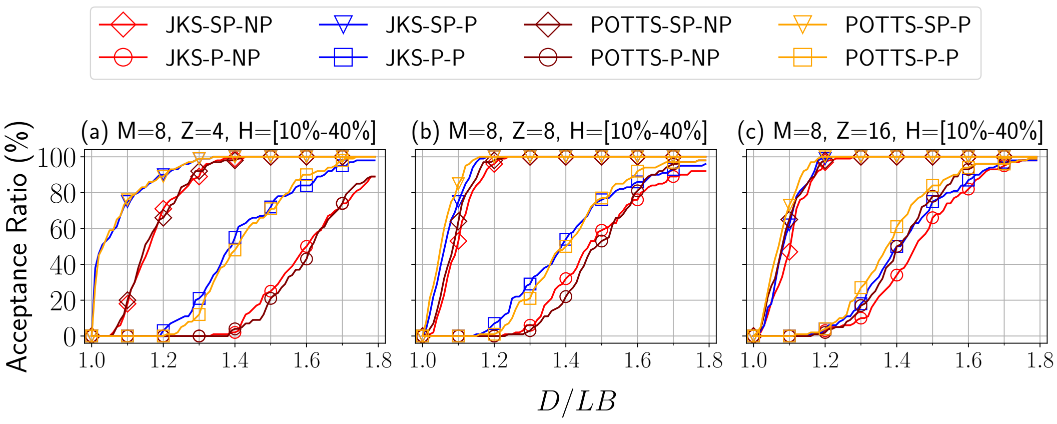


**Figure 4.9:** The makespan for frame-based OCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

and are denoted as SP and P, respectively. Here, we applied a simple heuristic task partitioning. All the initial non-critical sections were scheduled using list scheduling, and they precede any of the critical sections. Once the first non-critical section $\theta_{i,1}$ of task $\tau_i$ is assigned on a processor, the remaining execution of task $\tau_i$ is forced to be executed on that processor. [5] Please note that in our makespan evaluation, we did not apply the scheduling algorithms from Section 4.7. The primary aim of makespan evaluation was to demonstrate the *MS-OCS*makespan problem, its

[5]We also considered a task partitioning algorithm that is based on [SGW+17], where BFS* algorithm (an extension of the breadth-first-scheduling algorithm) is applied to generate the multiprocessor schedule for the dependency graphs. In our experiments regarding partitioned scheduling, the simple heuristic consistently performed better. All the presented results for partitioned scheduling are therefore based on the simple heuristic.

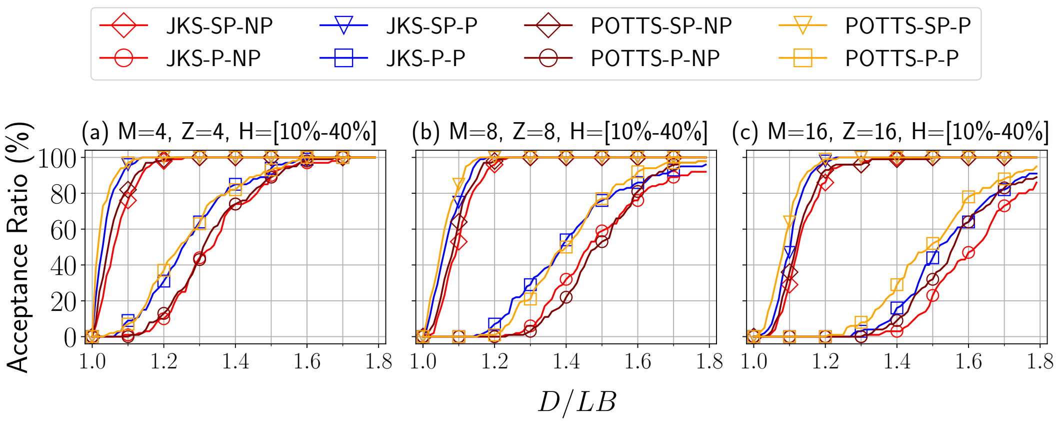


**Figure 4.10:** The makespan for frame-based OCS task systems with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of processor and the number of available shared resources simultaneously, i.e., $M = Z \in \{4, 8, 16\}$.

approximation ratio, and to compare various graph construction algorithms. The detailed comparisons of various scheduling algorithms are presented in Section 4.8.5.

We evaluated all 27 combinations under different settings. Due to the similarity, only a subset of the results are presented in Figures 4.8, 4.9, and 4.10. Generally, semi-partitioned scheduling algorithms outperform the partitioned strategies, independently of the algorithm used to construct the dependency graph. In addition, the preemptive scheduling policy with respect to the second computation segment is superior to the non-preemptive strategy and POTTS (usually) performs slightly better than JKS. Therefore, in subsequent schedulability evaluations, we only apply POTTS for the OCS task model. We analyze the effect of the three parameters individually by changing:

- **Workload of Shared Resources, i.e.,** $\boldsymbol{H \in \{[1\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}}$ (Figure 4.8): if the workload of the critical sections is increased, the difference between preemptive and non-preemptive scheduling is more significant.
- $\boldsymbol{Z}$ **for a fixed** $\boldsymbol{M}$, i.e., $Z \in \{4, 8, 16\}$ and $M = 8$ (Figure 4.9): when the number of resources is decreased compared to the number of processors, the performance gap between preemptive and non-preemptive scheduling increases.
- $\boldsymbol{M = Z \in \{4, 8, 16\}}$ (Figure 4.10): Increasing both $Z$ and $M$ slightly widens the gap between the semi-partitioned and partitioned approaches.

### 4.8.5 Evaluation Results for Schedulability

In our evaluation of non-nested resource access patterns, while keeping other configurations consistent, we analyzed the effect of three parameters individually by:

- increasing the percentage of total critical sections in each task;

- augmenting the number of available shared resources; and
- expanding both the number of available processors and shared resources.

For nested resource access patterns, we consider two additional scenarios:

- increasing the depth of nested resource accesses; and
- amplifying the probability that a critical section requests nested resources.

Our preliminary evaluation shows several key findings:

- our DGA-based approaches outperform the state-of-the-art in most of evaluated configurations;
- the LIST-EDF scheduling algorithms dominate the P-EDF scheduling algorithms; and
- the worst-fit partitioning algorithm exhibits superior performance compared to the federated-based partitioning approach.

Specifically, for frame-based task sets, our DGA-based approaches significantly outperform the state-of-the-art. Changing the number of resources or simultaneously increasing the number of processors and shared resources does not significantly influence the performance of our DGA-based approaches. Therefore, this section predominantly presents results concerning the growth in the percentage of total critical sections for frame-based task systems, while additional results can be found in Appendix A.1.2.

### Evaluation Results for OCS Task Systems

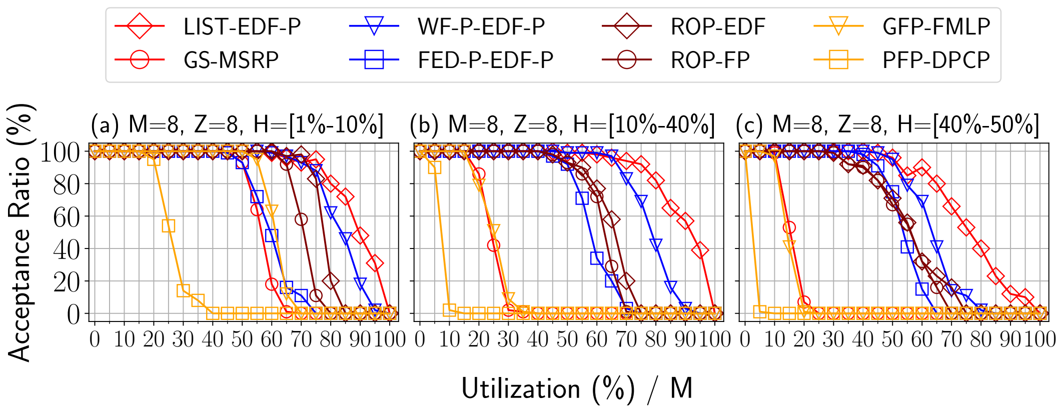


**Figure 4.11:** The evaluation results for periodic OCS task systems on 8 processors with 8 shared resources, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

We present a subset of our evaluation results for periodic task systems with the OCS task model in Figure 4.11, 4.12, and 4.13. Our evaluation results show three primary scenarios:

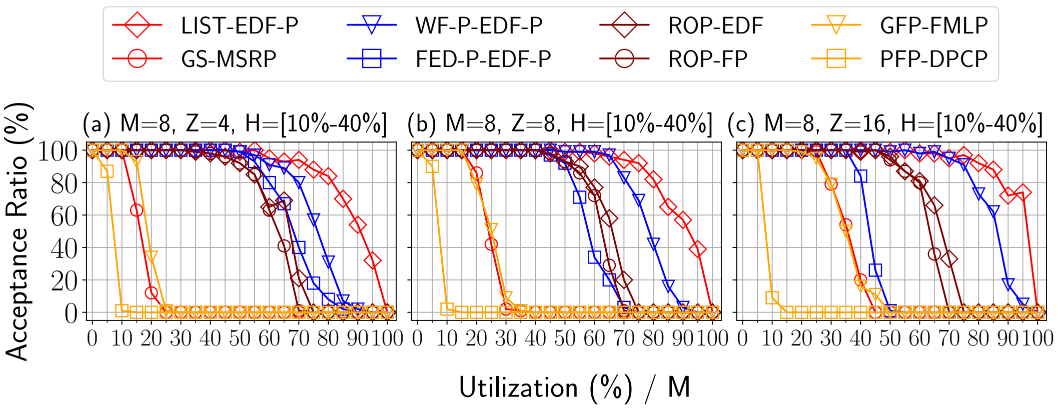


**Figure 4.12:** The evaluation results for periodic OCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

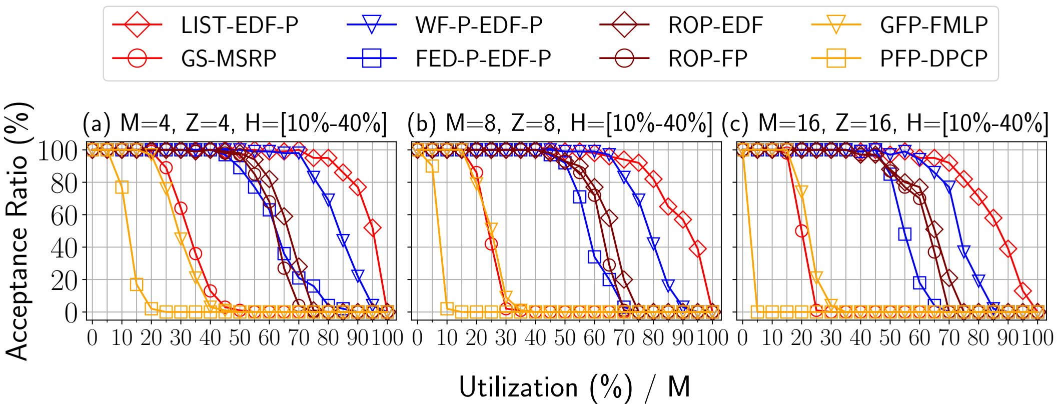


**Figure 4.13:** The evaluation results for periodic OCS task systems with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of processor and the number of available shared resources simultaneously, i.e., $M = Z \in \{4, 8, 16\}$.

- **Workload of Shared Resources, i.e., $\boldsymbol{H} \in \{[1\% - 10\%], [10\% - 40\%], [40\% - 50\%]\}$** (Figure 4.11): As the workload of critical sections rises, all methods experience performance deterioration. Particularly for $H = [40\% - 50\%]$, tasks with a period of 10 ms can have critical section execution times higher than 1 ms, causing direct deadline misses for tasks with a period of 1 ms in most methods. However, due to the non-work-conserving attribute, LIST-EDF-P manages to accommodate certain task sets, especially where the longest critical sections are between 1 and 2 ms.
- **Number of Available Shared Resources, i.e., $\boldsymbol{Z} \in \{4, 8, 16\}$** (Figure 4.12): When the number of shared resources is increased, compared to the number of processors, the performance of all the evaluated approaches is improved.

However, the performance gap between the LIST-EDF-P and the ROP based approaches, i.e., ROP-EDF and ROP-FP, increases. This indicates the increasing advantages of our non-work-conserving approaches with a larger number of shared resources.

- **Concurrent Expansion of Available Processor and Shared Resource, i.e.,**
  $\boldsymbol{M = Z \in \{4, 8, 16\}}$ (Figure 4.13): An integrated increase in both $M$ and $Z$ has a slight detrimental effect on the performance of all evaluated approaches.

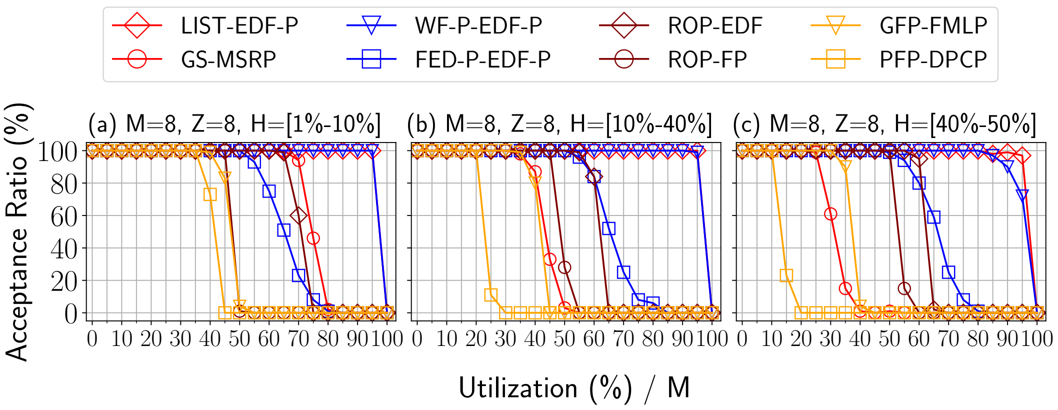


**Figure 4.14:** The evaluation results for frame-based OCS task systems on 8 processors with 8 shared resources, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

Figure 4.14 presents the evaluation results for frame-based OCS task systems on 8 processors with 8 shared resources. Three distinct levels of percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$, are presented. The results highlight the superior performance of DGA-based approaches in comparison to other methods. Impressively, even when the percentage of total critical sections reaches the range of $[40\%, 50\%]$, the DGA utilizing the LIST-EDF-P scheduling algorithm remains schedulable, sustaining an average processor utilization of 90%.

### Evaluation Results for MCS Task Systems

We present a subset of our evaluation results for periodic task systems employing the MCS task model in Figure 4.15, 4.16, and 4.17. These mirror the same three primary scenarios addressed for OCS task systems in Section 4.8.5. In addition to prior observations, the evaluation results show that our approaches significantly outperform other methods when the workload of the critical sections is high, i.e., $H = [40\% - 50\%]$, in Figure 4.15(c). Conversely, when the workload of the critical sections is on the lower side, particularly at $H = [1\% - 10\%]$ in Figure 4.15 (a), ROP-PCP exhibits superior performance. This can be attributed to the constraint programming of the problem $J_Z|r_j, l_j|L_{\max}$ when constructing the dependency graph,

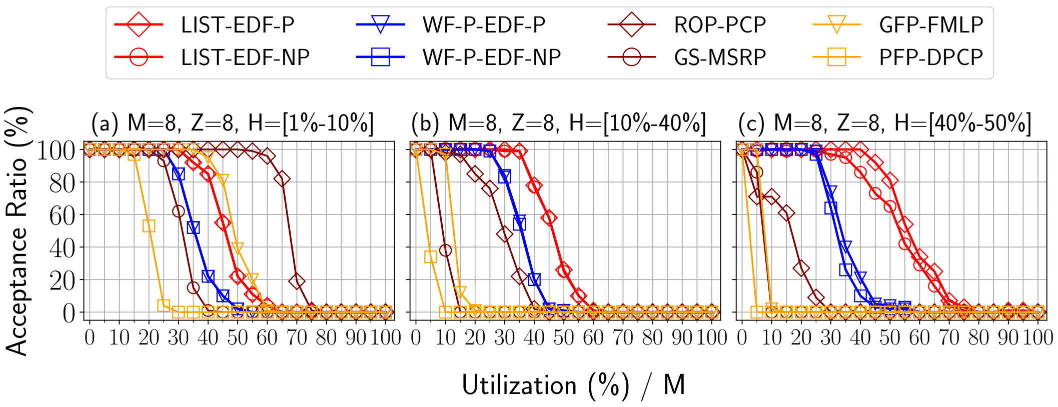


**Figure 4.15:** The evaluation results for periodic MCS task systems on 8 processors with 8 shared resources, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

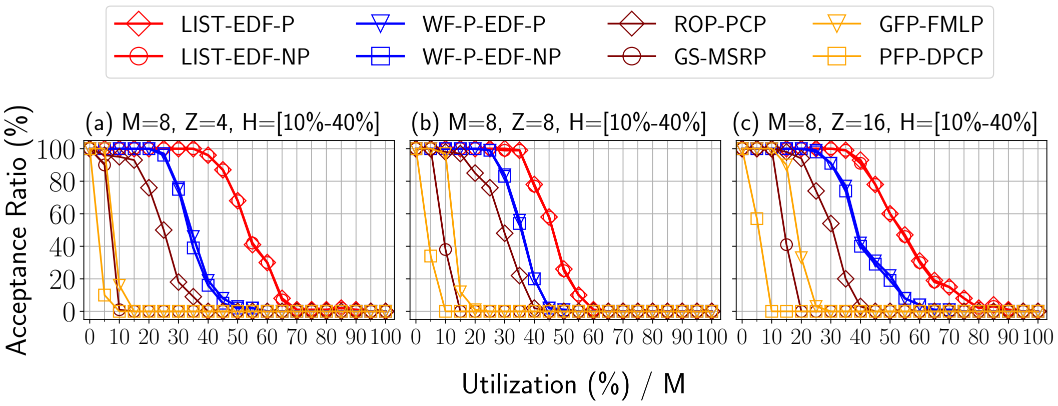


**Figure 4.16:** The evaluation results for periodic MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

which seeks to minimize the maximum lateness. It, however, ignores the execution order of the sub-jobs that do not impact the optimal lateness, potentially leading to sub-optimal performance when the non-critical sections' utilization is relatively high. In addition, the performance of preemptive and non-preemptive scheduling algorithms trends to be similar, since the optimized dependency graph and the pre-calculated deadline for each computational segment reduced the potential preemption due to earlier deadlines.

Figure 4.18 shows the evaluation results for frame-based MCS task systems on 8 processors with 8 shared resources. As the percentage of total critical sections increases, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$, the results show a rapid decline in the performance of state-of-the-art algorithms. Notably, these algorithms underperform compared to their counterparts in OCS task systems, as illustrated in

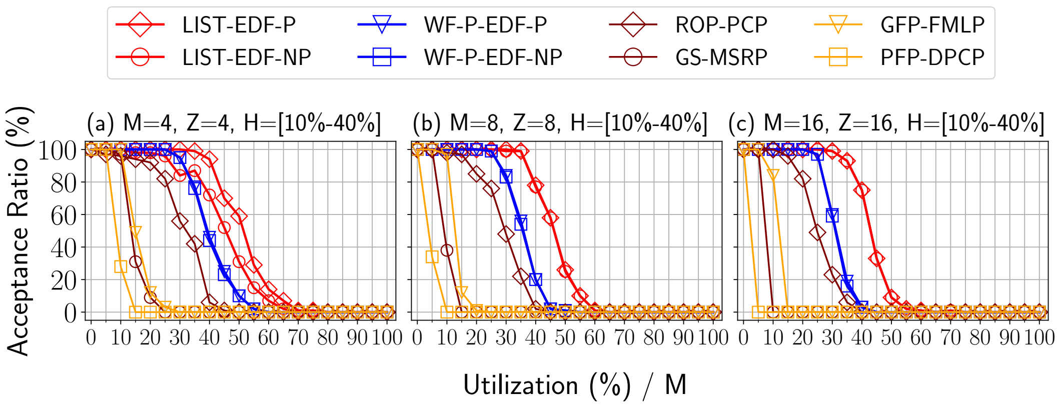


**Figure 4.17:** The evaluation results for periodic MCS task systems with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of processor and the number of available shared resources simultaneously, i.e., $M = Z \in \{4, 8, 16\}$.

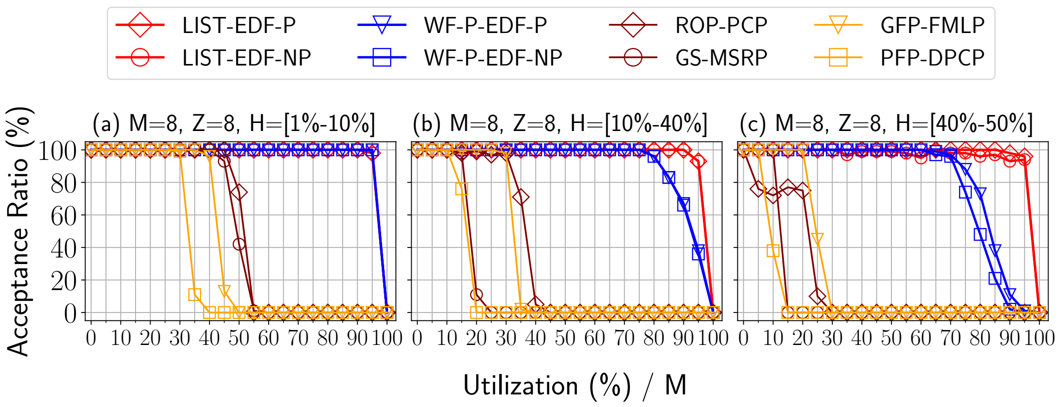


**Figure 4.18:** The evaluation results for frame-based MCS task systems on 8 processors with 8 shared resources, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

Figure 4.14. Moreover, the performance of DGA applying the P-EDF scheduling algorithms also downgrades. In contrast, the DGA with the LIST-EDF-P scheduling retains its schedulability, achieving an average processor utilization of 85%.

### Evaluation Results for Nested-MCS Task Systems

We present a subset of evaluation results for periodic task systems using the Nested-MCS task model in Figures 4.19, 4.20, 4.21, and 4.22. For all the evaluated configurations, DGA-based approaches consistently outperform state-of-the-art methods.

Figure 4.19 shows that when the utilization of critical sections increases, the performance of existing methods decreases considerably. In contrast, the efficiency of our newly proposed approach remains relatively unaffected. Notably, when the

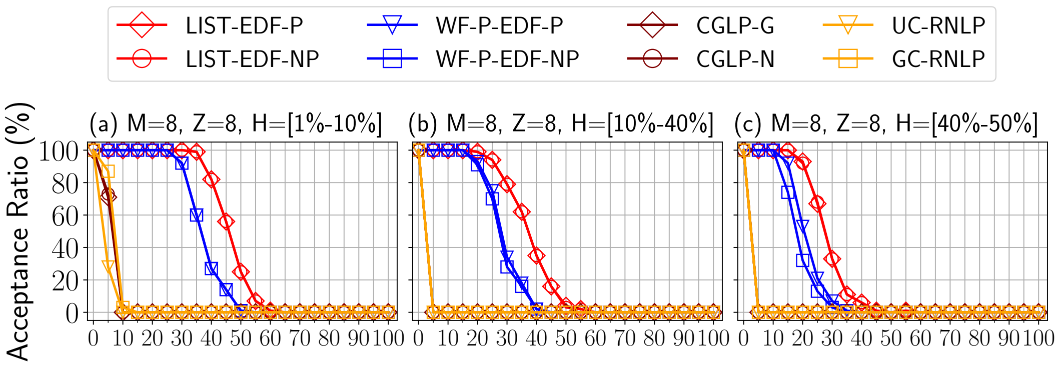


**Figure 4.19:** The evaluation results for periodic Nested-MCS task systems on 8 processors with 8 shared resources and a nested depth of 2, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

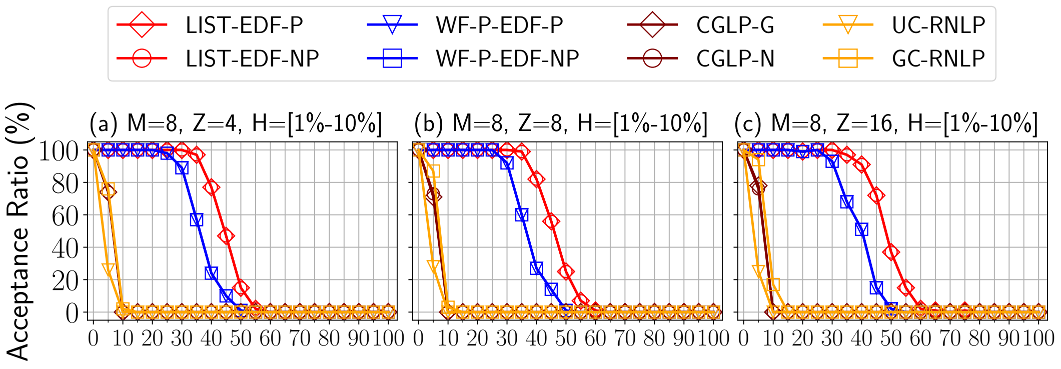


**Figure 4.20:** The evaluation results for periodic Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 2, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

utilization of critical sections is extremely high, reaching levels of $[40\% - 50\%]$ in Figure 4.19(c), existing methods fail. Our novel approach, on the other hand, continues to offer a reasonable acceptance ratio.

Figure 4.20 illustrates that simply increasing the number of shared resources, without changing other configurations, does not significantly impact the performance of any of the evaluated methods. However, increasing the depth of nested resource accesses (Figure 4.20 to Figure 4.21) results in a slight decline in the performance for all methods. Similarly, raising the probability of a critical section requesting nested shared resources, i.e., in Figure 4.22, also leads to a slight performance drop.

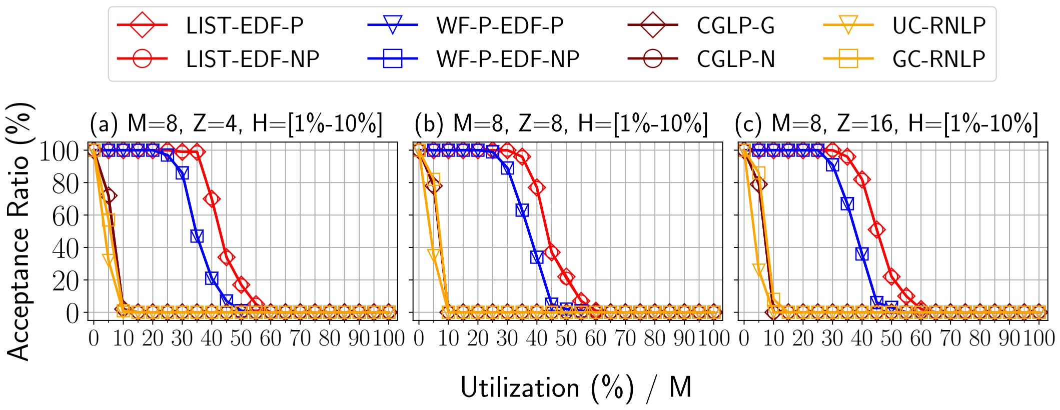


**Figure 4.21:** The evaluation results for periodic Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 4, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

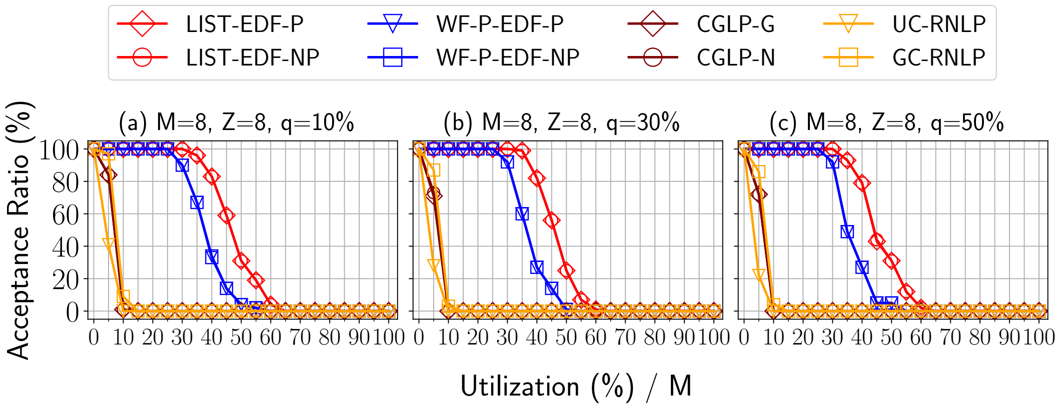


**Figure 4.22:** The evaluation results for periodic Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 2, emphasis on increasing the probability that a critical section requests nested shared resources, i.e., $q \in \{10\%, 30\%, 50\%\}$.

Figure 4.23 shows the evaluation results for frame-based Nested-MCS task systems operating on 8 processors with 8 shared resources and a nested depth of 2. As the percentage of total critical sections rises, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$, all the state-of-the-art approaches become ineffective when $H \geq 10\%$. In contrast, our DGA-based methods continue to offer a satisfactory performance in terms of acceptance ratio, even at relatively high percentages of total critical sections.

Additionally, the pattern of nested resource access also results in a negative influence on performance. For instance, while the DGA with the P-EDF scheduling algorithms manages an average processor utilization rate of just 60%, the DGA

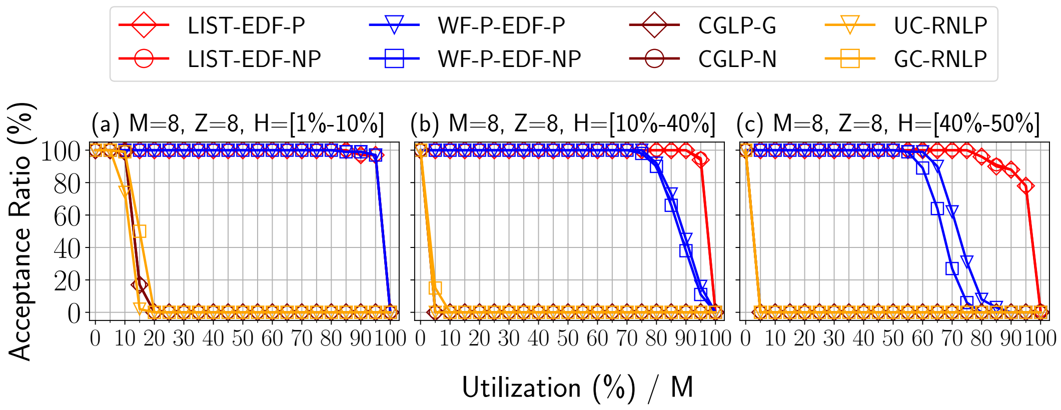


**Figure 4.23:** The evaluation results for frame Nested-MCS task systems on 8 processors with 8 shared resources and a nested depth of 2, emphasis on increasing percentage of total critical sections, i.e., $H \in \{[1\%, 10\%], [10\%, 40\%], [40\%, 50\%]\}$.

combined with the LIST-EDF-P scheduling strategy maintains its schedulability, reaching an average processor utilization rate of 75%.

## 4.9 Summary

In this chapter, we address the longstanding problem of resource synchronization for periodic tasks, aiming to provide an effective solution for a widely-adopted real-time task model. We seek to answer several fundamental questions related to the problem by starting with the simplest task models, namely the OCS and MCS task models, within frame-based task systems. Here is a short summary of our findings:

- The fundamental difficulty is mainly due to the sequencing of the mutually exclusive accesses to the shared resources (binary semaphores). Introducing more processors, eliminating periodicity or job recurrence, migrating tasks, or allowing preemption does not simplify the problem in terms of computational complexity. Notably, the problem in frame-based task systems, with either OCS or MCS task model, is $\mathcal{NP}$-hard in the strong sense.
- Our Dependency Graph Approach (DGA) employs non-work-conserving mechanisms specifically for critical sections. Although a critical section might be ready, it might not get executed due to artificially set precedence constraints. Most existing multiprocessor synchronization protocols assume work-conserving behaviors for critical section accesses via priority boosting. Our research highlights the potential benefits of adopting cautious, non-work-conserving synchronization, particularly for machine learning tasks, which often require extended critical sections when accessing the GPU.

- We introduce a structured design flow based on the DGA. This flow leverages algorithms from uni-processor non-preemptive scheduling (for OCS task model) and job/flow shop scheduling (for MCS task model), offering approximation ratios for the determined makespan in frame-based task systems.
- In our study, the performance gap between partitioned, semi-partitioned, and global scheduling largely arises from the ability to schedule sub-jobs that are constrained by the dependency graph. Recent research has shown that global scheduling might not consistently outperform partitioned or semi-partitioned algorithms [BG16; BS18]. Our evaluation results indicate that partitioned scheduling performed worse, largely attributable to the limited understanding of the problem $P|prec, tied|C_{\max}$. Further explorations are needed to understand these scheduling paradigms for a given dependency graph.

To enhance the applicability of DGA, we expanded from the OCS task model to encompass the MCS model. This allows for arbitrary configurations of the number of non-nested critical sections per task. Moreover, we adapted the system to handle nested resource accesses with multiple critical sections and adjusted it for periodic task systems. This is complemented by the integration of LIST-EDF scheduling, which is applied for both partitioned and global scheduling strategies.

Our evaluation results demonstrate that our approach excels in comparison to the state-of-the-art methods in the literature, proving its applicability across a majority of the studied task models and evaluated configurations.

One significant difference between our DGA-related approaches and the state-of-the-art methods is that DGA only supports periodic task systems and is not applicable to sporadic task systems. DGA requires that all tasks be known before deployment into the operating system, so that the execution order of all critical sections in job level for each shared resource can be pre-defined. Such a DAG construction process introduces additional overheads with respect to computational cost and storage. In addition, DGA forces each (sub-)job to execute in its worst-case execution time; no early finish is permitted in order to prevent multiprocessor timing anomalies.

Chapter 5

# Implementation and Verification of Resource Synchronization Protocols

**Contents**

## 5.1 Overview

Although many protocols appear theoretically sound, their empirical performance on real-world platforms often remains unverified. Moreover, although some protocols receive official support in RTOSes, there is no guaranteed assurance of their implementations, which can lead to potential mismatches between theoretical postulations and real-world outcomes. This chapter explores the intricacies of implementing resource synchronization protocols and subsequently introduces a formal verification framework for protocols implemented on an RTOS.

**Challenges in Contemporary Approaches:** Over the preceding decades, a number of resource synchronization protocols have been proposed and studied. While certain protocols, such as ROP [HYC16] and its derivatives [BCH+17], offer theoretically stringent worst-case response time guarantees, only a few of them have been implemented in actual computational platforms. Theoretical analyses often assume that operational overheads are negligible. However, this does not always align with real-world scenarios. For instance, task migration between cores can induce significant overheads [SCZ+17], underscoring the lack of in-depth research on protocol performance within actual RTOS environments.

Furthermore, while RTOSes, such as RTEMS, officially supports ICPP and MrsP, and LITMUS$^{\text{RT}}$ incorporates MPCP, DPCP, FMLP, and DFLP, the correctness of these implementations remains unverified. During the protocol design phase, a protocol's attributes are typically described as a set of rules, that can be implemented. However, the combination of these rules are complicated when all the details have to be considered, e.g., the order of priority modifications, the queue-based operations, and illegal inputs checking. One approach is to *test* sufficient inputs and validate the derived outputs. However, a sufficient test set that covers all possible situations is difficult to be derived, especially for multiprocessor systems. Executing a test case can only *validate* the behavior of the entire system, encompassing the OS kernel primitives, hardware-specific code, and the actual hardware or simulation platform. Any observed error can also be caused by the interplay of these low leveled components. Hence, it is difficult to pinpoint the real issues in the implementation of protocol itself.

In fact, the formal descriptions of a protocol are based on abstracting from operating system- or hardware-specific details. The chosen RTOS for protocol deployment might not necessarily comply with all of these foundational assumptions. Compensations might be required for successful implementation, potentially leading to mismatches between the formally described properties and the actual implementation. Instead of validation, all the properties that are desired to be achieved by a resource synchronization protocol can be formally proven (so-called *verification*) based on its formal descriptions. One approach is based on *model checking*. The considered system is first modeled in a formal language where all the required properties are specified in logic formulas, by which a model checker can be applied to automatically check the property specifications. However, the system model is difficult to be specified and there is no guarantee of correctness.

**This Dissertation:** Initially, we elucidate the fundamental principles of resource synchronization protocols. Subsequent sections enhance the open-source development procedures in both RTESM and LITMUS$^{\text{RT}}$, and detail the implementation of the DGA from Chapter 4 in both RTOSes. Operational overheads are measured on real platforms and compared against other officially supported protocols to underscore the applicability of DGA.

To overcome the drawbacks of traditional validation and verification approaches, we propose a formal verification framework wherein deductive verification [Hoa69; Flo93] is employed. This framework is designed to formally verify protocols implemented on an RTOS, operating under the assumption that all underlying layers are correctly supported. We chose RTEMS as our preferred RTOS due to its applicability across various domains. The verification specifics for the implementation of DGA, alongside two officially supported protocols, namely ICPP and MrsP, are executed.

## 5.2 Basic Rules of Synchronization Protocols

Resource synchronization protocols comprise a set of predefined rules, which every task must adhere to when accessing resources. The underlying logic behind these rules is often grounded in rigorous theory, to ensure tasks can meet their worst-case timing guarantees. In the following, we delineate the fundamental rules associated with these protocols:

**Scheduling Algorithms** A synchronization protocol must define its supported scheduling algorithms, e.g., either earliest deadline first (EDF) or fixed priority (FP). When EDF is applied, the job with the earliest absolute deadline has the highest priority, whereas the priorities for all tasks are predefined when FP is applied. For multiprocessor systems, global schedule, partitioned schedule and semi-partitioned schedule can be applied, details are stated in Section 2.2.2.

**Request Ordering** The order of concurrent requests for the same shared resource also has to be determined. When two or more tasks are blocked by the same shared resource, the waiting queue of these tasks has to be sequenced by a certain policy. Two common policies are FIFO queue and priority-based queue. In a FIFO queue, tasks are ordered by the requesting time, the task with earlier requesting time can acquire the corresponding shared resource earlier, which bounds the maximum waiting time for each task. In a priority-based queue, tasks are ordered by their current priorities, which are normally the tasks' scheduling priorities. Additionally, there is a third policy, i.e., in DGA, where the access order for each shared resource is predefined at the job level for all tasks within a single hyper-period. Therefore, the wait queue is sequenced by the predefined access order.

**Waiting Mechanism** The semantic, i.e., how a task is waiting for an occupied resource, must be identified. Under a suspension-based synchronization protocol, a task that is waiting for accessing to a currently unavailable resource is suspended by adding itself into a wait queue. Under a spin-based protocol, the task retains its privilege on the current processor, executing a spinning loop. It continuously checks for resource availability before accessing and starting its critical section.

**Bound Measure** The measure to bound the maximum blocking time and prevent unbounded priority inversions has to be set up. Under non-preemptive execution, once a task begins its critical section, no other tasks on the processor can preempt it, irrespective of their priorities and deadlines. Similarly, priority boosting allocates a boosted priority to each critical section, which is higher than the highest regular priority for scheduling of all tasks. However, under priority boosting, a critical section can still be potentially preempted by another critical section with higher boosted priority. Another notable approach is the priority ceiling: When a task executes its critical section, the priority can be prompted to the corresponding resource's ceiling priority, where the ceiling priority can be determined either statically or dynamically. When the ceiling priority of a shared resource is defined statically, it simply equals to the highest priority of any task that may request the resource. If dynamic ceiling priority is applied, the ceiling priority is defined as the highest priority of all tasks that are currently locking or will lock the shared resource, i.e., tasks in the corresponding waiting queue.

**Execution Place** Unlike protocols for uni-processor systems, multiprocessor resource synchronization protocols must specify where the critical sections are executed, whether locally or remotely. For the former, the critical sections of a task can be executed along with its non-critical sections on the processor where the task is currently assigned. For the latter, critical sections are executed on specified processor(s) where the corresponding resources are assigned on. In some protocols, the local executed critical sections can also be executed remotely. For example, the *help mechanism* in MrsP allows the current resource owner to execute its preempted critical section on a remote processor, where a task is spin-waiting for the same resource.

## 5.3 Implementation in RTOSes

In this section, we concentrate on the implementation of DGA in two widely-used operating systems in real-time research domain: RTEMS and LITMUS$^{RT}$. We start by discussing potential strategies for integrating DGA into LITMUS$^{RT}$, supplemented with the implementation of the proposed LIST-EDF scheduling algorithm. Subsequently, we explore the procedure for supporting new resource synchronization protocols in RTEMS, taking the implementation of DGA as a case study. Although we have provided support for other protocols such as MPCP, DPCP, FMLP, and DFLP in an earlier version of RTEMS [1] as detailed in [SPM+22], they are not the main focus of this dissertation and are not extensively discussed.

[1]These implementations were based on RTEMS 4.12. This version has been deleted and officially superseded by RTEMS 5.1. Transitioning the implementations from RTEMS 4.12 to RTEMS 5.1 requires necessary modifications and is beyond the scope of this dissertation.

