Deployment Designs for Multi-Core Real-Time Systems

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

vorgelegt von

ERJOLA LALO

aus Çorovodë, Albanien

genehmigt von der Fakultät für Mathematik/Informatik und Maschinenbau
der Technischen Universität Clausthal,

Tag der mündlichen Prüfung

12. Oktober 2023

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Bibliographic information published by the Deutsche Nationalbibliothek:
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; Detailed bibliographic data are available in the Internet at http://dnb.dnb.de.

Dissertation Technische Universität Clausthal, 2023
D 104

Dekan
Prof. Dr. Jörg P. Müller
Vorsitzender der Promotionskommission
Prof. Dr. Niels Neumann
Betreuer
Prof. Dr. Christian Siemers
Gutachter
Prof. Dr. Andreas Rausch
Prof. Dr. Jürgen Mottok

Umschlagabbildung: davstudio - stock.adobe.com

1. Auflage 2024
© 2024 Universitätsverlag, Regensburg
Leibnizstraße 13, 93055 Regensburg
Umschlaggestaltung: typegerecht berlin
Hintere Umschlagabbildung: Erjola Lalo
Satz: Erjola Lalo
Druck: docupoint magdeburg

ISBN 978-3-86845-180-1
ISBN 978-3-86845-181-8 (PDF)
DOI https://doi.org/10.61036/3868451818

Abstract

In modern embedded automotive systems, multi-core processors are used to provide higher execution performance and more computing capacity. However, they influence the way these systems are designed. Each change of the platform and application software results in an effort required to ensure and validate the functional correctness and timing requirements of these systems in a multi-core platform. For this reason, the use of the *Logical Execution Time (LET)* paradigm is advantageous for multi-core systems, as it satisfies these requirements by providing time and dataflow determinism, thereby reducing the development cycle and the impact of the platform in the design of such systems.

To integrate LET into these systems, special mechanisms are required to satisfy its semantics. Therefore, this work focuses on buffering and scheduling techniques for resource efficient integration of LET in automotive systems. A buffering mechanism and an automatic schedule synthesis are proposed to guarantee functional correctness, timing requirements, and LET semantics and to reduce the increased demands of LET for memory and processor resources. A static and global buffering protocol is proposed, which compared to lock-based protocols has plausible memory needs and zero-communication overheads at the boundaries of LET intervals. The automatic schedule synthesis considers the *Operating System (OS)* overheads such that scheduling is optimized with respect to context-switching overheads caused by preemption.

Buffering implements most of the LET semantics, but it provides no guarantee that tasks execute within their LET intervals. If scheduling is not designed to ensure that all tasks execute within their LET intervals, then functional correctness and LET semantics are violated. The manual design of a feasible schedule that satisfies the semantics and additional requirements of the application is inefficient and requires a significant amount of effort. Therefore, an automatic schedule generation approach for the *Fixed-Priority Scheduling (FPS)* and *Time-Triggered Scheduling (TTS)* is proposed. FPS is the widely used mechanism in the automotive domain because of its flexibility to handle dynamic changes of highly event-driven applications. Recently, TTS has attracted considerable attention in this domain to increase deterministic execution of tasks and efficient planning of processor resources. Therefore, schedule synthesis of FPS and TTS is provided to identify the efficiency, practicality, and resource optimization abilities of each approach for LET systems.

To show the practicality of LET for industrial automotive systems, its integration into the software architecture of the *classic platform* of *AUTomotive Open System ARchitecture (AUTOSAR)* is described and a case study is conducted using a real world system.

Kurzfassung

In modernen eingebetteten Systemen des Automobilbereichs werden Mehrkernprozessoren eingesetzt, um schnellere Ausführungsgeschwindigkeiten und höhere Rechenkapazitäten zu bekommen. Sie beeinflussen jedoch die Art und Weise, wie solche Systeme entworfen werden. Jedes Mal, wenn sich die Plattform oder die Anwendungssoftware ändert, hat dies den Aufwand zur Folge der benötigt wird, um die funktionale Korrektheit und die zeitlichen Anforderungen dieser Systeme in einer Mehrkernplattform sicherzustellen und zu validieren. Abhilfe schafft die Verwendung des *Logical Execution Time (LET)*-Paradigmas, da es Zeit- und Datenflussdeterminismus bietet und dadurch den Entwicklungszyklus verkürzt und den Einfluss der Plattform auf den Entwurf solcher Systeme reduziert.

Für die Integration von LET in Automobilsysteme, sind spezielle Mechanismen erforderlich, um dessen Semantik zu erfüllen. So werden ein Puffermechanismus und die automatische Synthese eines Schedulings vorgeschlagen, um die funktionale Korrektheit, die zeitlichen Anforderungen und die LET-Semantik zu gewährleisten und die erhöhten Anforderungen von LET an Speicher- und Prozessorressourcen zu reduzieren. Das statische und globale Pufferprotokoll besitzt im Vergleich zu Protokollen mit Zugriffssperren einen plausiblen Speicherbedarf und benötigt keinen Mehraufwand durch Kommunikation an den Grenzen der LET-Intervalle. Die automatische Synthese des Schedulings berücksichtigt das darunter liegende *Betriebssystems (OS)*, wodurch der Mehraufwand durch Kontextwechsel reduziert wird.

Die Pufferung realisiert den größten Teil der LET-Semantik, aber bietet keine Garantie dafür, dass Prozesse innerhalb ihrer LET-Intervalle ausgeführt werden. Denn für eine funktionale Korrektheit wird ein entsprechend ausgelegtes Scheduling benötigt. Der manuelle Entwurf eines realisierbaren Schedulings, der die Semantik und die zusätzlichen Anforderungen der Anwendung erfüllt, ist ineffizient und erfordert einen erheblichen Aufwand. Daher wird ein automatischer Ansatz für das *Fixed-Priority Scheduling (FPS)* und das *Time-Triggered Scheduling (TTS)* vorgestellt und deren Effizienz, Praktikabilität und die Fähigkeiten zur Ressourcenoptimierung für LET-Systeme ermittelt. FPS ist wegen seiner Flexibilität bei dynamischen Änderungen von stark ereignisgesteuerten Anwendungen der weit verbreitetste Mechanismus im Automobilbereich. Um die deterministische Ausführung von Prozessen und die effiziente Planung von Prozessorressourcen zu verbessern, hat TTS in diesem Bereich in letzter Zeit erhebliche Aufmerksamkeit erregt.

Die Praxistauglichkeit von LET für industrielle Automobilsysteme wird schließlich gezeigt, indem beschrieben wird, wie es sich in die *Classic Plattform* der *AUTomotive Open System ARchitecture (AUTOSAR)* integrieren lässt und eine Fallstudie anhand einer realen Anwendung durchgeführt wird.

To my family

Acknowledgements

This work was partially conducted during the OBZAS project ("Optimiertes Buffering für Zeitgesteuerte Automobile Software") funded by the Bavarian Research Foundation and the ARAMIS II project ("Development of Processes, Tools, and Platforms for Safety-Critical Multi-Core Systems") funded by the German Federal Ministry for Education and Research. I have been fortunate to be mentored and supported by a number of brilliant people. Foremost, I would like to express my deepest gratitude to my advisor Prof. Dr. Jürgen Mottok for his consistent guidance, invaluable advice, and the opportunity to enter the research world of embedded systems. I would like to express my deepest appreciation to my supervisor Prof. Dr. Christian Siemers for the supervision during my doctoral studies and for the opportunity to write this dissertation in his research group at the Clausthal University of Technology. I am extremely grateful for the inspiring discussions and his invaluable support. Furthermore, I would like to thank Prof. Dr. Andreas Rausch for his constructive feedback.