### 5.3.1 Implementation in LITMUS$^{RT}$

This section details our implementation in LITMUS$^{RT}$ and compares the implementation overheads of our approach with FMLP provided by LITMUS$^{RT}$ for both partitioned and global scheduling. Our implementation has been released in [Shi18] for the OCS task model and in [Shi19] for the MCS task model. In general, the implementation for the MCS task model can also be applied to task systems with the OCS task model. However, as they were designed for different task models, the implementations vary slightly, leading to different operational overheads. In this section, we only focus on the implementation for MCS task model.

### 5.3.2 DGA Implementation in LITMUS$^{RT}$

When implementing DGA in LITMUS$^{RT}$, we have two options a) apply the table-driven scheduling that LITMUS$^{RT}$ provides, or b) implement a new binary semaphore which enforces the execution order of sub-jobs of critical sections that access the same resource, with the execution order predefined by the dependency graph. In the first approach, a static scheduling table is generated for one hyper-period and then repeated periodically in a table-driven schedule. This table determines which sub-job is executed on which processor for each time point in the hyper-period. However, due to the possible large number of sub-jobs in one hyper-period and possible migrations among processors, the resulting table can be very large. To avoid this problem, we decided to implement a new binary semaphore that supports all the properties of our new approach instead.

The new approach is implemented under the plug-in Partitioned EDF with synchronization support (PSN-EDF) and the plug-in Global EDF with synchronization support (GSN-EDF). The original design of these two plug-ins guarantees the EDF feature of *LIST-EDF*. Therefore, we only need to provide the relative deadlines for all the sub-jobs of each task, and LITMUS$^{RT}$ will automatically update the absolute deadlines accordingly during runtime.

In order to enforce the sub-jobs to follow the execution order determined by the dependency graph, our implementation has to: a) let the all the sub-jobs inside one job follow the predefined order; b) force all the sub-jobs that access the same resource to follow the order determined by the graph.

The first order is ensured in LITMUS$^{RT}$ by default. The task deploy tool `rtspin` provided by the user-space library *liblitmus* defines the task structure, e.g., the execution order of non-critical sections and critical sections within one task, the related execution times, and the resource ID that each critical section accesses. Each job is executed in `rtspin` by using Algorithm 6 where `execution_for` is a simple spin loop function that emulates purely CPU-bound workloads. The function `execution_for(a, b, c)` has three inputs: `a` is the release time of the sub-job, `b` is the execution time of the sub-job, and `c` is the deadline of the sub-job. Note, that the release times of the sub-jobs are not predetermined but result from the moment the job is released (for the first sub-job), and the moments when the corresponding

**Algorithm 6** Inner order enforcement in `rtspin`

```
Input: Execution times for all sub-jobs: $C_{i,1}$, $C_{i,2}$, ..., and $C_{i,\eta_i}$, relative deadlines for
    each sub-job: $d_{i,1}$, $d_{i,2}$, ..., and $d_{i,\eta_i}$, and requested shared resources: $z_{i,1}$, $z_{i,2}$, ..., and
    $z_{i,\eta_i}$ (if the corresponding computational segment is non-critical section, the requested
    shared resource is $\varnothing$);
1: execution_for($r^{\ell}_{i,1}$, $C_{i,1}$, $d^{\ell}_{i,1}$);
2: semaphore_lock($z_{i,2}$);
3: execution_for($r^{\ell}_{i,2}$, $C_{i,2}$, $d^{\ell}_{i,2}$);
4: semaphore_unlock($z_{i,2}$);
5: ...;
6: semaphore_lock($z_{i,\eta_{i-1}}$);
7: execution_for($r^{\ell}_{i,\eta_{i-1}}$, $C_{i,\eta_{i-1}}$, $d^{\ell}_{i,\eta_{i-1}}$);
8: semaphore_unlock($z_{i,\eta_{i-1}}$);
9: execution_for($r^{\ell}_{i,\eta_i}$, $C_{i,\eta_i}$, $d^{\ell}_{i,\eta_i}$);
```

predecessor(s) are finished, i.e., a sub-job can be released only when its predecessor (if any) has finished its execution. Please note that for sub-jobs related to critical sections the release time is not only defined by its predecessor's finish time inside the same job, but also related to another predecessor that accesses the same resource (if one exists). The deadlines of the sub-jobs however are calculated beforehand and result from the dependency graph and the internal sub-job dependencies. All the commands in Algorithm 6 are executed sequentially, which directly ensures that the execution order of sub-jobs within one task. Moreover, the resource ID for each critical section is parsed by `rtspin`, so the critical section can find the correct semaphore to lock, and the execution of a critical section is protected by the corresponding `semaphore_lock(`$z_{i,j}$`)` and `semaphore_unlock(`$z_{i,j}$`)`. In our implementation we do not have to consider addressing the corresponding resources.

Consider a set of periodic real-time tasks where each task releases its first job simultaneously. The schedule for this task set will be repetitive across each hyper-period provided if:

- the scheduler is deterministic, always making consistent scheduling decisions for any given scenario,
- the WCRT of all jobs is smaller than the period of the tasks, i.e., it is ensured that at any point in time at most one job of each task is in the system, and
- no early completion of (sub)jobs is allowed.

A ticket system is applied to enforce the execution order of sub jobs that request the same shared resource. To be precise, we extended LITMUS$^{\text{RT}}$ data structure `rt_params` that describes task details, e.g., priority, period, and execution time for each task, by adding:

A ticket system is applied to enforce the execution order for sub-jobs accessing the same shared resource. Specifically, we augmented the LITMUS$^{\text{RT}}$ data structure, i.e., `rt_params`, which delineates task attributes such as priority, period, and execution time, with the following elements:

- `total_jobs`: An integer which represents the number of jobs of the related task in one hyper-period.
- `total_cs`: An integer that indicates the number of total critical sections in this task.
- `job_order`: An array which defines the total order of the sub-jobs associated with critical sections that access the same resource over one hyper-period. Additionally, for each shared resource, the array's final $Z$ elements record the total number of task set's critical sections. Therefore, the length of the array is the number of critical sections in one hyper-period of the described task plus the number of total shared resources, i.e., $\texttt{len}(\texttt{job\_order}) = \texttt{total\_jobs} \times \texttt{total\_cs} + Z$.
- `current_cs`: An integer that defines the index of the current critical section of the task that is being executed.
- `relative_ddls`: An array which records the relative deadlines for all sub-jobs of one task.

Furthermore, we implemented a new binary semaphore, named as `mdga_semaphore`, to make sure the execution order of all the sub-jobs that access the same resource follows the order specified by the dependency graph.

A semaphore has the following common components:

- `litmus_lock`: Protects the semaphore structure,
- `semaphore_owner`: Identifies the current owner of the semaphore, and
- `wait_queue`: Stores all jobs waiting for this semaphore.

We also introduced a parameter, `serving_ticket`, to regulate the non-work conserving access pattern of critical sections. That is, a job can lock the semaphore and start its critical section only if its ticket matches the corresponding `serving_ticket`.

The pseudo code in Algorithm 7 shows three main functions in our implementation. The details are as follows:

**The `get_cs_order` function** determines the position (or 'ticket') of a sub-job in the execution sequence of all sub-jobs accessing the same shared resource during run-time. This determination is a two-layered process.

In the first layer, the function identifies the current job's position within the current hyper-period. Within LITMUS$^{\mathrm{RT}}$, the `job_no` keeps track of the number of jobs a task releases. To obtain the precise position of this job in a hyper-period, we apply a modulo operation to `job_no` using `total_jobs` as the divisor.

The second layer finds the position of the current sub-job with respect to its associated shared resource. Since a job comprises multiple critical sections, the `current_cs` variable determines the position of the current critical section within the job. We generate the index by summing the critical sections of preceding jobs and the `current_cs` of the present job. Following this, the `cs_order` value is retrieved from `job_order` based on the deduced index.

**Table 5.1:** An example of the data structure for tasks.

| | total_jobs | total_cs | job_order | current_cs |
|---|---|---|---|---|
| $\tau_1$ | 2 | 2 | [1,3,6,8,9,9] | 1 |
| $\tau_2$ | 2 | 2 | [0,2,5,7,9,9] | 1 |
| $\tau_3$ | 2 | 2 | [1,3,6,8,9,9] | 0 |
| $\tau_4$ | 2 | 2 | [0,2,5,7,9,9] | 0 |
| $\tau_5$ | 1 | 1 | [4,4,9,9] | 0 |

We present an example with five tasks, four of which ($\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$) are detailed in Table 4.1 (Section 4.4.4). These tasks have identical periods: $T_1 = T_2 = T_3 = T_4 = 25$. A fifth task, $\tau_5$, has a period $T_5 = 50$ and follows the same pattern as $\tau_4$. Specifically, $\tau_5$ requests the resource $z_2$ in its second segment and resource $z_1$ in its fourth segment. Considering these configurations, the hyper-period for this task set becomes 50. In one hyper-period, tasks $\tau_1$ to $\tau_4$ release two jobs, while $\tau_5$ releases just one. The relevant data structure is illustrated in Table 5.1.

For example, task $\tau_1$'s `job_order` is [1, 3, 6, 8, 9, 9]. The first two elements, i.e., [1, 3], dictate the execution order of $J_1^1$'s two critical sections, while [6, 8] pertain to $J_1^2$. The final pair, [9, 9], indicates that nine sub-jobs request each of the resources, i.e., $z_1$ and $z_2$, within one hyper-period. Operating on a zero-based index and assuming that the `job_no` for $\tau_1$ is 13, Line 1 of Algorithm 7 yields the `current_jobno` as the second job of $\tau_1$ in the present hyper-period. Line 2 identifies the index of the appropriate critical section, i.e., index $= 3$, while Line 3 from the same algorithm determines the corresponding execution order. Hence, the $13th$ job of $\tau_1$ receives the execution order 8 (i.e., `cs_order` $= 8$) to access the resource.

**The `mdga_lock` function** is responsible for locking the semaphore to gain access to a dedicated resource. Upon invocation, it first obtains the appropriate execution order within a hyper-period using the `get_cs_order()` function. Following this, the semaphore's current status is assessed. If the semaphore is occupied by another job, the incoming job is immediately added to the `wait_queue`. Otherwise, the semaphore's `serving_ticket` is compared with the job's `cs_order`. If they match, the current job is granted ownership of the semaphore and can begin its critical section. However, if there is a mismatch, the job is added to the `wait_queue`.

Please note that our `wait_queue` is sorted increasingly by the `cs_order` of the jobs. This ensures the job with the smallest `cs_order` is at the front. Such prioritization ensures that when the active semaphore owner finishes its execution, only the first job in the `wait_queue` needs to be examined, eliminating the inefficiency of scanning an unordered `wait_queue`.

**Algorithm 7** DGA with multi-critical sections implementation

```
Input: Upcoming task $\tau_i${job_no, total_jobs, total_cs, current_cs, relative_ddls},
    and Requested semaphore $s_z${semaphore_owner, serving_ticket, wait_queue};

    Function get_cs_order():
 1: current_jobno ← $\tau_i$.job_no mod $\tau_i$.total_jobs;
 2: index ← current_jobno × $\tau_i$.total_cs + current_cs;
 3: cs_order ← $\tau_i$.job_order[index];

    Function mdga_lock():
 4: if $s_z$.semaphore_owner is NULL and
    $s_z$.serving_ticket equals to $\tau_i$.cs_order then
 5:     $s_z$.semaphore_owner ← $\tau_i$;
 6:     Update the deadline for $\tau_i$;
 7:     $\tau_i$ starts the execution of its critical section;
 8: else
 9:     Add $\tau_i$ to $s_z$.wait_queue;
10: end if

    Function mdga_unlock():
11: $\tau_i$ releases the semaphore lock;
12: Update the deadline for $\tau_i$;
13: $\tau_i$.current_cs++;
14: if $\tau_i$.current_cs = total_cs then
15:     Set $\tau_i$.current_cs ← 0;
16: end if
17: $s_z$.serving_ticket++;
18: if $s_z$.serving_ticket = num_cs then
19:     Set $s_z$.serving_ticket ← 0;
20: end if
21: Next task $\tau_{next}$ ← the head of the wait_queue (if exists);
22: if serving_ticket equals to $\tau_{next}$.cs_order then
23:     $s_z$.semaphore_owner ← $\tau_{next}$;
24:     $\tau_{next}$ starts the execution of its critical section;
25: else
26:     $s_z$.semaphore_owner ← NULL;
27:     Add $\tau_{next}$ to $s_z$.wait_queue;
28: end if
```

**The `mdga_unlock` function** is invoked when a job completes its critical section and attempts to unlock the semaphore. The task's `current_cs` is incremented by 1 to target the next potential critical section within the corresponding job. If `current_cs` reaches `total_cs`, it will be reset to 0. This indicates all critical sections in the job have been executed.

Subsequently, the semaphore's `serving_ticket` is incremented. The semaphore then becomes ready to be acquired by any successor in the dependency graph. If serving_ticket reaches the total number of critical sections associated with this resource within a hyper-period, denoted as num_cs, the serving_ticket resets to 0. This action initiates the next iteration. Importantly, the num_cs value resides within the last $Z$ elements of job_order, corresponding to the related resource id.

After this step, the first job in the wait_queue, named $\tau_{next}$, is examined. If the cs_order of $\tau_{next}$ matches the semaphore's serving_ticket, then the semaphore's ownership is assigned to $\tau_{next}$. Consequently, $\tau_{next}$ can begin the execution of its critical section. Otherwise, the owner of the semaphore is set to NULL, and the task $\tau_{next}$ is placed back in its respective wait_queue.

Furthermore, each sub-job has a unique modified deadline. This implies that a job may have varying deadlines during different segment executions. Careful attention is required during the deadline update in the implementation. When deploying a task using rtspin, the relative deadline of its initial sub-task is presented as the overall task's relative deadline. Given that two consecutive non-critical sections are not permissible, a sub-job's completion triggers either mdga_lock or mdga_unlock. When mdga_lock is invoked, the new critical section's deadline is updated using the relative_deadline. Invoking mdga_unlock lets only the completed critical section refresh the associated job's deadline for its subsequent section (if any).

The implementations for both global and partitioned plug-ins are similar. However, due to frequent preemptions or interruptions in global scheduling, preemption must be disabled during semaphore-related function executions to ensure the functions' integrity. If a job fails to acquire the semaphore and sleeps in the respective wait queue until the simulation's end, it enters an *uninterruptible sleep* status. To avoid this scenario, make sure the simulation runtime is a multiple of the hyper-period, as this guarantees task completions within their periods. Otherwise, restarting the system before the next iteration is recommended to prevent complications. In addition, our implementation does not fully address over-run scenarios. Therefore, a missed task deadline could potentially disrupt the ticket system.

### Overheads Comparison in LITMUS$^{RT}$

We evaluated the overheads of our implementation on an SMP system with an x86 architecture. Detailed configurations are provided in Section 2.4.3. For comparison, we also evaluated the FMLP supported in LITMUS$^{RT}$, which includes the P-FMLP for partitioned scheduling and the G-FMLP for global scheduling. All four protocols were tested using identical task sets, with each task having multiple critical sections. The overheads that we tracked are:

- **CXS**: context-switch overhead.
- **RELEASE**: time spent to enqueue a newly released job into a ready queue.
- **SCHED**: time spent to make a scheduling decision, i.e., find the next job to be executed.
- **SCHED2**: time spent to perform post context switch and management activities.

**Table 5.2:** Overheads of protocols in LITMUS$^{\mathrm{RT}}$ in $\mu s$.

| Max. (Avg.) | CXS | RELEASE | SCHED | SCHED2 | SEND-RES |
|---|---|---|---|---|---|
| P-FMLP | 29.51 (0.98) | 17.68 (0.96) | 31.85 (1.31) | 28.77 (0.18) | 66.33 (2.86) |
| PDGA-MCS | 30.65 (1.25) | 18.63 (1.02) | 31.09 (1.64) | 29.43 (0.19) | 59.09 (21.06) |
| G-FMLP | 30.51 (1.05) | 48.53 (3.75) | 45.99 (1.51) | 29.62 (0.16) | 72.26 (2.50) |
| GDGA-MCS | 26.87 (0.94) | 30.01 (2.19) | 30.25 (1.02) | 19.26 (0.14) | 72.53 (21.50) |
| PDGA-OCS | 18.76 (0.90) | 18.98 (1.06) | 48.50 (1.33) | 29.25 (0.16) | 38.3 (1.61) |
| GDGA-OCS | 30.87 (1.79) | 61.63 (12.06) | 59.05 (4.46) | 27.17 (0.25) | 72.09 (20.77) |

- **SEND-RES**: inter-processor interrupt latency, including migrations.

The overheads are reported in Table 5.2. This data illustrates that the overheads of our methodology are comparable to those of P-FMLP and G-FMLP. Furthermore, the implementations for the OCS task model, called PDGA-OCS and GDGA-OCS, were evaluated to examine the overhead and reported in Table 5.2 as well. A direct comparison of PDAG-OCS and PDGA-MCS (GDGA-OCS and GDGA-MCS, respectively) is not possible because they are designed for different scenarios, depending on the number of critical sections per task. As illustrated in Table 5.2, the overheads across different task model implementations appear to be of a similar magnitude.

### 5.3.3 Implementation in RTEMS

In this section, we start with an overview of the symmetric multiprocessing capabilities within RTEMS. Following that, we discuss the detailed implementation of DGA in RTEMS. To offer a unique perspective on the procedure, we utilize workflow diagrams to describe RTEMS's implementation details rather than providing a verbose algorithmic description. We conclude by evaluating the operational overheads of our approach, comparing them with the officially supported multiprocessor synchronization protocol, specifically MrsP, in RTEMS.

#### Symmetric Multiprocessing Support in RTEMS

RTEMS allows users to develop new resource synchronization protocols by adhering strictly to the RTEMS API. When creating a new semaphore, the `SEM_Initialize` function is invoked to designate particular attributes tailored for each synchronization protocol. Apart from the protocol-dependent semaphore creation, this section emphasizes the ubiquitous components present across all protocols. These include both lock and unlock directives, and application configurations.

**Lock and Unlock Directives** The workflow of the lock directive is demonstrated in Figure 5.1. When a task, i.e., $\tau_i$, requests a shared resource, it attempts to lock the relevant semaphore. After identifying the appropriate semaphore, denoted as `SEM`,

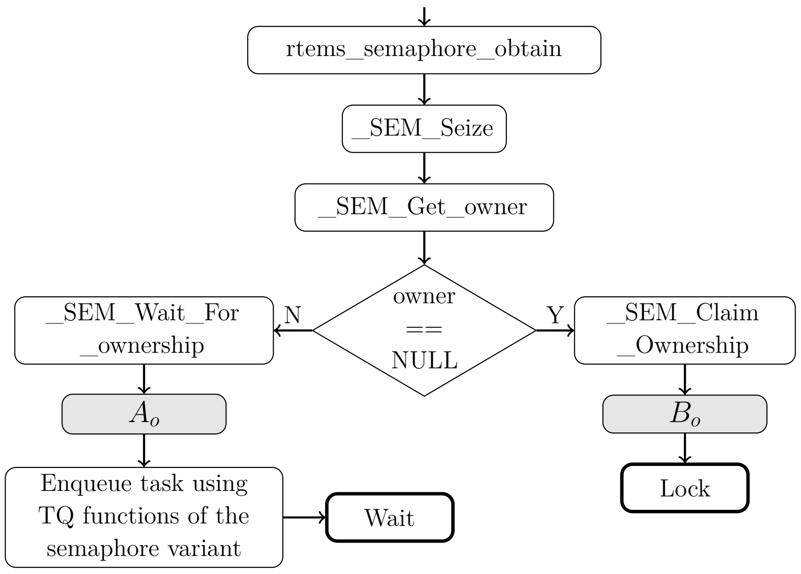


**Figure 5.1:** Workflow of the lock directive. Block $A_o$ and $B_o$ are specified according to the adopted protocols.

$\tau_i$ invokes the `_SEM_Seize` function. The semaphore's ownership is subsequently verified by examining the owner of the *Thread queue Control*. If the semaphore is locked by another task, $\tau_i$ must wait until the current owner releases the semaphore. The operations within block $A_o$ are defined based on the individual protocol designs. If there is no owner yet, $\tau_i$ usually becomes the semaphore's owner and initiates the execution of its critical section in work-conserving scenario. The actions within block $B_o$ might vary, contingent on the specific protocol design.

The workflow of the unlock directive is presented in Figure 5.2. This directive is activated when $\tau_i$ concludes its critical section's execution and intends to release the semaphore's lock. To check whether $\tau_i$ is the semaphore's current owner, the directive pinpoints the appropriate `_SEM_Surrender` function. If $\tau_i$ is not the owner, the semaphore remains locked. However, if it is the owner, $\tau_i$ can proceed to unlock the semaphore, with the main operation in block $A_r$ aimed at identifying the next potential owner if there is another task awaiting the semaphore. If no tasks are in queue, the ownership is redefined to `NULL`. The precise functions within $A_r$ are elaborated upon in subsequent sections dedicated to each protocol.

**Application Configuration** To support semi-partitioned scheduling in RTEMS, the configuration procedure outlined in Figure 5.3 should be followed. Initially, processors are associated with distinct scheduler instances using the `_RTEMS_-SCHEDULER_ASSIGN` macro, which is supported in RTEMS by default. Subsequently, tasks are assigned to specific scheduler instances through the `rtems_task_set_-scheduler` directive. Please note that a task is restricted to be executed on the processor associated with its scheduler instance.

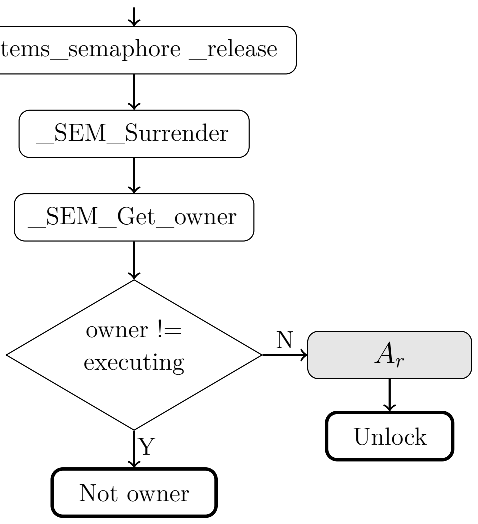


**Figure 5.2:** Workflow of the unlock directive. Block $A_r$ is specified according to the adopted protocols.

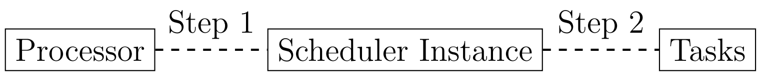


**Figure 5.3:** Configuration steps in RTEMS for semi-partitioned scheduling.

As shown in Figure 5.3, configuring an RTEMS application for SMP support necessitates the integration of multiple new functions. In the first step, a preliminary task, set to execute at the beginning of the RTEMS application, must be defined. This association between scheduler instances and processors is based on the official c-user guide. The dedicated scheduling algorithm for the scheduler instances has to be determined at first. Within the context of this dissertation, the default RTEMS-supported *Deterministic Priority SMP Scheduler* is employed, which is the same as the Fixed-Priority (FP) scheduler discussed in academic literature. Please note that to effectively support semi-partitioned scheduling, scheduler instance definitions should encompass all available processors within the system. This provision facilitates task migration across processors via adjustments in their scheduler nodes.

### DGA Implementation in RTEMS

In our DGA-based implementation within RTEMS, we specifically address the OCS task model. Similarly, a ticket system is employed in RTEMS to ensure forced order execution. Each task is allocated a set of tickets, with the ticket count equating to the number of jobs released within a single hyper-period. Sequentially released jobs from each task are designated their corresponding ticket, denoted as `job_ticket`.

The DGA semaphore control structure, as detailed in Listing 5.1, is defined by:

- `Wait_queue`: Manages the collection of tasks that are queued due to their pending requests for a shared resource.
- `ticket_order`: Stores the ticket sequence of jobs within a hyper-period. This sequence dictates the execution order of jobs that seizes this semaphore.
- `order_size`: Specifies the dimensions of the `ticket_order`.
- `current_position`: Acts as an array pointer, pinpointing the ticket of the next task permitted to lock the semaphore.

Semaphores are initialized using the `rtems_semaphore_create` function. The `current_position` is set to 0 initially. This value increments by one each time a task finishes its execution of a critical section protected by the corresponding semaphore. Once the `current_position` equals the `order_size`, it resets to 0.

```
typedef struct {
  Thread_queue_Control Wait_queue;
  Ticket_Control *ticket_order;
  int order_size;
  int current_position;
} DGA_Control;
```

**Listing 5.1:** DGA semaphore control structure.

The wait queue is implemented using the Red-Black Trees API of RTEMS. The ticket-based queue has a structure, named `Ticket_Node`, similar to the priority queues in RTEMS. This structure organizes the tickets of tasks in the wait queue and is detailed in Listing 5.2. We introduced a new field in the task control block that references its ticket node. When a task is created, it is associated with a ticket node, setting the `*owner` field of that node. The `Ticket_Control` is an integer representing the actual ticket number. Additionally, the `union` is utilized for the RBTree-based queue.

```
typedef struct {
  union {
    Chain_Node Chain;
    RBTree_Node RBTree
  } Node;
  Ticket_Control ticket;
  Thread_Control *owner;
} Ticket_Node;
```

**Listing 5.2:** Ticket Node structure.

In addition to the Chain Control structure, we added an `RBTree Control` to the thread queue heads. It assists in managing the `RBTree Nodes` introduced to the queue. The ticket queue now requires a new function table, as illustrated in Listing 5.3. The operations for the thread queue are listed as follows:

- `.priority_actions`: Priority actions are not defined for this semaphore because there is no need for priority-related actions.

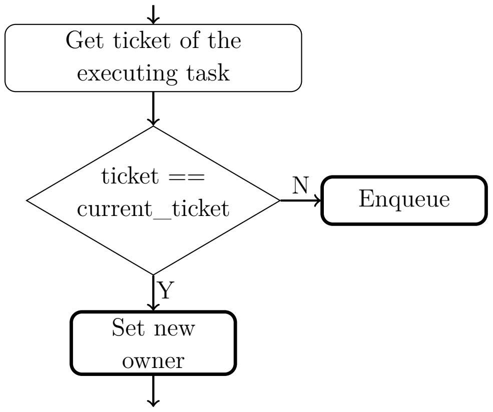


**Figure 5.4:** Locking operation of the DGA semaphore.

- `.enqueue`: If the corresponding RBTree has been initialized, the RBTree Node from the Ticket Node structure will be identified and positioned. If not, the RBTree will be initialized, the current node is served as the initial node.
- `.extract`: This operation removes a node from the RBTree.
- `.surrender`: This makes use of the *first* directive to identify the RBTree Node with the lowest ticket number, subsequently extracting it from the queue.
- `.first`: It identifies the node with the lowest ticket number in the tree. It then uses the `RTEMS_CONTAINER_OF` macro to get the associated `Ticket Node`. Finally, it determines the owning task for this node using the `*owner` pointer.

```
const Thread_queue_Operations _Thread_queue_Operations_TICKET = {
  .priority_actions = _Thread_queue_Do_nothing_priority_actions,
  .enqueue    = _Thread_queue_TICKET_enqueue,
  .extract   = _Thread_queue_TICKET_extract,
  .surrender   = _Thread_queue_TICKET_surrender,
  .first    = _Thread_queue_TICKET_first
};
```

**Listing 5.3:** Ticket related operations in thread queue.

The workflow for block $B_o$ in Figure 5.1 is detailed in Figure 5.4. When task $\tau_i$ requests a shared resource, it verifies the ownership status of the requested shared resource (semaphore). Contrary to work-conserving resource synchronization protocols, a task cannot directly lock the semaphore, even if the resource's owner is `NULL`. The task's `job_ticket` has to be compared with the semaphore's `current_-ticket`. A match authorizes the job to access the shared resource. Otherwise, the task is enqueued in a wait queue. This queue is organized in increasing order based on ticket numbers.

The workflow for block $A_r$ in Figure 5.2 is demonstrated in Figure 5.5. Once task $\tau_i$ completes its execution of critical section, it releases the semaphore. Subsequently, the value of `current_ticket` associated with the semaphore increments by 1. If

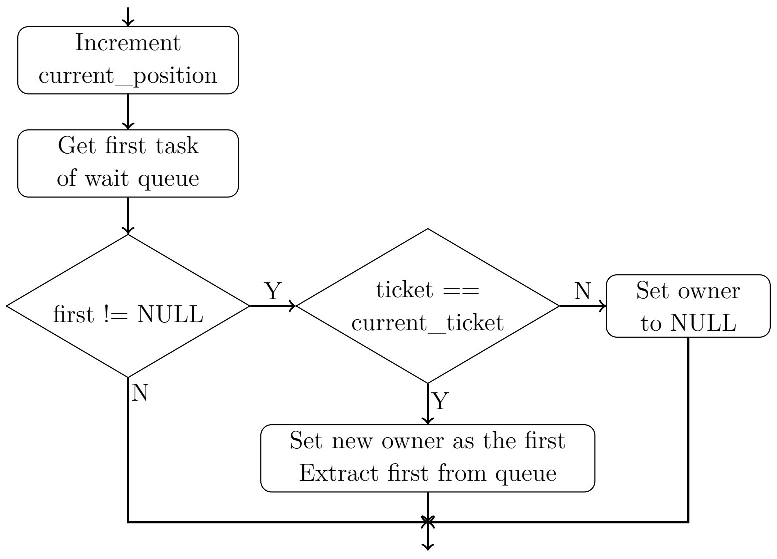

Increment current_position
Get first task of wait queue
first != NULL
Y
N
ticket == current_ticket
N
Y
Set owner to NULL
Set new owner as the first Extract first from queue

**Figure 5.5:** Releasing operation of the DGA semaphore.

the new ticket value matches `total_tickets`, which represents the total count of critical sections within a single hyper-period, then `current_ticket` is reset to 0. In scenarios where the wait queue is empty, the semaphore's owner is set to `NULL`. Otherwise, if the first task in the queue has a `job_ticket` equivalent to the updated `current_ticket`, then the new task is set as the new semaphore owner.

### Overheads Comparison in RTEMS

We conducted an evaluation of the operational overheads associated with our DGA implementation on an NXP QorIQ T4240 RDB reference design board. A comprehensive description of the hardware configurations is provided in Section 2.4.3.

Unlike LITMUS$^{RT}$, RTEMS lacks inherent tools for overhead measurement. To trace the overheads of our DGA implementation, we integrated timestamps both before and after the functions within our design. We specifically measured the `Seize` (commonly denoted as 'lock') and `Surrender` (or 'unlock') semaphore functions. Since MrsP is the only officially supported multiprocessor resource synchronization protocol in RTEMS, we limited our overhead comparison to MrsP and our DGA implementation. It is important to note that RTEMS does not support synchronized task releases by default, as also reported in [LGS+23]. This deviation disrupts the DGA's foundational assumption where all tasks release their first jobs synchronously, ensuring a consistent schedule within each hyper-period. After extended system operation, the ticket system's execution may deviate from its original design, potentially resulting in deadline misses. Our implementation only focuses on the forced sequential execution of critical sections for each shared resource. However, addressing synchronized release is out of the scope of this dissertation.

Table 5.3 demonstrates the overheads for two protocols. The data shows that the overheads of our DGA implementation align closely with those of MrsP. Directly comparing the overheads between the DGA implementations in RTEMS and LITMUS$^{RT}$ is not possible, due to the different objectives and architectures of the two RTOSes. We observed that the recorded overheads for both MrsP and our DGA are considerably reduced compared to those documented in [CBH+15; SPM+22], even when using the same testing script and hardware as presented in [SPM+22]. We attribute this performance enhancement to the update from version 4 to 5 of the RTEMS kernel. Nevertheless, a detailed exploration of this specific aspect was not included in this dissertation.

**Table 5.3:** Overheads of protocols in RTEMS in $ns$.

| Max. (Avg.) | Seize (Lock) | Surrender (Release) |
|---|---|---|
| MrsP | 255 (131) | 254 (134) |
| DGA-OCS | 247 (146) | 251 (133) |

## 5.4 Formal Verification Framework

In this section, we present the framework for the formal verification of resource synchronization protocols. First, we introduce the fundamental concept of deductive verification. This is followed by a discussion on the common assumptions made within our framework. We then illustrate the workflow of our formal verification framework. Finally, we discuss the necessary pre-processing steps required for the RTOS source code.

### 5.4.1 Deductive Verification

In this subsection, we first introduce the foundational elements of Hoare Triple and the concept of the Weakest Precondition. Afterwards, we explain the function contracts formulated by ACSL as applied within the context of Frama-C.

#### Hoare Triple and Weakest Precondition

A *Hoare Triple* of the form *PreprogramPost* is formulated, to specify a certain property of a program. In this structure, the postcondition *Post* holds if the *program* is executed with a fulfilled precondition *Pre*. If a postcondition is supposed to hold in any possible case, the precondition is just *true*. To verify such a property against a program, its *weakest precondition* that required to satisfy the post-condition is derived. If the defined precondition *Pre* implies the derived weakest precondition, the property is proven to hold for the analyzed program. The development of the weakest precondition is often performed backwards through the code by iteratively transforming the postcondition based on the code statements using the rules defined by Hoare [Gar19].

We provide an example using the function `absDiv(x,y)`, which divides $|x|$ by $|y|$, as shown in Listing 5.4. Since dividing by zero is not permitted (with the function

```
int absDiv(int x, int y){
  int d1, d2, res;
  if (x >= 0) {d1 = x;}
  else {d1 = x * -1;}
  if (y > 0) {d2 = y;}
  else if (y < 0) {d2 = y * -1;}
  else {return -1;}
  res = d1 / d2;
  return res;
}
```

**Listing 5.4:** One implementation of the function absDiv.

returning −1 in such cases), the desired behavior (*Property*) can be formulated as follows, where $\backslash res$ is the value returned by `absDiv`:

$$\{y \neq 0\} absDiv \{\backslash res = |x|/|y|\} \tag{5.1}$$

Furthermore, the result is expected to be non-negative since both the divisor and dividend are non-negative. Therefore, a new *Property* can be formulated:

$$\{y \neq 0\} absDiv \{\backslash res \geq 0\} \tag{5.2}$$

Statement (5.3) based on an implementation in Listing 5.4 concludes the derivation of the weakest precondition for *Property* (5.1).

$$\begin{aligned}
x \geq 0 \Rightarrow [&(y > 0 \Rightarrow \frac{x}{y} = \frac{|x|}{|y|}) \wedge (y < 0 \Rightarrow \frac{x}{-y} = \frac{|x|}{|y|}) \wedge \\
&(y = 0 \Rightarrow -1 = \frac{|x|}{|y|})] \wedge \\
x < 0 \Rightarrow [&(y > 0 \Rightarrow \frac{-x}{y} = \frac{|x|}{|y|}) \wedge (y < 0 \Rightarrow \frac{-x}{-y} = \frac{|x|}{|y|}) \wedge \\
&(y = 0 \Rightarrow -1 = \frac{|x|}{|y|})]
\end{aligned} \tag{5.3}$$

By replacing the $= \frac{|x|}{|y|}$ with $\geq 0$, the weakest precondition for *Property* (5.2) can be obtained as well.

## ACSL Function Contracts in Frama-C

To verify a function's specification consisting of Hoare-triples, also called *function contract*, against the source code of the implementation of a targeted protocol, Frama-C can be applied [FRA23]. The plugin *wp* (weakest precondition) of Frama-C provides the capabilities for static analysis and deductive verification of source code. The contracts are formulated by using the *ANSI/ISO C Specification language* (ACSL) [BCF+21]. It allows users to formally specify the behavior(s) of a function as a function contract in the form of annotations to its source code enclosed in special comments, i.e., `//@` or `/*@ ... */`.

Contracts can consist of different *behaviors*, each of which ensures a set of postconditions depending on different preconditions or assumptions and may be declared to be *complete* or *disjoint*. To ease the formulation of a specification, constructs like *predicates*, *logic functions* and *assertions* are provided. A predicate evaluates its parameters and returns either `true` or `false`. Predicates can be used within assertions, function contracts or other predicates. Logic functions can have any return type and perform assignments or computations. An ACSL function contract for the previous example is given in Listing 5.5. It describes the Property (5.1) and Property (5.2). The behaviors are declared to be disjoint, i.e., no two behaviors can occur as a consequence of one set of inputs. When Frama-C is invoked and given the annotated code of a function, the contract can be verified against the implementation with *wp*.

```
/*@ // auxiliary predicates and logic_functions
predicate IsZero(int x) = x == 0;
predicate NonZero(int x) = ! IsZero(x);
logic int abs_div(int x, int y) = \abs(x) / \abs(y);
*/
/*@ // function contract
behavior err:
requires IsZero(y);
ensures \result == −1;
behavior div:
requires NonZero(y);
ensures \result >= 0;
ensures \result == abs_div(x,y);
complete behaviors;
disjoint behaviors;
*/
int absDiv(int x, int y) {...}
```

**Listing 5.5:** An example of ACSL function contract using predicates, logic functions and builtin functions (\abs)

Another ACSL concept, i.e., *ghost code*, enables the use of C code within annotations, which is helpful when specifying more complex behaviors, e.g., loops. Ghost code makes implicit information explicitly visible and addressable in function contracts without affecting the behavior of the original source code under analysis [Bla21]. In this dissertation, *ghost code* is applied to transfer stateful information along a call hierarchy (in Listing 5.9) and to abstract from low-level operating system mechanisms (in Listing 5.12).

A *memory model* is employed to map the analyzed high-level memory concepts of types and pointers to a mathematical representation. An example is *wp*'s default *typed* memory model. To aid the abstraction, memory locations and pointers can be annotated with several terms and predicates predefined in ACSL. A valid pointer `p` that can be safely dereferenced, is declared by `\valid(p)`. All the modified memory locations are listed in the `assigns` clause within the contract. A function or its behavior that has no side-effects and assigns no non-local memory can be annotated with `assigns \nothing` [BCF+21]. In addition, the ACSL annotations are preprocessed and integrated into the *Abstract Syntax Tree* (AST), which is built using a modified form of the *C Intermediate Language* (CIL). It specifies the transformation of C programs into a reduced subset of C, which abstracts from low-level language concepts, supports compiler-specific extensions and facilitates automatic analyses. Furthermore, the program is type-checked during the transformation. Afterwards, several syntactical transformations are performed, e.g., a unified representation for loops and conditional branches and the removal of "syntactic sugar" like the convenience operator for dereferencing pointers [CKK+12; CCK+21; NMR+02].

Analyses with *wp* are launched either for a complete function contract or for its properties individually. For the selected properties, proof obligations are generated in a *wp*-own syntax that describe the goals to be proven based on the first order logic representation of the analyzed code and its annotations. These obligations are simplified by the builtin *Qed* engine, by either fully resolving them or adding further conditions facilitating the proof [Cor14]. If they are not resolvable by Qed, obligations are forwarded to an automatic SMT prover in the form of a *Why3* script [BFM+11]. If existing provers are not sufficient, interactive proof assistants such as Coq [Inr] can also be utilized to complete the verification [BBC+21].

### 5.4.2 Common Assumption

In our formal verification framework, only the implementation of a protocol is verified. The proposed framework is applied to verify the correctness of the implementation for all the specified properties from a given protocol, i.e., function contracts. Any other components that are not specified in the protocol definition are assumed to be functionally correct. We choose RTEMS as our target RTOS. The abstracted layers of the verification concept can be viewed in Figure 5.6. More precisely, the proposed framework assumes that an implemented resource synchronization protocol is based on a correct underlying operating system.

In order to verify the implementation layer separately from its underlying layers with deductive program verification, several abstractions have to be applied. First of all, the verification scope does not include the basic locking and scheduling operations, e.g., mutexes, queues and threads. The mutually exclusive execution of critical sections is considered as a part of the dependencies which are assumed to be correctly implemented. Furthermore, no notion of time is considered. Due to the verification perspective and the assumptions on underlying OS concepts, temporal

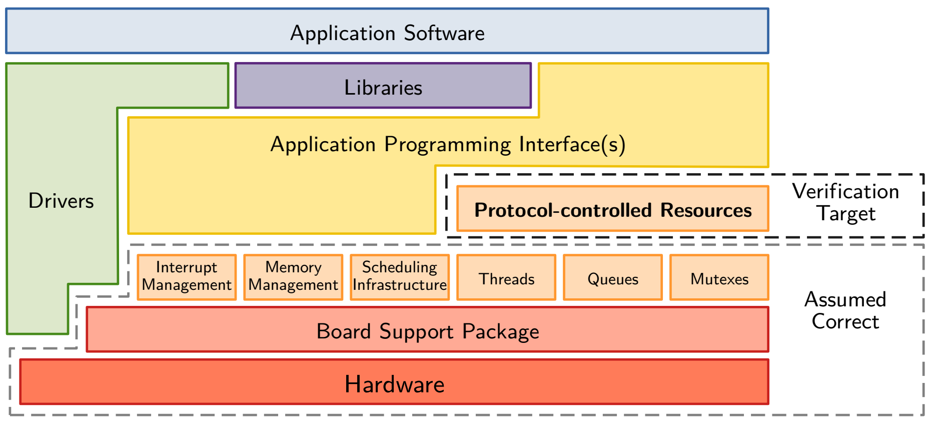


**Figure 5.6:** Abstracted layers of the verification concept within an RTOS.

properties are not necessary to verify the protocol specifications. Thus, if all the determined properties of a protocol have been verified to be implemented correctly, the protocol in the OS is formally verified.