I am extremely grateful to my industry advisor Dr. Andreas Sailer for his outstanding encouragement, mentoring, and guidance during my research work. This endeavor would not have been possible without his persistent support. I am grateful for the opportunity to have worked with Dr. Eugene Yip during the OBZAS project. I am particularly thankful for the joint experiences on scientific research and technical discussions. I am deeply indebted to my Vector Informatik GmbH colleagues, who contributed to the success of this dissertation, in particular to Philip Wagner, Christian Schütze, Raphael Weber, Michael Volk, Timo Schwendner, Manuel Strobel, Sascha Sommer, and Dr. Werner Thumann. Many thanks also to my colleagues Dr. Marc Weber, Marco Wierer, Katharina Engel, Erna Oklapi, Thomas Wilhelm, Daniel Wetzel, and Tamilselvan Shanmugam for their support during my work in Vector. Special thanks to my very good friend and colleague Arlinda Elmazi for her continuous encouragement, invaluable technical discussions, and the review of my dissertation. I could not have undertaken this journey without the support of my former colleagues of Timing Architects GmbH Dr. Michael Deubzer, Prof. Dr. Martin Hobelsberger, and Maximilian Rappl. I would like to express my sincere gratitude to my friends who have made my life more enjoyable during my doctoral studies.

Finally, and most importantly, I would like to thank my wonderful parents, my siblings, my uncle and my aunt who have always supported me and made possible that I pursue university studies and develop professionally. Most of all, I thank my husband, Rilion, who has inspired, supported, and encouraged me all these years, despite the long hours and days of working on my doctoral studies. You, my parents, my siblings, my uncle and my aunt are my best friends and mentors in life. I dedicate this dissertation to all of you.

March 2023, Regensburg

Contents

1 | Introduction

Embedded real-time systems are electronic processing units composed of software and hardware components. They can be found in domains such as automotive, avionic, and industrial automation. Their number is heavily growing and they are evolving to more advanced and complex functionalities such as autonomous driving and automated flight control. These systems are characterized by non-functional requirements such as timing, performance, robustness, safety, and reliability. These requirements imply that these systems must deliver correct results on time and have a deterministic execution flow and robust behavior in case of environmental disturbances.

Embedded real-time systems are divided into *hard*, *firm*, and *soft* real-time systems based on the deadline requirements. Violations of deadlines in *firm* and *soft* systems do not halt the system from functioning, but produced results are either discarded or have a low degree of usability. Violations of deadlines in *hard* real-time systems can cause catastrophic consequences. Therefore, these violations must be captured early in the development phase. An example of an in-vehicle system where these requirements are critical to ensuring passenger safety is the *Electric Power Steering (EPS)* system. The response time of this system to an under- or over-steering situation determines if the vehicle brakes and safely enters the lane on time [1]. Similarly, in airbag systems, fatal consequences can occur in the event of a vehicle accident if the airbag is not activated and opened on time. It is therefore imperative that such systems are of high quality. Sophisticated design solutions are required to ensure that functional and non-functional requirements are met, high performance is achieved, and development costs are kept low.

1.1 Motivation

Modern real-time automotive systems consist of thousands of functions [2]. These functions are highly interconnected, i.e., a large amount of data is exchanged between them, and a strict execution order of these functions is required for the proper functioning of the system. In addition, timing requirements such as data ages, deadlines, and end-to-end delays are specified for the outputs produced by these functions to ensure that the correct functionality is provided at the required time. Traditionally, the dataflow between these functions and the timing requirements are guaranteed during their integration to tasks. This process is highly complex because it is influenced not only by the complexity of the application itself, but also by the platform on which it runs, such as the hardware and *Operating System (OS)*. For example, scheduling is an integral components of the OS that also determines if timing requirements and the dataflow between functions of different tasks are fulfilled.

The technological expansion of automotive systems has shifted their development towards high-performance multi-core processors [3]. These processors offer computing capacity for more advanced functionalities, e.g., needed by autonomous driving, parking, and driving assistance systems. In addition to the increase in performance, a major advantage of multi-core processors is that the parallel execution of functions on different cores reduces the time required to deliver their outputs and end-to-end delays. However, these benefits come at the expense of a further increase on complexity of designing such systems. This is particularly challenging when integrating and maintaining different software increments or with each platform change.

When developing multi-core systems, software integrators are faced with the challenge of defining which functions to execute in parallel on different cores, while simultaneously ensuring that dataflow and timing requirements are met. Although approaches of supporting such design are available [4], maintaining software and platform changes is still a major challenge because most of these systems contain legacy code and redeveloping legacy functions for multi-core is not convenient due to the high development effort and costs. Furthermore, multi-core processors impose *multi-core effects* that must be also controlled during the design and integration phases of these systems. In these processors, multiple hardware components are shared among parallel executing functions such as memories, cores, and buses. The parallel, concurrent accesses to these resources cause delays and high inter-core communication overheads. Sharing of these resources leads to possible data stability issues that arise due to concurrent read and write operations to shared variables by functions running in parallel on different cores. Furthermore, deadlocks can occur due to parallel accesses on shared semaphores, which cannot be resolved by the classical priority-based resource protocols [5]. All these effects increase the response times of functions, the correct functional execution of the system, and can degrade the overall performance if not completely avoided or minimized. Therefore, changing the multi-core processor model, reassigning a function to a different core, or adding new functions to the system requires addressing all of the above challenges, which in turn increases the effort and time to design and integrate these systems and verify their timing requirements and functional correctness.

The *Logical Execution Time (LET)* paradigm [6] is an attractive approach to support the development of multi-core automotive systems because it provides time and dataflow determinism. LET defines for each task a *logical execution time* interval. Tasks read sensor data or input data produced by other tasks at the beginning of their interval and provide outputs to other tasks or actuators at the end of their interval. It shall be noted that a task contains one or multiple functions. Therefore, reading input and providing output data by a task means receiving and providing data by their respective functions. In LET, tasks deliver their results exactly at the end of their respective LET intervals, even if they complete their execution earlier than this time. In non-LET systems, tasks deliver their outputs during or at the end of their execution. Hence, in this case scheduling and allocation of tasks to cores defines when tasks deliver their outputs. For instance, higher-priority tasks deliver their outputs faster than lower-priority tasks. Furthermore, when tasks are executed in parallel on different cores, they deliver their results faster than when they are executed sequentially on the same core. In LET, changing of the platform, e.g., processor model, task-to-core allocation, and scheduling of tasks does not impact the time when tasks deliver their outputs because these tasks exchange data only at the boundaries of LET intervals. In this way, in LET, the dataflow between tasks is deterministic and end-to-end delays have a constant duration. In LET systems, the time of data exchanges between tasks is known and predictable at design and target execution, regardless of the platform.

The time and dataflow determinism provided by LET simplifies the verification process of the functional correctness and timing requirements of the system. LET provides composability, i.e., parts of the application encapsulated in LET intervals can be easily reused and integrated into new system increments without impacting the dataflow between functions. The same applies also when the system is extended with new functionalities. In LET, multi-core effects are controlled more efficiently because concurrent accesses on the same data during task execution do not occur and response times of tasks do not impact the dataflow and timing requirements of the overall application, unless deadline violations occur. Hence, the described advantages of LET significantly reduce the time and effort of developing automotive systems in multi-core platforms.

The integration of the LET paradigm into automotive systems involves several design aspects. This work focuses on two of the most important design aspects required to integrate and satisfy LET semantics. LET assumes zero communication and zero synchronization time for data exchange at the boundaries of LET intervals. This assumption is unrealistic for automotive applications because their functions are highly interconnected and a high amount of data is exchanged at the boundaries of their task's LET intervals. In such applications, LET has higher requirements regarding processing time and memory. To preserve the semantics, additional memory capacity is required to store the copies of data for all tasks. The data exchange operations take a reasonable amount of time, which increases the overall system load. Furthermore, tasks that execute beyond their LET intervals violate LET semantics and can cause in certain situations incorrect functional behavior. Therefore, to guarantee timing requirements, LET semantics, and to handle the increase of utilization, a feasible task's schedule must be defined. Scheduling must ensure that the execution of data exchanges at the boundaries of LET intervals must occur uninterruptedly and must not be delayed by other urgent tasks. This requirement is only fulfilled through deterministic scheduling strategies and algorithms that synthesize at design time the schedule of tasks considering the run-time of data exchange operations. The following section describes these aspects in detail.

1.2 Objectives and Assumptions

The fundamental objective of this work is resource efficient integration of LET paradigm into automotive multi-core systems. This objective is decomposed into several goals, which are described together with their motivation in the following paragraphs.

[Goal 1] Guarantee of LET semantics—The LET paradigm was initially designed for time-triggered aircraft systems [7]. To apply this paradigm in automotive systems, a buffering protocol must be developed. Such a protocol defines how the data is exchanged and synchronized at the boundaries of LET intervals. Although LET has a well-defined concept of exchanging data between tasks, it bases on assumptions that are not valid in practice for automotive applications. The assumptions of zero jitters and communication time of exchanged data at the boundaries of LET intervals are not true for standard lock-based buffering protocols such as the *Point-to-Point Protocol (PTP)* [8, 9]. Therefore, this goal is to design an alternative buffering approach for LET, such that the original communication semantics are guaranteed. This means that in the proposed buffering protocol the data exchange occurs with zero jitters at the boundaries of LET intervals and takes zero communication and synchronization time.

[Goal 2] Optimization of buffering load and timing of LET applications—A lock-based buffering protocol such as PTP [9] causes a significant increase of the system's load and the response times of tasks. This occurs due to data exchanges and synchronizations at the boundaries of LET intervals. The degree in which the load is increased depends on the amount of data that is copied to local variables at the beginning of LET intervals and the amount of data that is copied to global variables at the end of LET intervals. In PTP, to guarantee that the copy operations occur atomically at the boundaries of LET intervals, synchronization mechanisms are used, which, on the other hand, increase further the system's load. The additional load degrades the execution performance and leads to a system composition where most of the processor capacity is used to execute data flushes between memories and not for the actual execution of the application functions. Furthermore, the extra load decreases the possibility to find a feasible task schedule and increases the response times of tasks. Therefore, this goal focuses on designing an alternative buffering protocol that reduces the load caused by buffering operations and enables a more efficient scheduling of tasks. Specifically, this goal focuses on a static, wait-free global buffering strategy that reduces the buffering load by avoiding physical data exchanges and synchronizations at the boundaries of LET intervals.

[Goal 3] Optimization of the memory capacity for LET applications—The integration of LET paradigm into automotive systems, through any buffering protocol, requires additional memory capacity for storing the buffers added to preserve the LET semantics. The memory demands of LET are protocol dependent and increase with an unknown growth of application's complexity. Although multi-core processors are equipped with memories of adequate space, the PTP highly increases the memory space to store all data elements because it requires approximately $(N_R + N_W) * D$ additional memory for every data element, where N_R and N_W are respectively the number of reader and writer tasks and D is the data size of the data element. In a global buffering strategy [10], less memory is required because the reduction of LET intervals that overlap between LET tasks decreases the amount of buffer elements. Therefore, this goal targets the minimization of buffer memory via a buffering strategy, that is independent of the number of reader and writer LET tasks and that is sensitive to timing parameters such as the period and the duration of LET intervals. Moreover, strategies that focus on reducing the amount of buffers based on application characteristics and timing requirements are part of this goal.

[Goal 4] LET buffering for specific automotive systems characteristics—Despite the increase of system's load and high demands for memory capacity, PTP offers several benefits. PTP is easier to integrate in existing applications due to its simplified semantics. It handles at run-time the task-to-task communication independent of task's triggering pattern. Therefore, the focus of this goal is to identify application characteristics and use cases for which PTP is more practical to use than the proposed protocol. A comprehensive definition of PTP targeting the shortcomings of related work are also part of this goal.

[Goal 5] Scheduling of LET tasks—The schedule of LET tasks must be feasible regardless of the buffering protocol used to ensure LET semantics. To accomplish LET semantics, tasks must finish their execution within the boundaries of their LET intervals. If tasks execute beyond the end of their LET intervals, then outputs cannot be provided on time and, hence, LET semantics and functional correctness are violated. Incorrect execution of LET tasks impacts not only the time when they finish their execution, but also the correctness of the LET buffering. The data exchange and synchronization at the boundaries of LET in PTP is handled by so-called *communication* tasks. To provide a correct data exchange, the communication tasks must be executed in a defined order and finish without interruptions by any other task. Therefore, designing the scheduling of LET tasks during integration of LET paradigm into automotive systems is a crucial step.
This goal targets the automatic schedule synthesis of *Fixed-Priority Scheduling (FPS)* and *Time-Triggered Scheduling (TTS)* mechanisms to guarantee LET buffering correctness and fulfill several requirements. Specifically, it targets automatic schedule table generation of TTS and priority assignment of FPS. The FPS scheduling is a widely used approach in automotive systems due to its flexibility to handle dynamic changes and nondeterministic task arrival times of highly event-based applications. Compared to FPS, the TTS [11] offers benefits such deterministic execution of LET tasks and the ability to control overheads and better load distribution during the scheduling design.

This goal targets two different scheduling mechanisms to observe the advantages and practicality of each approach for LET systems. Finally, an automated approach to generate the schedule is intended to reduce the effort and complexity of scheduling design during the development of LET systems.

[Goal 6] Optimization of LET task scheduling—The OS handles task activation through a timer interrupt. The execution of the timer interrupt and of OS operations defined to activate, schedule, and terminate tasks affect the system's total utilization and timing requirements in two ways. Firstly, these operations take an amount of execution time, which is an additional run-time in the system. Secondly, the OS operations and the timer interrupt execute on a high priority level, which causes, in the worst-case, high preemption and start time delays to active tasks. These delays impact the response times of tasks and affect the end-to-end delays in general. Hence, it is essential to consider them during schedule construction in order to fulfill timing requirements under realistic conditions. Therefore, this goal targets these delays as part of the schedule generation and validation.
Scheduling decisions increase in general the overall system's load through the induced preemption overheads caused by context-switching between tasks. A high amount of preemptions decreases the execution performance of the system and leads to deadline violations and to an utilization that exceeds the processor's full potential. Therefore, this goal is to construct an optimized schedule of tasks for TTS and FPS approaches, considering task's activation, preemption, and termination overheads. This work focuses on minimizing the overall preemption overheads by reducing the number of preemptions caused by higher-priority tasks and the timer interrupt, and addressing the respective delays when defining the priorities and the start and end times of tasks. The aim is to validate the schedulability of tasks taking into account these overheads and delays during schedule construction, rather than through expensive schedulability tests that cannot be efficiently integrated into optimization algorithms and do not provide optimization capabilities.
A significant advantage of designing the schedule of tasks considering these overheads and delays is to avoid to an extent possible differences of the task execution planned at the design time and the one executed on the real target. In this way, unexpected violations of timing requirements are avoided and the required resources to ensure a feasible schedule are realistically planned.