### 5.4.3 Workflow of Formal Verification Framework

Verifying the implementation means we do *not* verify its compiled (i.e., compiler- and architecture-specific) results *nor* its runtime behavior. Instead, the source code of the corresponding implementation should be verified with formal specifications of all required properties of the resource synchronization protocol. This can be achieved by deductive program verification, which proves whether a program fulfills a set of post-conditions when assuming a set of preconditions.

When the deductive verification approach is applied, the implementation under verification can be written in high-level programming languages. Additionally, to allow for the separation of protocol-specific code and relied-upon OS primitives, the analysis should be performed in a modular way. That is, a set of conditions for a function body should be verified based on its statements and, for further called functions, based on their formal specification only. These called functions either need to be verified if they are specified by the protocol or can be assumed to be correct if they are provided by the RTOS. Hence, verification is confined to a specific layer or depth. The protocol-specific code that is to be verified and its dependencies have to be distinguished clearly. Precisely, the workflow of our framework can be described as follows:

1. Identify the subset of the targeted OS's source code that represents the protocol implementation.

2. Identify the resource synchronization protocol's properties and rules, e.g., the request and release of resources.
3. Design specifications consisting of Hoare-triples for the behavior of all utilized OS primitives.
4. Design specifications consisting of Hoare-triples for the protocol implementation based on the formal specification of the protocol.
5. Verify the specification of the implementation against its source code with deductive program verification, aided by appropriate software.

Please note that, unlike implementing with formal verification from scratch, our approach focuses on verifying an already-implemented synchronization protocol against its formal properties. This implementation is based on abstractions derived from the operating system or hardware specifications. In addition, the protocol's implementation is typically integrated throughout the operating system, and its implementation flow might not strictly adhere to the formally described properties. Therefore, point-to-point verification, i.e., directly linking each formally described property and its specification to the real implementation, is usually not possible.

In our verification framework, the source code of the implemented resource synchronization protocol is analyzed at first. All the implemented properties must be converted and matched to the formally described properties, and the specifications of Hoare triples need to be designed in accordance with these formal properties while also matching the abstractions of the implementation being verified.

### 5.4.4 Preprocessing the source code

To apply Frama-C for verification, the proposed framework only needs to utilize the cross-compilation toolchain without building or executing the OS code. However, the source code of the targeted RTOS needs to be preprocessed to resolve the inclusion dependencies and gain meta-information from macros and customized data types [CCK+21]. In order to describe the customized data types in the memory model, information on the bit width is also required. Furthermore, some header files are architecture-dependent. These headers may come with the cross-compilation toolchain or be generated during the source configuration.

To analyze the implementations in RTEMS, a separate source configuration is generated for uni-processor and multi-processor systems, respectively. To avoid compatibility problems, 32-bit PowerPC is chosen as the target architecture for the toolchain, which comes with SMP support in RTEMS and is supported by Frama-C as well [CCK+21]. Building the toolchain yields the required header `stddef.h`. Please note that the term *thread* in RTEMS is synonymous with *task* in this work and in real-time systems literature.

## 5.5 Verification of Protocols in RTEMS

In this section, we detail the verification process for officially supported ICPP and MrsP, and the new implemented DGA in RTEMS.

### 5.5.1 Verification of ICPP in RTEMS

In this section, we adopt the proposed verification framework to verify the Immediate Ceiling Priority Protocol (ICPP) officially implemented in RTEMS [BW09], which is a well-known synchronization protocol for uni-processor real-time systems [But11]. It is commonly considered as an advanced variant of the PCP [SRL90], as it has the same upper bound on the blocking time but less context switches. However, the standard implementation of ICPP has not been previously discussed. Any mismatch between the implementation and the formally proved properties can potentially lead to an error, e.g., deadlock.

Throughout our verification framework, we find out that the current implementation is in fact *not deadlock free*. To reach this serious conclusion, in the following, we present how we declare the function contracts for ICPP to employ our verification framework, and give a concrete example to illustrate how the deadlock can occur under the current implementation. The verified functions are listed in Table 5.4, where the `_CORE_ceiling_mutex` is the common prefix of all function names in the table. All the required properties of ICPP are as follows:

1. For a resource $z_j$, the priority ceiling is defined as $\Pi(R_j) = \max\left\{\pi(\tau_i) \mid \tau_i \text{ requests } R_j\right\}$, where $\pi(\tau_i)$ is the priority of task $\tau_i$.
2. The set of the resources' priority ceilings that a task $\tau_i$ holds at time $t$ is denoted as $C_{\tau_i,t} = \left\{\Pi(R_j) \mid \tau_i \text{ holds } R_j \text{ at time } t\right\}$
3. At any time $t$, a task runs at the highest priority among its base priority and the priority ceilings of its held resources: $\pi(\tau_i, t) = \max\left\{\pi(\tau_i),\ C_{\tau_i,t}\right\}$
4. Whenever a task $\tau_i$ requests a resource $z_j$ at time $t$, it is granted access and it immediately inherits $z_j$'s priority ceiling: $C_{\tau_i,t} = C_{\tau_i,t-1} \cup \left\{\Pi(R_j)\right\}$. Task $\tau_i$ executes its critical section with the priority following Rule 3.
5. When task $\tau_i$ releases a resource $z_j$ at time $t$, its priority ceiling is revoked from the set, i.e., $C_{\tau,t} = C_{\tau,t-1} \setminus \left\{\Pi(R_j)\right\}$. Afterwards, task $\tau_i$ is executed with its original priority if there is no following critical section, or it is executed for its next critical section with the priority derived by following Rule 3.

In the ICPP implementation of RTEMS, after a task successfully locks the semaphore[2], then the priority of the task is elevated to the ceiling priority if the original priority is lower than the ceiling priority. When a task or thread waits on a semaphore, it is added into a data structure, named as `thread_queue`. The priority queuing

[2]The locking protocols are originally for mutex, but they are realized by binary semaphores in RTEMS, which are technically as mutex locks. Here, we stick to the terminology of locking protocol to 'lock' a semaphore.

**Table 5.4:** Functions for acquiring and releasing a resource under ICPP.

| **Protocol Function** | **Purpose** |
|---|---|
| `_Seize` | Acquire an available or self-locked resource |
| ↪ `_Set_owner` | Check and inherit resource ceiling, set resource owner |
| `_Surrender` | Release a locked resource |

discipline simply orders the threads according to their current priority and in FIFO order in case of equal priorities.

### Preprocessing

Before the verification, we decouple the implementation of ICPP into two parts: a) the protocol-specific parts that will be analyzed, and b) the employed OS functionalities. To lock and unlock a resource, the RTEMS Classic API exposes the functions `rtems_semaphore_obtain` and `rtems_semaphore_release`. These functions lock an actual semaphore object from the passed system-wide ID and perform the demanded actions depending on its type. For a semaphore controlled by ICPP, the corresponding functions `_CORE_ceiling_mutex_Seize` and `_CORE_ceiling_mutex_Surrender` are called, where mutex locks are applied as binary semaphores to protect shared resources. Besides, another protocol-specific function is `_CORE_ceiling_mutex_Set_owner`. The remaining functions are lower-level primitives, which provide operations to update a thread's priority, achieve basic mutual exclusion for data consistency, and access the underlying non-protocol *Core Mutexes* and queues, are assumed to be implemented correctly. Since the implementation of a protocol may be spread across the source base, two header files are created to bundle these functions' specifications: a) `fc_common_stubs.h` is used for all implemented protocols; b) `fc_icpp_stubs.h` contains the ICPP-specific stub definitions.

### Abstractions and Function Contracts

The OS utilities are treated in two different ways when they are annotated to declare their (intended) behavior in the analysis. On the one hand, functions that have no effect in the analyzed situation, are *bypassed.* That is, the annotations do not declare their behavior, but assert their invocation has no side effects and can be ignored during verification. Listing 5.6 shows an example for bypassing calls for basic locking pairs, where the interrupt has already been disabled when the function is called in a uni-processor system. On the other hand, OS functions that perform actions that are not considered as a part of the protocol analysis but are critical to the ICPP implementation, have to be annotated with a description of their intended (and considered correct) behavior. The example in Listing 5.7 shows a contract that ensures the thread's priority either remains the same or corresponds to the passed priority node after its execution.

```
//@ assigns \nothing;
RTEMS_INLINE_ROUTINE void _CORE_mutex_Acquire_critical(
CORE_mutex_Control *the_mutex,
Thread_queue_Context *queue_context );
```

**Listing 5.6:** Bypassing of a system utility function.

```
/*@
requires \valid(the_thread) && \valid(priority_node);
assigns *the_thread, g_thread_inherited, g_prio_node;
ensures g_thread_inherited == the_thread && g_prio_node == priority_node;
behavior inherit_higher:
assumes priority_node->priority < Current_Priority(the_thread);
ensures Current_Priority(the_thread) == priority_node->priority;
behavior inherit_lower_or_equal:
assumes priority_node->priority >= Current_Priority(the_thread);
ensures Current_Priority(the_thread) ==
\old(Current_Priority(the_thread));
disjoint behaviors;
complete behaviors;
*/
void _Thread_Priority_add(
Thread_Control *the_thread,
Priority_Node *priority_node,
Thread_queue_Context *queue_context );
```

**Listing 5.7:** The system utility function adds a priority node to a thread.

### Contracts for ICPP-Seize

Once all necessary OS functions have been provided with contracts, the actual behaviors of the protocol operations *seize* and *surrender* can be specified. The following two properties are verified for seize in a pure ICPP:

- A task requesting a resource is granted the resource.
- After a resource is granted, the task runs on the highest ceiling priority of all currently held resources.

The implementation in RTEMS considers and checks more possible cases, which are not formally described by the protocol specifications. Overall, these additional scenarios lead to further properties:

- Acquiring a resource fails, if its priority ceiling is lower than the acquiring thread's base priority.
- Resources may be acquired again by the holding thread before release. The level of self-nested access is tracked.
- Acquiring a locked resource enqueues the thread into the resource's priority-based waiting queue. Such a request operation can be either successful by obtaining the resource eventually or failed if the request times out.

All the above five implemented properties are derived from the formally described properties 1-4 of ICPP. Therefore, if all these properties can be successfully verified, it implies that the first four formally described properties have been verified.

The check of the acquiring task's base priority is a legitimately safe measure to compensate incorrectly priority ceilings setting or unallowed resource accesses. Locking an already locked resource does not affect the ICPP. The last case happens only if a task suspends while holding a resource. This behavior is not considered in the definition of ICPP and would break the property. Towards this, the precondition is necessary that the requested resource is either available or locked by the current requesting task (Listing 5.8, Lines 4-5), which matches the formal property of ICPP, i.e., once a task starts its execution, all required resources must be available [BW09]. Therefore, the third case in the list is excluded in the analysis, which makes annotations for the responsible function `_CORE_mutex_Seize_slow` unnecessary.

These collected properties can be formulated as a function contract for the function `_CORE_ceiling_mutex_Seize`, shown in Listing 5.8. The preconditions in Lines 2-3 require that the pointers to the mutex and the executing thread are valid, i.e., dereferenceable, and their memory regions do not overlap. The precondition in Lines 4-5 expresses the invariant for a seize operation under ICPP, that the requested resource is free, and adds the situation that the shared resource has been acquired by the requesting task already. The default behavior, termed `behavior seize_successful`, ensures that the requesting thread locks the resource, and subsequently inherits the resource's ceiling priority. To ensure that the priority inheritance is passed from the acquired resource to the requesting thread, the `PriorityInherited` predicate is introduced (as shown in Listing 5.9). It checks if the passed thread and priority are the same as those set in the contract of `_Thread_Priority_add` (Listing 5.7) and whether the priority of the thread is updated after the change of the priority aggregation. The inheritance does not necessarily lead to a priority raise, since the thread may already hold a resource with a higher ceiling.

Within the seize function, the second relevant protocol-specific function from Table 5.4 is the `_Set_owner` function in Listing 5.10. It performs the check of the resource ceiling. If valid, the executing thread inherits the acquired resource's priority ceiling and is set as the resource owner. If the resource ceiling is violated due to the thread's priority, the operation fails. The conditions for the ceiling and the ceiling priority inheritance are known from the invoking seize function.

### Contracts for ICPP-Surrender

The contract for the ICPP surrender function, `_CORE_ceiling_mutex_Surrender`, can be designed in a similar sense. Since the ICPP does not allow threads to be enqueued and waiting for the shared resource, the contract is constructed with a precondition that the resource's queue has to be empty. The revocation of the formerly inherited priority is guaranteed by another predicate, i.e., `PriorityRevoked`.

```
/*@
requires \valid(the_mutex) && \valid(executing);
requires \separated(the_mutex, executing);
requires Mutex_Owner(the_mutex) == NULL ||
Mutex_Owner(the_mutex) == executing;
behavior seize_ceiling_violation:
assumes Mutex_Owner(the_mutex) == NULL && Base_Priority(executing) <
Mutex_Priority(the_mutex);
ensures \result == STATUS_MUTEX_CEILING_VIOLATED;
behavior seize_successful:
assumes Mutex_Owner(the_mutex) == NULL && Base_Priority(executing) >=
Mutex_Priority(the_mutex);
ensures PriorityInherited(executing, Mutex_Priority(the_mutex));
ensures Current_Priority(executing) <= Mutex_Priority(the_mutex);
ensures Mutex_Owner(the_mutex) == executing;
ensures \result == STATUS_SUCCESSFUL;
behavior seize_nested:
assumes Mutex_Owner(the_mutex) == executing;
assumes nested == _CORE_recursive_mutex_Seize_nested;
assigns the_mutex->Recursive.nest_level;
ensures Nest_Level(the_mutex) == \old(Nest_Level(the_mutex)) + 1;
ensures \result == STATUS_SUCCESSFUL;
disjoint behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _CORE_ceiling_mutex_Seize(
CORE_ceiling_mutex_Control *the_mutex,
Thread_Control *executing,
bool wait,
Status_Control ( *nested )( CORE_recursive_mutex_Control * ),
Thread_queue_Context *queue_context
) { /*...*/
  //@ calls _CORE_recursive_mutex_Seize_nested;
  status = ( *nested )( &the_mutex->Recursive );
  /*...*/ }
```

**Listing 5.8:** Contract declaring the ICPP functionality for the seize operation

The thread's dynamic priority after surrendering the resource is either lower than before, or remains the same if another resource with the same ceiling is still held. This is derived from the fifth formally described property.

### Verification and Mismatch

Due to the modular analysis, the seize function can be analyzed without inspecting the code of the called `_CORE_ceiling_mutex_Set_owner`. Instead, only its contract is used. Once the function under analysis fulfills that contract's preconditions, its postconditions are assumed to be fulfilled. This analysis successfully proves all stated properties. However, when attempting to verify the called set owner function, the verification fails to prove the ceiling check and parts of the successful acquisition.

```
/*@ ghost // variables declared in coremuteximpl.h
extern Thread_Control *g_thread_inherited;
extern Thread_Control *g_thread_revoked;
extern Priority_Node *g_prio_node;
extern bool prioritiesUpdated; */
/*@ predicate PriorityInherited(Thread_Control *t, Priority_Control p) =
t == g_thread_inherited && p == g_prio_node->priority && prioritiesUpdated;
*/
```

**Listing 5.9:** The predicate `PriorityInherited` checks priority inheritance.

```
/*@
requires \valid(the_mutex) && \valid(owner);
requires \separated(the_mutex, owner);
behavior set_owner_ceiling_violation:
assumes Base_Priority(owner) < Mutex_Priority(the_mutex);
ensures \result == STATUS_MUTEX_CEILING_VIOLATED;
behavior set_owner_successful:
assumes Base_Priority(owner) >= Mutex_Priority(the_mutex);
assigns *owner, *the_mutex, prioritiesUpdated, g_thread_inherited, g_prio_node;
ensures Current_Priority(owner) <= Mutex_Priority(the_mutex);
ensures PriorityInherited(owner, Mutex_Priority(the_mutex));
ensures Mutex_Owner(the_mutex) == owner;
ensures \result == STATUS_SUCCESSFUL;
disjoint behaviors;
complete behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _CORE_ceiling_mutex_Set_owner(
CORE_ceiling_mutex_Control *the_mutex,
Thread_Control *owner,
Thread_queue_Context *queue_context ){/*...*/}
```

**Listing 5.10:** Contract for the `_Set_owner` function.

The reason for the incapability to fulfill the conditions can be tracked down with further annotations.

After a successful ceiling check, the task's base priority is assumed to be lower or equal to the resource's ceiling. However, this assertion cannot be verified. We note that the resource's ceiling is not checked against the thread's base priority, but against its current dynamic priority derived from the task's priority aggregation. However, a resource's ceiling is required to be set as the highest base priority of all tasks that are requesting it. This mismatch may lead to a deadlock by erroneously denying legitimate nested resource access if resources are requested with descending order of priority ceilings. We give an example to illustrate such a case:

Consider two tasks $\tau_1$ and $\tau_2$. The priority of $\tau_1$ is greater than that of $\tau_2$, i.e., $\pi(\tau_1) > \pi(\tau_2)$. There are also two resources: $R_1$, which is accessed by both tasks, and $R_2$, which is only accessed by $\tau_2$. Their priority ceilings are set as $\Pi(R_1) = \pi(\tau_1)$ and $\Pi(R_2) = \pi(\tau_2)$, respectively. If it requests the second resource $R_2$, its dynamic priority is higher than $\Pi(R_2)$, which leads to a denial of the resource access by the implemented ceiling check. The consequence of this is a *deadlock*, i.e., $\tau_2$ holds $R_1$ but cannot successfully lock semaphore $R_2$ due to the implemented ceiling check, whilst $\tau_1$ cannot enter the critical section guarded by $R_1$. Such execution behavior with a deadlock can also be demonstrated by a running example in RTEMS, which will be released on Github. An acquisition in the opposite order would be accepted.

To correct the mismatch, an adaption to the priority ceiling check is proposed in Listing 5.11 for `coremuteximpl.h`. The current method checks the thread's potentially elevated dynamic priority. In our proposal, it would compare the thread's base priority with the priority ceiling of the newly requested resource. After applying the correction, all stated properties are successfully verified.

```
if (
owner->Real_priority.priority
< the_mutex->Priority_ceiling.priority
) {
  _Thread_Wait_release_default_critical( owner, &lock_context );
  _CORE_mutex_Release(&the_mutex->Recursive.Mutex, queue_context);
  return STATUS_MUTEX_CEILING_VIOLATED;
}
//@ assert Base_Priority(owner) >= Mutex_Priority(the_mutex);
```

**Listing 5.11:** Proposed correction for the priority ceiling check.

### 5.5.2 Verification of MrsP in RTEMS

In this section, we verify the MrsP [BW13] officially implemented in RTEMS, which is designed for semi-partitioned fixed priority task systems on multiprocessors. We adopt our verification framework to ensure whether the corresponding implementation derives the specified properties of the MrsP. One highlight of the MrsP is the help mechanism that employs a spin-waiting task to progress the execution of the current blocked task which holds the resource. However, a seize operation can be performed while being scheduled in the presence of the help mechanism. This requires the priority ceiling of the seized resource to be determined with a caution, which is of key interest in this work.

The protocol functions that are going to be verified are listed in Table 5.5, where `_MRSP` is the common prefix of all function names in the table. From the verification perspective, similar preprocessing in Section 5.5.1 is necessary for the implementation of MrsP as well. The help mechanism can be assumed to be implemented correctly as other OS utilities, as long as dynamic priorities and ceilings are managed correctly. In addition, the verification is based on one arbitrary thread that performs the analyzed operation. Any other threads which might interact with it are assumed to behave correctly. A successful verification implies that this assumption holds as well. The properties of the original MrsP are as follows:

1. Each task $\tau_i$ is assigned to a specified processor $P_m$, and critical sections are executed locally, unless the *help mechanism* is applied.
2. Each resource $z_j$ has one local priority ceiling for each processor $P_m$, which is defined by the highest priority of every task assigned to $P_m$ that requests the resource: $\Pi(R_j, P_m) = \max\left\{\pi(\tau_i) \mid \tau_i \text{ requests } R_j \text{ on } P_m\right\}$.
3. For local resources that are not shared between processors, the ICPP rules are applied.
4. For global resources, the ICPP inheritance mechanism is applied with their local priority ceilings. If the requested global resource is not available, the requesting tasks spin-wait on their own processor in a FIFO order.
5. Help mechanism: a spin-waiting task for accessing to a resource must be able to *help* (by offering its computation time to) the current owner of the resource in case the owner is preempted within the critical section.

In fact, the help mechanism in the original design of the MrsP by Burns and Wellings [BW13] can cause additional local blocking, since threads are allowed to acquire priority-promoting resources while being helped on other processors, which may preempt threads dispatched on their home processors. Garrido et al. [GZB+17] suggested to resolve this issue by postponing the effect of inherited priorities to the time when the thread returns to its home processor. The verified implementation coincidentally realizes the same concept by dispatching idle threads to run subsidiary for threads that migrate to seek help by Catellani et al. [CBH+15].

### Abstractions and Function Contracts

The multiprocessor setup requires further abstractions and adaptations. While some OS utilities' stub contracts designed for the verification of ICPP can be reused, the others need to be wrapped with a new contract. For example, the function in Listing 5.12 retrieves a thread's *home node*, i.e., the scheduler node for its original processor (The first formally described property). However, it is retrieved as a chain element via several nested function calls and then extracted by a macro. This macro is expanded over multiple definitions and is eventually based on a compiler-specific offset function, which is not able to be formulated in ACSL contracts or logic functions.

Since the derivation reaches deeply into the OS specific functions, it becomes a target for abstraction. Instead of tracing the complete call and macro hierarchy, we declare a global ghost pointer `g_homenode` of the type `Scheduler_Node` to represent the executing thread's home scheduler node in the context of the verification. The `getter` function is specified by an ACSL contract to return a reference to that scheduler node in Listing 5.12. The ghost object is then said to be valid by the preconditions of the verified functions, which ensures the validity of dereferencing and access to its fields. Therefore, we assume the ghost object is equivalent to a scheduler node retrieved by the original utility function.

```
// variable defined in mrspimpl.h
//@ ghost extern Scheduler_Node *g_homenode;
/*@ requires \valid(the_thread);
assigns \nothing;
ensures \result == g_homenode; */
RTEMS_INLINE_ROUTINE Scheduler_Node *_Thread_Scheduler_get_home_node(const
    Thread_Control *the_thread);
```

**Listing 5.12:** Abstraction of the function that returns a thread's home node.

**Table 5.5:** Functions for acquiring and releasing a resource under MrsP.

| **Protocol Function** | **Purpose** |
|---|---|
| `_Seize` | Acquire an available or wait for a locked resource |
| ↪ `_Claim_ownership` | Performed if the resource is free |
| ↪ `_Wait_for_ownership` | Performed if the resource is used on another processor |
| ↪ `_Raise_priority` | Always performed to run at the resource's local ceiling |
| `_Surrender` | Release a locked resource and pass it to the first (if any) waiting task |

We also need to abstract the local resource priority ceilings for each processor (The second formally described property). A task has to raise its priority to the local priority ceiling of the resource that is requested. From the verification perspective, the task's assigned processor may be arbitrary, but fixed, and can be modeled by another ghost variable `const int g_core`. The maximum number of processors configured in the architecture-specific `cpu.h` files is 32, which can be formulated as a constraint for the variable `g_core`. It also defines the number of valid entries for a resource's local ceilings. The detailed function contract is omitted here, but is available in our Github release [Egi22].

The inheritance and revocation of priorities are modeled by the predicates `PriorityInherited` and `PriorityRevoked` similarly to the verification of ICPP.

The deep integration of the helping mechanism (the fifth formally described property) into the scheduler infrastructure of RTEMS allows it to be separated from MrsP's remaining properties. From a verification perspective, the helping mechanism can be assumed to be implemented correctly, similar to other operating system utilities, as long as dynamic priorities and ceilings are managed correctly. Furthermore, assuming correctness for queue operations implies that both an enqueue operation and a surrender operation on a non-empty queue will always be successful.

### Contracts for MrsP-Seize

When designing the contracts for the seize operation and its corresponding functions, a ceiling check similar to the function, i.e., `_CORE_ceiling_mutex_Set_owner` in ICPP is detected, which matches the second formally described property. Inside `_MRSP_Raise_priority`, the priority of the executing thread is checked against the local ceiling of the requested resource. The comparison is performed with the current scheduler node's dynamic priority rather than with the thread's base priority. The above implemented properties match the third and forth formally described properties. As a result, the current implementation of the seize operation of MrsP in RTEMS does not allow an arbitrary sequence of resource requests if they are not properly nested. However, the implementation is valid only under one assumption: *a thread acquiring nested resources always requests them in a non-descending order of priority ceilings.* The assumption is translated to a precondition in the successful behaviors of the affected functions.

A resource that is additionally acquired while being helped by a waiting thread does not necessarily have a priority ceiling for the foreign processor to which the thread has migrated. Instead, the migrated thread always inherits the resource's local ceiling stored for its home processor and does not affect the priority of the helping thread. This feature indicates that the ceiling check correction in Section 5.5.1 could be applied for multiprocessor systems as well. As long as the thread still holds the resource it is helped with, it stays in the migrated-to processor and runs at a legitimate priority. The inheritance of a new ceiling becomes effective as soon as the thread returns back to its home scheduler.

Please note that the operation described in the fourth formally described property, where the requesting tasks spin-wait on their own processor in a FIFO order when the requested resource is not available, is assumed to be correctly implemented. This assumption is based on the spinning macro and queue operations being provided by the system-level functionalities.

### Contracts for MrsP-Surrender

The surrender operation for MrsP has to be handled carefully due to possible waiting threads. The contract is shown in Listing 5.13, where the possible waiting tasks can revoke the surrendered resource's priority ceiling. Threads waiting for the resource spin in the corresponding FIFO queue. Therefore, based on the contents of that queue, we can distinguish the behavior with the assistance of the predicate `MrsPThreadsWaiting` in Listing 5.14. In case there is no waiting thread, the resource owner (which corresponds to the queue owner) is simply reset to `NULL`. On the other hand, if the waiting queue is not empty, the first thread is set to be the succeeding owner. These operations are ensured by the stub contract for the queue's surrender function `_Thread_queue_Surrender_sticky` in Listing 5.15. This function ensures that the ownership is passed to the next thread and the affected tasks' priorities are updated. The new resource owner, represented by the thread, can be abstracted by the ghost variable `g_new_owner`. In both cases, the surrendering thread loses the inherited priority.

```
/*@
requires \valid(mrsp) && \valid(&mrsp->Wait_queue.Queue) &&
\valid(executing) && \valid(g_homenode);
behavior surrender_no_successor:
assumes MrsP_Owner(mrsp) == executing;
assumes ! MrsPThreadsWaiting(mrsp);
ensures MrsP_Owner(mrsp) == NULL;
ensures PriorityRevoked(executing, MrsP_Ceiling(mrsp));
ensures Executing_Priority >= \old(Executing_Priority);
ensures \result == STATUS_SUCCESSFUL;
behavior surrender_successor:
assumes MrsP_Owner(mrsp) == executing;
assumes MrsPThreadsWaiting(mrsp);
ensures PriorityRevoked(executing, MrsP_Ceiling(mrsp));
ensures Executing_Priority >= \old(Executing_Priority);
ensures MrsP_Owner(mrsp) == g_new_owner;
ensures \result == STATUS_SUCCESSFUL;
behavior surrender_fail:
assumes MrsP_Owner(mrsp) != executing;
ensures \result == STATUS_NOT_OWNER;
disjoint behaviors;
complete behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _MRSP_Surrender(
MRSP_Control *mrsp,
Thread_Control *executing,
Thread_queue_Context *queue_context ){/*...*/}
```

**Listing 5.13:** Function contract of the MrsP surrender operation.

The actual transfer of the ownership happens in the counterpart during the waiting thread's seize operation, by the queue function `_Thread_queue_Enqueue_-sticky`. When the function is called successfully, the calling thread is guaranteed to receive the ownership of the queue. Furthermore, the waiting thread is expected to have raised its priority to the resource's local priority ceiling as expressed by the annotations of the seize operation. Therefore, the surrendering thread does not have to take care of priority manipulations for other tasks.

```
/*@ predicate MrsPThreadsWaiting(MRSP_Control *m) =
m->Wait_queue.Queue.heads != NULL;
*/
```

**Listing 5.14:** Helper predicate to determine if a resource wait queue is empty.

```
/*@
requires \valid(queue);
requires queue->owner == NULL;
assigns queue->owner, prioritiesUpdated;
ensures queue->owner == g_new_owner;
ensures prioritiesUpdated;
*/
void _Thread_queue_Surrender_sticky(
Thread_queue_Queue *queue,
Thread_queue_Heads *heads,
Thread_Control *previous_owner,
Thread_queue_Context *queue_context,
const Thread_queue_Operations *operations );
```

**Listing 5.15:** Stub contract for the surrender operation on a queue for MrsP.

### Verification with Frama-C

As explained earlier, the official implementation of the seize operation of MrsP in RTEMS does not allow an arbitrary sequence of resource requests if they are not properly nested. Additionally, we introduce an assumption that a thread acquiring nested resources must request them in a non-descending order of priority ceilings. This is combined with the derived function contracts. After applying the remedy of ICPP and the introduced assumption, all the implemented functions are successfully verified with Frama-C and *wp*, i.e., satisfying the original definition of MrsP. We also ensure that the help mechanism conforms to the suggestions proposed by Garrido et al.[GZB+17]. In case the implementation of the ceiling check and the priority retrieval are modified, the annotations can be adapted accordingly and the verification process can be reattempted.

## 5.5.3 Verification of DGA in RTEMS

In this section, we verify the implementation of DGA in RTEMS. The current implementation only supports the OCS task model. This implies that each task contains only one non-nested critical section.

Table 5.6 lists the protocol functions set for verification, with all function names bearing the common prefix `_DGA`. From a verification perspective, we applied a preprocessing approach to the DGA implementation, as detailed in Section 5.5.1.

The core properties of DGA that are to be verified are as follows:

1. Forced Execution Order: Within a single hyper-period, each task receives a unique set of orders for its jobs. This order designates the sequence of jobs for the dedicated shared resource accesses during that hyper-period.
2. Non-Work-Conserving Mechanism: Each resource retains an execution order. A job can only access the shared resource if it holds the corresponding order. If not, the job is not able to lock the shared resource, even if it is available, i.e., not held by another task.

Table 5.6: Functions for acquiring and releasing a resource under DGA.

| Protocol Function | Purpose |
|---|---|
| `_Set_Thread` | Set the ticket value |
| `_Seize` | Acquire an available or self-locked resource |
| ↪ `_Set_owner` | Set resource owner |
| ↪ `_Wait_for_ownership` | Performed if the resource is used on another processor or ticket does not match |
| `_Surrender` | Release a locked resource and check the head of the wait queue |

As highlighted in Section 5.3.3, the core principle of the DGA implementation is ensuring critical sections of jobs requesting the same shared resource execute in a predefined order. This is achieved using a ticket-based approach to regulate execution order. Consequently, in addition to the seize and surrender functions, the `Set_Thread` function to initialize the ticket system is also verified.

### DGA Set Thread

This section describes the function invoked by the API, which is designed to set the ticket value for the calling task. All associated contracts are detailed in Listing 5.16.

To ensure data integrity and thread safety, the function initiates by entering a critical section, thereby safeguarding the data structure from concurrent access. Following this, it checks if the executing thread's ticket number is 0. This status indicates the thread's first-time access to the `DGA_Control` structure. In such cases, the function assigns a ticket number to the thread and subsequently logs it into the `ticket_order` array of the `DGA_Control` structure at a predetermined position. However, if the executing thread's ticket number is not 0, the function, instead of allocating a new ticket number, logs the existing ticket number in the `ticket_order` array of the `DGA_Control` structure. The execution order is forced by using the ticket mechanism, which matches the first formally described property.

Upon completing these steps, the function exits the critical section. Upon successfully executing its operations, the function returns a `STATUS_SUCCESSFUL` status. The encompassing functional contract in 5.16 details two behaviors while also underlining the dependencies altered during the thread setting function's execution.

### DGA Seize

In the DGA seize operation, the function checks the ownership status of the shared resource. When the semaphore owner is set to `NULL` and the requesting task has a valid ticket, the function triggers `_DGA_Claim_ownership`, thereby ensuring that the executing thread claims ownership of the DGA semaphore. However, if the semaphore is already under the ownership of the requesting task, the function

```
/*@
requires \valid(dga) && \valid(executing) && \valid(queue_context) ;
assigns executing->ticket.ticket, dga->ticket_order[position], executing->ticket.owner ;
ensures \result == STATUS_SUCCESSFUL;
behavior no_ticket_number:
assumes executing->ticket.ticket == 0;
assigns executing->ticket.ticket, dga->ticket_order[position], executing->ticket.owner ;
ensures dga->ticket_order[position] == executing->ticket.ticket;
behavior else:
assumes executing->ticket.ticket != 0;
assigns dga->ticket_order[position];
ensures dga->ticket_order[position] == executing->ticket.ticket;
complete behaviors;
disjoint behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _DGA_Set_thread(
DGA_Control *dga,
Thread_Control *executing,
Thread_queue_Context *queue_context,
int position
);
```

**Listing 5.16:** Function contracts for _DGA_Set_thread.

promptly issues a `STATUS_UNAVAILABLE` status, which indicates that the thread is not permitted to seize the DGA semaphore again.

In scenarios where the current task either lacks a valid ticket or the requested shared resource is already in a locked state, i.e., `owner != NULL`, the function proceeds to assess the `wait` argument. If this argument is set to true and the semaphore owner differs from the requesting task, the function employs `_DGA_Wait_for_ownership` to wait for the semaphore's ownership. Conversely, if the argument is false and the semaphore owner is not the requesting task, the function directly returns a `STATUS_UNAVAILABLE` status, denoting the thread's inability to acquire the DGA semaphore. Such operations match the second formally described property.

All subsequent function calls are rigorously verified within the framework to ensure the integrity of the Seize functionality. A detailed representation of these behaviors, along with their associated dependencies and results, can be found in Listing 5.17. Please note that the ordered ticket queue is implemented using a red-black tree. This functionality is facilitated by the RTEMS API, which is, for the scope of this discussion, assumed to function correctly.

### DGA Surrender

Within the scope of the DGA surrender operation, the function first assesses whether the executing thread holds the current ownership of the DGA. If the current thread is not the owner, the function promptly returns a `STATUS_NOT_OWNER` status. Conversely, once ownership is confirmed, the function augments the DGA semaphore's

```
/*@
requires \valid(dga) && \valid(executing) && \valid(queue_context) ;
behavior noOwner:
assumes DGAMutex_Owner(dga) == NULL && DGA_has_valid_ticket(dga, executing);
assigns dga->Wait_queue.Queue.owner;
ensures \result == STATUS_SUCCESSFUL;
behavior semaphoreExecuting:
assumes DGAMutex_Owner(dga) == executing;
ensures \result == STATUS_UNAVAILABLE;
behavior semaphoreWaitForOwnership:
assumes DGAMutex_Owner(dga) != executing && DGAMutex_Owner(dga) != NULL &&
    wait;
ensures \result == STATUS_SUCCESSFUL;
behavior else:
assumes DGAMutex_Owner(dga) != executing && DGAMutex_Owner(dga) != NULL && !
    wait;
ensures \result == STATUS_UNAVAILABLE;
disjoint behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _DGA_Seize(
DGA_Control *dga,
Thread_Control *executing,
bool wait,
Thread_queue_Context *queue_context
);
```

**Listing 5.17:** Function contracts for _DGA_Seize.

**Table 5.7:** Derived proof goals and required time of the verification of ICPP, MrsP and DGA.

| | **Proof Goals** | |
|---|---|---|
| | Qed | Alt-Ergo |
| **ICPP Functions** | 51 | 44 |
| **MrsP Functions** | 75 | 27 |
| **DGA Functions** | 109 | 7 |

ticket value by 1. Following this, it seeks the subsequent owner, identifying it as the lead thread within the DGA wait queue through the `_Thread_queue_First_locked` function. Given the task presents a valid ticket, the function subsequently extracts this owner from the wait queue using the `_Thread_queue_Extract_critical` function. In the absence of a valid ticket, the semaphore's owner is reset to `NULL`.

As a final step in its sequence of operations, the function returns a `STATUS_SUCCESSFUL` status. The detailed breakdown of these functionalities can be reviewed in Listing 5.18.

```
/*@
requires \valid(dga) && \valid(executing)&& \valid(queue_context);
assigns dga->Wait_queue.Queue.owner,dga->current_position;
behavior notOwner:
assumes DGAMutex_Owner(dga) != executing;
ensures \result == STATUS_NOT_OWNER;
behavior surrenderPossible:
assumes DGAMutex_Owner(dga)!= NULL && DGAMutex_Owner(dga) == executing;
ensures \result == STATUS_SUCCESSFUL;
complete behaviors;
disjoint behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _DGA_Surrender(
DGA_Control *dga,
Thread_Control *executing,
Thread_queue_Context *queue_context
);
```

**Listing 5.18:** Function contracts for _DGA_Surrender.

### 5.5.4 Overhead and Discussion

To verify the protocol-specific functions, we annotate every encountered function, together with appropriate abstractions from the OS's details. The corresponding source code can be reviewed in [Egi22]. Please note that the verification framework does not build or execute the annotated source code, so the annotation is only for the verification purpose.

To measure the verification time, we ran Frama-C without its GUI and verified the protocol functions using the argument `-wp-fct f1,f2,...,fn` to Frama-C. The process ran in a single-threaded mode on an Intel Core i7-8700K CPU with 16 GB of RAM. The computation time for each protocol is less than 10 seconds, which includes preprocessing, transformation and normalization of the protocol implementation, the generation and simplification of proof obligations as well as the delegation of selected proofs to Alt-Ergo [OCa20]. The results for derived proof goals of the verification of ICPP, MrsP, and DGA are listed in Table 5.7.

Although there are corresponding functions for the acquire and release operations in ICPP, MrsP, and DGA protocols with `..._Seize` and `..._Surrender`, the distinction of protocol-specific functions is still challenging. In RTEMS, the implementations of the protocol rules are spread over various sub-functions. Furthermore, the implementation includes several additional checks and actions, which are not formally described by the protocol specifications.

For ICPP, several additional check mechanisms are added:

- check the correctness of setting the resource's priority ceiling,
- check if the task reclaims an already locked resource or claims a locked resource by enqueueing,
- check if the task is allowed to access the resource, and

- the resource owner actively switches its state to blocked, i.e., self-suspending.

For MrsP, the priority ceiling check is included as well, along with an extension which runs idle tasks to substitute helped tasks during their migration phase to improve nested resource access.

These rules above have to be split across the function contracts, where a called function has to be one part of the caller's function contract. The call for one function can generate a call chain of protocol-specific functions. A leaf function's preconditions have to be ensured by the calling functions. In turn, its ensured postconditions can be relevant to the root function along with its contract.

Our case studies show that a protocol is not necessarily implemented exactly as specified. In practice, the implementation has to cover a broad variety of possible configurations, e.g., a task requests a self-locked resource or the resource owner is self-suspending. The approach to handle such difference in this work is to require ICPP's invariant as a precondition. Then the protocol-conforming subset of the actual implementation can be verified efficiently.

With the introduced techniques, further properties could be verified. For example, the basic locking which was not considered as part of the verification, could be included. A possible approach is to introduce a boolean ghost status variable for each locking level, e.g., for the thread-, mutex- and MrsP-queues. Since the analyzed functions are called from an API-function, the state of the locks at the time of the call must be included in the preconditions of the called functions. The contracts of the locking primitives that are currently bypassed could be changed to ensure the correct state of their affected locks in the postcondition. The contracts of the protocol functions could then express the properties of the locks by referring to the ghost variables' states. To complete the specification of memory assignments, some top-level functions, e.g., `_MRSP_Seize` and `_MRSP_Surrender` would have to be complemented with `assigns` clauses in order to make sure that they have no unspecified side-effects. Such annotations become important if these functions are included in a possible verification of the invoking API functions.

The proposed verification framework can also be applied on other OSes, with the assumption that all the low-layer functions are implemented correctly in the targeted OS, e.g., mutexes, queues, and threads. For each targeted OS, the definitions of the used helper predicates and logic functions have to be adapted. Once the necessary OS utilities are abstracted, these basis can be used to verify the implementations of multiple protocols. For each implemented protocol, the properties derived from the formal definition for the seize and the surrender operation should be portable to the targeted OS with reasonable effort. In the end, the function contracts for the detailed implementation of a dedicated resource synchronization protocol can be designed and verified.

## 5.6 Summary

In the domain of real-time systems, numerous resource synchronization protocols have been proposed and studied since the 1990s to tackle the challenges posed by concurrent tasks sharing resources. While a significant portion of research has been dedicated to theoretical worst-case timing analysis, the practical realization details and associated pitfalls have received less attention. Even though various protocols are now supported across RTOSes, ensuring that these implemented protocols (often contributed by multiple individuals) consistently uphold proven properties remains a pressing challenge for the community.

In this chapter, we demonstrate the applicability of the newly proposed DGA for multiprocessor resource synchronization by detailing its implementation in LITMUS$^{\mathrm{RT}}$ and RTEMS. Through comprehensive synthetic experiments, the measured overheads show that our implementations are comparable to FMLP and MrsP, which are officially supported in LITMUS$^{\mathrm{RT}}$ and RTEMS, respectively. When accounting for real system overhead, this provides system designers with a clearer and more decisive understanding of the performance of resource synchronization protocols. For LITMUS$^{\mathrm{RT}}$, the source code for the OCS task model can be reviewed in [Shi18], and for the MCS task model in [Shi19]. For RTEMS, the source code corresponding to the OCS task model is available in [SPS22].