This work is based on the following assumptions:
A system composed of periodic tasks that communicate with each other via shared memory (global or local) under the LET semantics is considered. The distributed communication is not considered but assumed that data shared between tasks executing on other *Electronic Control Unit (ECU)s* is done such that the data is sent before or at the end of producer task's LET interval and received before or at the start of read of consumer task's LET interval, depending on whether this data uses LET semantics. A-periodic and sporadic tasks are assumed to not execute on the same core as periodic tasks. Communication between periodic LET tasks and other non-LET sporadic tasks

is not explicitly considered and evaluated. It is assumed that this communication is done with other communication mechanisms and not via LET, and it is assumed that the communication costs are part of the *Worst-Case Execution Time (WCET)* of the tasks. It is assumed that data elements exchanged via LET are exchanged only between periodic tasks. In buffer synthesis, it is assumed that tasks call functions with unique names. In legacy systems, this is not the case, as different tasks call the same functions, but at different levels of their call tree. Finally, this work considers a system where the WCET is predictable and estimated considering as well multi-core effects.

1.3 Contributions

This work provides the following key contributions.

C1 *Resource efficient buffering mechanism for LET systems* – The proposed *Static Buffering Protocol (SBP)* [12] reduces communication overheads and improves execution performance by avoiding the physical data exchanges at the boundaries of LET intervals. It reduces memory demands of LET by enforcing a global buffering mechanism and by suppressing writes of unnecessary outputs and of outputs that satisfy data age constraints. The SBP protocol was designed and developed during the OBZAS research project ("Optimiertes Buffering für Zeitgesteuerte Automobile Software") funded by the Bavarian Research Foundation [13]. The research activities regarding SBP were carried out equally by the project partners. Therefore, SBP was published together in the paper [12].
A comprehensive definition of the PTP buffering approach as an alternative way to integrate the LET paradigm into automotive systems considering specific application characteristics is given. The detailed description in this work addresses the shortcomings of PTP described in the related work. Moreover, advantages of PTP are derived and its buffering performance is evaluated against the proposed SBP. This contribution accomplishes Goal 1 - Goal 4.

C2 *Resource efficient scheduling of LET systems* – Approaches to automatically synthesize and optimize the schedule of LET tasks for TTS and FPS strategies are given. The proposed automatic overhead-aware synthesis algorithms are designed to fulfill simultaneously timing, performance, resource, and LET requirements. They are designed to optimize the schedule, i.e., to minimize preemption overheads by reducing the number of preemptions. To handle the impact of start and preemption delays, the schedule is constructed considering timer, terminate, and context-switching overheads. Unlike related work, approaches are proposed to validate the schedulability of tasks considering these overheads during schedule generation, rather than through expensive schedulability tests that cannot be efficiently integrated into optimization algorithms. This contribution accomplishes Goal 5 and Goal 6. Parts of this contribution are published in [14, 15].

C3 Case studies conducted considering characteristics of industrial applications and the extensibility of these applications with new functionalities are given. Considering that the *AUTomotive Open System ARchitecture (AUTOSAR)* standard is successfully used in automotive systems, part of this contribution is the integration of the LET paradigm into AUTOSAR software architecture as a use case to show the practicality of LET. In this way, real challenges in integrating LET into highly event-driven classic AUTOSAR systems are identified. The practicality of LET for automotive systems is demonstrated through a case study with a real world *Antilock Braking System (ABS)* running on a multi-core ECU. This contribution targets the feasibility of contributions targeting Goal 1 - Goal 6 in the industrial environment.

In this work, a tool that synthesizes buffering information of an automotive application was developed to evaluate the memory consumption and the execution performance of SBP and PTP. The *TA.Simulation* option of TA Tool Suite [16] was extended to simulate the buffering behavior of SBP and PTP. Similarly, a tool was developed to generate the schedule of LET tasks that use SBP and PTP buffering protocols to exchange data. The tool constructs the schedule for each TTS and FPS scheduling mechanism separately. In order to show the practicality of LET for industrial in-vehicle application, the *Runtime Environment (RTE)* of an existing AUTOSAR OS is extended to support LET buffering.

1.4 Outline

An overview of the key contributions addressed by each chapter is provided in Table 1.1. The remainder of this work is structured as follows. Chapter 2 introduces the fundamentals of real-time embedded systems, the software application requirements, multi-core effects, the LET paradigm, and the AUTOSAR standard. Chapter 3 describes the integration and optimization strategies of LET in automotive applications in terms of data exchange and determinism. It provides a comprehensive evaluation

	[C1]	[C2]	[C3]
Chapter 1			
Chapter 2			
Chapter 3	✓		✓
Chapter 4		✓	✓
Chapter 5	✓		✓

Table 1.1: Overview of contributions targeted in each chapter.

of the proposed strategies using models with characteristics of industrial applications. The scheduling of LET tasks and optimizations in terms of scheduling overheads are covered in Chapter 4. The schedule construction of two scheduling approaches and their related work is described throughout the chapter. The comprehensive evaluation of both approaches emphasizes the benefits and the degree in which the determinism of LET is fulfilled by each scheduling technique. Chapter 5 describes the integration of LET paradigm into the software architecture of classic AUTOSAR systems. The practicality of LET in automotive applications is shown by conducting a case study using a real world AUTOSAR in-vehicle application executing on a real platform. Chapter 6 concludes this work and outlines the future work.

2 | Fundamentals

This chapter describes the fundamentals of embedded real-time systems. An overview of hardware and software platform of modern *Electronic Control Unit (ECU)*s is introduced briefly. The chapter describes the impact that multi-core processors have on the time and dataflow determinism, performance, and design of in-vehicle applications. The fundamentals of the *Logical Execution Time (LET)* paradigm are described, as well as the advantages LET brings to the development of in-vehicle applications. Finally, the basic concepts of the classic *AUTomotive Open System ARchitecture (AUTOSAR)* systems are introduced.

2.1 Embedded Real-Time Systems

An embedded real-time system is composed of the *application software*, the *operating system* and the *hardware* components. The fundamentals of each component are described throughout this chapter.

2.1.1 Hardware Architecture

Multi-core processors are a special kind of multiprocessors and consist of at least two or more execution units, so-called cores, integrated in a single processor socket and interconnected via crossbars or dedicated bus networks. They provide more computation power than single-core processors through the parallel work of two or more cores. The advent of multi-core is driven by the need of more computation power. Increasing of the single-core processors frequency leads to more energy consumption and decreases the longevity of the device. Therefore, in order to handle the increasing need of computation power, multi-core processors are invented.

A multi-core processor has three classes of memory design. In *distributed memory* architecture, each core has its own local memory. In *shared memory* architecture only one global memory exists and it is accessed by all cores via a shared network. The *hybrid* design is a combination of shared and distributed memory architecture. In this architecture, all memories (local and global) are accessed by all cores, but with dedicated crossbars. Hence, cores access their local memory with a faster crossbar than the local memories of other cores or the global memory. In terms of processing speed, the multi-core processors are classified in *homogeneous* and *heterogeneous* types.

In the first type, the cores have identical instruction sets. Whereas, the heterogeneous processors have different types of instruction sets for cores. The multi-core design is further classified by the way cores operate. In asymmetric multiprocessing (AMP), cores run independently and in symmetric multiprocessing (SMP) it is the operating system that arbitrates core operations. This work assumes that the multi-core processor is either homogeneous or heterogeneous. Their impact is abstracted only in the *Worst-Case Execution Time (WCET)* of all functions of the application software. This work assumes that the accessing delays of local memories are lower when accessed by the local core and higher when accessed by other cores. Similarly, the delays for accessing shared memory is higher than accessing the local memory of the core. These delays have an extensive impact on the communication run-time between functions of the application software.

2.1.2 Software Architecture

The embedded *application software* describes a set of functionalities implemented by several functions. Examples of these applications are Braking Assistance, Engine Control Management and Pedestrian Recognition applications that are found in modern cars. Functions of the application software are mapped to different *tasks*. Tasks are the schedulable units recognized by the *Operating System (OS)*. They are executed periodically, sporadically, or a-periodically. Periodic tasks are activated and executed repeatedly every fixed *"periodic"* time. The arrival of sporadic and a-periodic tasks is unpredictable and only known at run-time. For sporadic tasks their minimum and maximum arrival time is known at design phase, which is used to evaluate their impact on the execution of periodic tasks. This work focuses only on periodic tasks.

2.1.2.1 Safety Composition

Faulty behaviors of embedded systems occur due to hardware and software malfunctions. In safety standards, safety integrity levels [17] are defined to indicate the consequences caused by such failures. Safety requirements are specified and guaranteed at design and implementation via *fault isolation* and *fault tolerance* handling. Fault isolation is ensured via partitioning of applications, such that critical parts of the system remain unaffected by the hardware or software failure. Fault tolerance ensures that failures are handled at run-time, such that software application, i.e., allocated in the failed hardware, continues execution in a correct manner. Fault tolerance is employed by replicated functions running on multiple processors. Hence, the application software is composed by several applications, each mapped to dedicated partition. Partitions are a safety pattern, specified in ARINC - 653 [18] and ISO 26262 standards [17], to isolate spatially and temporally critical applications and protect them against faulty functions. Partitions have dedicated resources such as memory and processor time. *Spatial* partitioning determines that functions of different partitions must not interfere with each other in terms of memory (functions operate on memory regions dedicated to the partition they belong). *Temporal* partitioning ensures that shared resources, e.g., processor time are strictly adhered to a partition, and no interferences occur. Each partition takes a dedicated time-slot during execution that is not shared among other partitions. Partitioning protects critical parts of the application software.

In AUTOSAR OS [19], temporal partitions are implemented in the form of *timing protections*. For instance, task overruns are ensured by the *time budget* concept of the timing protection mechanism. A time budget is assigned to the execution time of every task and if any task exceeds this budget the OS takes the necessary actions (e.g., by forcibly terminating the task). In this work, a safe upper bound of WCET for every task is assumed, which corresponds to the time budget. This work construct the schedule of tasks under this assumption, but it does not consider the run-time of OS to handle time budget violation. In typical safety systems, criticality levels are assigned to tasks to indicate the consequences of failure to execute those tasks. For instance, high-criticality (HI) tasks of an application should be free from interference and any non-occurrence of these tasks leads to catastrophic consequences. The zero-interference execution of high critical tasks must be as well guaranteed at schedule construction. The constructed schedules typically go in the process of certification and are certified only if task requirements are fulfilled. Schedules of low-critical tasks do not require certification although these tasks must as well meet their deadline requirements. In this work, safety is not explicitly addressed, but it is assumed that all tasks must meet their deadlines regardless of their criticality.

Virtualization is an alternative method to ensure zero-interference between software applications of different criticality levels. Hypervisors [20] isolate critical applications in safety-critical OS and non-critical ones in other OS. All OSs run in the same virtual machine, but each having dedicated resources.

2.1.2.2 Communication

Embedded functions are highly interconnected. They exchange data in different ways such as for instance by means of shared memory or by means of messages. In the shared memory method, tasks exchange data by reading and/or writing the same data elements. These data elements are allocated to shared memory components, such that they are accessed by all cores to which tasks are allocated for execution. Two tasks exchange data if one task produces and the other consumes the same global data element. The task that consumes the data is referred to as *reader task* and the one that produces the data is referred to as *writer task*. Data protection, i.e, consistency mechanisms [5, 21, 22] such as *lock-based*, *lock-free* and *wait-free*, are proposed to avoid the read of unstable data due to concurrent read and write operations on data elements shared among different tasks. Locks allow exclusive operating rights on data elements. While a task accesses a shared data element, it holds the lock in order to

avoid concurrent accessing and guarantees that it is operating in a consistent value. This approach can cause long blocking delays and can lead to deadlocks. The *Priority Ceiling Protocol (PCP)* is used [23] to avoid deadlocks for tasks in single-core systems. Lock-free approaches [24] are used to allow accesses on data elements without holding a lock. At the end of the access, reader tasks check whether another task updated concurrently the data element while consuming the data and retries until a stable value is consumed. The number of retries can be unpredictable, but an upper bound is defined in [24, 25] to estimate its impact in the WCET.
In wait-free methods, the readers do not wait for the writers to finish and the writers are not blocked by the readers. Instead, different versions of the data are stored in buffers for every data element. Reader tasks read the latest version of the data from the respective buffer element. This approach has the disadvantage of requiring more memory for buffer storage, but it ensures exclusive operations on data without deadlocks or unnecessary retry delays. To reduce the memory required to store buffers, buffer elements not consumed by any reader task are used to store new versions of data produced by writer tasks.

Tasks executing in different ECUs, communicate via distributed bus in form of messages using CAN, FlexRay, or Ethernet protocols [26]. Distributed communication distinguishes between *Time-Triggered (TT)* and *Event-Triggered (ET)* approaches of transmitting the messages. In the time-triggered approach, e.g. with the FlexRay protocol, the transmissions are specified in a schedule table in which time slots define the time for sending messages. In the event-triggered approach, e.g. with the CAN protocol, messages are arbitrated using a fixed priority assigned to each message. The tasks of the different partitions communicate with each other either via inter-partition communication or via a distributed bus. In general, the communication between partitions of the same ECU happens via buffers, where the data is transferred by the kernel OS from one partition to another. Only the kernel OS knows the memory ranges accessed by both partitions and it is the OS performing this action. The drawback of this approach is that buffer overruns might occur. Another method is the shared buffer approach (buffer is located in shared memory), in which sender/receiver partitions can read from or write to. This case is more efficient because the kernel is not involved.

This work focuses only on task communication based on shared memory approach using the LET paradigm [7].Two LET communication protocols for data exchange between tasks are described, the *Static Buffering Protocol (SBP)* and *Point-to-Point Protocol (PTP)*. Let $S = \{S_s | s \in \mathbb{N}^+\}$ denote the set of all global data elements in the software application exchanged using SBP or PTP. Each data element has a primitive type (e.g., Integer, Float, Boolean) or composite type (e.g., Array, Struct). This notation of data elements is used throughout this work.

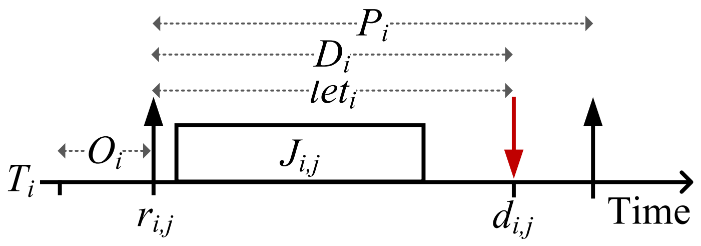


Figure 2.1: Attributes and scheduling parameters of task T_i its job $J_{i,j}$. The first vertical black arrow indicates the release time $r_{i,j}$ of job $J_{i,j}$ and the red arrow the absolute deadline $d_{i,j}$. The release time $r_{i,j}$ is defined based the offset O_i and period P_i.