Addressing the imperative of implementation accuracy, we introduce a robust framework to formally verify existing protocol implementations within RTOSes. Our approach specifies the intended behaviors of the implementation in the form of function contracts. We then apply deductive verification to determine whether each component aligns with its formal description, assuming the foundational primitives are implemented correctly. Our verification framework enables modular formal verification of functional components. By defining and isolating the analysis target from its dependencies, the workflow of the proposed framework retains its conceptual independence from the platform.

Practical applications of our verification framework to the ICPP and MrsP protocols in RTEMS underscore its applicability and importance to real-world RTOSes. Moreover, the discovery of mismatches in the RTEMS implementations highlights its effectiveness. Post the introduction of a recommended correction, the implementations of ICPP and MrsP (subject to one additional assumption) were successfully verified. Furthermore, the implementation of the DGA in RTEMS was formally verified by applying the proposed framework, solidifying its correctness.

Chapter 6

# Model-based Optimization on Distributed Embedded Systems

## Contents

## 6.1 Overview

Statistical and machine learning algorithms often possess a high degree of parameterizability, making their performance particularly sensitive to hyper-parameter configurations. For instance, the Multi-layer Perceptron (MLP) [GD98] requires careful selection of various hyper-parameters, such as the number of layers, the number of neurons in each layer, the type of activation functions, and learning strategies. Proper configuration of these parameters is imperative before deploying a machine learning model in a real-world scenario.

While hyper-parameter tuning is vital for achieving optimal predictive performance, it becomes increasingly resource-intensive as the size of the data grows and the search space expands. Model-Based Optimization (MBO), also known as Bayesian optimization [JSW98], offers a solution to this challenge. MBO addresses this expensive optimization by using Gaussian process (GP) regression to approximate the predictive performance based on the hyper-parameters.

However, the conventional approach to hyper-parameter tuning, which involves training and evaluating a dedicated machine learning model on centralized data, faces limitations in distributed settings. Centralized data collection and processing may be infeasible due to concerns about privacy and constraints on computational capacity and storage on each device.

**Challenges in current approaches:** The conventional tuning process, which depends on centralized data, is becoming increasingly infeasible in distributed embedded systems. An emergent alternative suggests that each node tunes its hyper-parameters independently using its local data. However, this can introduce variability in the performance of the machine learning algorithm due to the limited size of individual training datasets which can lead to issues like generalization error, poor convergence, and inconsistent performance.

The primary challenge of hyper-parameter tuning in distributed embedded systems is to effectively utilize these decentralized subsets of data. The goal is to derive a universal hyper-parameter configuration that is effective across all nodes in the system. We seek an innovative method for this purpose. The method should simultaneously: a) increase the average prediction accuracy; b) enhance statistical stability; and c) improve run-time efficiency.

To the best of our knowledge, only a handful of studies in the field have accounted for resource constraints [KRL+17; KSL+19], and optimized the execution of MBO on single multi-core embedded systems with a comprehensive dataset [Kot18]. Notably, none have applied Model-Parallelism in conjunction with MBO for distributed embedded systems, which possess limited computational resources and store different sub-datasets on each node.

**This dissertation:** We address the challenge of hyper-parameter tuning for machine learning algorithms on resource-constrained distributed embedded systems. This challenge is formulated as a distributed black-box optimization problem. That is, each node, utilizing its local data, operates as an individual black box while the entire distributed embedded system acts as a collective black box. The goal is to optimize the system's performance in terms of mean prediction accuracy, statistical stability, and run-time efficiency.

To tackle this, we introduce a novel framework called *MODES*. This framework employs Model-Based Optimization on resource-constrained Distributed Embedded Systems to fine-tune hyper-parameters for machine learning algorithms both locally and efficiently. *MODES* offers two distinct optimization modes:

- Black-box mode (*MODES*-B) in Section 6.3.2: This mode considers the entire ensemble as a singular black box and optimizes the hyper-parameters of each individual model jointly by considering the weights for different nodes
- Individual mode (*MODES*-I) in Section 6.3.3: Here, all models are considered as replicates of a single black box, facilitating parallelized optimization in a distributed context.

Notably, *MODES* is designed for versatility, catering to a broad spectrum of applications with minimal adjustments and the capability for mode switching.

For clarity, we outline the foundational principles of Model-Based Optimization in Section 6.2. Following this, Section 6.3 offers an exhaustive exposition of our distributed model-based optimization strategy, which encapsulates the underlying system model settings and assumptions, designs of both modes tailored to specific optimization objectives. We also provide a comparative analysis of these modes. Lastly, we present experimental evaluations, demonstrating the effectiveness of our approach, in Section 6.4.

## 6.2 Model Based Optimization

Model-Based Optimization (MBO) aims to solve the following optimization problem:

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x} \in \mathcal{X}} f(\boldsymbol{x})$$

for a given function $f(\boldsymbol{x}) \colon \mathcal{X} \to \mathbb{R}$ with $\mathcal{X} \subset \mathbb{R}^p$. We assume that the true expensive black box function can be approximated through a surrogate. This surrogate is a regression method that is comparably inexpensive to be evaluated. In this work we use a Gaussian process regression, which is a typical choice for MBO. To start the optimization, an initial design $\mathcal{D}$ of $k$ points, laid out in a Latin hyper-cube design across $\mathcal{X}$, is evaluated on the expensive function and yields the outcomes $\boldsymbol{y}$. In the following, the sequential model-based optimization iteratively repeats the following steps until a predefined budget is exhausted:

1. Using all past evaluations $\mathcal{D}$ and their outcomes $\boldsymbol{y}$, a Gaussian process is fitted, which then serves as a surrogate to globally estimate $f$.
2. An acquisition function is derived from the current surrogate.
3. The acquisition function is optimized to determine the most promising point $\hat{\boldsymbol{x}}$: $\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x} \in \mathcal{X}} acq(\boldsymbol{x})$.
4. $y = f(\hat{\boldsymbol{x}})$ is evaluated, $\hat{\boldsymbol{x}}$ and $y$ are added to $\mathcal{D}$ and $\boldsymbol{y}$.

The acquisition function must balance two key considerations. First, it must explore, i.e., evaluate points where the surrogate's prediction is uncertain. Second, it must exploit, i.e., evaluate points predicted to be optimal by the surrogate. The final optimal result $\hat{\boldsymbol{x}}^*$ is the input that leads to the maximal observed objective value, e.g., prediction accuracy.

A popular acquisition function is the expected improvement (EI). Using a Gaussian process as a surrogate yields a Gaussian posterior with mean $\hat{\mu}(\boldsymbol{x})$ and standard deviation $\hat{s}(\boldsymbol{x})$ at each point $\boldsymbol{x}$. Accordingly the expected improvement can be derived as follows:

$$\begin{aligned} \mathrm{EI}(\boldsymbol{x}) &= \mathbb{E}(max(\hat{\mu}(\boldsymbol{x}) - y_{\max}), 0) \\ &= (\hat{\mu}(\boldsymbol{x}) - y_{\max})\Phi(\frac{\hat{\mu}(\boldsymbol{x}) - y_{\max}}{\hat{s}(\boldsymbol{x})}) + \hat{s}(\boldsymbol{x})\phi(\frac{\hat{\mu}(\boldsymbol{x}) - y_{\max}}{\hat{s}(\boldsymbol{x})}) \end{aligned} \tag{6.1}$$

where $\Phi$ is the distribution, and $\phi$ is the density function of the standard Gaussian distribution. Here, $y_{\max}$ represents the best observed value in $\boldsymbol{y}$ so far. This classical formulation of MBO only yields one proposal $\hat{\boldsymbol{x}}$ in each iteration. In our approach, obtaining multiple proposals in each iteration is necessary to leverage parallel computing infrastructures. The Batch expected improvement (qEI) [GLC10; RMW14] is introduced as an acquisition function designed for multiple proposals. It transforms the $p$-dimensional optimization problem for finding one promising point into a $p \cdot q$-dimensional optimization problem for finding $q$ promising points. As the qEI lacks of an exact analytical representation for $q > 2$ it is usually solved approximately by Monte Carlo (MC) sampling methods. In BoTorch [BKJ+20] the qEI for $X = (\boldsymbol{x}_1 \dots \boldsymbol{x}_q)'$ is calculated as follows: We sample $\tilde{\boldsymbol{y}} \in \mathbb{R}^q$ from the joint posterior of $X$, which is given by the Gaussian process surrogate. We calculate the individual improvements $I = \max(\tilde{\boldsymbol{y}} - y_{\max}, 0)$. Then, we obtain $\max(I)$ for the current sample. Finally, we repeat those steps multiple (e.g. 1000) times and average over the obtained maximal improvements to obtain an MC approximation of the qEI for a given $X$. To obtain the set of $q$ points that maximizes the qEI, BoTorch uses gradient-based optimization.

The extensions of the qEI to noisy problems, namely the qNEI [BKJ+20], can be derived by replacing the fixed value $y_{\max}$ with $\max(\tilde{\boldsymbol{y}}_{\text{obs}})$ from the sample $(\tilde{\boldsymbol{y}}, \tilde{\boldsymbol{y}}_{\text{obs}})'$ of the joint posterior of $(\boldsymbol{x}_1 \dots \boldsymbol{x}_q \ \boldsymbol{x}_{\text{obs},1} \dots \boldsymbol{x}_{\text{obs},t})'$, with $D = (\boldsymbol{x}_{\text{obs},1} \dots \boldsymbol{x}_{\text{obs},t})'$.

Similarly the NEI can be calculated by introducing MC-sampling into the calculation of the EI. Therefore, we replace $y_{\max}$ with the average of multiple samples of $\max(\tilde{\boldsymbol{y}}_{\text{obs}})$.

For single proposal MBO, we apply the EI and NEI methods in *MODES*-B. In contrast, *MODES*-I employs parallelization with multiple proposals using the qEI and qNEI criteria.

## 6.3 Distributed Model Based Optimization

This section first introduces the model of the distributed embedded system. Afterwards, to achieve improved mean prediction accuracy, enhanced statistical stability, and better run-time efficiency, two categories of proposed *MODES* with different structures are explained in detail.

### 6.3.1 Distributed Embedded Systems

Within a distributed embedded system, also referred to as a *cluster*, multiple embedded devices collaborate to achieve a unified objective. In this dissertation, we focus on a homogeneous cluster[1] in which all nodes exhibit identical attributes. Specific assumptions for this cluster are as follows:

- The cluster comprises $n$ nodes, labeled as $ES_1$, $ES_2$, ... $ES_n$. Each of these nodes represents a distinct embedded device.
- Every node has finite storage capacity, constraining the volume of data it can retain.
- The datasets from different nodes, which can be viewed as subsets of a comprehensive dataset, vary at least partially.
- Node interconnections have bandwidth restrictions. This limitation constrains data transfers to small sizes, particularly hyper-parameter configurations and performance metrics like classification accuracy.

In our proposed architecture, a host-client paradigm is adopted across all nodes. While each node operates a designated machine learning algorithm, solely one node executes the MBO algorithm. This particular node, termed the *host*, concurrently runs both the MBO and its designated machine learning algorithm. In contrast, the other nodes, i.e., *clients*, solely execute their specific machine learning algorithm.

Due to the inherent computational constraints of our embedded systems setup, we deploy only lightweight machine learning algorithms. This results in a comparably diminutive hyper-parameter search space. The number of hyper-parameters intrinsic to the machine learning algorithm is denoted by $p$.

## 6.3.2 Black-box Mode *MODES*-B

In *MODES*-B, we treat the entire distributed system as a singular black box. Both the hyper-parameter settings and the weights of individual nodes are jointly optimized, aiming to enhance performance using ensemble learning. The whole system only generates one prediction at a time. This approach finds utility in various applications. For instance, it can be applied for air quality predictions using all embedded sensors in a given area [BXF+18], and object recognition by using images taken from different angles [GDO+12].

The structure of *MODES*-B is shown in Figure 6.1, and the corresponding workflow is presented in Algorithm 8. Initially, MBO constructs the surrogate, denoted as $\mathcal{S}$. At the beginning of each iteration, MBO only generates one set of hyper-parameters with the highest expected improvement with respect to the current surrogate, comprising a total of $(n \times p + n)$ hyper-parameters, represented as $X = x_1, \dots, x_n, w_1, \dots, w_n$. In each setting, first set $x_1$ contains $p$ elements that represent the hyper-parameters of the dedicated machine learning model for the first node, second set $x_2$ represents the hyper-parameters for the second node and so on.

[1]While our proposed method can be extended to heterogeneous clusters, doing so would necessitate efforts to synchronize the operations across different nodes. For instance, one might assign heavier workloads to nodes with more resources and superior computational capabilities. Such considerations, however, are out of the scope of this dissertation.

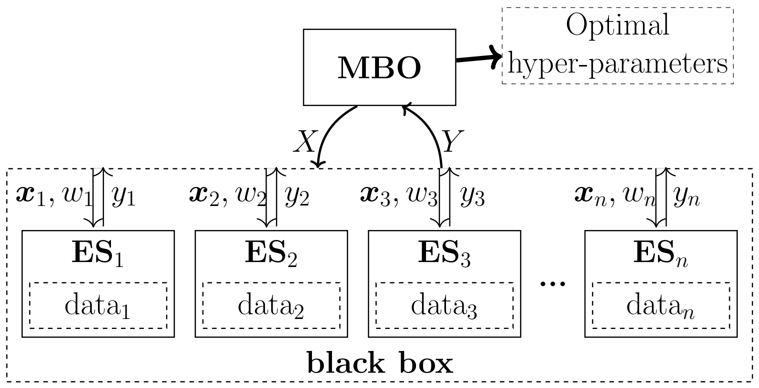


**Figure 6.1:** *MODES*-B: The whole distributed embedded system is treated as a single black box.

Moreover, $n$ weights indicating the importance of each node and its local data are represented through $X$ as well.

The dedicated machine learning model $ML$ is trained on each node by using the given hyper-parameter setting (i.e., $x_j$ where $j$ is the node id) and the local sub-dataset. Each node generates one local performance result (accuracy of classification) of the trained machine learning model by using an evaluation test set. The final result, $Y$, is computed as the weighted average of results from all nodes: $Y = \sum_{j=1}^{n} w_j \times y_j$, where $y_j$ is the local performance result of node $j$, and $\sum_{j=1}^{n} w_j = 1$. In practice, each weight lies in the range [0.1, 1]. Afterwards, a normalization is operated to obtain the real weight for accuracy calculation, i.e., $w_j = \frac{w_j}{\sum_{i=1}^{n} w_i}$. At the end of each iteration, this final result updates the surrogate in the MBO. The process is repeated until the maximum number of iterations is reached or the time budget is exhausted.

In this mode, the search space dimensionality amounts to $n \times p + n$. Therefore, the large number of nodes ($n$) in the dedicated cluster and/or the large number of hyper-parameters ($p$) of the dedicated machine learning model can lead to a high-dimensional search space. The computational demand for MBO to update its surrogate and propose new settings scales with the search space size. However, due to the limited computational capability, embedded systems may not be able to find the optimal hyper-parameter setting within such an expansive search space and a given time frame.

To address this challenge, we mandate that all nodes adopt identical hyper-parameter settings, while maintaining distinct weights, i.e., $\forall i, j \le n, i \ne j : x_i = x_j$ and $\exists i, j \le n, i \ne j : w_i \ne w_j$. As a result, the search space is significantly reduced to $(p + n)$ dimensions. In each MBO iteration, all the nodes receive the same set of hyper-parameters, and train the dedicated machine learning model using their local datasets independently. Afterwards, the evaluation test set is utilized to evaluate the performance of these trained machine learning models on different nodes, and the weighted mean is returned to the *host* node, which is used to update MBO

**Algorithm 8** Workflow of *MODES*-B

```
Input: number of nodes $n$, dedicated machine learning model $ML$, number of
    hyper-parameters $p$, time budget $T$, and maximum tuning iterations $Itr$;
Output: Optimal hyper-parameter setting: $HP\text{-}B$;
 1: Initialize: MBO surrogate $\mathcal{S}$, iteration $i \leftarrow 0$, time $t \leftarrow 0$;
 2: while $i \leq Itr$ and $t \leq$ T do
 3:     $X \leftarrow$ MBO $(\mathcal{S}, n, p)$;
 4:     for $j$ from 1 to $n$ do
 5:         $y_j = ML(\boldsymbol{x}_j, ES_j, data_j)$
 6:     end for
 7:     $Y \leftarrow \sum_{j=1}^{n} w_j \times y_j$;
 8:     Update surrogate according to $(X, Y)$;
 9:     $i$ ++;
10:     Accumulate consumed time $t$;
11: end while
12: MBO generates the optimized $HP\text{-}B$ according to current surrogate;
13: return $HP\text{-}B$;
```

surrogate. In the end, we obtain a set of optimized hyper-parameters complemented by the node weights.

Please note, our proposed *MODES*-B, which allows distinct hyper-parameters for each node (i.e., $\exists i, j \leq n, i \neq j : x_i \neq x_j$), is also compatible with high-capacity distributed systems. However, assessing performance on these systems is out of scope for this work.

### 6.3.3 Individual Mode *MODES*-I

In the *MODES*-I mode, each node is treated as an instance of the same black box. The whole cluster acts like a multi-processor system and each node serves as a single processor. This structure enables the parallel application of MBO. In this scenario, the performance of multiple proposed hyper-parameter settings can be evaluated at the same time, i.e., each node trains a dedicated machine learning model using one set of the proposed hyper-parameter settings and its local dataset. In this mode, improving the timing efficiency is the most important objective, e.g., in some real world timing-sensitive applications like autonomous driving systems [LAB+11].

Figure 6.2 illustrates the structure of *MODES*-I, while Algorithm 9 presents the corresponding workflow. In each iteration, MBO proposes $n$ different hyper-parameter settings drawing from the insights of the current surrogate and utilizing either the qEI or qNEI acquisition function as explained in Section 6.2. Each node uses one hyper-parameter setting to independently train the dedicated machine learning model using their local data. Afterwards, these trained models are evaluated by using a local evaluation test set. The individual performance measures, i.e., the

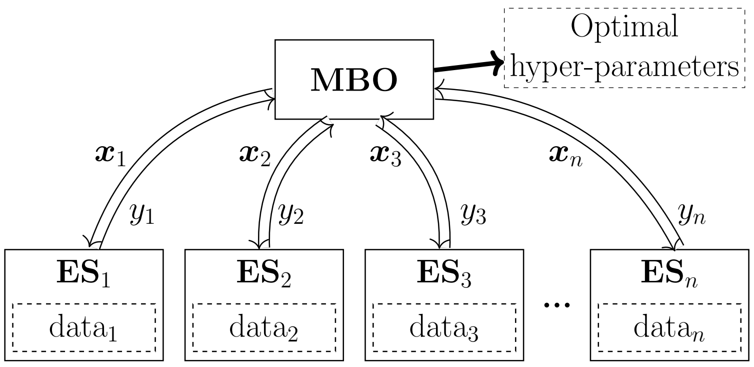


**Figure 6.2:** *MODES*-I: Each embedded system acts as an individual black box.

**Algorithm 9** Workflow of *MODES*-I

```
Input: number of nodes $n$, dedicated machine learning model $ML$, number of
    hyper-parameters $p$, time budget $T$, and maximum tuning iterations $Itr$;
Output: Optimal hyper-parameter setting: $HP\text{-}I$;
 1: Initialize: MBO surrogate $\mathcal{S}$, iteration $i \leftarrow 0$, time $t \leftarrow 0$;
 2: while $i \leq Itr$ and $t \leq$ T do
 3:     $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n\} \leftarrow$ MBO $(\mathcal{S}, n, p)$;
 4:     for $j$ from 1 to $n$ do
 5:         $y_j \leftarrow$ ML($\{\boldsymbol{x}_j, ES_j, data_j\}$);
 6:     end for
 7:     Update surrogate according to $\{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\}$;
 8:     $i \leftarrow i +$ n;
 9:     Accumulate consumed time $t$;
10: end while
11: MBO generates the optimized $HP\text{-}I$ according to current surrogate;
12: return $HP\text{-}I$;
```

accuracy of classification, are sent back to the *host* node. In our setting, synchronized updating of surrogate is applied, where the surrogate is updated by MBO only after all nodes have completed their evaluations. Therefore, the execution duration of each iteration corresponds to the maximum time taken by any of the nodes. The iterations are repeated until the time budget is exhausted or the maximum number of iterations is reached. The optimization result is one hyper-parameters setting that can be utilized for all the nodes. The entire system can formulate predictions by averaging outcomes, assigning equal weights across nodes. As an alternative, an individual node can handle predictions autonomously with compromised robustness.

*MODES*-I significantly improves the run-time efficiency of the hyper-parameter tuning process, by fully harnessing the computational resources of every node within the distributed system, i.e., it evaluates $n$ proposed settings in parallel by considering all the information from the local data across various nodes. Although the performance of the tuned hyper-parameters may not be improved significantly, due to the fact that different data in different nodes creates noisy results, its applicability remains robust, especially for time-sensitive operations on distributed embedded systems. Consider the case of real-time traffic flow prediction, which demands instantaneous feedback from embedded systems, such as mobile devices, prioritizing tuning speed over incremental accuracy enhancements. Another illustrative instance is applications focused on human activity recognition via mobile devices, e.g., mobile phone or smart watch, which needs fast response (recognition time) according to the sensor's signal and the computation power is restricted.

### 6.3.4 Comparison between *MODES*-B and *MODES*-I

The aforementioned *MODES*-B and *MODES*-I focus on different requirements with different assumptions. *MODES*-B seeks to enhance overall system performance with respect to prediction accuracy and statistical stability by acknowledging node heterogeneity. While *MODES*-I aims at parallelizing the tuning process to improve efficiency, working on the assumption of high similarity among the hardware setting of nodes and their local data subsets.

For *MODES*-B, the entire distributed embedded system operates as a collective ensemble. Each hyper-parameter setting involves not only the hyper-parameters for the dedicated machine learning model, but also the weights for different models. In each iteration of optimization process, only one single proposal is trained and evaluated in the entire system. In the end, the obtained optimized hyper-parameter setting is applied for the whole ensemble, and only one classification result is generated by the system. Theoretically, since the tuned weights represent the importance of different nodes and corresponding sub-datasets, *MODES*-B can outperform other hyper-parameter tuning algorithms if sub-datasets held by different nodes are imbalanced or some sub-datasets have great noise.

In *MODES*-I, multiple nodes in a distributed embedded system are treated as multiple clones of a single node. In addition, the local sub-datasets are considered as subsets of a consistent dataset. This treatment relies on an assumption that the optimal hyper-parameters of the dedicated machine learning model for different nodes are high similar. Therefore, multiple proposals are trained and evaluated on all the available nodes at the same time, in order to accelerate the optimization of the corresponding surrogate. Ideally, the tuning process can be sped up by $n$ times, where $n$ is the number of nodes in the dedicated distributed embedded system. However, due to the variation of the execution times for different hyper-parameters on different nodes, some nodes, finishing earlier, must wait for the node taking the longest time, ensuring synchronized surrogate updates. Hence the improvement of efficiency is less than $n$ times.

Although asynchronous parallel strategies [JRG+12] as well as scheduling methods [RKB+16; KSL+19] are developed for heterogeneous run-time of different

**Table 6.1:** The comparative analysis of *MODES*-B, *MODES*-I, Single, and Central approaches (notations: + improved, o no change, - decreased).

| | ***MODES*-B** | ***MODES*-I** | **Single** | **Central** |
|---|---|---|---|---|
| **Privacy** | + | + | + | - |
| **Data parallelism** | o | + | o | o |
| **Model parallelism** | + | o | o | o |
| **Accuracy** | + | o | o | + |
| **Efficiency** | o | + | o | - |
| **Statistical Stability** | + | o | - | + |

proposals, the comparison of different surrogate updating strategies is considered out of the scope. When there are many nodes, the resulting surrogate may not be able to generate a sufficient number of valuable proposals for evaluating the machine learning algorithms in parallel in the next iteration. That is, some of the proposed hyper-parameter settings to be evaluated have to be generated randomly without any contributions to the corresponding surrogate. Moreover, since each node can make the prediction independently, *MODES*-I offers greater scalability than *MODES*-B. Thus, adding or removing nodes has no detrimental effect on the distributed system's functionality.

Table 6.1 compares all the aforementioned schemes, where Single is that each node tunes on its local data and Central tunes on centralized data from all nodes. The performance related features, i.e., accuracy, efficiency, and statistical stability will be demonstrated in the following section. Please note that *Central* is not evaluated as it is considered out of scope.

## 6.4 Experimental Evaluation

To evaluate the performance of *MODES*, we deploy it on a distributed embedded system comprising four nodes. We utilize the emulation platform delineated in Section 2.4.3, wherein a high-performance AMD server simulates the computational workload of embedded devices, and a separate GPU server manages the MBO operations. Our implementation leverages BoTorch [BKJ+20], a Bayesian optimization library built on PyTorch.

### 6.4.1 Experimental Setup

We evaluate the *MODES* framework using four popular real-world datasets, each with up to $60,000$ instances:

1. The MNIST [LCB98] dataset: it contains $60,000$ handwritten digits (from 0 to 9) images with $28 \times 28$ grey-scale resolution. The MNIST dataset is widely

**Table 6.2:** The 5 hyper-parameters that are tuned for MLP.

| Parameter | Type | Range |
|---|---|---|
| Number of layers | $\mathbb{N}$ | [1, 15] |
| Units per layer | $\mathbb{N}$ | [10, 150] |
| Activation | Dict. | {identity, logistic, tanh, relu} |
| L2 penalty | $\mathbb{R}$ | $[-5, -2]$ (log-10 scale) |
| Learning rate for Adam | $\mathbb{R}$ | $[-4, -1]$ (log-10 scale) |

used for evaluating the performance of machine learning algorithms. Here, we fit our learning task as an image classification problem on the MNIST dataset.

2. The Fashion-MNIST [XRV17] dataset: it consists of Zalando's article images, where the statistics are exactly the same as the original MNIST dataset, i.e., with the same number of instances, the same image size, and the same distribution of different classes. The Fashion-MNIST is more representative for modern computer vision tasks. It usually serves as a replacement for the original MNIST dataset when benchmarking machine learning algorithms, since the original MNIST classification task is easy (e.g., MLP can easily achieve the accuracy of 95%) and overused in the machine learning domain.
3. The Covertype [BD99] dataset: it is a non-vision dataset as well, coming from the US Forest Service inventory information. This dataset is originally used to predict forest cover type from cartographic variables, and it is sensitive for the model settings (parameter tuning) of some popular machine learning algorithms (e.g., MLP, SVM and RF). The original dataset contains $581,012$ instances and 7 classes. However, the number of instances for different classes are extremely unbalanced, i.e., 100 times difference. Hence, we downsized the dataset according to the size of the smallest class, i.e., each class now contains $2,747$ instances, and in total $19,229$ instances.
4. The HAR [AGO+13] dataset: it consists of $10,299$ instances, which are built from the recordings of 30 subjects performing activities of daily living while carrying a waist-mounted smartphone with embedded inertial sensors. Therefore, the HAR dataset naturally fits the distributed embedded systems scenario and it satisfies the assumptions of *MODES* well. As a sensing dataset, six human activities are included, i.e., walking, climbing the stairs, walking down the stairs, sitting, standing, and laying.

Based on the selected datasets and the computational power of the platform, two machine learning algorithms that represent the state-of-the-art are selected as the optimization targets: a) Multi-Layer Perceptron (MLP) [GD98] and b) Random Forest (RF) [LW+02]. The performance of these two benchmark machine learning algorithms have been well-reported on the aforementioned datasets, where they can be used as the references for the performance of our *MODES*. Moreover, the

**Table 6.3:** The 7 hyper-parameters that are tuned for Random Forest.

| Parameter | Type | Range |
|---|---|---|
| number of trees | $\mathbb{N}$ | [5, 150] |
| maximal # of features at every split | Dict. | {auto, sqrt, log2} |
| maximal # of levels in trees | $\mathbb{N}$ | [2, 40], or Auto mode |
| minimal # of samples to split a node | $\mathbb{N}$ | [2, 20] |
| minimal # of samples at leaf node | $\mathbb{N}$ | [1, 20] |
| function to measure the quality of a split | Dict. | {gini, entropy} |
| usage of bootstrap samples | Boolean | {True, False} |

performances of MLP and RF are both sensitive to the hyper-parameters, which makes MBO tuning necessary.

To efficiently evaluate the performance of fine-tuned machine learning algorithms, for the most accuracy-sensitive hyper-parameters among all adjustable hyper-parameters in MLP and RF, we select values based on experience. There are 5 hyper-parameters for MLP and 7 hyper-parameters for RF need to be tuned, details can be found in Table 6.2 and Table 6.3, respectively.

To simulate possible patterns of distributed data storage, datasets are preprocessed. First, each dataset is randomly split into a training set, an evaluation test set, and an unseen final test set at a ratio of $10:1:1$. The evaluation test set is used only for hyper-parameter tuning, i.e., to verify the performance of the proposed hyper-parameter settings. The results are then used to update the MBO surrogate. The unseen final test set is used to evaluate the final performance of hyper-parameters, which were optimized using different methods, in their respective data storage scenarios. Please note that although different evaluation and test sets can be applied to various nodes in real applications, we use the same evaluation and test sets for all nodes in our evaluation to eliminate any potential disturbance from the evaluation and test datasets. Finally, to simulate data storage scenarios in real distributed embedded systems, a sub-dataset for each node is generated from the overall training set using the following strategies:

- **Uniform Split (D1)**: Divide the training set equally into four parts.
- **Duplicated Split (D2)**: Each of the four training sets from D1 is augmented with 30% of data, randomly selected from the other three parts. As a result, each sub-dataset overlaps with the others.
- **Unbalanced Split (D3)**: Divide the training set unequally with shares of $20\%, 20\%, 30\%$, and $30\%$.

### 6.4.2 Selection of Baselines

In order to compare the performance of our proposed methods, 9 algorithms are evaluated. These algorithms are named based on the following rules: a) B/I/S in the first part: *MODES*-B, *MODES*-I, or Single is applied. b) EI/NEI in the second part: expected improvement or noisy expected improvement is applied to generate proposals for next iteration. For *MODES*-I, qEI/qNEI is correspondingly applied when $q$ proposals are generated for parallel execution.

Each MBO tuning procedure has a maximum budget of 100 iterations and 12 hours of runtime. For *MODES*-I, only 25 iterations and 3 hours run-time are assigned, since it can evaluate four different hyper-parameter settings at the same time in each iteration. In total 100 proposals are evaluated at the end. The optimized hyper-parameters are applied to train the dedicated machine learning algorithms. To be fair, the training datasets are the same during hyper-parameter tuning. Finally, the identical testing data, which is unseen by all methods, is adopted. MBO inherently involves randomized decisions, including the selection of initial points and proposals based on surrogates. Thus, analyzing variance is essential to ensure the accuracy of our evaluation results. We repeated each experimental setting 10 times, to show the statistical stability of proposed methods.

### 6.4.3 Experimental Results

We evaluated all combinations and subsequently report the accuracy of the classification results for two machine learning algorithms and three data splitting strategies separately for the different datasets. Since MLP and RF architecture are modularized and standardized (i.e., Scikit-learn [PVG+11]), the randomness from the algorithm itself in reloading (training with the same hyper-parameters and the training set) can be ignored by averaging. This implies that even slight improvements in accuracy are only due to better hyper-parameter settings. The results are shown in Figures 6.3, 6.4, 6.5, and 6.6.

These results show that the B-EI outperforms all the other methods in most of the evaluated cases with respect to the mean prediction accuracy and/or statistical stability. Generally, EI-based methods outperform NEI-based methods in terms of mean prediction accuracy and statistical stability. This indicates that the MLP and RF algorithms are inherently noiseless. Although *MODES*-I (I-q(N)EI) shows less competitiveness in classification accuracy. It significantly improves the run-time efficiency, the detailed improvement will be presented in Section 6.4.4. Additionally, when more overlapping data is incorporated into the training dataset (i.e., transitioning from D1 to D2), the performance of I-qEI notably improves. This enhancement is observed in most evaluated cases and can be attributed to the augmented data size and heightened similarity across various datasets.

For both MNIST and Fashion-MNIST datasets (Figure 6.3 and 6.4), B-EI shows its advantages if the data size is unbalanced in different nodes, i.e., D3 datasets. However, B-EI performs worse than I-qEI and S-EI for D2 datasets, because: a) high similarity of datasets in different nodes reduces the influence from tuning the weights of nodes, i.e., simple average in I-qEI already performs well. b) the increased size of

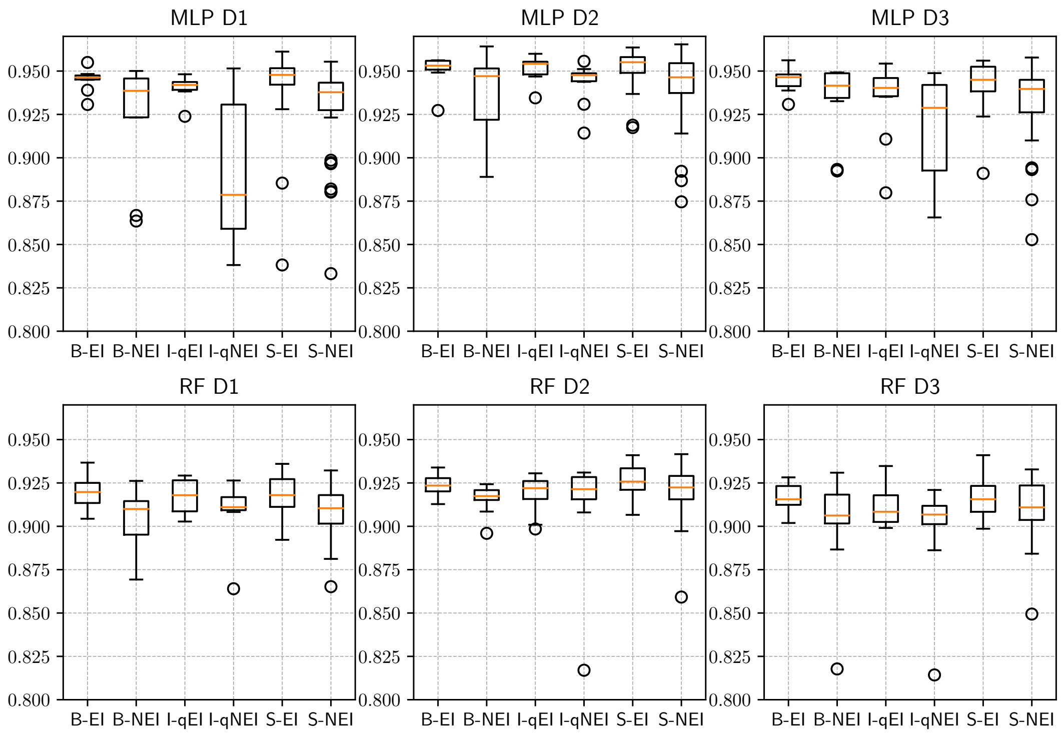


**Figure 6.3:** The accuracy of two machine learning algorithms using different hyper-parameter tuning methods on **MNIST** dataset.

training data allows each single node to train a machine learning model individually with good prediction accuracy.

For Covertype dataset, RF outperforms MLP in all the three data splitting strategies. Hence, only the results for RF are analyzed. In both D1 and D2, B-EI performs slightly worse than S-EI, since each node can train a machine learning model with good prediction accuracy based on the relatively easy dataset. However, for the D3 datasets, where the size of datasets in different nodes is unbalanced, B-EI outperforms all other methods by considering the weights of different nodes.

Since the HAR dataset has fewer dimensions than MNIST (562:784), considering the much smaller sample size (1:6), HAR is more difficult to train especially by MLP. Compared to RF, which is an ensemble of decision trees, MLPs are attributed to have a higher sensitivity to inputs, which tend to result in a higher risk of deviations in the case of relatively high dimensional and low-sample sized inputs. Specifically, when data size in each node is relatively small (D1), B-EI can better reconstruct the true distribution of HAR dataset through the weighted optimization scheme so as to achieve a higher accuracy on test dataset, i.e., increase the weights of nodes (learning with bias) that the distribution of data is closer to the true one. However, when the size of each dataset increases (D2), the risk of over-fitting decreases [HTF09], i.e., the noise compensated in MBO dominates the optimization to prevent over-fitting. Consequently, NEI-based methods outperform EI-based methods on D2 datasets.

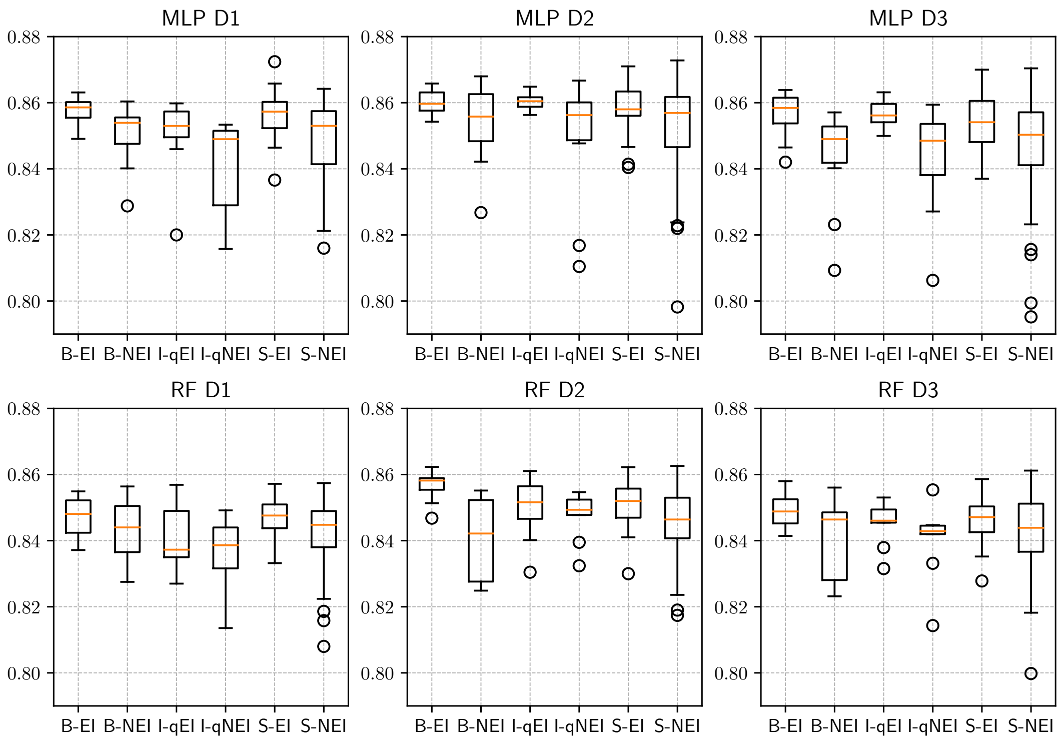


**Figure 6.4:** The accuracy of two machine learning algorithms using different hyper-parameter tuning methods on **Fashion-MNIST** dataset.

The inherent data imbalance in D3 presents a minor trade-off between weight and noise. This leads to comparable performance between NEI- and EI-based methods. The similar phenomenon is also discussed in [CH09]. In most of the evaluated cases, one of our proposed *MODES* still outperforms the Singles with respect to mean prediction accuracy, and show better statistical stability. In contrast, RF shows more robust behavior for HAR dataset than MLP does, where the results are promising and similar to what we have observed on the previous dataset. In summary, for a great variety of datasets and/or applications without data aggregation, the method *MODES*, with two different modes, outperforms the traditional approach S-EI in terms of either accuracy (*MODES*-B) or run-time efficiency (*MODES*-I) without much accuracy degradation.

### 6.4.4 Scalability and Applicability

In order to investigate the scalability of the *MODES*, we evaluated it on the Infinite-MNIST [LCB07] date set with 16 (emulated) nodes. The Infinite-MNIST (also known as MNIST8M) dataset produces an infinite supply of digit images derived from the well-known MNIST dataset using pseudo-random deformations and translations.