2.1.2.3 Tasks Composition

The embedded real-time system considered in this work consists of $n \in \mathbb{N}$ periodic tasks $\tau = \{T_1, ..., T_n\}$ and $m \in \mathbb{N}$ processing units, i.e., cores $C = \{C_1, ..., C_m\}$. Each task $T_i \in \tau$ is activated periodically at period $P_i \in \mathbb{N}$ with an offset $O_i \in \mathbb{N}$, has deadline $D_i \in \mathbb{N}$ equal to the LET interval duration let_i (where $D_i \leq P_i$), and is mapped for execution to any of the cores in C, where $1 \leq i \leq n$. The WCET of T_i is defined as $wcet_i$. Each task T_i is instantiated to $n_i \in \mathbb{N}$ number of jobs within duration hp of the *Hyper-Period (HP)*. The hp is calculated as the *Least Common Multiple (LCM)* of periods of tasks $hp = lcm(\{P_1, P_2, ..., P_n\})$ and $n_i = \frac{hp}{P_i}$. The HP defines the time interval of identical repetition of the activation pattern of tasks. Let $J_{i,j}$ define the j^{th} job of T_i, where $j \in [1, n_i]$. Let $r_{i,j}$ and $d_{i,j}$ be the release time and the absolute deadline of each job $J_{i,j}$ of each task $T_i \in \tau$, respectively, defined as

$$r_{i,j} = O_i + (j-1) * P_i, \tag{2.1}$$

$$d_{i,j} = r_{i,j} + D_i. \tag{2.2}$$

The absolute deadline $d_{i,j}$ also corresponds to the end of the LET interval of job $J_{i,j}$. The above attributes of the task T_i^C are shown in Figure 2.1.

Tasks are divided into different categories based on their deadlines and offsets. Tasks with deadlines equal to their period ($P_i = D_i$) are referred to as tasks with *implicit* deadlines. Tasks with deadlines less than the period ($P_i < D_i$) are referred to as tasks with *constrained* deadlines and the ones with deadline greater than the period ($P_i > D_i$) are referred to as tasks with *arbitrary* deadlines. Based on the offset, tasks are divided into *synchronous* and *asynchronous* tasks. Synchronous tasks have all the same offsets and asynchronous tasks have different ones.

2.1.2.4 Timers and Task Activation

In embedded microcontrollers, timers are special counter registers that measure the progression of time. Their main purpose is activation of tasks or setting of events. Timers are incremented at each processor cycle or at each prescaler register overflow. Prescalers are dedicated registers that are used to measure higher values of timers. When counter reaches its configured maximal value, it overflows and an interrupt is released, referred to as *timer interrupt*. The timer interrupt is executed in the context of the OS and checks if any task must be activated or if any event must be set. The time required for the counter to overflow defines the period of the timer. Timer's period has a high impact on activation time of tasks. The timer interrupt causes task activation delays, referred differently to as *activation jitter*, because of its period and its run-time. Nevertheless, timers could be configured such that activation delays are reduced. For example, the timer should have a period of 1 ms in order to activate every job of tasks T_{1ms} (period 1 ms) and T_{2ms} (period 2 ms). If the timer's period is configured 2 ms, then every second job of T_{1ms} is not activated before the release of the next job. If the timer's period is configured 1.5 ms, then the maximum activation jitter for both tasks is the run-time of timer interrupt plus 1.5 ms.
In typical embedded multi-core microcontrollers, each core has its dedicated counter that is used to activate tasks that run on that core. Two types of timers are defined to activate tasks: the hardware and the software timer. Unlike the hardware timer, the software timer must be incremented manually in the software. However, the hardware timer is used as a baseline in the software timer too. A software timer can be a periodic or a high-resolution timer.

Figure 2.2 shows an example of three tasks activated by a periodic timer. The period of the timer is set to 4 ms, which defines that every 4 ms the timer interrupt *Timer* is released to activate tasks. It checks for arrived events, which indicate a request to activate tasks, and activates the requested tasks. Task T_0 has a period of 4 ms and is activated by the timer every 4 ms but with an insignificant activation jitter, defined by the time difference between the red and black arrows, which is due to the point of time the timer interrupt sets the task in active state during its execution. Task T_1 has a period of 5 ms and is activated by the timer irregularly, i.e., at a higher activation jitter than of task T_0. This occurs because the 4 ms period of the timer does not harmonize with the period of task T_1. For instance, the activation of the second job of T_1 happens only by the third release of the timer interrupt between 8 ms – 9 ms (indicated by the red vertical arrow). This causes a maximal activation jitter of 4 ms for the second job of task T_1. Similarly, task T_2 is activated by a large activation jitter as in the case of task T_1. To avoid large activation jitters, the timer's period must be configured properly. For instance, in the example of Figure 2.2 a timer period of 1 ms would activate tasks T_1 and T_2 at correct times without major activation jitters. However, this configuration has other drawbacks. Triggering the timer to often can cause high interruption load, which on the other hand degrades the performance and can lead to deadline violations of tasks. Hence, if the timer's period is set to 1 ms, then tasks are

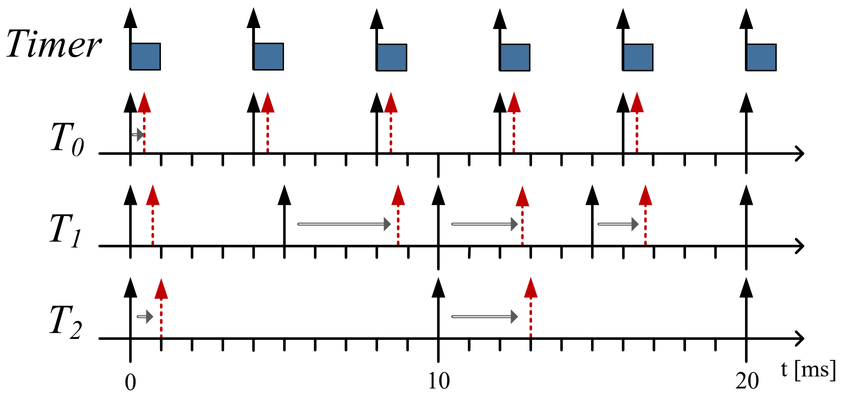


Figure 2.2: Periodic Timer. Tasks T_0, T_1, and T_2 are activated by the periodic timer interrupt *Timer*. The period of *Timer* is set to 5 ms. The vertical black arrows indicate the expected activation times of tasks and timer interrupt. The red arrows indicate the actual activation of tasks. The horizontal black arrows indicate the shift of activation of tasks. The solid blue colored boxes indicate the execution of the timer interrupt. Activation jitters of tasks are caused by the period and execution of *Timer*.

activated at correct points of time but with the drawback that no time is left for tasks to execute because the processor is occupied mainly by the timer interrupt. Therefore, finding the correct period of the timer in an application with non-harmonic periods is not only challenging, but often a trade-off between lower activation jitters and higher execution load caused by the timer.

To handle drawbacks and configuration challenges of periodic timers, the high-resolution timers are defined in AUTOSAR [19] to activate tasks without major delays and without causing high interruption load. The maximum value of the counter of a high-resolution timer is reset every time the timer reaches its maximal value to a value that might differ from the previous one. Therefore, these timers know the next activation event of the task and the timer interrupt is triggered exactly at these times. Figure 2.3 shows the example of tasks T_0, T_1, and T_2 activated by the high-resolution timer. The maximal value of the counter is assigned such that the timer interrupt is triggered at times 0 ms, 5 ms, 6 ms, 8 ms, 10 ms, 12 ms, 15 ms, 16 ms and 20 ms, which correspond to the planned activation times of tasks. The maximal activation delay of 1 ms is unavoidable due to the run-time of the timer interrupt. High-resolution timers reduce the high execution load caused by periodic timers in applications with non-harmonic periodic tasks, but in practice do not enable high accuracy in task activation and have a higher execution load per instance of the timer interrupt. This is due to their internal algorithm for determining the next activation point.

Timer interrupts have a higher priority than computation tasks and can preempt tasks during execution. These preemptions lead to preemption delays and context-switching overheads, which affect execution performance of the system and task response times. The impact of timers on tasks activation and scheduling is an essential part of this work. Because high-resolution software timers cause fewer interruptions,

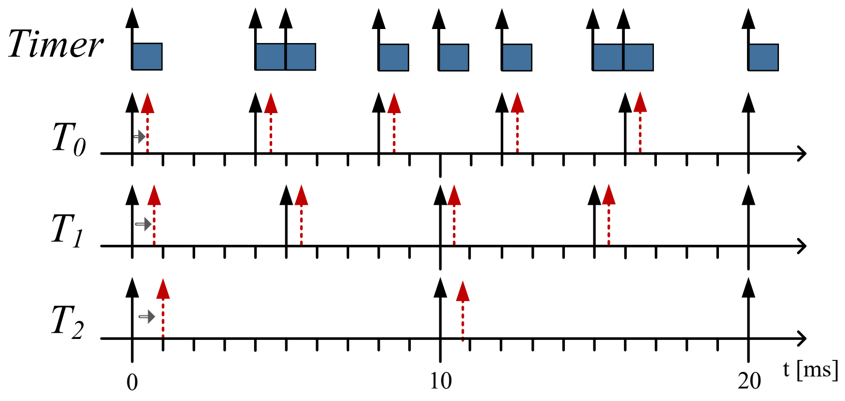


Figure 2.3: High-Resolution Timer. Tasks T_0, T_1, and T_2 are activated by the high-resolution software timer handler *Timer*. The activation of *Timer* is not based on a fixed period, but is defined by setting the maximal value of the timer each time the timer overflows. The vertical black arrows indicate the expected activation times of tasks and timer interrupt. The red arrows indicate the actual activation of tasks. The horizontal black arrows indicate the shift of activation of tasks. The solid blue colored boxes indicate the execution of the timer interrupt. Activation jitters of tasks are caused only due to the execution of *Timer*.

shorter activation delays for non-harmonic periodic tasks, and lower execution load, they are used to activate tasks of applications targeted in this work.

2.1.3 Real-Time Scheduling

Scheduling is the component of an embedded real-time OS that determines the execution of tasks on processing units. It has an enormous impact on satisfaction of the real-time requirements. In multi-core systems, its definition is often associated with *allocation*, that defines the assignment of tasks to cores or processors. The *scheduling algorithm* defines standard rules that determine the execution order of jobs. The direct outcome of a scheduling algorithm is a *schedule*, which consists of a list of jobs with their respective order of execution. A schedule is valid if all the jobs meet their deadlines. An optimal scheduling algorithm can schedule any feasible task set. A task set is *feasible* if at least one valid schedule exists. Schedule feasibility is evaluated through schedulability analysis which are constructed based on complex tests, so called schedulability tests. They verify that for a scheduling algorithm and a given task set a valid schedule is found. These tests are classified as *sufficient*, *necessary* and *exact* [27]. Schedulability tests are often hard to construct as they consists of mathematical equations that denote the relation between several attributes of tasks and the platform. They are often constructed based on *Worst-Case Response Time (WCRT)* formulations. WCRT analysis construct worst-case bounds for response times of tasks, but because they are pessimistic they do not represent the reality and are hard to apply. Alternatively, simulation [16] is used to check schedule feasibility and deadline violations. Although highly practical in multi-core case, verification using simulation

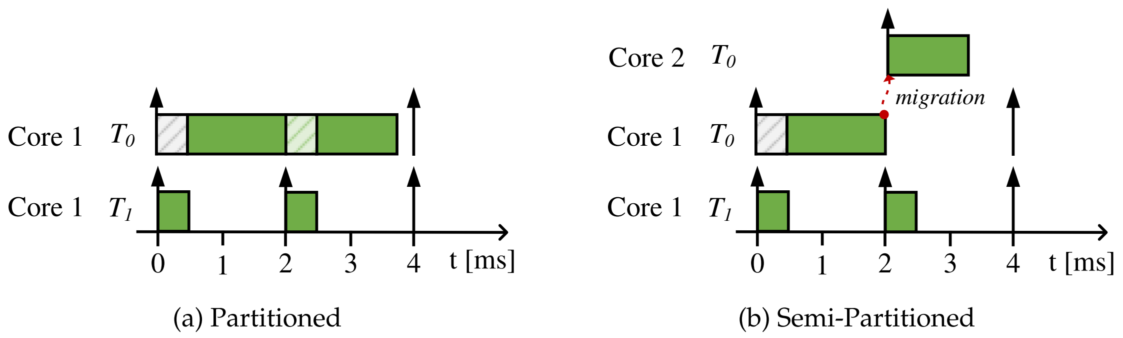


Figure 2.4: Example of **(a)** partitioned and **(b)** semi-partitioned scheduling. The light green box indicates the preemption time of a task. The gray colored boxes indicate the start delay. The red arrow indicates a migration of the task to the other core. The black arrows indicate the release of tasks.

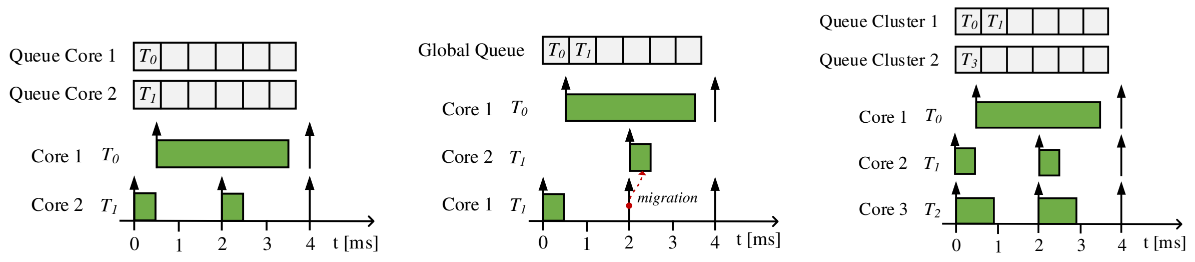
is not constructive compared to WCRT analysis because they do not consider all execution paths of the system.

2.1.3.1 Scheduling classification

The following paragraphs provide a summary of scheduling types categorized based on various aspects.

Partitioned vs. Semi-partitioned—This category concerns the resource allocation of tasks. In *partitioned* scheduling, tasks are assigned for execution to a core and all its jobs execute only on this core. *Semi-partitioned* is a derivative of partitioned scheduling. It allows a subset of tasks to migrate for execution onto other cores. Migration refers in this context to the reassignment of a task to a core that is different to the initially assigned one. The migration of a task occurs either before starting its execution on the initially assigned core or after it is preempted and resumes on the other core. The aim of migration is to improve schedulability and to balance the load between cores.