To mitigate the effect of inadequate training samples in each node, e.g., a machine learning model may not be well trained if only small size of training data

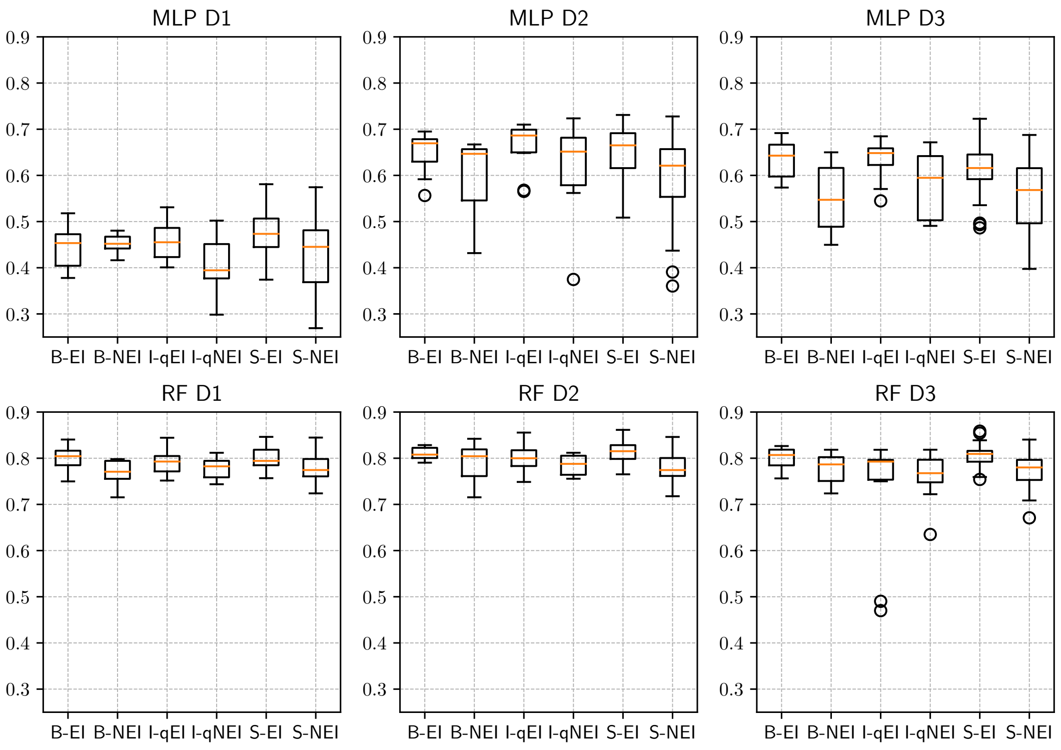


**Figure 6.5:** The accuracy of two machine learning algorithms using different hyper-parameter tuning methods on **Covertype** dataset.

is available, following the size of MNIST dataset used in Section 6.4.1 (i.e., 60,000 training samples for 4 nodes), we enlarge the size of dataset linearly with the same termination condition. In our experiments, we individually chose a total of 240, 000 training samples across both datasets. We applied strategies similar to those used for generating sub-datasets in Section 6.4.1: a) For **D1**, we equally divided the training set into 16 sets; b) For **D2**, we extended each sub-dataset from D1 by adding 5,000 samples randomly selected from the remaining samples; c) For **D3**, we divided the training samples unequally, i.e., 8 sets with 5% share and 8 sets with 7.5% share.

The results of the Infinite-MNIST dataset are shown in Figure 6.7. In general, the *MODES*-B outperforms other methods in all the evaluated cases for MLP and most cases for RF. The performance of *MODES*-I for MLP shows large variance, since the key assumption of *MODES*-I, i.e., data chunks in different nodes have high similarity that the optimal hyper-parameters are similar for these (sub-)datasets in different nodes, does not hold any more for 16 nodes. Optimizing models for different nodes introduces significant noise to the *centralized* MBO surrogate model of *MODES*-I, which can make the tuned hyper-parameter infeasible for the final test set. This variability in optimization could also explain the outlier observed for the S-EI method on the D3 dataset. When the similarity of data in different nodes increases, i.e., in D2 datasets, the variance of I-EI reduces significantly. Therefore,

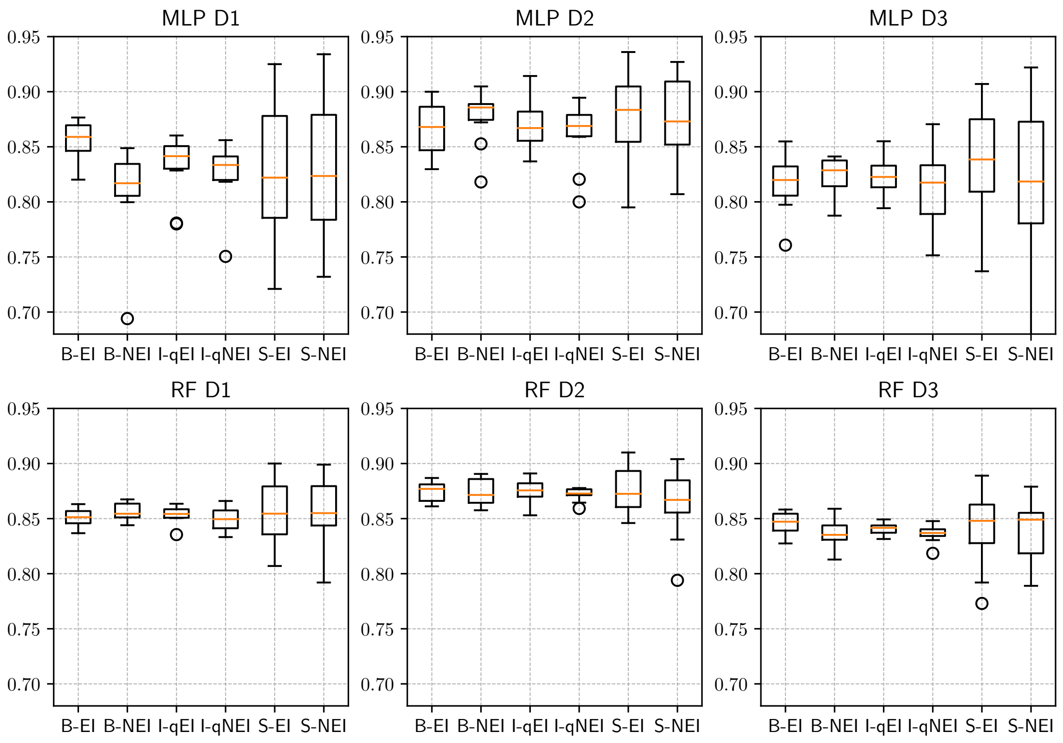


**Figure 6.6:** The accuracy of two machine learning algorithms using different hyper-parameter tuning methods on **HAR** dataset.

while *MODES*-B performs well as the number of nodes increases, *MODES*-I requires the applied data subsets on each node to have certain similarities.

To evaluate the applicability of *MODES* in distributed embedded systems, we utilize a physical platform comprising four embedded devices. A detailed description can be found in Section 2.4.3. Each of these devices runs specific machine learning algorithms (either MLP or RF) for a designated task. The nodes are interconnected, facilitating data transmission between them. Real-world testing yielded accuracy metrics consistent with our earlier findings. Remarkably, *MODES*-I, when executed on an actual cluster, demonstrated speed improvements of 2.2 to 3.7 times over the alternative methods.

## 6.5 Summary

In order to optimize the deployment of machine learning model on a distributed embedded system with constrained resources, we proposed *MODES*, a novel framework for model-based optimization on distributed embedded systems. Instead of aggregating all the data at a centralized server, *MODES* leverages the local data on each node to obtain the optimized hyper-parameter setting of dedicated machine learning algorithms without any raw data sharing. Specifically, two modes are

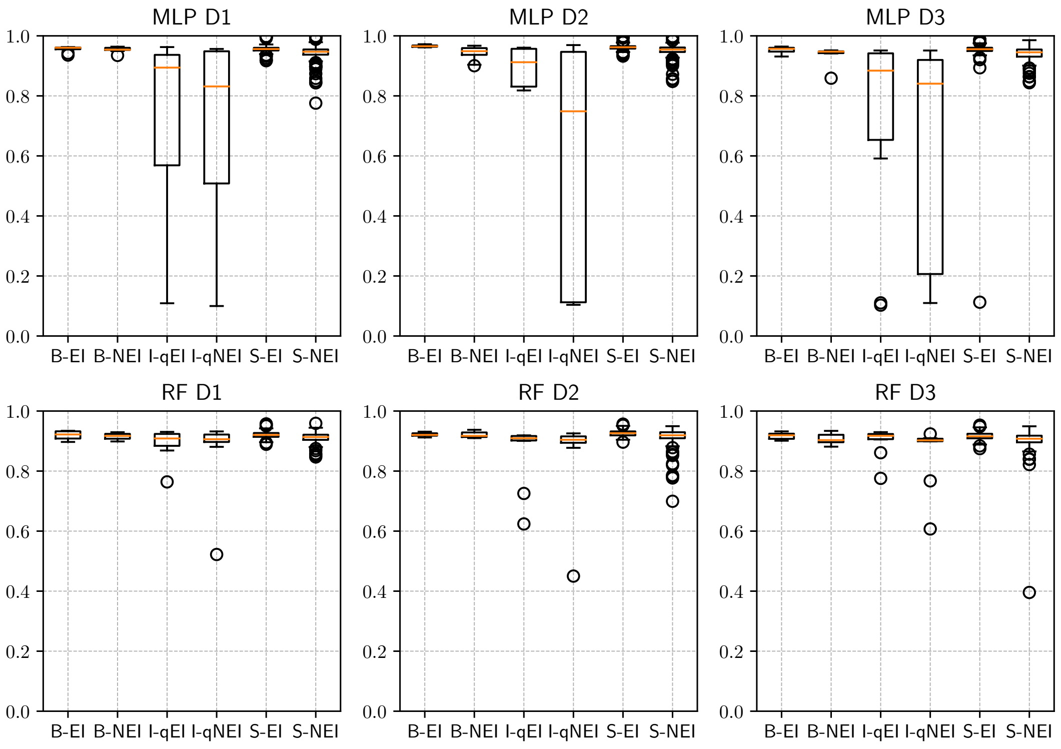


**Figure 6.7:** The accuracy of two machine learning algorithms using different hyper-parameter tuning methods on **Infinite-MNIST** dataset.

considered: *MODES*-B treats the whole system as a single black box and tunes the hyper-parameters jointly; *MODES*-I treats each node as a copy of the same black box and optimizes the hyper-parameters in parallel.

We conduct extensive evaluations on real-world datasets to compare both modes of *MODES* with a baseline method, in which each single node tunes its own hyper-parameter setting by applying MBO using its local data independently. The results show that: a) *MODES*-B outperforms all the other methods in most of the evaluated cases. b) *MODES*-I highly improves the run-time efficiency, where the improvement depends upon the number of nodes in the distributed system, at a cost of slightly degraded performance in some cases. The implementation of *MODES* and corresponding experiments are released in [SBR21].

Chapter 7

# Reinforcement Learning for Average Task Execution Time Minimization with $(m, k)$ Robustness Guarantee

**Contents**

## 7.1 Overview

Safety-critical systems frequently encounter *transient faults*, which may lead to *soft errors* with catastrophic consequences. Therefore, error-handling must be addressed by design. While comprehensive fault protection is resource-intensive, many safety-critical applications can afford a marginal error rate. In practice, some safety-critical applications can tolerate a limited number of errors at the cost of temporarily downgrading the quality of service (QoS), without catastrophic consequences if some constraints of error tolerance are guaranteed. For instance, certain robotic applications can effectively accomplish their missions even in the presence of a limited error count [CBC+16; YCC18]. Such applications often adopt the $(m, k)$ robustness constraint, ensuring at least $m$ error-free jobs out of any $k$ consecutive ones. Static patterns, such as the deeply red pattern (R-pattern)[KS95] and the evenly distributed pattern (E-pattern)[QH00], are used to dictate job execution modes, either an *error-free* mode, which might elongate the worst-case execution time, or an *error-prone* mode, which offers the benefit of reduced execution time.

Besides ensuring execution accuracy, another primary objective is to minimize the system's overall energy consumption [CBC+16; NQ06; NZ20]. Processors exhibit distinct power consumption levels in their active and idle states. The system's average energy consumption is calculated based on the average time spent in each state, factoring in the state's respective power consumption. Specifically, reducing the time in the active or busy state leads directly to a decrease in power consumption. Therefore, energy conservation is fundamentally associated with limiting the system's average utilization.

**Challenges in current approaches:** While static patterns have demonstrated their efficacy in energy-aware designs, they often result in resource over-provisioning due to the relatively low probability of errors. To enhance the adaptability of these static-pattern-based methods, Chen et al. [CBC+16] suggested a strategy that continuously monitors resilience during runtime to tweak the patterns as required. More precisely, their pattern-based scheduler defers resource-intensive reliable executions until the last feasible moment. This is achieved by tracking the number of upcoming jobs that can still be faulty without violating the constraints.

Although this method incorporates a degree of adaptability, it may still inadvertently lead to a conservative resource allocation, especially when the actual soft error probability isn't factored in. Furthermore, its primary emphasis is on schedulability analysis, ensuring the worst-case response time, rather than focusing on energy optimization and minimizing average utilization. Moreover, certain situations can introduce a dynamic element, causing the error probability to fluctuate during runtime, leading to increased system complexity.

To the best of our knowledge, no current research both addresses energy consumption optimization under the $(m, k)$ constraints and offers high adaptability for scenarios with low or even variable error probabilities.

**This dissertation:** We consider a periodic and constrained-deadline task set, denoted as $\mathbf{T}$. For each task $\tau_i$, the $\ell$-th job, denoted as $J_i^\ell$, has a distinct arrival time $a_i^\ell$ and finishing time $f_i^\ell$. Every job operates in one of four modes: *unreliable*, *reliable*, *detected*, or a composite mode, i.e., *detected* + *reliable*. In the composite mode, a job starts in the *detected* mode and switches to the *reliable* mode if a soft error is identified during the same interval $[a_i^\ell, f_i^\ell)$. The mode in which a job $J_i^\ell$ is executed determines the probability of soft errors within the interval $[a_i^\ell, f_i^\ell)$. Specifically, while jobs in *unreliable* and *detected* modes can be faulty, those in *reliable* or *detected* + *reliable* modes are assumed to be correct.

Our primary objective is to design a mode selection strategy for the upcoming job. This strategy relies on past errors, the current job's mode (informed by observed soft errors in the *detected* and *reliable* modes), and an expected error probability $p_e$. This mode selection approach must invariably ensure that the task in question complies with the stipulated $(m, k)$-constraints. Simultaneously, it seeks to curtail the expected execution time of the task.

To this end, we introduce an adaptive state-based algorithm. This algorithm leverages explicit information and/or predictions about soft error probabilities to guarantee $(m, k)$-compliance for each task, even under real-time constraints. It effectively curtails the anticipated task execution time. We provide an optimal solution for a known error probability $p_e$ and present an RL-based methodology to adapt to fluctuating or unspecified error probabilities $p_e$.

In Section 7.2, we present the formulation of all $(m, k)$-compliant states of a task using a minimal automaton, allowing transitions only between compliant states. Based on the soft error probability, a Markov chain model is constructed based on a stochastic transition system. Section 7.3 then focuses on the optimization of the expected execution time. Assuming a stationary and known soft error probability, we introduce an optimization algorithm based on the Markov chain model. This algorithm aims to compute the stochastic parameters pivotal to the job selection strategy. Finally, in Section 7.4, we apply a reinforcement learning (RL) strategy for job mode selection, designed specifically for scenarios where soft error probabilities remain unknown.

## 7.2 Minimal Compliant Automata Construction

In this section, we first define the problem of minimizing average utilization while adhering to the $(m, k)$ constraints. Subsequently, we introduce an algorithm designed to construct $(m, k)$-compliant automata, optimizing for a minimal number of states.

### 7.2.1 Primitives

**Definition 20** (Correctness Indication)**.** We denote the correctness of a job at the end of its execution using 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.

The correctness of the $\ell$-th job $J_i^\ell$ of task $\tau_i$ is represented by the *character* $c_\ell \in \Sigma$. We represent the correctness of a job sequence from $J_i^1$ to $J_i^n$, using a (possibly infinite) sequence of these characters, i.e., a word $w = c_1 \circ c_2 \circ \cdots \circ c_n$, where $n \in \mathbb{N}^*$. We denote the sub-word of $w$ that starts at index $a$ and ends at index $b$ as

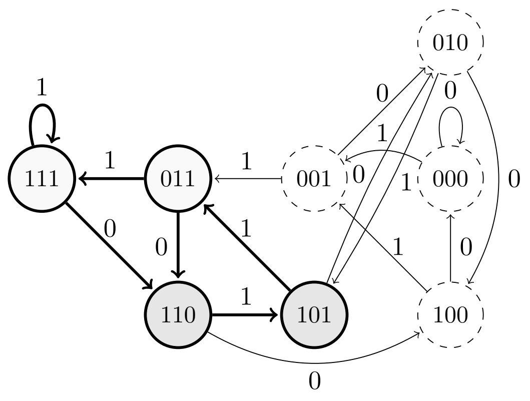


**Figure 7.1:** An exemplary $k$-error-automata $\mathcal{A}_k$ and the $(2,3)$-compliant automata $\mathcal{A}_k^*$ is highlighted in bold, where the darker states are *critical states* and the lighter states are *nominal states*.

$w(a,b) = c_a \circ \cdots \circ c_b$ for $a < b$. The $w(a,:)$ denotes the sub-word starting at index $a$ and continuing to the end. Conversely, $w(:,b)$ represents the sub-word from the beginning up to the index $b$.

To eventually guarantee $(m,k)$-compliance of a task, i.e., of an infinite sequence of jobs, every 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 $2^k$ different outcomes $Q := \{00\dots0,\dots,11\dots1\}$ for which we define a $k$-error-automata.

**Definition 21** ($k$-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 $q_s := 11\dots1 \in Q$ denotes the unique starting state, $\Sigma := \{0,1\}$ denotes the input alphabet, and $\delta$ defines the transition system $\delta : (Q,\Sigma) \mapsto Q$ such that for any state $q \in Q := \{00\dots0,\dots,11\dots1\}$

$$\delta(q,0) = q(2,:) \circ 0 \in Q \tag{7.1}$$

$$\delta(q,1) = q(2,:) \circ 1 \in Q \tag{7.2}$$

An exemplary 3-error-automata $\mathcal{A}_3$ is illustrated in Figure 7.1. While a $k$-error-automata models all error sequences in $k$ consecutive jobs, not each sequence is $(m,k)$ compliant.

**Definition 22** ($(m,k)$-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 $Q^* \subseteq Q$.

In order to verify if a task satisfies its $(m,k)$ constraint after the finishing of the $\ell$-th job given the indication $c_\ell$ it must be tested if every sub-word of length $k$ in $w = c_1 \circ \cdots \circ c_\ell$ for $\ell \geq k$ contains at least $m$ correct executions.

**Definition 23** (Job Sequence Induced State)**.** Given a word $w = c_1 \circ \cdots \circ c_\ell$ for $\ell \geq k$, that 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 2.1.3, in the *unreliable mode*, the correctness of $(j+k)$-th job, i.e., $c_{j+k} = 0$ with probability 1. In *detected mode*, if an 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 released only after detecting an error. This operation ensures that the current instance is correct, thus maintaining the specified $(m, k)$ constraint.

Recall that **our objective** is to design a state-based execution mode selection strategy that can satisfy $(m, k)$ compliance and minimize expected execution time simultaneously. 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

$$\alpha : Q^* \mapsto \{u, d, r, d + r\} \tag{7.3}$$

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 $\mathbb{P}(\psi(w(j+1, j+k)) \notin Q^* \mid \psi(w(j, j+k-1) \in Q^*, \alpha(\psi(w(j, j+k-1))) = 0$ 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 as an error to guarantee $(m, k)$ compliance, since the outcome is not observable. From a design perspective, we have to design the transitioning system of $\mathcal{A}_k$ such that only the compliant states $Q^*$ are reachable.

**Definition 24** (Compliant Transitions)**.** A transition system $\delta$ of a $k$-error-automata $\mathcal{A}_k$ is $(m,k)$-compliant if and only if for any given word $w$ with $j \geq 1$ the following implication holds

$$\psi(w(j, j+k-1)) \in Q^* \implies \delta(\psi(w(j, j+k-1)), c_{j+k}(\alpha)) \in Q^*$$

**Definition 25** (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 26** (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., $w(j+1, j+k-1)$.

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 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 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 that must be met by any feasible strategy.

**Lemma 13** (Compliant Mapping Strategy)**.** Any mapping strategy $\alpha$ for the $k$-error-automata $\mathcal{A}_k$ that satisfies the constraints

$$\alpha(\psi(x_j)) = \begin{cases} r \vee (d+r) & \text{if } \psi(x_j) \text{ is a critical state} \\ u \vee d \vee r & \text{if } \psi(x_j) \text{ is a nominal state} \end{cases} \tag{7.4}$$

leads 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 2 and Corollary 3, we know that for any induced state $q := \psi(x_j) \in Q^*$, the strategy $\alpha(q)$ must 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 leads to a compliant state. □

An $\alpha$-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 7.1.

### 7.2.2 States Reduction and Minimal Automata Construction

We propose an algorithm to generate a minimal $(m, k)$-compliant automaton $\mathcal{A}_k^*(\alpha)$, which is necessary to reduce the computational complexity of our to be designed expected execution time minimization algorithms. We note that the approach to generate minimal finite-state machines used by 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*. In contrast, our Algorithm 10 utilizes the specificity of the problem to only generate compliant states right away.

**Definition 27.** 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.

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 14.** 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 not $n$-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 15.** 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 4.** The initial set of *nominal states* can be minimized to a set of representatives of the form $\ast\cdots\ast\circ v$ 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 14 and Lemma 15, since we know that states $q, q'$ are merged up to $n$-step equivalence if $\mathbb{1}[q(n+1,:)] \geq m$ and thus $\mathbb{1}[\ast\cdots\ast\circ q(n+1,:)] \geq m$ 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 24** (Minimal Automata)**.** *The minimal number of compliant states $Q^*$ of a $(m,k)$-compliant $\mathcal{A}_k^*$ is given by*

$$|Q^*| = \frac{k!}{m! \times (k-m)!} \tag{7.5}$$

*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 15, 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 $\ell$ the number of combinations for $\ell-1$ must be substracted. This is due to the fact that by Lemma 14, we know that each state is represented by the maximal equivalence representative and the combinations with $m$ ones in the last $\ell$ characters can be extended to combinations with $m$ ones in the last $\ell+1$, which would then be covered by the representative of $\ell$. In consequence, we have that $|Q^*|$ is given by

$$\binom{k-1}{m-1} + \binom{m}{m} + \sum_{\ell=1}^{k-m} \binom{m+\ell}{m} - \binom{m+\ell-1}{m} = \binom{k}{m}$$

which proves the theorem. □

**Algorithm 10** Generation of minimal compliant $\mathcal{A}_k^*$

```
Input: Constraint $(m, k)$;
1: $\mathcal{A}_k^* \leftarrow (q_s, Q^* := \varnothing, \delta := \varnothing, \Sigma := \{0, 1\})$;
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 $\delta$;
6:     add transition $\delta(q, 0) = q(2, :) \circ 0$ to $\delta$;
7: end for
8: for each $q \in \{w \in b(k - 1, m - 1) \mid 1 \circ w\}$ do
9:     add $q$ to $Q^*$;
10:    add transition $\delta(q, 1) = q(2, :) \circ 1$ to $\delta$;
11: end for
12: return $\mathcal{A}_k^*$;
```

Let $b(z, n)$ represent 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 $\ell$ many $*$-characters as follows:

$$\bigcup_{z=0}^{k-m-1} *_{k-m-z} \circ 1 \circ b(m + z - 1, m - 1) \tag{7.6}$$

For instance in the case of $(2, 4)$ constraints, the minimal *nominal* states are given by Equation (7.6) 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 $\delta(q, 1) = q(2, :) \circ 1$ to the automata.

## 7.3 Minimization of Expected Execution Time

In this section, we explain our mapping strategy for associating execution modes with jobs by considering different strategies for *critical* and *nominal* states. Subsequently, we propose an optimization strategy based on the induced Markov Chain. Lastly, we provide an example to illustrate the workflow of our proposed strategy.

### 7.3.1 Mapping Strategy

Our mapping strategy leverages the design flexibility across various state categories to select the execution mode for the 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 task instance with *reliable* mode, i.e., $\alpha(q) = r$.
2. Release a task instance with *detected* mode and only release an immediate follow-up instance with *reliable* mode 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 $(1-p_e) \cdot C_d + p_e \cdot C_r$. 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 task instance with *reliable* mode, i.e., $\alpha(q) = r$ and $c(\alpha(q)) = 1$ with probability 1.
2. Release a task instance with *detected* mode, 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 a task instance with *unreliable* mode, 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, our preference is to select either a *detected* or an *unreliable* mode instance in each nominal state $q$ at random. That is, we choose a *detected* mode instance with probability $p_d$ or an *unreliable* mode instance with probability $p_u$, ensuring $p_d + p_u = 1$. 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 $\mathbb{P}(c(\alpha(q)) = 0) = p_d \cdot p_e + p_u$.

### 7.3.2 Induced Markov Chain

Using the mapping strategy $\alpha$, we can derive an $\alpha$-induced Markov Chain from the automata $\mathcal{A}_k^*$.

**Observation 1** (Induced Markov Chain)**.** The $\alpha$-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 $\alpha$.

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. Moreover, the specific transition probabilities are derived based on the error probability $p_e$ and the stochastic state-based mode selection, i.e., if $q$ is a nominal state then by our strategy $\alpha$ the following state transitions are given:

$$\mathbb{P}(x_{n+1} = \delta(q,0)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 0) = p_d \cdot p_e + p_u$$
$$\mathbb{P}(x_{n+1} = \delta(q,1)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 1) = p_d \cdot (1 - p_e)$$

and for critical states

$$\mathbb{P}(x_{n+1} = \delta(q,0)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 0) = 0$$
$$\mathbb{P}(x_{n+1} = \delta(q,1)|x_n = q) = \mathbb{P}(c(\alpha(q)) = 1) = 1$$

**Definition 28** (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 $\xi$ is said to be a stationary distribution or invariant distribution if

$$A \cdot \xi = \xi \tag{7.7}$$

We then obtain the corresponding stationary distribution $\xi$ according to Equation (7.7) by treating $\xi$ as an eigenvector of $A$ with an eigenvalue 1 that can be efficiently numerically solved by e.g., eigenvalue decomposition (spectral theorem [PTV+07]). Let the stationary distribution $\xi^T = (\xi_1, \dots, \xi_r)$ where the $\xi_i$ correspond to the stationary probability to be in state $q_i \in Q^*$. In consequence, the expected average execution time is:

$$\mathbb{E}(C) \coloneqq \sum_{i=1}^{r} \xi_i \cdot (p_d \cdot C_d + p_u \cdot C_u) \cdot [q_i \text{ is nominal}]$$
$$+\xi_i \cdot \min\{C_r, (1-p_e) \cdot C_d + p_e \cdot C_r\} \cdot [q_i \text{ is critical}]$$

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 $\alpha$-induced $(m,k)$-compliant Markov Chain always has a stationary distribution.

**Theorem 25** (Renewal Theorem [GS20])**.** *A finite, irreducible Markov Chain has a unique stationary distribution.*

**Definition 29** (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 26.** *The $\alpha$-induced $(m,k)$-compliant $\mathcal{A}_k^*(\alpha)$ is a finite and irreducible discrete-time Markov Chain.*

*Proof.* From Theorem 24, it immediately follows that $\mathcal{A}_k^*(\alpha)$ has finite states. Moreover, since the $\alpha$-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

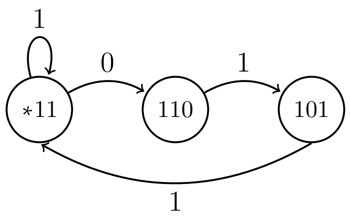


**Figure 7.2:** A minimal $(2,3)$-compliant 3-error-automata $\mathcal{A}_3^*$.

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.

Let $q, q'$ any two states in the non-minimized $\alpha$-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 $q \rightsquigarrow 11\ldots1$ ($k$ ones) and $11\ldots1 \rightsquigarrow q'$. Since for each feasible state $q \in Q$ the transition $\delta(q,1) \in Q$ is defined for *critical*- and *nominal states*, the state $11\ldots1$ can be reached from any state $q$ by successive 1-transitions. Secondly, for any given $(m,k)$-constraints, starting from state $11\ldots1$, (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 $11\ldots1$, since $q = \delta(11\ldots1, q)$ and $\mathbb{1}[q] + \not\mathbb{1}[q] = k$ and thus $\not\mathbb{1}[q] = k - \mathbb{1}[q] \leq k - m$. □

### 7.3.3 An Illustrative Example

To illustrate our approach, we provide a detailed 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 10, the generated automata is shown in Figure 7.2. The ordered set of states, denoted as $Q$, is given by $\langle Q \rangle = \langle 11, 110, 101 \rangle$. Here, $*11$ represents a *nominal* state, while both 110 and 101 represent *critical* states. By using the mapping strategy described in Section 7.3.1, 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:

$$A = \begin{bmatrix} 0.9 \cdot p_d & 0 & 1 \\ 1 - 0.9 \cdot p_d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.8}$$

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 results in the following expected average execution time $\xi_1 \cdot p_u \cdot C^u + \xi_1 \cdot p_d \cdot C^d + 2 \cdot \xi_2 \cdot \min\left(C_r, C^d + p_e \cdot C^r\right)$ and evaluates to

$$\frac{p_u \cdot C^u + p_d \cdot C^d + (1 - 0.9 \cdot p_d) \cdot \min\left\{C_r, C^d + 0.9 \cdot C^r\right\}}{3 - 1.8 \cdot p_d}$$

$$\implies \frac{230 - 137 \cdot p_d}{150 - 90 \cdot p_d} \text{ for any } p_d \in [0, 1] \tag{7.9}$$

The function in Equation (7.9) 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* $q$, the selection is always $\alpha(q) = d$. In summary, we obtain the following mapping strategy:

$$\alpha(q) = \begin{cases} d & \text{if } q \in \{101, 110\} \\ u & \text{if } q \in \{*11\} \end{cases} \tag{7.10}$$

that results in a minimal expected average execution time of 1.533. However, when we reduce the error probability to 1%, with $p_e = 0.01$, the expected average execution time becomes:

$$\frac{1150 - 766 \cdot p_d}{750 - 495 \cdot p_d} \text{ for any } p_d \in [0, 1] \tag{7.11}$$

Unlike Equation (7.9), Equation (7.11) shows a monotonically decreasing trend over the interval $[0, 1]$. Consequently, the minimum expected average execution time is reached when $p_d = 1$. This results in the altered strategy to select $\alpha(q) = d$ for any *nominal* state.

## 7.4 Reinforcement Learning Based Approach

When the error probability for each task is unknown, the probabilities of state transitions are no longer explicit, which makes the static optimization approach proposed in Section 7.3 inapplicable. To this end, we propose an artificial expert (agent) based on reinforcement learning (RL), to optimize the selections of execution modes dynamically during the runtime. In this section, we first give a short overview of RL. Afterwards, we demonstrate how the execution mode selection with $(m, k)$ constraints problem is formulated to the RL-solvable problem. Furthermore, the barrier function that assures the $(m, k)$ constraint is discussed. Finally, we present the learning policy for RL agent for the studied problem.

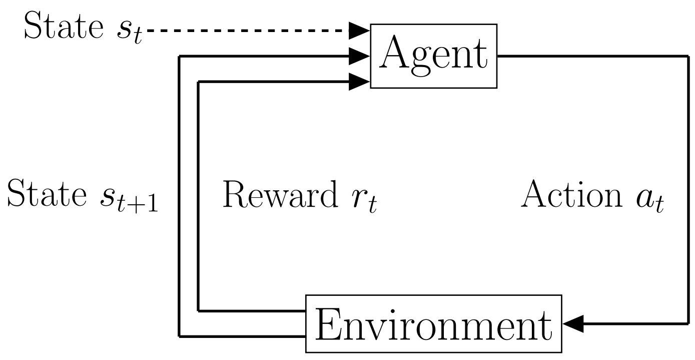


**Figure 7.3:** Agent environment interaction.

## 7.4.1 Overview of Reinforcement Learning

As one of the machine learning paradigms, RL problems are often represented or formulated as Markov Decision Processes (MDPs). Figure 7.3 shows the basic components of a MDP. The *environment* $E$ is defined as a *state* space, which is denoted as $S$. Each state instance, i.e., $s_t \in S$, is a description of the environment at time $t$. All the actions that an agent can take formulate the action space, i.e., $A$. One iteration of the MDP is shown in Figure 7.3: when an action $a_t \in A$ is taken based on current state $s_t$, the environment is transited to a new state $s_{t+1}$. During the transition, a reward $r_t$ is given to the agent based on the reward function $R$. Therefore, reinforcement learning problem can be formulated as a four tuple, i.e, $(S, A, P, R)$, where $P: S \times A \times S \mapsto \mathbb{R}$ represents the probability of state transitions $P(s_{t+1} \mid s_t, a_t)$, and $R: S \times A \times S \mapsto \mathbb{R}$ denotes the corresponding reward function. 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.

The objective of a reinforcement learning task is for the agent to learn a policy $\pi$, which is a probability density function to describe the state-to-action mapping. That is, an agent can take an action according to the policy and current state, i.e., $a = \pi(s)$. A policy $\pi$ can be formed in two different ways: a) the deterministic policy $\pi : S \mapsto A$ is a unique mapping from state to action; b) uniform stochastic policy $\pi : S \times A \mapsto A$ defines the probability distribution of actions according to a given state, i.e., $\pi(a_t|s_t) = \mathbb{P}(a = a_t \mid s = s_t)$ where $\sum_{a_t \in A} \pi(a_t|s_t) = 1$. The learned policy is evaluated by the cumulative future reward, i.e., $W_t = \sum_{i=t}^{\infty} r_i$. Since future reward is often less valuable than present reward, a discount rate $\gamma \in (0, 1]$ is included, i.e., $W_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. In addition, value function is applied to estimate the expected future reward. On the one hand, state-value function is defined as $V_\pi(s_t) = \mathbb{E}_A[Q_\pi(s_t, A)]$, which defines the expected cumulative reward from state $s_t$ by applying policy $\pi$. Action-value function for a policy $\pi$ 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 $a_t$ under state $s_t$.

According to the learning objectives, model-free RL approaches can be divided into three main categories:

- Policy-Based: the objective is to train the (deep) neural network to map the current state to the best probabilistic action to take, i.e., with the highest expected state value. The policy gradient ascent approach [SKM00] can be applied to train the policy network by maximizing $\mathbb{E}_S[V(S;\theta)]$, where the $\theta$ is the parameters of the neural network.
- Value-Based: the action-value function is applied to find the most valuable action, i.e., $a_t^* = \arg\max_a Q^*(s_t, a)$, while being in state $s_t$. Commonly, one deep Q network (DQN), i.e., $Q(s, a, \omega)$, is deployed to approximate the $Q^*(s, a)$, where $\omega$ is the parameters of the DQN, which can be trained using the temporal difference (TD) learning [Tes95].
- Actor-Critic: both the policy network (actor) to approximate $\pi(a|s)$ and the value network (critic) to approximate $Q_\pi(s, a)$ are utilized simultaneously. 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.

Besides the above three approaches for model-free RL, other model-based RL approaches can also be applied when an additional model is deployed to simulate the environment. One of the most famous model-based RL applications is the AlphaGo Zero [SHM+16; SSS+17], which utilizes Monte Carlo Tree Search (MCTS) to find moves based on previously learned moves. Although MCTS can provide more precise estimation of the reward for each action, it is still time consuming. Therefore, only the policy network is applied after the training process in some time-sensitive applications.

### 7.4.2 RL Formulation

We consider each task to be an agent. Hence, multiple agents exist within the same task system. These multiple agents in this work operate independently. In other words, they differ from the multi-agent reinforcement learning paradigm. Each agent independently takes its own actions based on its observation without requiring any information from the others. Hence, in the following, we focus on the selection strategy of execution modes for one task.

As shown in Section 7.3.2, the job level execution modes selections for each task is an independent MDP. The action space for each task is, selecting the execution mode for its next job, i.e., $A = \{0: \textit{unreliable},\ 1: \textit{detected},\ 2: \textit{reliable}\}$. The environment state comprises the execution statuses for a task's corresponding jobs, i.e., $s_t = [s_t^1, s_t^2, \ldots, s_t^\ell]$. The execution status of each job, i.e., $s_t^j$, has the following four attributes:

- *Correctness*: This is a binary variable indicating the correctness of the corresponding job, i.e., according to Definition 20, 0 denotes error execution and 1 denotes correct execution.
- *Execution mode*: This records the execution mode of a job, corresponding to an element from the action space.

- *Expected execution time*: This indicates the expected execution time of each job according to the corresponding execution mode, e.g., $C^u$, $C^d$, or $C^r$. Please note, the expected execution times here are the same as the WCET of different modes, which are different to the expected execution times in Section 7.3.1.
- *Real execution time*: This is used for recording the real execution time of a job. For both *unreliable* and *reliable* modes, a job should have the same real execution time as the expected execution time. However, for the *detected* mode, the real execution time may be much longer than expected when a job is forced to be correct according to $(m, k)$ constraint but is detected as *error*. The job has to re-execute the *reliable* mode to satisfy the $(m, k)$ requirement, where the real execution time equals to $C^d + C^r$.

The length of an environment state, represented by $\ell$, should be at least $k$ to ensure that the execution statuses of a minimum of $k$ jobs can be recorded and checked against the $(m, k)$ constraint. Once the length of the environment state is determined and fixed ($\ell = 2k$ in this work), a FIFO stack-like mechanism is applied, i.e., the state matrix only record the latest $\ell$ jobs' execution statuses.

An example of the environmental state transition is illustrated in Figure 7.4, corresponding to a task with the $(m = 3, k = 5)$ constraint. The length of the environment state is set to $k = 5$. The task takes the action to execute the *detected* mode, i.e., $a_t = 1:$ *detected* for its next new job $J^6$. The second and third elements of $J^6$'s status have been determined as $\begin{pmatrix} 1 \\ C^d \end{pmatrix}$ by default. By checking against the $(m = 3, k = 5)$ constraint, the $J^6$ must be executed correctly. Therefore, the first element is marked as 1. However, in the real execution, $J^6$ is detected as *error*, an additional *reliable* mode has to be executed immediately. As a result, the last element of $J^6$'s status is $C^d + C^r$. Subsequently, the state transitions from $s_t$ to $s_{t+1}$, achieved by adding the status of $j^6$ and removing the status of $J^1$.

$$
\begin{array}{c}
\begin{array}{ccccc} J^1 & J^2 & J^3 & J^4 & J^5 \end{array} \\
\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 2 \\ C^d & C^u & C^d & C^d & C^r \\ C^d & C^u & C^d & C^d & C^r \end{pmatrix} \\
s_t
\end{array}
\xrightarrow[a_t = 1]{}
\begin{array}{c}
\begin{array}{ccccc} J^2 & J^3 & J^4 & J^5 & J^6 \end{array} \\
\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 & 1 \\ C^u & C^d & C^d & C^r & C^d \\ C^u & C^d & C^d & C^r & C^d + C^r \end{pmatrix} \\
s_{t+1}
\end{array}
$$

**Figure 7.4:** An example for state transition of a task.

Please note, different environment construction approaches can also be applied. For example, a three dimensional tensor can be applied to record several latest aforementioned two dimensional matrices, so that the more comprehensive information is recorded without discarding.

### 7.4.3 Barrier Function

To achieve the objective of minimizing the average execution time for each task, the reward is intuitively set to be inversely proportional to a job's real execution time, i.e., the longer real execution time a job has, the less reward the agent obtains. Besides the objective of the RL-based approach, i.e., maximize the cumulative reward, the $(m, k)$ constraint has to be satisfied as well. To address this, we introduce a barrier function that checks the category of the current state before the agent deploys a job with the selected execution mode. If the current state $s_t \in S_{nom}$, the barrier function bypasses the check as all three execution modes are permissible for the next job. However, if the current state $s_t \in S_{crt}$, the result of next job has to be correct. If the agent decides to execute the *unreliable* mode, the barrier function forbids the action and limit the options to only execute the *detected* or the *reliable* mode. In addition, an extremely large negative reward is returned to the agent. The barrier function serves two purposes: a) it ensures adherence to the $(m, k)$ constraint, and b) it teaches the agent to avoid selecting the *unreliable* mode when $s_t \in S_{crt}$.

Although our agent operates in a model-free manner, the barrier function implicitly enforces adherence to the R-pattern in the worst-case scenario:

**Theorem 27.** *The worst case execution pattern of RL-based approach is the same as the R-pattern adopted in [CBC+16; KS95].*

*Proof.* The worst case execution R-pattern in [CBC+16] contains $(k - m)$ incorrect jobs that are executed in the *detected* mode, and $m$ correct jobs that are executed in the *detected* mode and the *reliable* mode, where all the results of $k$ jobs with the *detected* mode are incorrect. In our RL-based approach, the agent is limited to select one execution mode for the next job. Subsequent to this choice, the barrier function is applied to check the feasibility of the selected execution mode and the correctness of the result when a job finishes its execution. If the current state $s_t \in S_{nom}$, no additional effort is needed. Only when the current state $s_t \in S_{crt}$, the execution pattern, i.e., the *detected* mode and *reliable* mode are executed for the same job, can happen. The worst case of the RL approach is that the agent always decides to execute *detected* mode and all the execution results are incorrect. Therefore, $m$ jobs with *reliable* mode are executed right after the *detected* mode, which is the same as the R-pattern in [CBC+16; KS95]. □

**Theorem 28.** *The derived schedule from the RL-based approach under the barrier function is schedulable if the schedulability based on the R-pattern has been ensured.*

*Proof.* Given a static pattern, a task is schedulable if it passes the schedulability test in Lemma 1 from [CBC+16]. As shown in Theorem 27, in the worst case the RL-based approach performs the same as the R-pattern. Therefore, if a task has passed the schedulaiblity test in Lemma 1 from [CBC+16] based on the R-pattern, the derived schedule for this task from the RL-based approach must be schedulable in the worst case, which concludes the proof. □

### 7.4.4 Learning Policy

In this work, we utilize the deep Q network (DQN) agent as our RL model, where Boltzmann Q Policy is applied to estimate the Q value of each action. While exploring, the agent creates an action distribution, which describes how optimal an action is according to the data gathered. Afterwards, Boltzmann policy turns the agent's exploration behavior into a spectrum between picking the action randomly (random policy) and always picking the most optimal action (greedy policy). The DQN agent is constructed by a 10-layer neural network, which contains 1 input layer, 1 activation layer, 1 flatten layer, 6 fully connected layers, and 1 output layer.

Please note that the proposed RL-based approach is not limited to any specific learning policy, all learning approaches that support a discrete action space are applicable. Finding the best policy to train a DQN agent is considered out of scope.

## 7.5 Experimental Evaluation

To evaluate the effectiveness of our proposed approaches, we numerically simulate the task system. We then compare the performance of the proposed mapping strategy (both when $p_e$ is known and when $p_e$ is unknown) against state-of-the-art methods across a wide range of configurations. The adopted hardware platform was the GPU server of the emulation platform in Section 2.4.3. Overall, the following approaches are evaluated, namely:

- Optimized mapping strategy in Section 7.3 (OPT): the selection of execution mode for each state follows the optimized mapping between states and execution modes.
- RL-based approach in Section 7.4 (RL): the environment is constructed by using OpenAI Gym [BCP+16]. The implementation of RL agent relies on Keras-rl package [Pla16] and TensorFlow [AAB+16].
- Adaptive approach (ADP) [CBC+16]: R-pattern is applied, i.e., postpone the forced-correct jobs as late as possible, which can benefit this approach due to the flexibility.
- Static approach (STA) [NQ06]: executes $m$ jobs in the reliable mode and $(k - m)$ jobs in the unreliable mode for any consecutive jobs. Here, R-pattern will be equal to E-patterns in terms of utilization reduction, e.g., energy saving, regardless of the given error probabilities.

### 7.5.1 Single Task Evaluation

We conducted evaluations on a single task with various experimental settings, such as the $(m, k)$ constraint and error probability. The $m$ was chosen from a set, i.e., $m \in \{2, 4, 6, 8\}$, $k$ was a constant number, i.e., 10, and the error probability was given as $p_e \in \{0.05, 0.15, 0.3\}$. We set $C^d$ to $1.5 \times C^u$ and $C^r$ to $3.5 \times C^u$ to emulate

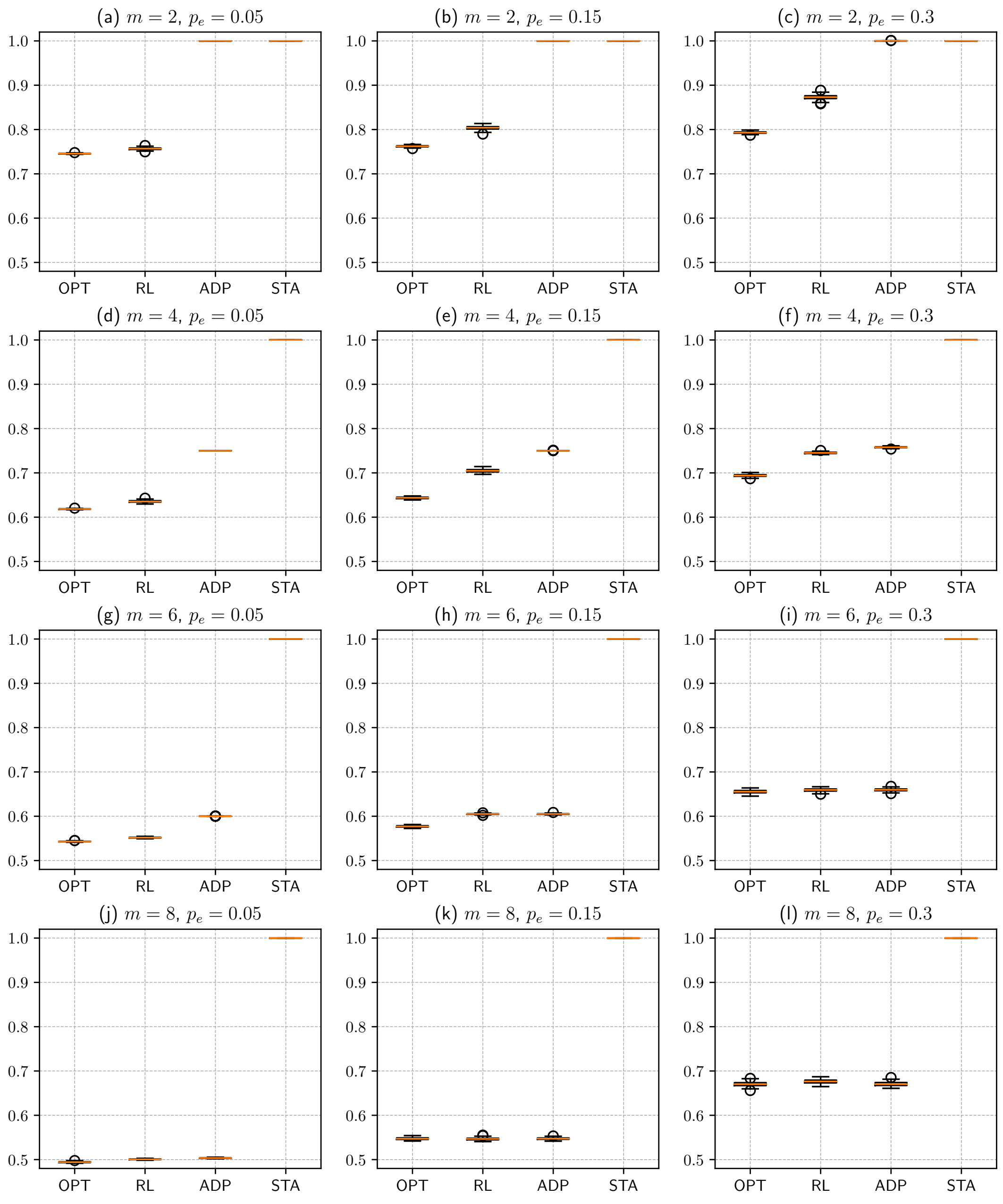


**Figure 7.5:** Results of normalized utilization for tasks with different settings.

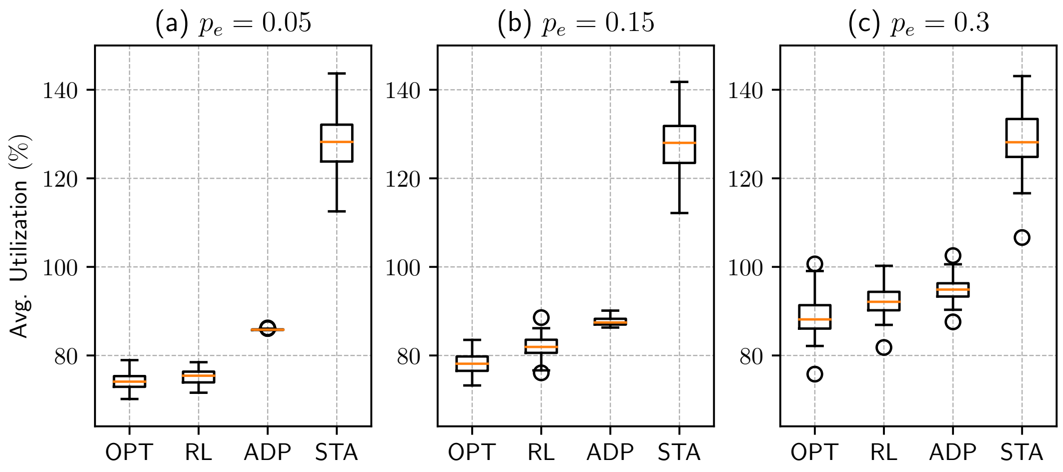


**Figure 7.6:** Results for multitask systems with different error probabilities.

software-based error detection and recovery. Each task released $10,000$ jobs for one iteration, and 100 iterations were performed.

Figure 7.5 shows the results for one single task, where the y-axis represents the normalized average execution time for jobs of one task, i.e., the lower the better. In general, the OPT approach outperforms all the other approaches in all the evaluated cases. When the error probability $p_e$ or the ratio $m/k$ is relatively low, e.g., in Figure 7.5 (a)-(h), (j), and (k), the OPT approach outperforms other approaches significantly. When both error probability and the ratio $m/k$ increase, e.g., in Figure 7.5 (i) and (l), the options to select the execution modes become more limited, which results in a negligible difference between the OPT, RL, and ADP approaches.

The RL approach also dominates in most of the evaluated cases of ADP and STA approaches, as seen in Figure 7.5 (a)-(g) and (j). However, it always performs worse than the OPT approach (without knowing the error probability in advance). When the error probability is relatively low, e.g., in Figure 7.5 (a), (d), (g) and (j), or both error probability and the ratio $m/k$ are relatively high, e.g., in Figure 7.5 (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 7.5, 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 7.5, we can observe that the achievable benefit from the RL approach significantly decreases. It is because the agent tends to execute the *unreliable* modes first to maximize the cumulative reward. However, such an intention also reduces the resilience, causing the subsequent jobs in the *detected* mode to often switch to the *reliable* mode immediately for a detected error.

## 7.5.2 Multitask System Evaluation

We conducted the evaluation for multitask on multiprocessor systems as well, where tasks were scheduled by partitioned scheduling. We considered 100 task sets, each of them contained 40 tasks that were scheduled on 4 processors. The total utilization for each task set $U_{\mathbf{T}} \in [20\%, 200\%]$ with each step 20%, when all tasks only executing in the *reliable* modes. For each task, the utilization was generated by applying the Dirichlet-Rescale (DRS) algorithm [GBD20], where utilization for each task was not higher than 50%. The task periods $T_i$ were randomly selected from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$, which are the periods recommended for automotive systems [KZH15]. The execution time of *reliable* mode for each task was calculated, i.e., $C_i^r = U_i * T_i$, and $C_i^u$ and $C_i^d$ are calculated with the same ratios in Section 7.5.1. We considered worst-fit partitioning algorithm. That is, tasks are sorted decreasingly at first. Afterwards, each unassigned task (with the highest utilization) was allocated to the processor with the lowest utilization. The configurations for $(m, k)$ constraints were the same as described in Section 7.5.1. We set each system with only one unified error probability for all the tasks running on it. We set a hyper-period as 10,000 time units, and obtain the average utilization of the system by dividing the accumulated execution time from all jobs running in the system by the hyper-period.

Due to the similarity over results, we selectively show the task systems with total utilization 200% in Figure 7.6 to present the trends. In general, the results show that the both OPT approach and RL approach can decrease the utilization for multitask systems in all the evaluated configurations. In particular, both the OPT and RL approaches are most effective when the error probability is relatively low, as seen in Figure 7.6 (a).

### 7.5.3 Discussions of Overheads

For each task, the optimal approach generates a lookup table offline for on-the-fly mode selection. The offline computational overhead 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. The runtime overhead is only table lookup and negligible.

To evaluate the overhead of training and mode selection, we deployed our RL-base approach on both the GPU server in Section 2.4.3 and Nvidia Jetson AGX Xavier (32G) board in Section 2.4.3. On the GPU server, the training process for each task, with a single configuration, took 450 seconds on average. The training process is repeated 20 times. The trained DQN with the highest reward is selected, and the overhead for each task to select the execution mode for the next job is 300 microseconds. On the Nvidia Jetson AGX Xavier board, we evaluated two power modes with different power budgets: a) the default mode with a 15$W$ power budget and 4 processors running at 2188 MHz, and b) the MAXN mode, which has no power budget limitation and has 8 processors running at 2265.6 MHz. The detailed configurations can be found in [NVI21].

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 default mode, the training process took 19.7 minutes, and execution mode selection took 1.06 milliseconds on average. In MAXN mode, the training process took 15 minutes, and execution mode selection took 1 milliseconds on average. We observe that the increase in the number of processors does not proportionally reduce the training time, and none of the processors are fully 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.

For real-world applicability, a lookup table can also be utilized since the state space in our application, i.e., the minimal legal space $S^*$, is limited. That is, the trained DQN network can be converted to a table, that 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 or changing error probabilities, as one safe policy, the ADP proposed in [CBC+16] can be first applied to estimate a safe error 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.

## 7.6 Summary

We study how to selectively deploy fault-tolerance techniques as different execution modes under $(m, k)$ constraints to reduce the number of expensive executions and eventually save energy, while satisfying the schedulability. Through formulating the mapping between states and selections for execution modes of jobs, we propose two different adaptive approaches. When the error probability is known, we provide a Markov chain based approach to optimize the mapping strategy. When the error probability is unknown, an RL-based agent is trained to aid the selection of the execution mode for its next job. To demonstrate the applicability of our approaches, we provide extensive numerical evaluations. The results show that both proposed approaches outperform the state-of-the-art in most of the evaluated cases, especially when the error probability is relatively low under the same $(m, k)$ constraint.

Chapter 8

# Conclusion and Future Work

## Contents

In the previous chapters of this dissertation, we investigated the analysis and optimizations of timing-constrained embedded systems, approaching the topic from two distinct perspectives. Machine learning, with its remarkable adaptivity, prowess in data-driven decision-making, and capacity to manage large-scale data, has emerged as a key solution provider across various application domains, such as medical diagnosis, autonomous driving, automation control, and environmental monitoring. Concurrently, embedded systems, known for their determinism, real-time functionality, reliability, and efficiency, are emerging as favored platforms for deploying machine learning models, particularly within industrial domains.

Despite the synergies, the deployment of machine learning models in embedded systems presents significant challenges. On the one hand, such applications often come with strict timing requirements, necessitating careful consideration due to the inherent complexity of machine learning tasks. On the other hand, the constrained resources of embedded systems, coupled with the high computational demands of many machine learning tasks, further complicate this integration.

Moreover, considering machine learning's adaptability to dynamic environments, it presents itself as a source of innovative solutions to the existing challenges within the embedded systems domain. This adaptability becomes a crucial factor in enhancing the functionality and efficiency of these systems, particularly in dynamically changing scenarios.

In the subsequent sections, we summarize the contributions of this dissertation. The conclusion offers a concise recapitulation, structured according to the chapters where the main subjects are detailed. Following that, we discuss ongoing and future research directions, aiming to address the limitations observed in our current study and to enhance the effectiveness and applicability of our approaches proposed in this dissertation.

## 8.1 Conclusion

In this section, we summarize three contributions to optimize and analyze timing-constrained embedded systems. First, we focus on resource synchronization, specifically considering relatively long critical sections. To validate the proposed approach, we provide the implementation in two widely applied operating systems and propose a formal verification framework to ensure the correctness. Secondly, we optimize the deployment of machine learning models on resource-constrained distributed embedded systems. Additionally, we discuss an application that employs reinforcement learning to address challenges inherent to embedded systems. We also discuss some limitations observed in our current study.

### 8.1.1 DGA for Multiprocessor Real-Time Synchronization

**Contributions:** Chapter 4 answers some fundamental questions regarding multiprocessor real-time synchronization. A key finding is that the computational complexity of the simplest synchronization setting is categorized as NP-hard. In this setting, each task comprises a single non-nested critical section, i.e., OCS task model, and all tasks have an identical period and deadline. Introducing additional processors or permitting task preemption and migration does not alleviate this complexity.

To address the challenges of extended critical sections in machine learning tasks, for instance, those accessing large datasets or utilizing GPUs for acceleration, we introduced the dependency graph approach. This non-work-conserving method contains two steps. In the first step, dependency graphs are constructed to determine the execution sequence of critical sections for each shared resource. Towards this, the approach employs uni-processor non-preemptive scheduling algorithms specific to the OCS task model. We further expanded the approach to more general task systems, such as those accommodating multiple non-nested or nested critical sections per task, i.e., MCS and Nested-MCS task model, respectively. For these models, constraint programming for job shop scheduling is applied to construct dependency graphs. In the second step, for scheduling the constructed dependency graphs, we introduced several LIST-EDF-based algorithms. These algorithms set deadlines for each sub-job by considering the precedence constraints. These algorithms are supplemented with both global and partitioned scheduling strategies. For the latter, we introduced two distinct partitioning methods: one based on federated scheduling to segregate tasks according to shared resources, and another utilizing a worst-fit heuristic. In addition, periodic task systems are supported by unrolling jobs for all tasks in one hyper-period, and dependency graphs are constructed at the job level.

Empirical results underscore the performance advantages of our proposed DGA. It outperforms state-of-the-art methods in the literature across a majority of the evaluated configurations. This superiority is particularly evident in configurations where critical sections occupy a relatively long duration, specifically, longer than 10% of a task's total execution time.

**Limitations:** The proposed DGA has two main limitations. First, only periodic task systems are supported, wherein each task must release its jobs strictly according to its period, and all tasks simultaneously release their initial jobs. Second, in certain configurations, specifically when the critical section occupies between 1% and 10% of the total execution time, our DGA, in certain cases, underperforms compared to some state-of-the-art methods, such as ROP.

### 8.1.2 Implementation and Verification of Protocols

**Contributions:** Chapter 5 details how the proposed dependency graph approach is implemented in both LITMUS$^{RT}$ and RTEMS. Measured overheads demonstrate that our implementations closely align with those of officially supported multiprocessor resource synchronization protocols.

To guarantee the accuracy and reliability of these implementations, we introduced a framework capable of formally verifying an implemented protocol. The framework operates under the premise that the protocol is implemented in a correctly functioning RTOS. Upon utilizing this verification framework, we identified a long-standing inconsistency between ICPP and MrsP implementations with their formally described properties in RTEMS. Subsequently, we proposed a solution for ICPP and MrsP (under a specific condition regarding nested resource access). Specifically, when a thread acquiring nested resources always requests them in a non-descending order of priority ceilings.

**Limitations:** The dependency graph approach does not actually need the locking/unlocking mechanism anymore since the dependency graph handles mutual exclusion of the critical sections. However, our current implementation in both LITMUS$^{RT}$ and RTEMS is still based on the existing locking mechanism. There is still a lack of research on how to implement the dependency graph approach in modern real-time operating systems with low overhead. Furthermore, our formal verification framework operates under a significant presumption. That is, the RTOS, wherein the protocols are implemented, is assumed to be functionally correct. This assumption may not always hold true in real-world scenarios. Therefore, a thorough verification process for the RTOS implementation itself is imperative.

### 8.1.3 MBO on Distributed Embedded Systems

**Contributions:** Chapter 6 introduces MODES, a hyper-parameter tuning approach designed for distributed embedded systems that takes resource constraints and data privacy into account. MODES consists of two branches, each with different objectives, i.e., improving prediction performance and enhancing tuning efficiency, respectively. In MODES-B, the distributed embedded system is treated as a single black box, with the objective of tuning a set of hyper-parameters for the machine learning model in each node while simultaneously adjusting the corresponding weights for these nodes. Conversely, MODES-I treats each node as a duplicate of the same black box, allowing the tuning process to be parallelized. This allows multiple proposed sets of hyper-parameters to be evaluated independently across different nodes.

Evaluation results demonstrate that MODES-B outperforms the baseline approach, where each node tunes its hyper-parameters in isolation. This superiority is evident in terms of both prediction accuracy and statistical stability. Meanwhile, MODES-I greatly improves the efficiency of the tuning process and retains comparable performance levels.

**Limitations:** Our approach is primarily designed for homogeneous distributed embedded systems. However, heterogeneous distributed embedded systems are also common in real-world scenarios. Therefore, modifications are necessary to adapt to these heterogeneous systems. Additionally, we assumed a consistent environment, implying that optimized hyper-parameters will remain relevant for future applications. However, with the shifting nature of environments and data patterns, the optimized set of hyper-parameters might become inapplicable. This is a challenge commonly termed as the *concept drift* problem. Currently, our framework lacks the flexibility to address this issue.

### 8.1.4 RL for Average Task Execution Time Minimization

**Contributions:** Chapter 7 explores an application in which reinforcement learning is employed to address challenges in the embedded systems domain. This study focuses on minimizing the average task execution time by selectively deploying fault-tolerance techniques as different execution modes under $(m, k)$ constraints. The objective is to reduce the number of expensive executions, thereby conserving energy, without compromising schedulability. We formalize this challenge by associating states with choices of execution modes for tasks.

In scenarios where the error probability is known and stable, an optimal method is designed. This method conducts mapping optimization by transforming all permissible states into a Markov chain. Conversely, when the error probability remains unknown or unstable, an RL-driven agent is trained to assist in selecting of the execution mode for its next job.

Our evaluation results indicate that both introduced strategies outperform existing methods in a majority of evaluated scenarios, especially when the error probability is relatively low under the same $(m, k)$ constraint.

**Limitations:** The RL-based strategy, while promising, exhibits non-negligible operational overheads on actual hardware. This limitation restricts its practical applicability. Hence, dedicated efforts to optimize the implementation and reduce these overheads are imperative.

## 8.2 Ongoing and Future Work

In light of the identified limitations of our present study, we propose several directions to advance the deployment of machine learning tasks in embedded systems. These encompass: a) improving performance with respect to timing requirements, b) considering a more generalized task model, c) ensuring a robust implementation, and d) enhancing overall performance in dynamic environments. Furthermore, we will discuss our ongoing efforts that leverage reinforcement learning to elevate the average performance of embedded systems.

**Exploration of Different DAG Construction Approaches** As highlighted in Section 4.8.5, the primary objective of job-shop scheduling optimization in constraint programming is to minimize the maximum lateness. However, this singular optimization objective might inadvertently neglect the execution order of intermediate sub-jobs, especially when these sub-jobs do not directly impact the optimal lateness. Such an oversight could result in performance degradation, especially when the utilization of critical sections is relatively low.

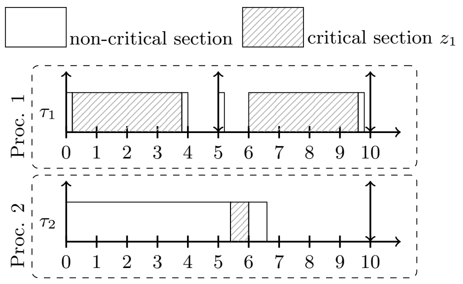
Our future direction aims to investigate alternative optimization objectives for constructing DAGs associated with shared resources. Such approaches would prioritize every sub-job, ensuring a comprehensive treatment. One prospective strategy might involve deploying two optimization procedures in succession, each with distinct objectives.

In preliminary experiments, we combined the optimization of maximum lateness with a workload averaging objective. This combination exhibited promising results, particularly when the overall utilization of critical sections was in the range of $[1\%, 10\%]$. The optimization of maximum lateness is detailed in Section 2.2.1. The workload averaging objective aims to distribute the workloads of critical sections from various shared resources uniformly across a specified time frame. To implement this for a given task set, the following steps are necessary:

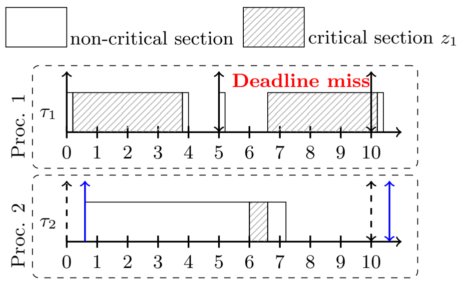
1. Operate the optimization to minimize the maximum lateness as stated in Section 4.6.
2. Segment the hyper-periods into evenly sized time windows.
3. Strive to minimize the disparities in the number (or utilization) of critical sections across these time windows, ensuring that the maximum lateness, as established in the initial step, remains unaffected.

The detailed configuration of this approach, specifically the size of the time windows, requires further investigation. Additionally, for sub-jobs that don't directly influence either objective, their execution order still needs to be determined.

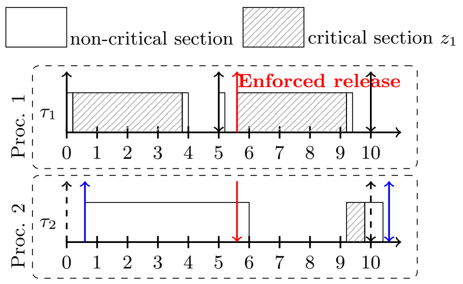
**Incorporating Release Jitters into DGA Design** In the foundational design of DGA, the primary focus was on frame-based task systems and periodic task systems. A fundamental assumption was that every task within a set would release its first job synchronously at time 0, and that subsequent jobs would be released at the start of each period, i.e., $r_i^\ell = (\ell - 1) \cdot T_i$. Consequently, the first step of DGA enables the predetermined execution order of sub-jobs accessing the same shared resource through offline strategies.



**(a)** A DGA compatible schedule.



**(b)** An infeasible schedule with release jitter.



**(c)** A feasible schedule with enforced release.

**Figure 8.1:** Different scheduling scenarios using DGA considering release jitters.

While direct application of DGA to purely sporadic task systems may be infeasible due to these predetermined execution orders of critical sections, there is potential to adapt it for systems with release jitters. In practical scenarios, such release jitters imply $r_i^\ell > (\ell - 1) \cdot T_i$. Introducing these jitters without adjusting the execution orders defined by the DAGs can lead to missed deadlines.

To illustrate, consider an example with two tasks under the OCS task model, where both $\tau_1$ and $\tau_2$ contain three computational segments and the middle segment request shared resource $z_1$, i.e., $\tau_1 = (((0.2, 0), (3.6, 1), (0.2, 0)), 4, 5, 5)$ and $\tau_2 = (((5.4, 0), (0.6, 1), (0.6, 0)), 6.6, 10, 10)$. Both tasks require the shared resource $z_1$. The Potts method constructed graph suggests the critical sections' execution order as: $J_{1,2}^1 \rightarrow J_{2,2}^1 \rightarrow J_{1,2}^2$. Scheduled through Partitioned LIST-EDF, $\tau_1$ is allocated to processor 1 and $\tau_2$ to processor 2. Without a release jitter, a feasible schedule is produced in Figure 8.1a. However, the introduction of a release jitter for $\tau_2$, i.e., $r_2^1 = 0.6$ in Figure 8.1b, makes the schedule infeasible when following the predefined execution order of critical sections.

To address release jitters, we intend to employ the *release enforcement approach*. This strategy enforces the release of a sub-job under certain conditions. Specifically, if a sub-job's predecessor is not released by the latest permissible release time, the sub-job can bypass the dependency graph constraints and release itself directly.

We denote the latest release time of a sub-job's predecessor as $\hat{r}^{\ell}_{i,j}$. This is defined by the minimum of two times: the deadline of the sub-job's predecessor minus its WCET, and the deadline of the current sub-job minus the sum of its WCET and its predecessor's WCET, i.e., $\hat{r}^{\ell}_{i,j} = \min d_{PredJ^{\ell}_{i,j}} - C_{PredJ^{\ell}_{i,j}}, d^{\ell}_{i,j} - C_{PredJ^{\ell}_{i,j}} - C^{\ell}_{i,j}$. Considering the previously discussed task set, the sub-job $J^2_{1,2}$ becomes ready at time 5.2. The combined execution time for all its successors is 3.8 time units. The worst-case execution time (WCET) of its predecessor, $J^1_{2,2}$, is 0.6 time units. To ensure schedule feasibility, $J^1_{2,2}$ must be released no later than $d^2_1 - C_(1,2) - C_{1,3} - C_{2,2}$, i.e., 5.6. This computation yields a time of 5.6. If $J^1_{2,2}$ is still not released by this time,$J^2_{1,2}$ can commence its execution immediately. The resultant feasible schedule with enforced release times is shown in Figure 8.1c.

Jobs that miss their latest release times are subsequently added to the wait queue of the corresponding shared resource(s). This wait queue is prioritized based on the deadlines of these critical sections.

Although this strategy offers some advantages, the comprehensive worst-case response time guarantee after applying these adjustments under different scenarios remains an open question for further research.

**Supporting Resource Synchronization Protocols in the Latest RTEMS Release with Formal Verification** In our previous work [SPM+22], we integrated support for MPCP, DPCP, FMLP, and DFLP in RTEMS version 4.12. However, the 4.12 release branch of RTEMS has been superseded by the 5.1 release. Moreover, our initial implementations lacked the rigor of formal verification.

To the best of our knowledge, no existing studies or projects focus on both the implementation of resource synchronization protocols and their formal verification within the RTEMS environment. To fill this gap, we plan to convert our previous implementations to align with the latest RTEMS release, specifically 5.1, and incorporate formal verification processes to ensure their correctness. Additionally, it is necessary to expand the formal verification process to some key components of the RTOS. Verifying the EDF scheduler in RTEMS is our next step. We aim to offer valuable insights and tools to our community peers by making our refined patch available as open source.

**Considering Concept Drift When Deploying Machine Learning Models** Machine learning model deployment often faces fluctuating environments. A model optimal in one environment may lose its efficacy when the environment changes, a phenomenon termed *concept drift*. Addressing this dynamic nature of environments when optimizing machine learning models is tantamount to solving a *dynamic optimization problem*.

An example of concept drift is the *bit flip errors*. Grounded in contemporary computer architecture, data in memory devices is encapsulated as binary digits, e.g., an 8-bit binary sequence can symbolize 256 distinct decimal values. Memory hardware is susceptible to transient faults, known as bit flip errors, which can stem from manufacturing discrepancies, ambient radiation, or temperature fluctuations. Essentially, a bit flip error denotes an inadvertent switch from 0 to 1 or its reverse. To illustrate, an unsigned integer, 10 (binary: 00001010), could transform into 26 (binary: 00011010) due to a bit flip in its fourth binary bit. We assume the probability of any bit undergoing such a flip remains consistent.

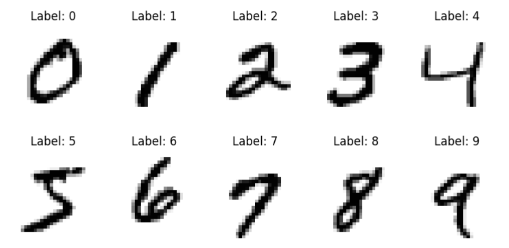
Despite the existence of error detection and mitigation solutions like RAM parity and ECC memory, their uptake in low-power embedded systems remains low, mainly due to cost and space constraints. Ferroelectric FET (FeFET), an emergent non-volatile memory (NVM) technology, holds significant promise. However, its reliability is intrinsically temperature-dependent. As temperatures rise, the FeFET's bit flip error rate increases correspondingly. Many embedded systems applications operate across a broad temperature spectrum, and most active cooling mechanisms, like fans, only kick in at critical temperature thresholds.This constraint arises from considerations of cost, size, and energy consumption. Consequently, the broad range of feasible operating temperature of applied hardware leads to significant variability in the bit flip error rate.

A machine learning model's performance often hinges on its training data and the new, unseen test data. To gauge how bit flip errors in a specific dataset influence a machine learning model's performance, we artificially introduced bit flips into the memory storing the renowned MNIST dataset [LCB98]. Here, each handwritten digit is represented using 784 integers, with each integer encapsulated by an 8-bit binary sequence. Using this data's bit representation in the model involves introducing bit flips into the memory. In particular, these bit errors manifest as multi-bit flips that are symmetric, i.e., the likelihood of a 0 becoming a 1 mirrors that of a 1 becoming a 0. This presupposition depicts the bit flip probability during each read from this unreliable, or approximate, memory. Figure 8.2 showcases a sample containing ten handwritten digits, albeit with divergent error rates.

In our upcoming research, we aim to refine MODES by enhancing its adaptability to changing environments, e.g., the bit flip errors with changing error rates.

**Applying Machine Learning Approaches for Scheduling Real-Time Tasks**
Recent research in the domain of embedded and real-time systems has shown growing interest in machine learning methodologies, especially reinforcement learning, due to their potential applications. For example, Bo et al.[BQL+21] proposed an innovative scheduling technique for multiprocessor real-time systems based on deep RL. Similarly, Xu et al.[XKH+23] introduced an approach leveraging RL to optimize the path latency across all scheduled runnables in automotive systems.

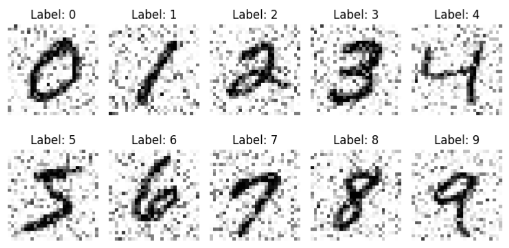
Despite these promising works, there remains a gap in research concerning the application of RL for resource synchronization in multiprocessor environments. One potential exploration is to involve RL within the DGA framework, where it assists in the construction of the dependency graph. Here, the feasibility of the generated schedule would determine the reward structure for the RL agent: a positive reward



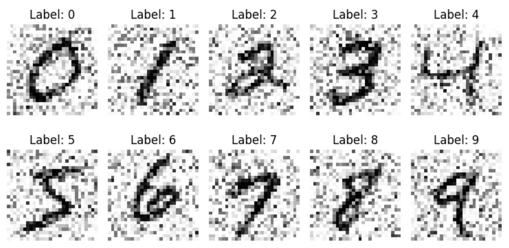
**(a)** A subset of MNIST with 0% error rate.

Label: 0
Label: 1
Label: 2
Label: 3
Label: 4
Label: 5
Label: 6
Label: 7
Label: 8
Label: 9

**(b)** A subset of MNIST with 5% error rate.



**(c)** A subset of MNIST with 10% error rate.



**(d)** A subset of MNIST with 15% error rate.

**Figure 8.2:** Examples of visualized MNIST dataset with different error rates.

for feasible schedules and a negative one otherwise. Alternatively, designing an RL-based scheduler dedicated to synchronize resource access, ensuring mutually exclusive accesses, presents another interesting direction.

However, However, a significant emerging challenge in this context is RL's inconsistency in guaranteeing hard real-time properties. While RL may enhance performance metrics in an aggregate sense, there remain instances where strict real-time deadlines may not be consistently met.

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography

[AA05] T. A. AlEnawy and H. Aydin. “Energy-Constrained Scheduling for Weakly-Hard Real-Time Systems”. In: *Proceedings of the 26th IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 2005, pp. 376–385. DOI: 10.1109/RTSS.2005.18. URL: https://doi.org/10.1109/RTSS.2005.18 (Cited on page 48).

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467 (Cited on page 190).

[AAB15] H. Almatary, N. C. Audsley, and A. Burns. “Reducing the Implementation Overheads of IPCP and DFP”. In: *2015 IEEE Real-Time Systems Symposium, RTSS*. IEEE Computer Society, 2015, pp. 295–304. DOI: 10.1109/RTSS.2015.35. URL: https://doi.org/10.1109/RTSS.2015.35 (Cited on page 43).

[Age07] A. A. Ageev. “A 3/2-Approximation for the Proportionate Two-Machine Flow Shop Scheduling with Minimum Delays”. In: *Approximation and Online Algorithms, 5th International Workshop, WAOA, Revised Papers*. Ed. by C. Kaklamanis and M. Skutella. Vol. 4927. Lecture Notes in Computer Science. Springer, 2007, pp. 55–66. DOI: 10.1007/978-3-540-77918-6_5. URL: https://doi.org/10.1007/978-3-540-77918-6%5C_5 (Cited on page 76).

[AGO+13] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz. “A Public Domain Dataset for Human Activity Recognition using Smartphones”. In: *21st European Symposium on Artificial Neural Networks, ESANN*. 2013. URL: https://www.esann.org/sites/default/files/proceedings/legacy/es2013-84.pdf (Cited on page 165).

[AHP+10] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. “Automated Verification of a Small Hypervisor”. In: *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE, Proceedings*. Ed. by G. T. Leavens, P. W. O’Hearn, and S. K. Rajamani. Vol. 6217. Lecture Notes in Computer Science. Springer, 2010, pp. 40–54. DOI: 10.1007/978-3-642-15057-9_3. URL: https://doi.org/10.1007/978-3-642-15057-9%5C_3 (Cited on page 42).

[AMP+19] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. “Leveraging Rust Types for Modular Specification and Verification”. In: *Object-Oriented Programming Systems, Languages, and Applications*. Vol. 3. 2019, 147:1–147:30. DOI: 10.1145/3360573 (Cited on page 42).

[ANN+20] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. “An Empirical Survey-based Study into Industry Practice in Real-time Systems”. In: *41st IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2020, pp. 3–11. DOI: `10.1109/RTSS49844.2020.00012`. URL: `https://doi.org/10.1109/RTSS49844.2020.00012` (Cited on page 50).

[Aud91] N. C. Audsley. “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times”. In: (1991) (Cited on page 102).

[BA07] B. Brandenburg and J. Anderson. “Feather-trace: A lightweight event tracing toolkit”. In: *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*. Citeseer. 2007, pp. 19–28 (Cited on page 5).

[BA08] B. B. Brandenburg and J. H. Anderson. “An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS$^{RT}$”. In: *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, Proceedings*. IEEE Computer Society, 2008, pp. 185–194. DOI: `10.1109/RTCSA.2008.13`. URL: `https://doi.org/10.1109/RTCSA.2008.13` (Cited on page 5).

[BA10] B. B. Brandenburg and J. H. Anderson. “Optimality Results for Multiprocessor Real-Time Locking”. In: *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS*. IEEE Computer Society, 2010, pp. 49–60. DOI: `10.1109/RTSS.2010.17`. URL: `https://doi.org/10.1109/RTSS.2010.17` (Cited on pages 39 sq.).

[BA11] B. B. Brandenburg and J. H. Anderson. “Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks”. In: *Proceedings of the 11th International Conference on Embedded Software, EMSOFT*. Ed. by S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister. ACM, 2011, pp. 69–78. DOI: `10.1145/2038642.2038655`. URL: `https://doi.org/10.1145/2038642.2038655` (Cited on page 40).

[Bae11] O. K. Baek. *Data-centric distributed computing*. US Patent 8,060,464. 2011 (Cited on page 46).

[Bak91] T. P. Baker. “Stack-based Scheduling of Realtime Processes”. In: *Real Time Syst.* 3.1 (1991), pp. 67–99. DOI: `10.1007/BF00365393`. URL: `https://doi.org/10.1007/BF00365393` (Cited on pages 2, 39).

[Bar15] S. K. Baruah. “The federated scheduling of systems of conditional sporadic DAG tasks”. In: *2015 International Conference on Embedded Software, EMSOFT*. Ed. by A. Girault and N. Guan. IEEE, 2015, pp. 1–10. DOI: `10.1109/EMSOFT.2015.7318254`. URL: `https://doi.org/10.1109/EMSOFT.2015.7318254` (Cited on page 64).

[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` (Cited on page 8).

[BB12] J. Bergstra and Y. Bengio. “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305. DOI: `10.5555/2503308.2188395`. URL: `https://dl.acm.org/doi/10.5555/2503308.2188395` (Cited on page 45).

[BB20] S. Bozhko and B. B. Brandenburg. “Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle (Artifact)”. In: *Dagstuhl Artifacts Ser.* 6.1 (2020), 03:1–03:2. DOI: 10.4230/DARTS.6.1.3. URL: https://doi.org/10.4230/DARTS.6.1.3 (Cited on page 44).

[BBB+21] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. “The dogged pursuit of bug-free C programs: the Frama-C software analysis platform”. In: *Commun. ACM* 64.8 (2021), pp. 56–68. DOI: 10.1145/3470569. URL: https://doi.org/10.1145/3470569 (Cited on page 36).

[BBC+21] P. Baudin, F. Bobot, L. Correnson, Z. Dargaye, and A. Blanchard. *WP Plug-in Manual.* version 22.0 (Titanium). 2021. URL: https://frama-c.com/download/wp-manual-22.0-Titanium.pdf (Cited on pages 36, 134).

[BBL01] G. Bernat, A. Burns, and A. Llamosí́. “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 (Cited on page 47).

[BBW16] A. Biondi, B. B. Brandenburg, and A. Wieder. “A Blocking Bound for Nested FIFO Spin Locks”. In: *2016 IEEE Real-Time Systems Symposium, RTSS.* IEEE Computer Society, 2016, pp. 291–302. DOI: 10.1109/RTSS.2016.036. URL: https://doi.org/10.1109/RTSS.2016.036 (Cited on page 40).

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5. URL: https://doi.org/10.1007/978-3-662-07964-5 (Cited on page 43).

[BCC+18] S. Buschjäger, K.-H. Chen, J.-J. Chen, and K. Morik. “Realization of Random Forest for Real-Time Evaluation through Tree Framing”. In: *IEEE International Conference on Data Mining, ICDM.* IEEE Computer Society, 2018, pp. 19–28. DOI: 10.1109/ICDM.2018.00017. URL: https://doi.org/10.1109/ICDM.2018.00017 (Cited on page 7).

[BCF+21] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language.* version 1.16 for Frama-C 22.0. 2021. URL: https://frama-c.com/download/acsl-implementation-22.0-Titanium.pdf (visited on 08/09/2023) (Cited on pages 133 sq.).

[BCH+16] G. von der Brüggen, K.-H. Chen, W.-H. Huang, and J.-J. Chen. “Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments”. In: *2016 IEEE Real-Time Systems Symposium, RTSS.* IEEE Computer Society, 2016, pp. 303–314. DOI: 10.1109/RTSS.2016.037. URL: https://doi.org/10.1109/RTSS.2016.037 (Cited on page 48).

[BCH+17] G. von der Brüggen, J.-J. Chen, W.-H. Huang, and M. Yang. “Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems”. In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS.* Ed. by E. Bini and C. Pagetti. ACM, 2017, pp. 287–296. DOI: 10.1145/3139258.3139287. URL: https://doi.org/10.1145/3139258.3139287 (Cited on pages 40, 101, 116).

[BCP+16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. “OpenAI Gym”. In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540 (Cited on page 190).

[BD99] J. A. Blackard and D. J. Dean. “Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables”. In: *Computers and electronics in agriculture* 24.3 (1999), pp. 131–151 (Cited on page 165).

[BFM+11] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages.* 2011, pp. 53–64 (Cited on page 134).

[BG16] B. B. Brandenburg and M. Gul. “Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations”. In: *IEEE Real-Time Systems Symposium, RTSS.* IEEE Computer Society, 2016, pp. 99–110. DOI: 10.1109/RTSS.2016.019. URL: https://doi.org/10.1109/RTSS.2016.019 (Cited on page 113).

[BKJ+20] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy. “BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS, virtual.* Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.-F. Balcan, and H.-T. Lin. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/f5b1b89d98b7286673128a5fb112cb9a-Abstract.html (Cited on pages 158, 164).

[BKL18] A. Blanchard, N. Kosmatov, and F. Loulergue. “A Lesson on Verification of IoT Software with Frama-C”. In: *2018 International Conference on High Performance Computing & Simulation, HPCS.* IEEE, 2018, pp. 21–30. DOI: 10.1109/HPCS.2018.00018. URL: https://doi.org/10.1109/HPCS.2018.00018 (Cited on page 36).

[Bla21] A. Blanchard. *Introduction to C program proof with Frama-C and its WP plugin.* 2021. URL: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf (visited on 08/09/2023) (Cited on page 133).

[BLB+07] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. “A Flexible Real-Time Locking Protocol for Multiprocessors”. In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA.* IEEE Computer Society, 2007, pp. 47–56. DOI: 10.1109/RTCSA.2007.8. URL: https://doi.org/10.1109/RTCSA.2007.8 (Cited on pages 39, 102).

[BLL+83] K. R. Baker, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. “Preemptive Scheduling of a Single Machine to Minimize Maximum Cost Subject to Release Dates and Precedence Constraints”. In: *Oper. Res.* 31.2 (1983), pp. 381–386. DOI: 10.1287/opre.31.2.381. URL: https://doi.org/10.1287/opre.31.2.381 (Cited on pages 89, 101).

[BNG+18] J. Berk, V. Nguyen, S. Gupta, S. Rana, and S. Venkatesh. "Exploration Enhanced Expected Improvement for Bayesian Optimization". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD, Proceedings, Part II.* Ed. by M. Berlingerio, F. Bonchi, T. Gärtner, N. Hurley, and G. Ifrim. Vol. 11052. Lecture Notes in Computer Science. Springer, 2018, pp. 621–637. DOI: 10.1007/978-3-030-10928-8_37. URL: https://doi.org/10.1007/978-3-030-10928-8%5C_37 (Cited on page 45).

[BQL+21] Z. Bo, Y. Qiao, C. Leng, H. Wang, C. Guo, and S. Zhang. "Developing Real-Time Scheduling Policy by Deep Reinforcement Learning". In: *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS.* IEEE, 2021, pp. 131–142. DOI: 10.1109/RTAS52030.2021.00019. URL: https://doi.org/10.1109/RTAS52030.2021.00019 (Cited on pages 48, 202).

[Bra11] B. B. Brandenburg. "Scheduling and locking in multiprocessor real-time operating systems". PhD thesis. University of North Carolina, Chapel Hill, USA, 2011. DOI: 10.17615/x1zq-v169. URL: https://doi.org/10.17615/x1zq-v169 (Cited on pages 5, 42).

[Bra13] B. B. Brandenburg. "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling". In: *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS.* IEEE Computer Society, 2013, pp. 141–152. DOI: 10.1109/RTAS.2013.6531087. URL: https://doi.org/10.1109/RTAS.2013.6531087 (Cited on page 102).

[BRB+17] B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang. "mlrMBO: A modular framework for model-based optimization of expensive black-box functions". In: *arXiv preprint arXiv:1703.03373* (2017) (Cited on page 45).

[BS15] T. Bund and F. Slomka. "Sensitivity Analysis of Dropped Samples for Performance-Oriented Controller Design". In: *IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC.* IEEE Computer Society, 2015, pp. 244–251. DOI: 10.1109/ISORC.2015.16. URL: https://doi.org/10.1109/ISORC.2015.16 (Cited on pages 46 sq.).

[BS18] A. Biondi and Y. 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. DOI: 10.1007/s11241-018-9303-1. URL: https://doi.org/10.1007/s11241-018-9303-1 (Cited on page 113).

[BUC+17] G. von der Brüggen, N. Ueter, J.-J. Chen, and M. Freier. "Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS.* Ed. by E. Bini and C. Pagetti. ACM, 2017, pp. 108–117. DOI: 10.1145/3139258.3139273. URL: https://doi.org/10.1145/3139258.3139273 (Cited on page 100).

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition.* Vol. 24. Real-Time Systems Series. Springer, 2011. ISBN: 978-1-4614-0675-4. DOI: 10.1007/978-1-4614-0676-1. URL: https://doi.org/10.1007/978-1-4614-0676-1 (Cited on page 137).

[BVB+22] K. Bedarkar, M. Vardishvili, S. Bozhko, M. Maida, and B. B. Brandenburg. “From Intuition to Coq: A Case Study in Verified Response-Time Analysis 1 of FIFO Scheduling”. In: *IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2022, pp. 197–210. DOI: 10.1109/RTSS55097.2022.00026. URL: https://doi.org/10.1109/RTSS55097.2022.00026 (Cited on page 44).

[BW09] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages - Ada, Real-Time Java and C / Real-Time POSIX, Fourth Edition.* International computer science series. Addison-Wesley, 2009. ISBN: 978-0-321-41745-9 (Cited on pages 39, 137, 140).

[BW13] A. Burns and A. J. Wellings. “A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP”. In: *25th Euromicro Conference on Real-Time Systems, ECRTS*. IEEE Computer Society, 2013, pp. 282–291. DOI: 10.1109/ECRTS.2013.37. URL: https://doi.org/10.1109/ECRTS.2013.37 (Cited on pages 3, 39 sq., 143 sq.).

[BXF+18] J. Bian, H. Xiong, Y. Fu, and S. K. Das. “CSWA: Aggregation-Free Spatial-Temporal Community Sensing”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press, 2018, pp. 2087–2094. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16281 (Cited on pages 12, 159).

[BYC13] J. Bergstra, D. Yamins, and D. D. Cox. “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures”. In: *Proceedings of the 30th International Conference on Machine Learning, ICML*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 115–123. URL: http://proceedings.mlr.press/v28/bergstra13.html (Cited on page 45).

[Car82] J. Carlier. “The one-machine sequencing problem”. In: *European Journal of Operational Research* 11.1 (1982), pp. 42–47 (Cited on page 61).

[CBC+16] K.-H. Chen, B. Bönninghoff, J.-J. Chen, and P. Marwedel. “Compensate or ignore? meeting control robustness requirements through adaptive soft-error handling”. In: *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES*. Ed. by T.-W. Kuo and D. B. Whalley. ACM, 2016, pp. 82–91. DOI: 10.1145/2907950.2907952. URL: https://doi.org/10.1145/2907950.2907952 (Cited on pages 8 sq., 47, 174, 189 sq., 194).

[CBC18] K.-H. Chen, G. von der Brüggen, and J.-J. Chen. “Reliability Optimization on Multi-Core Systems with Multi-Tasking and Redundant Multi-Threading”. In: *IEEE Trans. Computers* 67.4 (2018), pp. 484–497. DOI: 10.1109/TC.2017.2769044. URL: https://doi.org/10.1109/TC.2017.2769044 (Cited on page 8).

[CBH+15] S. Catellani, L. Bonato, S. Huber, and E. Mezzetti. “Challenges in the Implementation of MrsP”. In: *Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Proceedings*. Ed. by J. A. de la Puente and T. Vardanega. Vol. 9111. Lecture

Notes in Computer Science. Springer, 2015, pp. 179–195. DOI: 10.1007/978-3-319-19584-1_12. URL: https://doi.org/10.1007/978-3-319-19584-1%5C_12 (Cited on pages 5, 35, 42, 131, 144).

[CBH+17] J.-J. Chen, G. von der Brüggen, W.-H. Huang, and R. I. Davis. "On the Pitfalls of Resource Augmentation Factors and Utilization Bounds in Real-Time Scheduling". In: *29th Euromicro Conference on Real-Time Systems, ECRTS*. Ed. by M. Bertogna. Vol. 76. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 9:1–9:25. DOI: 10.4230/LIPIcs.ECRTS.2017.9. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2017.9 (Cited on page 31).

[CBS+18] J.-J. Chen, G. von der Brüggen, J. Shi, and N. Ueter. "Dependency Graph Approach for Multiprocessor Real-Time Synchronization". In: *39th IEEE Real-Time Systems Symposium, RTSS*. 2018, pp. 434–446 (Cited on pages 10, 16 sq.).

[CCC+18] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook. "Continuous Formal Verification of Amazon s2n". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC, Proceedings, Part II*. Ed. by H. Chockler and G. Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 430–446. DOI: 10.1007/978-3-319-96142-2_26. URL: https://doi.org/10.1007/978-3-319-96142-2%5C_26 (Cited on page 43).

[CCK+21] L. Correnson, P. Cuoq, F. Kirchner, A. Maroneze, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. *Frama-C User Manual*. version 22.0 (Titanium). 2021. URL: https://frama-c.com/download/user-manual-22.0-Titanium.pdf (visited on 08/09/2023) (Cited on pages 36, 134, 136).

[CDH+09] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs, Proceedings*. Ed. by S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2. URL: https://doi.org/10.1007/978-3-642-03359-9%5C_2 (Cited on page 42).

[CGS13] S. Chaki, A. Gurfinkel, and O. Strichman. "Verifying periodic programs with priority inheritance locks". In: *Formal Methods in Computer-Aided Design, FMCAD*. IEEE, 2013, pp. 137–144. URL: https://ieeexplore.ieee.org/document/6679402/ (Cited on page 44).

[CH+08] G. Claeskens, N. L. Hjort, et al. *Model selection and model averaging*. Vol. 330. Cambridge University Press Cambridge, 2008 (Cited on page 46).

[CH09] Y.-B. Chan and P. Hall. "Scale adjustments for classifiers in high-dimensional, low sample size settings". In: *Biometrika* 96.2 (2009), pp. 469–478 (Cited on page 169).

[Che16] J.-J. Chen. "Federated scheduling admits no constant speedup factors for constrained-deadline DAG task systems". In: *Real Time Syst.* 52.6 (2016), pp. 833–838. DOI: 10.1007/s11241-016-9255-2. URL: https://doi.org/10.1007/s11241-016-9255-2 (Cited on page 64).

[Che18] Z. Chen. *SET-MRTS: Schedulability Experimental Tools for Multiprocessors Real Time Systems*. 2018. URL: https://github.com/RTLAB-UESTC/SET-MRTS-public (visited on 08/01/2023) (Cited on page 101).

[CK07] J.-J. Chen and C.-F. Kuo. "Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms". In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA*. IEEE Computer Society, 2007, pp. 28–38. DOI: 10.1109/RTCSA.2007.37. URL: https://doi.org/10.1109/RTCSA.2007.37 (Cited on page 8).

[CKK+12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C - A Software Analysis Perspective". In: *Software Engineering and Formal Methods - 10th International Conference, SEFM, Proceedings*. Ed. by G. Eleftherakis, M. Hinchey, and M. Holcombe. Vol. 7504. Lecture Notes in Computer Science. Springer, 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16. URL: https://doi.org/10.1007/978-3-642-33826-7%5C_16 (Cited on page 134).

[CKZ19] H. Choi, H. Kim, and Q. Zhu. "Job-Class-Level Fixed Priority Scheduling of Weakly-Hard Real-Time Systems". In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. Ed. by B. B. Brandenburg. IEEE, 2019, pp. 241–253. DOI: 10.1109/RTAS.2019.00028. URL: https://doi.org/10.1109/RTAS.2019.00028 (Cited on page 47).

[CLB+06] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. "LITMUS$^{RT}$ : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers". In: *Proceedings of the 27th IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 2006, pp. 111–126. DOI: 10.1109/RTSS.2006.27. URL: https://doi.org/10.1109/RTSS.2006.27 (Cited on page 5).

[Cor14] L. Correnson. "Qed. Computing What Remains to Be Proved". In: *NASA Formal Methods - 6th International Symposium, NFM, Proceedings*. Ed. by J. M. Badger and K. Y. Rozier. Vol. 8430. Lecture Notes in Computer Science. Springer, 2014, pp. 215–229. DOI: 10.1007/978-3-319-06200-6_17. URL: https://doi.org/10.1007/978-3-319-06200-6%5C_17 (Cited on page 134).

[CPW98] B. Chen, C. N. Potts, and G. J. Woeginger. "A review of machine scheduling: Complexity, algorithms and approximability". In: *Handbook of Combinatorial Optimization: Volume1–3* (1998), pp. 1493–1641 (Cited on page 75).

[CRE+20] M. A. R. Coy, F. Rehbach, A. E. Eiben, and T. Bartz-Beielstein. "Parallelized bayesian optimization for problems with expensive evaluation functions". In: *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume*. Ed. by C. A. C. Coello. ACM, 2020, pp. 231–232. DOI: 10.1145/3377929.3390017. URL: https://doi.org/10.1145/3377929.3390017 (Cited on page 45).

[CSB+22] J.-J. Chen, J. Shi, G. von der Brüggen, and N. Ueter. "Scheduling of Real-Time Tasks With Multiple Critical Sections in Multiprocessor Systems". In: *IEEE Trans. Computers* 71.1 (2022), pp. 146–160 (Cited on pages 10, 16 sq.).

[CSB16] F. Cerqueira, F. Stutz, and B. B. Brandenburg. “PROSA: A Case for Readable Mechanized Schedulability Analysis”. In: *28th Euromicro Conference on Real-Time Systems, ECRTS*. IEEE Computer Society, 2016, pp. 273–284. DOI: 10.1109/ECRTS.2016.28. URL: https://doi.org/10.1109/ECRTS.2016.28 (Cited on page 44).

[CTB94] C.-M. Chen, S. K. Tripathi, and A. Blackmore. “A Resource Synchronization Protocol for Multiprocessor Real-Time Systems”. In: *Proceedings of the 1994 International Conference on Parallel Processing. Volume I: Algorithms & Applications*. Ed. by J. Chandra. CRC Press, 1994, pp. 159–162. DOI: 10.1109/ICPP.1994.44. URL: https://doi.org/10.1109/ICPP.1994.44 (Cited on page 40).

[DB11] R. I. Davis and A. Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Comput. Surv.* 43.4 (2011), 35:1–35:44. DOI: 10.1145/1978802.1978814. URL: https://doi.org/10.1145/1978802.1978814 (Cited on page 28).

[DGK+21] K. Dole, A. Gupta, J. Komp, S. Krishna, and A. Trivedi. “Event-Triggered and Time-Triggered Duration Calculus for Model-Free Reinforcement Learning”. In: *42nd IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2021, pp. 240–252. DOI: 10.1109/RTSS52674.2021.00031. URL: https://doi.org/10.1109/RTSS52674.2021.00031 (Cited on page 48).

[EA09] A. Easwaran and B. Andersson. “Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling”. In: *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS*. Ed. by T. P. Baker. IEEE Computer Society, 2009, pp. 377–386. DOI: 10.1109/RTSS.2009.37. URL: https://doi.org/10.1109/RTSS.2009.37 (Cited on pages 39, 102).

[Egi22] C.-C. von Egidy. *The source code of this work*. 2022. URL: https://github.com/tu-dortmund-ls12-rt/Resource-Synchronization-Protocols-Verification-RTEMS (visited on 04/02/2022) (Cited on pages 145, 152).

[EMK18] D. Efremov, M. U. Mandrykin, and A. V. Khoroshilov. “Deductive Verification of Unmodified Linux Kernel Library Functions”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA, Proceedings, Part II*. Ed. by T. Margaria and B. Steffen. Vol. 11245. Lecture Notes in Computer Science. Springer, 2018, pp. 216–234. DOI: 10.1007/978-3-030-03421-4_15. URL: https://doi.org/10.1007/978-3-030-03421-4%5C_15 (Cited on page 42).

[Evi21] Evidence Srl. *Erika Enterprise: a royalty free automotive OSEK/VDX certified Hard Real Time Operating System*. 2021. URL: https://www.erika-enterprise.com/ (visited on 04/02/2022) (Cited on page 42).

[FGM+19] P. Fradet, X. Guo, J.-F. Monin, and S. Quinton. “CertiCAN: A Tool for the Coq Certification of CAN Analysis Results”. In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. Ed. by B. B. Brandenburg. IEEE, 2019, pp. 182–191. DOI: 10.1109/RTAS.2019.00023. URL: https://doi.org/10.1109/RTAS.2019.00023 (Cited on page 44).

[FLC10] D. Faggioli, G. Lipari, and T. Cucinotta. "The Multiprocessor Bandwidth Inheritance Protocol". In: *22nd Euromicro Conference on Real-Time Systems, ECRTS*. IEEE Computer Society, 2010, pp. 90–99. DOI: 10.1109/ECRTS.2010.19. URL: https://doi.org/10.1109/ECRTS.2010.19 (Cited on page 40).

[FLC12] D. Faggioli, G. Lipari, and T. Cucinotta. "Analysis and implementation of the multiprocessor bandwidth inheritance protocol". In: *Real Time Syst.* 48.6 (2012), pp. 789–825. DOI: 10.1007/s11241-012-9162-0. URL: https://doi.org/10.1007/s11241-012-9162-0 (Cited on page 40).

[FLM+18] P. Fradet, M. Lesourd, J.-F. Monin, and S. Quinton. "A Generic Coq Proof of Typical Worst-Case Analysis". In: *2018 IEEE Real-Time Systems Symposium, RTSS*. IEEE Computer Society, 2018, pp. 218–229. DOI: 10.1109/RTSS.2018.00039. URL: https://doi.org/10.1109/RTSS.2018.00039 (Cited on page 44).

[Flo93] R. W. Floyd. "Assigning meanings to programs". In: *Program Verification: Fundamental Issues in Computer Science*. Springer, 1993, pp. 65–81 (Cited on page 117).

[FRA23] FRAMA-C. *Frama-C software analyzers*. 2023. URL: https://frama-c.com/ (visited on 04/02/2022) (Cited on pages 36, 133).

[GAB16] S. Gadia, C. Artho, and G. Bloom. "Verifying Nested Lock Priority Inheritance in RTEMS with Java Pathfinder". In: *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM, Proceedings*. Ed. by K. Ogata, M. Lawford, and S. Liu. Vol. 10009. Lecture Notes in Computer Science. 2016, pp. 417–432. DOI: 10.1007/978-3-319-47846-3_26. URL: https://doi.org/10.1007/978-3-319-47846-3%5C_26 (Cited on page 43).

[Gar19] P.-L. Garoche. *Formal verification of control system software*. Vol. 67. Princeton University Press, 2019 (Cited on page 132).

[GBD] D. Griffin, I. Bate, and R. I. Davis. *dgdguk/drs*. DOI: 10.5281/zenodo.4118058. URL: https://doi.org/10.5281/zenodo.4118058 (Cited on page 32).

[GBD20] D. Griffin, I. Bate, and R. I. Davis. "Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests". In: *41st IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2020, pp. 76–88. DOI: 10.1109/RTSS49844.2020.00018. URL: https://doi.org/10.1109/RTSS49844.2020.00018 (Cited on pages 32, 193).

[GD98] M. W. Gardner and S. Dorling. "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences". In: *Atmospheric environment* 32.14-15 (1998), pp. 2627–2636 (Cited on pages 6, 155, 165).

[GDD19] S. Ghosh, S. Dey, and P. Dasgupta. "Synthesizing Performance-Aware (m, k)-Firm Control Execution Patterns Under Dropped Samples". In: *32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems, VLSID*. IEEE, 2019, pp. 1–6. DOI: 10.1109/VLSID.2019.00019. URL: https://doi.org/10.1109/VLSID.2019.00019 (Cited on page 46).

[GDO+12] Y. Gu, H. M. Do, Y. Ou, and W. Sheng. “Human gesture recognition through a Kinect sensor”. In: *2012 IEEE International Conference on Robotics and Biomimetics, ROBIO*. IEEE, 2012, pp. 1379–1384. DOI: 10.1109/ROBIO.2012.6491161. URL: https://doi.org/10.1109/ROBIO.2012.6491161 (Cited on pages 12, 159).

[GG16] Y. Gal and Z. Ghahramani. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML*. Ed. by M.-F. Balcan and K. Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1050–1059. URL: http://proceedings.mlr.press/v48/gal16.html (Cited on page 45).

[GGB13] Y. Gao, S. K. Gupta, and M. A. Breuer. “Using explicit output comparisons for fault tolerant scheduling (FTS) on modern high-performance processors”. In: *Design, Automation and Test in Europe, DATE*. ACM, 2013, pp. 927–932. DOI: 10.7873/DATE.2013.195. URL: https://doi.org/10.7873/DATE.2013.195 (Cited on pages 8, 47).

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7 (Cited on pages 41, 72).

[GLC10] D. Ginsbourger, R. Le Riche, and L. Carraro. “Kriging is well-suited to parallelize optimization”. In: *Computational intelligence in expensive optimization problems*. Springer, 2010, pp. 131–162 (Cited on pages 45, 158).

[GLN01] P. Gai, G. Lipari, and M. D. Natale. “Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip”. In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 2001, pp. 73–83. DOI: 10.1109/REAL.2001.990598. URL: https://doi.org/10.1109/REAL.2001.990598 (Cited on pages 3, 39, 102).

[Goo23] Google Developers. *Google OR-Tools*. 2023. URL: https://developers.google.com/optimization/ (visited on 07/01/2023) (Cited on pages 75, 101).

[Gra11] A. Graves. “Practical Variational Inference for Neural Networks”. In: *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems, Proceedings*. Ed. by J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger. 2011, pp. 2348–2356. URL: https://proceedings.neurips.cc/paper/2011/hash/7eb3c8be3d411e8ebfab08eba5f49632-Abstract.html (Cited on page 45).

[Gra69] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal of Applied Mathematics* 17.2 (1969), pp. 416–429 (Cited on pages 29, 63 sq., 75, 79, 89, 99).

[GRT21] X. Guo, L. Rieg, and P. Torrini. “A generic approach for the certified schedulability analysis of software systems”. In: *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*. IEEE, 2021, pp. 83–92. DOI: 10.1109/RTCSA52859.2021.00018. URL: https://doi.org/10.1109/RTCSA52859.2021.00018 (Cited on page 44).

[GS20] G. Grimmett and D. Stirzaker. *Probability and random processes*. Oxford university press, 2020 (Cited on page 183).

[GSC+16] R. Gu, Z. Shao, H. Chen, X. ( Wu, J. Kim, V. Sjöberg, and D. Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. Ed. by K. Keeton and T. Roscoe. USENIX Association, 2016, pp. 653–669. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu (Cited on page 43).

[GSD20] A. Garg, A. K. Saha, and D. Dutta. "Direct Federated Neural Architecture Search". In: *CoRR* abs/2010.06223 (2020). arXiv: 2010.06223. URL: https://arxiv.org/abs/2010.06223 (Cited on page 46).

[GZB+17] J. Garrido, S. Zhao, A. Burns, and A. J. Wellings. "Supporting Nested Resources in MrsP". In: *Reliable Software Technologies - Ada-Europe 2017 - 22nd Ada-Europe International Conference on Reliable Software Technologies, Proceedings*. Ed. by J. Blieberger and M. Bader. Vol. 10300. Lecture Notes in Computer Science. Springer, 2017, pp. 73–86. DOI: 10.1007/978-3-319-60588-3_5. URL: https://doi.org/10.1007/978-3-319-60588-3%5C_5 (Cited on pages 40, 144, 148).

[HAA20] C. He, M. Annavaram, and S. Avestimehr. "FedNAS: Federated Deep Learning via Neural Architecture Search". In: *CoRR* abs/2004.08546 (2020). arXiv: 2004.08546. URL: https://arxiv.org/abs/2004.08546 (Cited on page 46).

[Har19] Hardkernel co., Ltd. *ODROID-N2*. 2019. URL: https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram/ (visited on 10/25/2019) (Cited on page 35).

[HAZ17] M. A. Haque, H. Aydin, and D. 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. DOI: 10.1109/TPDS.2016.2600595. URL: https://doi.org/10.1109/TPDS.2016.2600595 (Cited on page 31).

[HDK+17] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. "Communication Centric Design in Complex Automotive Embedded Systems". In: *29th Euromicro Conference on Real-Time Systems, ECRTS*. Ed. by M. Bertogna. Vol. 76. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 10:1–10:20. DOI: 10.4230/LIPIcs.ECRTS.2017.10. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2017.10 (Cited on page 100).

[HDZ+21] M. Hu, J. Ding, M. Zhang, F. Mallet, and M. Chen. "Enumeration and Deduction Driven Co-Synthesis of CCSL Specifications using Reinforcement Learning". In: *42nd IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2021, pp. 227–239. DOI: 10.1109/RTSS52674.2021.00030. URL: https://doi.org/10.1109/RTSS52674.2021.00030 (Cited on page 48).

[HHL11] F. Hutter, H. H. Hoos, and K. Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Learning and Intelligent Optimization - 5th International Conference, LION, Selected Papers*. Ed. by C. A. C. Coello. Vol. 6683. Lecture Notes in Computer Science. Springer, 2011, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40. URL: https://doi.org/10.1007/978-3-642-25566-3%5C_40 (Cited on page 45).

[HHL12] F. Hutter, H. H. Hoos, and K. Leyton-Brown. "Parallel Algorithm Configuration". In: *Learning and Intelligent Optimization - 6th International Conference, LION, Revised Selected Papers*. Ed. by Y. Hamadi and M. Schoenauer. Vol. 7219. Lecture Notes in Computer Science. Springer, 2012, pp. 55–70. DOI: 10.1007/978-3-642-34413-8_5. URL: https://doi.org/10.1007/978-3-642-34413-8%5C_5 (Cited on page 45).

[HHL13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. "An evaluation of sequential model-based optimization for expensive blackbox functions". In: *Genetic and Evolutionary Computation Conference, GECCO '13, Companion Material Proceedings*. Ed. by C. Blum and E. Alba. ACM, 2013, pp. 1209–1216. DOI: 10.1145/2464576.2501592. URL: https://doi.org/10.1145/2464576.2501592 (Cited on page 45).

[HLK11] P.-C. Hsiu, D.-N. Lee, and T.-W. Kuo. "Task synchronization and allocation for many-core real-time systems". In: *Proceedings of the 11th International Conference on Embedded Software, EMSOFT*. Ed. by S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister. ACM, 2011, pp. 79–88. DOI: 10.1145/2038642.2038656. URL: https://doi.org/10.1145/2038642.2038656 (Cited on page 40).

[HO01] N. Hansen and A. Ostermeier. "Completely Derandomized Self-Adaptation in Evolution Strategies". In: *Evol. Comput.* 9.2 (2001), pp. 159–195. DOI: 10.1162/106365601750190398. URL: https://doi.org/10.1162/106365601750190398 (Cited on page 45).

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (Cited on page 117).

[Hop71] J. Hopcroft. "An n log n algorithm for minimizing states in a finite automaton". In: *Theory of machines and computations*. Elsevier, 1971, pp. 189–196 (Cited on page 179).

[HQE20] Z. A. H. Hammadeh, S. Quinton, and R. 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. DOI: 10.1145/3356865. URL: https://doi.org/10.1145/3356865 (Cited on page 47).

[HS92] L. A. Hall and D. B. Shmoys. "Jackson's Rule for Single-Machine Scheduling: Making a Good Heuristic Better". In: *Math. Oper. Res.* 17.1 (1992), pp. 22–35. DOI: 10.1287/moor.17.1.22. URL: https://doi.org/10.1287/moor.17.1.22 (Cited on pages 61 sqq., 87).

[HSJ08] E. Henriksson, H. Sandberg, and K. H. Johansson. "Predictive compensation for communication outages in networked control systems". In: *Proceedings of the 47th IEEE Conference on Decision and Control, CDC*. IEEE, 2008, pp. 2063–2068. DOI: 10.1109/CDC.2008.4739306. URL: https://doi.org/10.1109/CDC.2008.4739306 (Cited on pages 46 sq.).

[HTF09] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. ISBN: 9780387848570. DOI: 10.1007/978-0-387-84858-7. URL: https://doi.org/10.1007/978-0-387-84858-7 (Cited on page 168).

[HXW+20] C. Huang, S. Xu, Z. Wang, S. Lan, W. Li, and Q. Zhu. "Opportunistic Intermittent Control with Safety Guarantees for Autonomous Systems". In: *57th ACM/IEEE Design Automation Conference, DAC*. IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218742. URL: https://doi.org/10.1109/DAC18072.2020.9218742 (Cited on page 48).

[HYC16] W.-H. Huang, M. Yang, and J.-J. Chen. "Resource-Oriented Partitioned Scheduling in Multiprocessor Systems: How to Partition and How to Share?" In: *2016 IEEE Real-Time Systems Symposium, RTSS*. IEEE Computer Society, 2016, pp. 111–122. DOI: 10.1109/RTSS.2016.020. URL: https://doi.org/10.1109/RTSS.2016.020 (Cited on pages 40, 101, 116).

[Inr] Inria. *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (visited on 04/02/2022) (Cited on page 134).

[Jac55] J. R. Jackson. "Scheduling a production line to minimize maximum tardiness". In: (1955) (Cited on page 87).

[JRG+12] J. Janusevskis, R. L. Riche, D. Ginsbourger, and R. Girdziusas. "Expected Improvements for the Asynchronous Parallel Global Optimization of Expensive Functions: Potentials and Challenges". In: *Learning and Intelligent Optimization - 6th International Conference, LION, Revised Selected Papers*. Ed. by Y. Hamadi and M. Schoenauer. Vol. 7219. Lecture Notes in Computer Science. Springer, 2012, pp. 413–418. DOI: 10.1007/978-3-642-34413-8_37. URL: https://doi.org/10.1007/978-3-642-34413-8%5C_37 (Cited on pages 46, 163).

[JSW98] D. R. Jones, M. Schonlau, and W. J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: *J. Glob. Optim.* 13.4 (1998), pp. 455–492. DOI: 10.1023/A:1008306431147. URL: https://doi.org/10.1023/A:1008306431147 (Cited on pages 7, 45, 155).

[JWA15] C. E. Jarrett, B. C. Ward, and J. H. Anderson. "A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems". In: *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS*. Ed. by J. Forget. ACM, 2015, pp. 3–12. DOI: 10.1145/2834848.2834874. URL: https://doi.org/10.1145/2834848.2834874 (Cited on pages 40, 102).

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: formal verification of an OS kernel." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP*. Ed. by J. N. Matthews and T. E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596 (Cited on page 43).

[KGC+12] P. Kumar, D. Goswami, S. Chakraborty, A. Annaswamy, K. Lampka, and L. Thiele. "A hybrid approach to cyber-physical systems verification". In: *The 49th Annual Design Automation Conference 2012, DAC*. Ed. by P. Groeneveld, D. Sciuto, and S. Hassoun. ACM, 2012, pp. 688–696. DOI: 10.1145/2228360.2228484. URL: https://doi.org/10.1145/2228360.2228484 (Cited on page 46).

[KIM79] H. Kise, T. Ibaraki, and H. Mine. "Performance analysis of six approximation algorithms for the one-machine maximum lateness scheduling problem with ready times". In: *Journal of the Operations Research Society of Japan* 22.3 (1979), pp. 205–224. DOI: 10.15807/jorsj.22.205 (Cited on pages 62, 101).

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7. URL: https://doi.org/10.1007/s00165-014-0326-7 (Cited on page 36).

[KMY+16] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. "Federated Learning: Strategies for Improving Communication Efficiency". In: *CoRR* abs/1610.05492 (2016). arXiv: 1610.05492. URL: http://arxiv.org/abs/1610.05492 (Cited on page 46).

[KN03] Y. Karuno and H. Nagamochi. "A Better Approximation for the Two-Machine Flowshop Scheduling Problem with Time Lags". In: *Algorithms and Computation, 14th International Symposium, ISAAC, Proceedings*. Ed. by T. Ibaraki, N. Katoh, and H. Ono. Vol. 2906. Lecture Notes in Computer Science. Springer, 2003, pp. 309–318. DOI: 10.1007/978-3-540-24587-2_33. URL: https://doi.org/10.1007/978-3-540-24587-2%5C_33 (Cited on page 76).

[Kot18] H. Kotthaus. "Methods for efficient resource utilization in statistical machine learning algorithms". PhD thesis. Technical University of Dortmund, Germany, 2018. URL: https://hdl.handle.net/2003/36929 (Cited on page 156).

[KP00] B. Kalyanasundaram and K. Pruhs. "Speed is as powerful as clairvoyance". In: *J. ACM* 47.4 (2000), pp. 617–643. DOI: 10.1145/347476.347479. URL: https://doi.org/10.1145/347476.347479 (Cited on page 30).

[KRL+17] H. Kotthaus, J. Richter, A. Lang, J. Thomas, B. Bischl, P. Marwedel, J. Rahnenführer, and M. Lang. "RAMBO: Resource-Aware Model-Based Optimization with Scheduling for Heterogeneous Runtimes and a Comparison with Asynchronous Model-Based Optimization". In: *Learning and Intelligent Optimization - 11th International Conference, LION, Revised Selected Papers*. Ed. by R. Battiti, D. E. Kvasov, and Y. D. Sergeyev. Vol. 10556. Lecture Notes in Computer Science. Springer, 2017, pp. 180–195. DOI: 10.1007/978-3-319-69404-7_13. URL: https://doi.org/10.1007/978-3-319-69404-7%5C_13 (Cited on page 156).

[KS95] G. Koren and D. E. Shasha. "An Approach To Handling Overloaded Systems That Allow Skips". In: *16th IEEE Real-Time Systems Symposium, Proceedings*. IEEE Computer Society, 1995, pp. 110–119. DOI: 10.1109/REAL.1995.495201. URL: https://doi.org/10.1109/REAL.1995.495201 (Cited on pages 8, 47, 174, 189).

[KSL+19] H. Kotthaus, L. Schönberger, A. Lang, J.-J. Chen, and P. Marwedel. "Can Flexible Multi-Core Scheduling Help to Execute Machine Learning Algorithms Resource-Efficiently?" In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems, SCOPES*. Ed. by S. Stuijk. ACM, 2019, pp. 59–62. DOI: 10.1145/3323439.3323986. URL: https://doi.org/10.1145/3323439.3323986 (Cited on pages 46, 156, 163).

[KSZ17] H.-P. Kriegel, E. Schubert, and A. Zimek. “The (black) art of runtime evaluation: Are we comparing algorithms or implementations?” In: *Knowl. Inf. Syst.* 52.2 (2017), pp. 341–378. DOI: 10.1007/s10115-016-1004-2. URL: https://doi.org/10.1007/s10115-016-1004-2 (Cited on page 7).

[KZH15] S. Kramer, D. Ziegenbein, and A. Hamann. “Real world automotive benchmarks for free”. In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS).* Vol. 130. 2015 (Cited on pages 100, 193).

[LAB+11] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. R. Pratt, M. Sokolsky, G. Stanek, D. M. Stavens, A. Teichman, M. Werling, and S. Thrun. “Towards fully autonomous driving: Systems and algorithms”. In: *IEEE Intelligent Vehicles Symposium (IV).* IEEE, 2011, pp. 163–168. DOI: 10.1109/IVS.2011.5940562. URL: https://doi.org/10.1109/IVS.2011.5940562 (Cited on pages 12, 161).

[LBO+12] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade - Second Edition.* Ed. by G. Montavon, G. B. Orr, and K.-R. Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, 2012, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8%5C_3 (Cited on page 45).

[LBW+21] S. Lee, H. Baek, H. Woo, K. G. Shin, and J. Lee. “ML for RT: Priority Assignment Using Machine Learning”. In: *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS.* IEEE, 2021, pp. 118–130. DOI: 10.1109/RTAS52030.2021.00018. URL: https://doi.org/10.1109/RTAS52030.2021.00018 (Cited on page 48).

[LCA+14] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah. “Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks”. In: *26th Euromicro Conference on Real-Time Systems, ECRTS.* IEEE Computer Society, 2014, pp. 85–96. DOI: 10.1109/ECRTS.2014.23. URL: https://doi.org/10.1109/ECRTS.2014.23 (Cited on pages 64, 94).

[LCB07] G. Loosli, S. Canu, and L. Bottou. “Training invariant support vector machines using selective sampling”. In: *Large scale kernel machines* 2.1 (2007), pp. 301–320. URL: http://leon.bottou.org/papers/loosli-canu-bottou-2006 (Cited on page 169).

[LCB98] Y. LeCun, C. Cortes, and C. J. Burges. “The MNIST database of handwritten digits, 1998”. In: *URL http://yann. lecun. com/exdb/mnist* (1998) (Cited on pages 164, 202).

[LDS+07] Y. Liu, R. P. Dick, L. Shang, and H. Yang. “Accurate temperature-dependent integrated circuit leakage power estimation is easy”. In: *2007 Design, Automation and Test in Europe Conference and Exposition, DATE.* EDA Consortium, 2007, pp. 1526–1531. URL: https://dl.acm.org/citation.cfm?id=1266701 (Cited on page 9).

[LGS+23] C.-C. Lin, M. Günzel, J. Shi, T. T. Seidl, K.-H. Chen, and J.-J. Chen. “Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies”. In: *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS.* IEEE, 2023, pp. 161–173. DOI: 10.1109/RTAS58335.2023.00020. URL: https://doi.org/10.1109/RTAS58335.2023.00020 (Cited on page 130).

[LLK+93] E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys. “Chapter 9 Sequencing and scheduling: Algorithms and complexity”. In: *Logistics of Production and Inventory*. Ed. by S. C. Graves, A. H. G. R. Kan, and P. H. Zipkin. Vol. 4. Handbooks in Operations Research and Management Science. North-Holland, 1993, pp. 445–522. DOI: 10.1016/s0927-0507(05)80189-6. URL: https://doi.org/10.1016/s0927-0507(05)80189-6 (Cited on page 75).

[LNR09] K. Lakshmanan, D. de Niz, and R. Rajkumar. “Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors”. In: *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS*. Ed. by T. P. Baker. IEEE Computer Society, 2009, pp. 469–478. DOI: 10.1109/RTSS.2009.51. URL: https://doi.org/10.1109/RTSS.2009.51 (Cited on page 40).

[LR79] J. Lenstra and A. Rinnooy Kan. “Computational complexity of discrete optimization problems”. In: *Ann. Discrete Math.* 4 (1979) (Cited on pages 11, 41, 72).

[LRB77] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. “Complexity of Machine Scheduling Problems”. In: *Annals of Discrete Mathematics* 1 (1977), pp. 343–362 (Cited on page 56).

[LS09] D. Leinenbach and T. Santen. “Verifying the Microsoft Hyper-V Hypervisor with VCC”. In: *FM 2009: Formal Methods, Second World Congress, Proceedings*. Ed. by A. Cavalcanti and D. Dams. Vol. 5850. Lecture Notes in Computer Science. Springer, 2009, pp. 806–809. DOI: 10.1007/978-3-642-05089-3_51. URL: https://doi.org/10.1007/978-3-642-05089-3%5C_51 (Cited on page 42).

[LW+02] A. Liaw, M. Wiener, et al. “Classification and regression by randomForest”. In: *R news* 2.3 (2002), pp. 18–22 (Cited on page 165).

[LWJ+20] H. Liang, Z. Wang, R. Jiao, and Q. Zhu. “Leveraging Weakly-hard Constraints for Improving System Fault Tolerance with Functional and Timing Guarantees”. In: *IEEE/ACM International Conference On Computer Aided Design, ICCAD*. IEEE, 2020, 101:1–101:9. DOI: 10.1145/3400302.3415717. URL: https://doi.org/10.1145/3400302.3415717 (Cited on page 47).

[LXG+19] L. Li, H. Xiong, Z. Guo, J. Wang, and C.-Z. Xu. “SmartPC: Hierarchical Pace Control in Real-Time Federated Learning System”. In: *IEEE Real-Time Systems Symposium, RTSS*. IEEE, 2019, pp. 406–418. DOI: 10.1109/RTSS46320.2019.00043. URL: https://doi.org/10.1109/RTSS46320.2019.00043 (Cited on page 46).

[MBB22] M. Maida, S. Bozhko, and B. B. Brandenburg. “Foundational Response-Time Analysis as Explainable Evidence of Timeliness (Artifact)”. In: *Dagstuhl Artifacts Ser.* 8.1 (2022), 07:1–07:2. DOI: 10.4230/DARTS.8.1.7. URL: https://doi.org/10.4230/DARTS.8.1.7 (Cited on page 44).

[MF75] G. McMahon and M. Florian. “On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness”. In: *Oper. Res.* 23.3 (1975), pp. 475–482. DOI: 10.1287/opre.23.3.475. URL: https://doi.org/10.1287/opre.23.3.475 (Cited on page 61).

[MG22] L. Miedema and C. Grelck. "Strategy Switching: Smart Fault-Tolerance for Weakly-Hard Resource-Constrained Real-Time Applications". In: *Software Engineering and Formal Methods - 20th International Conference, SEFM, Proceedings*. Ed. by B.-H. Schlingloff and M. Chai. Vol. 13550. Lecture Notes in Computer Science. Springer, 2022, pp. 129–145. DOI: 10.1007/978-3-031-17108-6_8. URL: https://doi.org/10.1007/978-3-031-17108-6%5C_8 (Cited on page 47).

[MGL+23] A. Matovic, R. Graczyk, F. Lucchetti, and M. Völp. "Consensual Resilient Control: Stateless Recovery of Stateful Controllers". In: *35th Euromicro Conference on Real-Time Systems, ECRTS*. Ed. by A. V. Papadopoulos. Vol. 262. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 14:1–14:27. DOI: 10.4230/LIPIcs.ECRTS.2023.14. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2023.14 (Cited on page 47).

[NAA18] C. E. Nemitz, T. Amert, and J. H. Anderson. "Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts (Artifact)". In: *Dagstuhl Artifacts Ser.* 4.2 (2018), 02:1–02:3. DOI: 10.4230/DARTS.4.2.2. URL: https://doi.org/10.4230/DARTS.4.2.2 (Cited on page 40).

[NAA19] C. E. Nemitz, T. Amert, and J. H. Anderson. "Real-time multiprocessor locks with nesting: optimizing the common case". In: *Real Time Syst.* 55.2 (2019), pp. 296–348. DOI: 10.1007/s11241-019-09328-w. URL: https://doi.org/10.1007/s11241-019-09328-w (Cited on page 40).

[NAG+19] C. E. Nemitz, T. Amert, M. Goyal, and J. H. Anderson. "Concurrency groups: a new way to look at real-time multiprocessor lock nesting". In: *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS*. Ed. by J. Ermont, Y.-Q. Song, and C. D. Gill. ACM, 2019, pp. 187–197. DOI: 10.1145/3356401.3356404. URL: https://doi.org/10.1145/3356401.3356404 (Cited on page 102).

[NAG+21] C. E. Nemitz, T. Amert, M. Goyal, and J. H. Anderson. "Concurrency groups: a new way to look at real-time multiprocessor lock nesting". In: *Real Time Syst.* 57.1-2 (2021), pp. 190–226. DOI: 10.1007/s11241-020-09361-0. URL: https://doi.org/10.1007/s11241-020-09361-0 (Cited on page 40).

[NBF+14] M. Nasri, S. K. Baruah, G. Fohler, and M. Kargahi. "On the Optimality of RM and EDF for Non-Preemptive Real-Time Harmonic Tasks". In: *22nd International Conference on Real-Time Networks and Systems, RTNS*. Ed. by M. Jan, B. B. Hedia, J. Goossens, and C. Maiza. ACM, 2014, p. 331. DOI: 10.1145/2659787.2659806. URL: https://doi.org/10.1145/2659787.2659806 (Cited on page 87).

[NF16] M. Nasri and G. Fohler. "Non-work-conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions". In: *28th Euromicro Conference on Real-Time Systems, ECRTS*. IEEE Computer Society, 2016, pp. 165–175. DOI: 10.1109/ECRTS.2016.11. URL: https://doi.org/10.1109/ECRTS.2016.11 (Cited on page 87).

[NK06] S. Nijssen and J. Kok. "Frequent subgraph miners: runtimes don\'t say everything". In: *Proceedings of the Workshop on Mining and Learning with Graphs*. 2006, pp. 173–180 (Cited on page 7).

[NK14] M. Nasri and M. Kargahi. "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks". In: *Real Time Syst.* 50.4 (2014), pp. 548–584. DOI: 10.1007/s11241-014-9203-y. URL: https://doi.org/10.1007/s11241-014-9203-y (Cited on page 87).

[NKT+23] P. R. Nikiema, A. Kritikakou, M. Traiola, and O. Sentieys. "Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead". In: *35th Euromicro Conference on Real-Time Systems, ECRTS.* Ed. by A. V. Papadopoulos. Vol. 262. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 15:1–15:22. DOI: 10.4230/LIPIcs.ECRTS.2023.15. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2023.15 (Cited on page 47).

[NLB+21] O. Nicole, M. Lemerre, S. Bardin, and X. Rival. "No Crash, No Exploit: Automated Verification of Embedded Kernels". In: *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS.* IEEE, 2021, pp. 27–39. DOI: 10.1109/RTAS52030.2021.00011. URL: https://doi.org/10.1109/RTAS52030.2021.00011 (Cited on page 43).

[NMR+02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Proceedings.* Ed. by R. N. Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 213–228. DOI: 10.1007/3-540-45937-5_16. URL: https://doi.org/10.1007/3-540-45937-5%5C_16 (Cited on pages 36, 134).

[NNB10] F. Nemati, T. Nolte, and M. Behnam. "Partitioning Real-Time Systems on Multiprocessors with Shared Resources". In: *Principles of Distributed Systems - 14th International Conference, OPODIS, Proceedings.* Ed. by C. Lu, T. Masuzawa, and M. Mosbah. Vol. 6490. Lecture Notes in Computer Science. Springer, 2010, pp. 253–269. DOI: 10.1007/978-3-642-17653-1_20. URL: https://doi.org/10.1007/978-3-642-17653-1%5C_20 (Cited on page 40).

[NQ06] L. Niu and G. Quan. "Energy minimization for real-time systems with (m, k)-guarantee". In: *IEEE Transactions on Very Large Scale Integration Systems* 14.7 (2006), pp. 717–729. DOI: 10.1109/TVLSI.2006.878337. URL: https://doi.org/10.1109/TVLSI.2006.878337 (Cited on pages 8 sq., 174, 190).

[NVI21] NVIDIA CORPORATION & AFFILIATES. *NVIDIA Jetson AGX Xavier Supported Modes and Power Efficiency.* 2021. URL: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3261/Tegra%20Linux%20Driver%20Package%20Development%20Guide/power_management_jetson_xavier.html#wwpID0E0X00HA (visited on 03/01/2022) (Cited on pages 35, 193).

[NZ20] L. Niu and D. Zhu. "Reliable and Energy-Aware Fixed-Priority (m, k)-Deadlines Enforcement with Standby-Sparing". In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE.* IEEE, 2020, pp. 424–429. DOI: 10.23919/DATE48585.2020.9116398. URL: https://doi.org/10.23919/DATE48585.2020.9116398 (Cited on pages 8 sq., 174).

[NZ86] E. Nowicki and S. Zdrzałka. “A note on minimizing maximum lateness in a one-machine sequencing problem with release dates”. In: *European journal of operational research* 23.2 (1986), pp. 266–267 (Cited on page 61).

[OCa20] OCamlPro SAS. *An SMT Solver For Software Verification*. 2020. URL: https://alt-ergo.ocamlpro.com/ (visited on 04/02/2022) (Cited on page 152).

[OSM02] N. Oh, P. P. Shirvani, and E. J. McCluskey. “Control-flow checking by software signatures”. In: *IEEE Trans. Reliab.* 51.1 (2002), pp. 111–122. DOI: 10.1109/24.994926. URL: https://doi.org/10.1109/24.994926 (Cited on page 8).

[Ous82] J. K. Ousterhout. “Scheduling Techniques for Concurrent Systems”. In: *Proceedings of the 3rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 1982, pp. 22–30 (Cited on page 78).

[Pla16] M. Plappert. *keras-rl*. https://github.com/keras-rl/keras-rl. 2016 (Cited on page 190).

[Pot80] C. N. Potts. “Technical Note - Analysis of a Heuristic for One Machine Sequencing with Release Dates and Delivery Times”. In: *Oper. Res.* 28.6 (1980), pp. 1436–1441. DOI: 10.1287/opre.28.6.1436. URL: https://doi.org/10.1287/opre.28.6.1436 (Cited on pages 61 sq., 87, 101).

[Pra86] D. K. Pradhan, ed. *Fault-Tolerant Computing: Theory and Techniques; Vol. 1*. USA: Prentice-Hall, Inc., 1986. ISBN: 013308230X (Cited on page 31).

[PST+97] C. A. Phillips, C. Stein, E. Torng, and J. Wein. “Optimal Time-Critical Scheduling via Resource Augmentation (Extended Abstract)”. In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing*. Ed. by F. T. Leighton and P. W. Shor. ACM, 1997, pp. 140–149. DOI: 10.1145/258533.258570. URL: https://doi.org/10.1145/258533.258570 (Cited on page 30).

[PTV+07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: the art of scientific computing, 3rd Edition*. Cambridge University Press, 2007. ISBN: 9780521706858. URL: https://www.worldcat.org/oclc/123285342 (Cited on page 183).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. DOI: 10.5555/1953048.2078195. URL: https://dl.acm.org/doi/10.5555/1953048.2078195 (Cited on page 167).

[QH00] G. Quan and X. S. Hu. “Enhanced Fixed-Priority Scheduling with (m, k)-Firm Guarantee”. In: *Proceedings of the 21st IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 2000, pp. 79–88. DOI: 10.1109/REAL.2000.895998. URL: https://doi.org/10.1109/REAL.2000.895998 (Cited on pages 8, 47, 174).

[Raj90] R. Rajkumar. “Real-Time Synchronization Protocols for Shared Memory Multiprocessors”. In: *10th International Conference on Distributed Computing Systems ICDCS*. IEEE Computer Society, 1990, pp. 116–123. DOI: `10.1109/ICDCS.1990.89257`. URL: `https://doi.org/10.1109/ICDCS.1990.89257` (Cited on pages 3, 39, 102).

[Ram99] P. 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. DOI: `10.1109/71.774906`. URL: `https://doi.org/10.1109/71.774906` (Cited on pages 46 sq.).

[RBR19] L. Rakotomalala, M. Boyer, and P. Roux. “Formal Verification of Real-time Networks”. In: *JRWRTC 2019, Junior Workshop RTNS 2019*. TOULOUSE, France, 11/2019. URL: `https://hal.science/hal-02449140` (Cited on page 44).

[RKB+16] J. Richter, H. Kotthaus, B. Bischl, P. Marwedel, J. Rahnenführer, and M. Lang. “Faster Model-Based Optimization Through Resource-Aware Scheduling Strategies”. In: *Learning and Intelligent Optimization - 10th International Conference, LION, Revised Selected Papers*. Ed. by P. Festa, M. Sellmann, and J. Vanschoren. Vol. 10079. Lecture Notes in Computer Science. Springer, 2016, pp. 267–273. DOI: `10.1007/978-3-319-50349-3\_22`. URL: `https://doi.org/10.1007/978-3-319-50349-3%5C_22` (Cited on pages 46, 163).

[RMW14] D. J. Rezende, S. Mohamed, and D. Wierstra. “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. In: *Proceedings of the 31th International Conference on Machine Learning, ICML*. Vol. 32. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 1278–1286. URL: `http://proceedings.mlr.press/v32/rezende14.html` (Cited on page 158).

[RQB22] P. Roux, S. Quinton, and M. Boyer. “A Formal Link Between Response Time Analysis and Network Calculus (Artifact)”. In: *Dagstuhl Artifacts Ser.* 8.1 (2022), 03:1–03:3. DOI: `10.4230/DARTS.8.1.3`. URL: `https://doi.org/10.4230/DARTS.8.1.3` (Cited on page 44).

[RSL88] R. Rajkumar, L. Sha, and J. P. Lehoczky. “Real-Time Synchronization Protocols for Multiprocessors”. In: *Proceedings of the 9th IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 1988, pp. 259–269. DOI: `10.1109/REAL.1988.51121`. URL: `https://doi.org/10.1109/REAL.1988.51121` (Cited on pages 3, 39, 102).

[RZH20] I. Rasheed, L. Zhang, and F. Hu. “A privacy preserving scheme for vehicle-to-everything communications using 5G mobile edge computing”. In: *Comput. Networks* 176 (2020), p. 107283. DOI: `10.1016/j.comnet.2020.107283`. URL: `https://doi.org/10.1016/j.comnet.2020.107283` (Cited on page 6).

[SBR+21] J. Shi, J. Bian, J. Richter, K.-H. Chen, J. Rahnenführer, H. Xiong, and J.-J. Chen. “MODES: model-based optimization on distributed embedded systems”. In: *Mach. Learn.* 110.6 (2021), pp. 1527–1547 (Cited on pages 13, 17).

[SBR21] J. Shi, J. Bian, and J. Richter. *Model-based Optimization on Distributed Embedded System*. `https://github.com/Strange369/MODES-public`. 2021. URL: `https://github.com/Strange369/MODES-public` (Cited on page 172).

[SCZ+17] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings. “Implementation and Evaluation of Multiprocessor Resource Synchronization Protocol (MrsP) on LITMUSRT”. In: *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT).* 2017 (Cited on pages 5, 42, 116).

[SEC+22] J. Shi, C.-C. von Egidy, K.-H. Chen, and J.-J. Chen. “Formal Verification of Resource Synchronization Protocol Implementations: A Case Study in RTEMS”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.11 (2022), pp. 4157–4168 (Cited on pages 12, 17).

[SGW+17] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. “Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks”. In: *2017 IEEE Real-Time Systems Symposium, RTSS.* IEEE Computer Society, 2017, pp. 92–103. DOI: `10.1109/RTSS.2017.00016`. URL: `https://doi.org/10.1109/RTSS.2017.00016` (Cited on page 103).

[Shi18] J. Shi. *Dependency Graph Approaches for OCS Task Model Implemented LITMUS-RT.* 2018. URL: `https://github.com/JJShi92/Dependency-Graph-Approach-for-Periodic-Tasks` (visited on 08/01/2023) (Cited on pages 119, 154).

[Shi19] J. Shi. *Dependency Graph Approaches for MCS Task Model Implemented LITMUS-RT.* 2019. URL: `https://github.com/JJShi92/DGA-Multiple-CriticalSections` (visited on 08/01/2023) (Cited on pages 119, 154).

[SHK+14] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting”. In: *J. Mach. Learn. Res.* 15.1 (2014), pp. 1929–1958. DOI: `10.5555/2627435.2670313`. URL: `https://dl.acm.org/doi/10.5555/2627435.2670313` (Cited on page 45).

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. “Mastering the game of Go with deep neural networks and tree search”. In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: `10.1038/nature16961`. URL: `https://doi.org/10.1038/nature16961` (Cited on page 187).

[SKM00] S. P. Singh, M. J. Kearns, and Y. Mansour. “Nash Convergence of Gradient Dynamics in General-Sum Games”. In: *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence.* 2000, pp. 541–548. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1%5C&smnu=2%5C&article%5C_id=63%5C&proceeding%5C_id=16` (Cited on page 187).

[SKT20] M. Shirazi, M. Kargahi, and L. Thiele. “Performance maximization of energy-variable self-powered (m, k)-firm real-time systems”. In: *Real Time Syst.* 56.1 (2020), pp. 64–111. DOI: `10.1007/s11241-020-09344-1`. URL: `https://doi.org/10.1007/s11241-020-09344-1` (Cited on page 47).

[SLA12] J. Snoek, H. Larochelle, and R. P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems, Proceedings*. Ed. by P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. 2012, pp. 2960–2968. URL: `https://proceedings.neurips.cc/paper/2012/hash/05311655a15b75fab86956663e1819cd-Abstract.html` (Cited on page 45).

[SLD+03] H. Su, F. Liu, A. Devgan, E. Acar, and S. R. Nassif. “Full chip leakage estimation considering power supply and temperature variations”. In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2003, pp. 78–83. DOI: `10.1145/871506.871529`. URL: `https://doi.org/10.1145/871506.871529` (Cited on page 9).

[SPD+22] V. P. Suresh, R. R. Pai, D. D’Souza, M. D’Souza, and S. K. Chakrabarti. “Static Race Detection for Periodic Programs”. In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings*. Ed. by I. Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 290–316. DOI: `10.1007/978-3-030-99336-8\_11`. URL: `https://doi.org/10.1007/978-3-030-99336-8%5C_11` (Cited on page 44).

[SPM+22] J. Shi, J. D. T. Pham, M. Münch, J. V. Hafemeister, J.-J. Chen, and K.-H. Chen. “Supporting Multiprocessor Resource Synchronization Protocols in RTEMS”. In: *16th annual workshop on Operating Systems Platforms for Embedded Real-Time applications, OSPERT*. 2022. eprint: `2104.06366` (Cited on pages 42, 118, 131, 201).

[SPS22] J. Shi, J. D. T. Pham, and S. Subramanian. *The source code of this work*. 2022. URL: `https://github.com/JJShi92/RTEMS-Resource-Synchronization-Protocols` (visited on 10/03/2023) (Cited on page 154).

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”. In: *IEEE Trans. Computers* 39.9 (1990), pp. 1175–1185. DOI: `10.1109/12.57058`. URL: `https://doi.org/10.1109/12.57058` (Cited on pages 2, 38, 40, 137).

[SS95] Y. N. Sotskov and N. V. Shakhlevich. “NP-hardness of Shop-scheduling Problems with Three Jobs”. In: *Discret. Appl. Math.* 59.3 (1995), pp. 237–266. DOI: `10.1016/0166-218X(95)80004-N`. URL: `https://doi.org/10.1016/0166-218X(95)80004-N` (Cited on pages 41, 72).

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. “Mastering the game of Go without human knowledge”. In: *Nat.* 550.7676 (2017), pp. 354–359. DOI: `10.1038/nature24270`. URL: `https://doi.org/10.1038/nature24270` (Cited on page 187).

[SUB+19a] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Multiprocessor Synchronization of Periodic Real-Time Tasks Using Dependency Graphs”. In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2019, pp. 279–292 (Cited on pages 10, 16 sq.).

[SUB+19b] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Partitioned Scheduling for Dependency Graphs in Multiprocessor Real-Time Systems”. In: *25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*. 2019, pp. 1–12 (Cited on pages 10, 16).

[SUB+21] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. “Graph-Based Optimizations for Multiprocessor Nested Resource Sharing”. In: *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*. 2021, pp. 129–138 (Cited on pages 10, 16).

[SUC+23] J. Shi, N. Ueter, J.-J. Chen, and K.-H. Chen. “Average Task Execution Time Minimization under (m, k) Soft Error Constraint”. In: *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2023, pp. 1–13 (Cited on pages 14, 17).

[SZY+20] I. Singh, H. Zhou, K. Yang, M. Ding, B. Lin, and P. Xie. “Differentially-private Federated Neural Architecture Search”. In: *CoRR* abs/2006.10559 (2020). arXiv: `2006.10559`. URL: `https://arxiv.org/abs/2006.10559` (Cited on page 46).

[TEH+16] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. “System-level timing feasibility test for cyber-physical automotive systems”. In: *11th IEEE Symposium on Industrial Embedded Systems, SIES*. IEEE, 2016, pp. 121–130. DOI: `10.1109/SIES.2016.7509419`. URL: `https://doi.org/10.1109/SIES.2016.7509419` (Cited on page 100).

[Tes95] G. Tesauro. “Temporal Difference Learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68. DOI: `10.1145/203330.203343`. URL: `https://doi.org/10.1145/203330.203343` (Cited on page 187).

[The21] The RTEMS Project. *Real-Time Executive for Multiprocessor Systems*. 2021. URL: `http://www.rtems.org/` (visited on 04/01/2022) (Cited on page 5).

[VPC02] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. “Transient-Fault Recovery Using Simultaneous Multithreading”. In: *29th International Symposium on Computer Architecture ISCA*. Ed. by Y. N. Patt, D. Grunwald, and K. Skadron. IEEE Computer Society, 2002, pp. 87–98. DOI: `10.1109/ISCA.2002.1003565`. URL: `https://doi.org/10.1109/ISCA.2002.1003565` (Cited on page 31).

[VPM22] N. Vreman, R. Pates, and M. Maggio. “WeaklyHard.jl: Scalable Analysis of Weakly-Hard Constraints”. In: *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. IEEE, 2022, pp. 228–240. DOI: `10.1109/RTAS54340.2022.00026`. URL: `https://doi.org/10.1109/RTAS54340.2022.00026` (Cited on pages 47, 179).

[VRN+22] F. Vanhems, V. Rusu, D. Nowak, and G. Grimaud. “A Formal Correctness Proof for an EDF Scheduler Implementation”. In: *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. IEEE, 2022, pp. 281–292. DOI: `10.1109/RTAS54340.2022.00030`. URL: `https://doi.org/10.1109/RTAS54340.2022.00030` (Cited on page 43).

[WA12] B. C. Ward and J. H. Anderson. “Supporting Nested Locking in Multiprocessor Real-Time Systems”. In: *24th Euromicro Conference on Real-Time Systems, ECRTS*. Ed. by R. Davis. IEEE Computer Society, 2012, pp. 223–232. DOI: 10.1109/ECRTS.2012.17. URL: https://doi.org/10.1109/ECRTS.2012.17 (Cited on page 40).

[WA13] B. C. Ward and J. H. Anderson. “Fine-grained multiprocessor real-time locking with improved blocking”. In: *21st International Conference on Real-Time Networks and Systems, RTNS*. Ed. by M. Auguin, R. de Simone, R. I. Davis, and E. Grolleau. ACM, 2013, pp. 67–76. DOI: 10.1145/2516821.2516843. URL: https://doi.org/10.1145/2516821.2516843 (Cited on pages 40, 81).

[WA14] B. C. Ward and J. H. Anderson. “Multi-resource Real-Time Reader/Writer Locks for Multiprocessors”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2014, pp. 177–186. DOI: 10.1109/IPDPS.2014.29. URL: https://doi.org/10.1109/IPDPS.2014.29 (Cited on page 40).

[WB13a] A. Wieder and B. B. Brandenburg. “Efficient partitioning of sporadic real-time tasks with shared resources and spin locks”. In: *8th IEEE International Symposium on Industrial Embedded Systems, SIES*. IEEE, 2013, pp. 49–58. DOI: 10.1109/SIES.2013.6601470. URL: https://doi.org/10.1109/SIES.2013.6601470 (Cited on page 40).

[WB13b] A. Wieder and B. B. Brandenburg. “On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks”. In: *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS*. IEEE Computer Society, 2013, pp. 45–56. DOI: 10.1109/RTSS.2013.13. URL: https://doi.org/10.1109/RTSS.2013.13 (Cited on page 102).

[WHK+21] Z. Wang, C. Huang, H. Kim, W. Li, and Q. Zhu. “Cross-Layer Adaptation with Safety-Assured Proactive Task Job Skipping”. In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021), 100:1–100:25. DOI: 10.1145/3477031. URL: https://doi.org/10.1145/3477031 (Cited on page 48).

[XHD+15] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. “Petuum: A New Platform for Distributed Machine Learning on Big Data”. In: *IEEE Trans. Big Data* 1.2 (2015), pp. 49–67. DOI: 10.1109/TBDATA.2015.2472014. URL: https://doi.org/10.1109/TBDATA.2015.2472014 (Cited on page 46).

[XKH+23] R. Xu, M. Kühl, H. von Hasseln, and D. Nowotka. “Reducing Overall Path Latency in Automotive Logical Execution Time Scheduling via Reinforcement Learning”. In: *Proceedings of the 31st International Conference on Real-Time Networks and Systems, RTNS*. ACM, 2023, pp. 212–223. DOI: 10.1145/3575757.3593658. URL: https://doi.org/10.1145/3575757.3593658 (Cited on pages 48, 202).

[XRV17] H. Xiao, K. Rasul, and R. Vollgraf. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *CoRR* abs/1708.07747 (2017). arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747 (Cited on page 165).

[YCC18] M. Yayla, K.-H. Chen, and J.-J. Chen. “Fault Tolerance on Control Applications: Empirical Investigations of Impacts from Incorrect Calculations”. In: *4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems, EITEC@CPSWeek*. IEEE Computer Society, 2018, pp. 17–24. DOI: 10.1109/EITEC.2018.00008. URL: https://doi.org/10.1109/EITEC.2018.00008 (Cited on pages 8, 47, 174).

[YHL04] W. Yu, H. Hoogeveen, and J. K. Lenstra. “Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard”. In: *J. Sched.* 7.5 (2004), pp. 333–348. DOI: 10.1023/B:JOSH.0000036858.59787.c2. URL: https://doi.org/10.1023/B:JOSH.0000036858.59787.c2 (Cited on pages 41, 73).

[ZJ22] H. Zhu and Y. Jin. “Real-Time Federated Evolutionary Neural Architecture Search”. In: *IEEE Trans. Evol. Comput.* 26.2 (2022), pp. 364–378. DOI: 10.1109/TEVC.2021.3099448. URL: https://doi.org/10.1109/TEVC.2021.3099448 (Cited on page 46).

[ZMC01] D. Zhu, R. G. Melhem, and B. R. Childers. “Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems”. In: *Proceedings of the 22nd IEEE Real-Time Systems Symposium RTSS*. IEEE Computer Society, 2001, pp. 84–94. DOI: 10.1109/REAL.2001.990599. URL: https://doi.org/10.1109/REAL.2001.990599 (Cited on page 99).

[ZV10] X. Zhang and S. L. van de Velde. “Polynomial-time approximation schemes for scheduling problems with time lags”. In: *J. Sched.* 13.5 (2010), pp. 553–559. DOI: 10.1007/s10951-009-0134-8. URL: https://doi.org/10.1007/s10951-009-0134-8 (Cited on page 76).

[ZW17] S. Zhao and A. J. Wellings. “Investigating the correctness and efficiency of MrsP in fully partitioned systems”. In: *10th York doctoral symposium on computer science and electronic engineering*. York. 2017 (Cited on page 42).

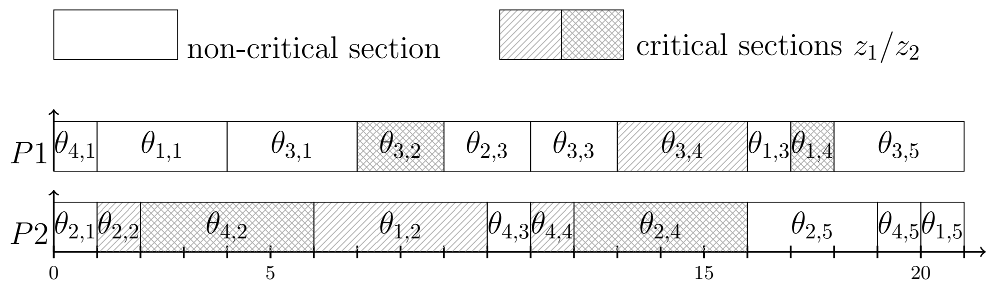
Appendix A

# Appendix

## A.1 Appendix for Chapter 4

### A.1.1 Detailed Schedules for Illustrative Examples

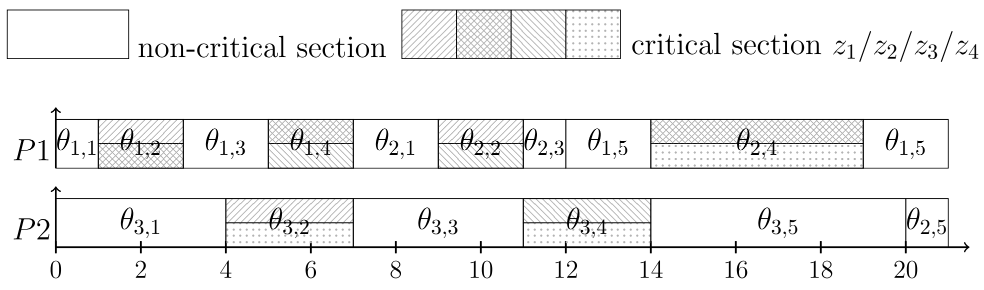
By applying the LIST-EDF scheduling algorithm on two processors, the concrete schedule for the dependency graph presented in Figure 4.2 is depicted in Figure A.1.



non-critical section
critical sections $z_1/z_2$
P1
$\theta_{4,1}$ $\theta_{1,1}$ $\theta_{3,1}$ $\theta_{3,2}$ $\theta_{2,3}$ $\theta_{3,3}$ $\theta_{3,4}$ $\theta_{1,3}$ $\theta_{1,4}$ $\theta_{3,5}$
P2
$\theta_{2,1}$ $\theta_{2,2}$ $\theta_{4,2}$ $\theta_{1,2}$ $\theta_{4,3}$ $\theta_{4,4}$ $\theta_{2,4}$ $\theta_{2,5}$ $\theta_{4,5}$ $\theta_{1,5}$
0
5
15
20

**Figure A.1:** Schedule the dependency graph from Figure 4.2 on 2 processors using LIST-EDF.

Similarly, by utilizing the LIST-EDF scheduling algorithm on two processors, the schedule for the dependency graph from Figure 4.3 is illustrated in Figure A.2.



non-critical section
critical section $z_1/z_2/z_3/z_4$
P1
$\theta_{1,1}$ $\theta_{1,2}$ $\theta_{1,3}$ $\theta_{1,4}$ $\theta_{2,1}$ $\theta_{2,2}$ $\theta_{2,3}$ $\theta_{1,5}$ $\theta_{2,4}$ $\theta_{1,5}$
P2
$\theta_{3,1}$ $\theta_{3,2}$ $\theta_{3,3}$ $\theta_{3,4}$ $\theta_{3,5}$ $\theta_{2,5}$
0
2
4
6
8
10
12
14
16
18
20

**Figure A.2:** Schedule the dependency graph from Figure 4.3 on 2 processors using LIST-EDF.

### A.1.2 Evaluation Results for Frame-based Task Sets

We present the remaining evaluation results for frame-based task sets across various task models.

#### Evaluation Results for OCS Task Systems

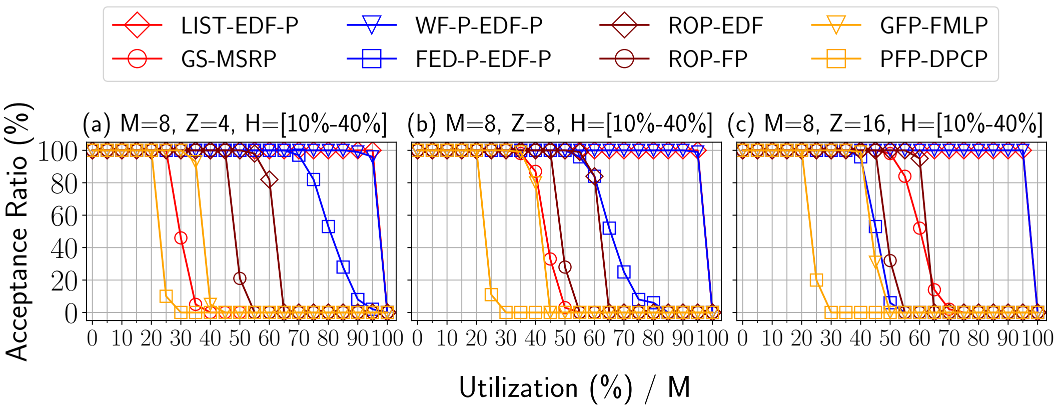


**Figure A.3:** The evaluation results for frame-based OCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

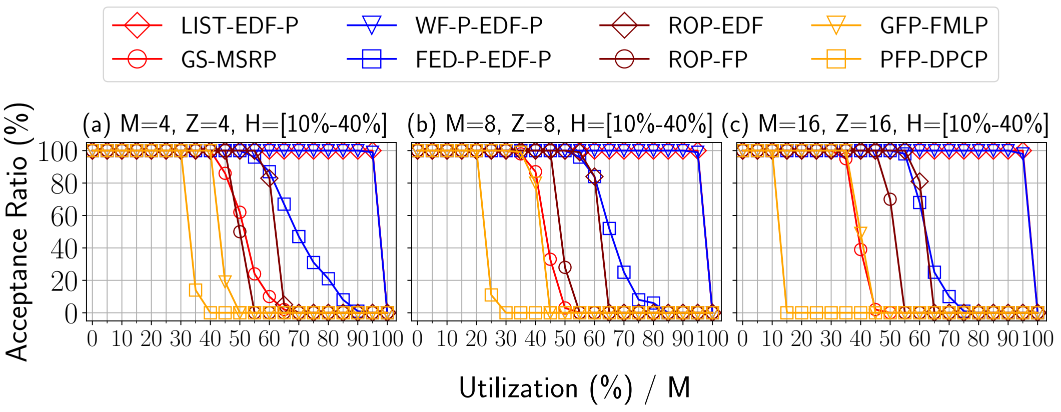


**Figure A.4:** The evaluation results for frame-based OCS task systems with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of processor and the number of available shared resources simultaneously, i.e., $M = Z \in \{4, 8, 16\}$.

A subset of our evaluation results for frame-based task systems with the OCS task model can be found in Figures A.3 and A.4. These evaluation results demonstrate two additional scenarios:

- **Number of Available Shared Resources, i.e., $Z \in \{4, 8, 16\}$** (Figure A.3): When the number of shared resources increases relative to the number of processors, the performance of most evaluated approaches improves. However, the performance of DGA with the federated-based partitioning algorithm declines. This suggests that the federated-based partitioning algorithm struggles when the number of available shared resources is considerably large.
- **Concurrent Expansion of Available Processor and Shared Resource, i.e., $M = Z \in \{4, 8, 16\}$** (Fig. 4.13): A simultaneous increase in both $M$ and $Z$ does not significantly affect the performance of the evaluated approaches.

### Evaluation Results for MCS Task Systems

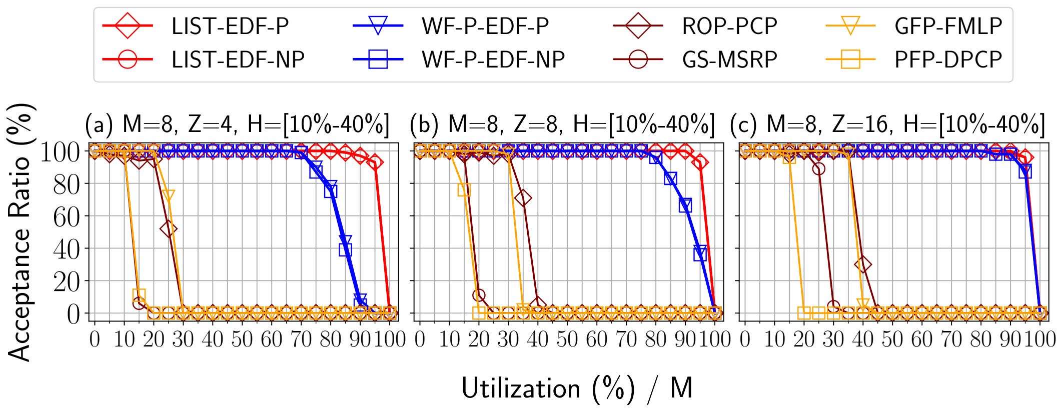


**Figure A.5:** The evaluation results for frame-based MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

A subset of our evaluation results for frame-based task systems using the MCS task model is presented in Figures A.5 and A.6.

As observed previously, the results show that when the number of shared resources increases relative to the number of processors (as seen in Figure A.5), the performance of all evaluated approaches improves. Please note that in the MCS task model, only the worst-fit heuristic was evaluated as the partitioning algorithm for DGA. A concurrent increase in both $M$ and $Z$ (as shown in Figure A.6) slightly worsens the performance of all evaluated approaches.

### Evaluation Results for Nested-MCS Task Systems

We present a subset of our evaluation results for frame-based task systems using the Nested-MCS task model in Figures A.7, A.8, and A.9.

In Figure A.7, we observe that an increase in the number of shared resources, without modifying other parameters, does not substantially affect the performance of the evaluated algorithms. However, increasing the depth of nested resource accesses

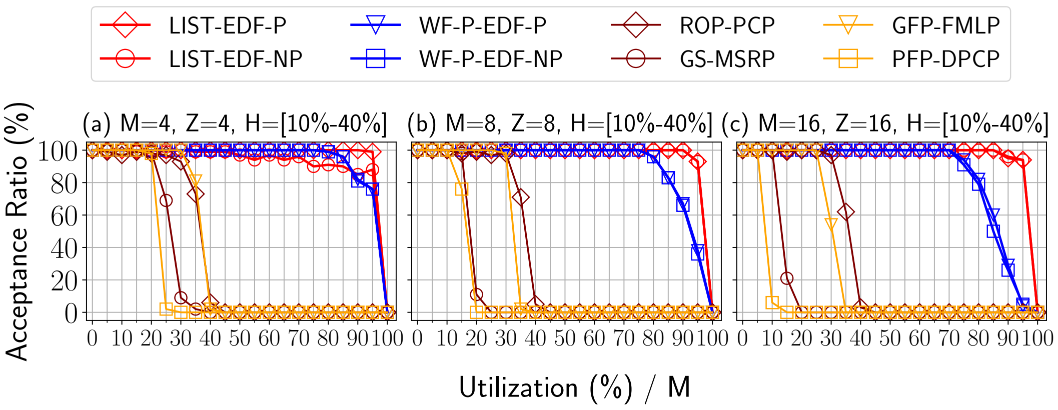


**Figure A.6:** The evaluation results for frame-based MCS task systems with $[10\%, 40\%]$ workload for critical sections, emphasis on increasing the number of processor and the number of available shared resources simultaneously, i.e., $M = Z \in \{4, 8, 16\}$.

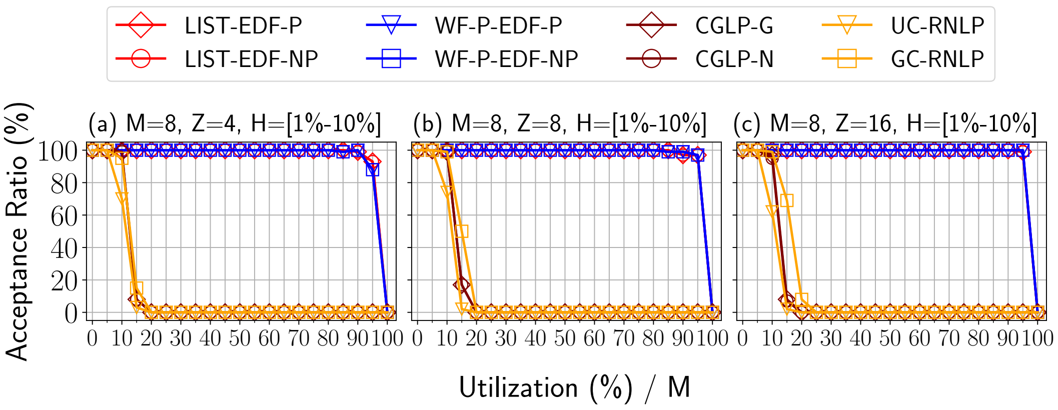


**Figure A.7:** The evaluation results for frame-based Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 2, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

from 2 to 4 (Figure A.7 to Figure A.8) results in a slight decline in the performance for all methods. In deep nested resource accesses, i.e., depth is 4, an increase in the number of shared resources leads to a performance improvement for the DGA with the partitioned scheduling algorithm.

Moreover, an increase in the probability of a critical section requesting nested shared resources (Figure A.9) does not significantly influence the performance of the evaluated algorithms.

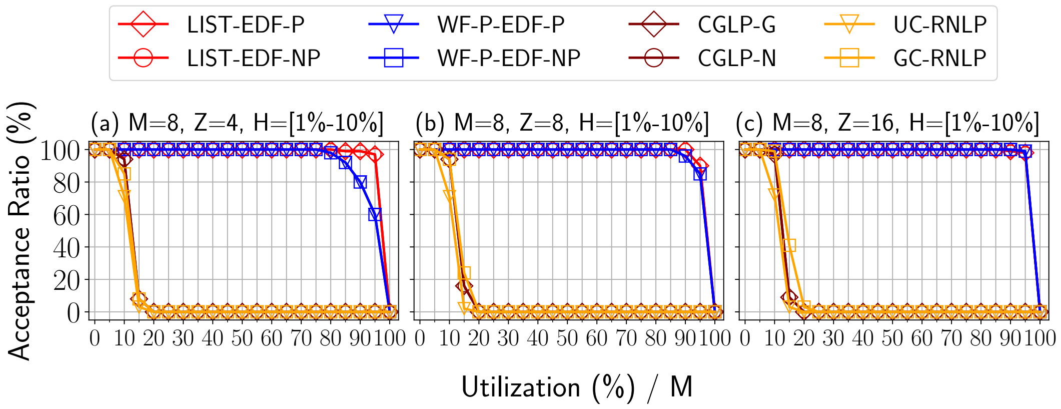


**Figure A.8:** The evaluation results for frame-based Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 4, emphasis on increasing the number of available shared resources, i.e., $Z \in \{4, 8, 16\}$.

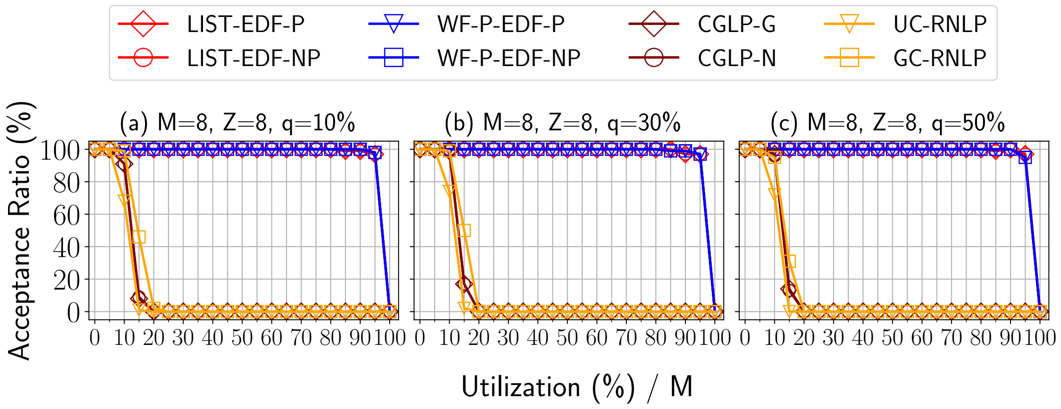


**Figure A.9:** The evaluation results for frame-based Nested-MCS task systems on 8 processors with $[10\%, 40\%]$ workload for critical sections, and a nested depth of 2, emphasis on increasing the probability that a critical section requests nested shared resources, i.e., $q \in \{10\%, 30\%, 50\%\}$.