An example of two tasks T_0 and T_1 that are scheduled using partitioned and semi-partitioned scheduling is shown in Figure 2.4. The execution of T_0 and T_1 based on partitioned scheduling is shown in Figure 2.4a. Task T_1 is allocated on the same core as T_0 and has the highest priority. It preempts task T_0 at time 2 ms. After termination of T_1, task T_0 resumes on the same core and continues execution until termination. Figure 2.4b shows the execution of tasks T_0 and T_1 considering semi-partitioned scheduling. Task T_0 resumes on Core 2 after its preemption at time 2 ms by task T_1. Although the practicality of semi-partitioned scheduling has been evaluated [28, 29], it must be used occasionally to keep migration overhead low. Compared to semi-partitioned, the partitioned scheduling lacks sufficient utilization of cores and it does not scale well with the increase of the number of cores. In practice, it is



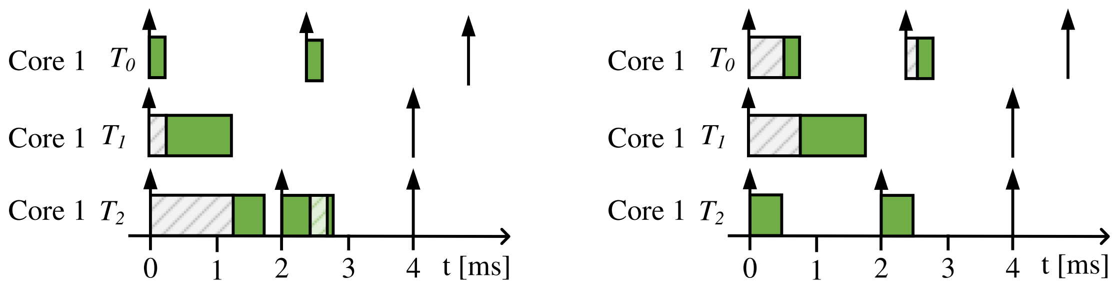
(a) Local (b) Global (c) Clustered

Figure 2.5: Example of **(a)** local, **(b)** global, and **(c)** clustered scheduling. The green colored boxes indicate the execution of tasks.

advised that a software application is designed in such a way that tasks are divided into multiple tasks and scheduled via partitioned scheduling [30] and are allocated to different cores at design time. This avoids migration overheads and increases the predictability. The partitioned scheduling is the widely used in industrial automotive systems and is the approach used in this work to allocate tasks to cores.

Local vs. Global vs. Clustered—In *local* scheduling, every core has its own scheduler. Each task, mapped for execution to a core, is scheduled by the local scheduler of the core. In *global* scheduling, tasks are scheduled by one scheduler and all active jobs are stored in a global queue. Example of global scheduling algorithms are Pfair, EDZL, LLF, Global-EDF. The Pfair scheduling algorithm [31] is based on fairness and is classified as fluid scheduling strategy. Tasks are executed fairly, i.e., each task takes an equal quantum of execution time. This algorithm is optimal in multiprocessors for periodic tasks with implicit deadlines and performs well in up to 100 % load. However, it is impractical due to high migration overheads.
Optimal schedulers, successors to Pfair, are PF, PD, PD2, ERFair, and PFair staggered model. The EDZL (Earliest Deadline until Zero Laxity) [32] assigns the highest priority to tasks with zero laxity. Tasks are not preempted during execution. Similarly, LLF (Least Laxity First) assigns priorities based on laxity. Laxity defines the time difference between the deadline and the WCET of a task. LLF is optimal for uniprocessor case and performs well in terms of schedulability, but again suffers from high migration overheads. Global-EDF is not optimal for multiprocessors [33]. *Clustered* scheduling is a hybrid approach between local and global scheduling. It consists of clustered sets of processors, where tasks are scheduled globally within a cluster.

An example of local, global, and clustered scheduling is shown in Figure 2.5. The application consists of tasks T_0 and T_1. Figure 2.5a shows their execution based on the schedule defined by the local scheduling. In this example, tasks are allocated for execution on different cores. Task T_0 is scheduled by the local scheduler of Core 1 and T_1 by the local scheduler of Core 2. Each scheduler has its own queue for storing ready tasks. Figure 2.5b shows the execution of T_0 and T_1 defined by a global scheduler, which handles allocation of tasks to cores by using a global queue to store ready tasks.



(a) Static (b) Dynamic

Figure 2.6: Example of **(a)** static and **(b)** dynamic scheduling. The green colored boxes indicate the execution of tasks. The gray colored boxes indicate the start delay. The light green box indicates the preemption time of a task.

The allocation of tasks to cores differs based on the available cores. For instance, the second job of task T_1 is activated at time 2 ms and is assigned by the global scheduler to execute on Core 2, instead of Core 1, in which the first job has executed. Figure 2.5c shows an example of tasks T_0, T_1, and T_2. Tasks T_0 and T_1 are scheduled by the clustered scheduling, named Cluster 1, which uses global scheduler to allocate and execute tasks on Core 1 and Core 2. Task T_2 is scheduled by the clustered scheduling, named Cluster 2, which is a local scheduler to allocate tasks to Core 3.

Static vs. Dynamic—A schedule is determined either completely before the system runs (*static*) or at run-time (*dynamic*). In multiprocessor systems, the *static* scheduling defines the fixed allocation of tasks to cores and the fixed order of execution of jobs at design time. The schedule is not changed during the run-time. This design is generally more efficient at run-time but not very flexible, because the complete schedule must be known before system's operation. In high load conditions, the low priority tasks could suffer from deadline violations due to strict partitioned design, i.e., the allocation of jobs to cores remains unchanged at run-time. *Fixed-priority* and *fixed-job* scheduling are static, in which a fixed priority is assigned respectively to tasks and jobs. Example algorithms are *Rate Monotonic (RM)* and *Deadline Monotonic (DM)*. RM assigns priorities based on the rule "smallest period - highest priority" and DM based on "earliest deadline - highest priority".
Dynamic scheduling constructs the schedule at run-time and can change it dynamically. For instance, priorities are assigned at run-time via RM or DM heuristics. Compared to static scheduling, they are more flexible and resource efficient. But in hard real-time systems they add unpredictable behavior and are hard to verify. Additionally, they are characterized by a high number of task/job migrations that lead to increase of the utilization and deadline violations. Migration overhead impacts the performance, making this scheduling approach impractical.

An example of static and dynamic scheduling is shown in Figure 2.6. The application consists of tasks T_0 and T_1, which execute on the same core. Figure 2.6a shows their

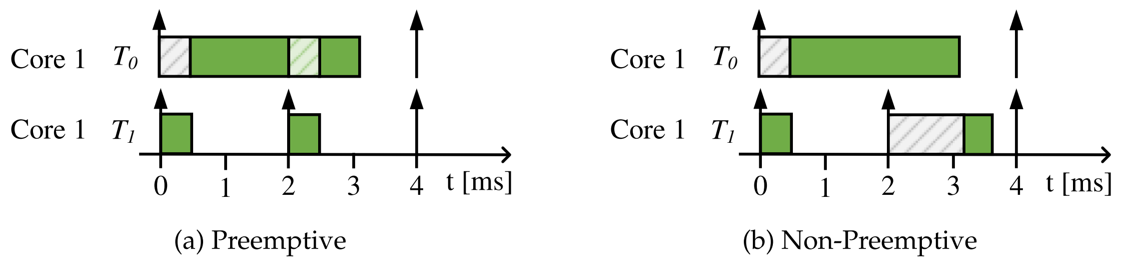


Figure 2.7: Example of **(a)** preemptive and **(b)** non-preemptive scheduling. The green colored boxes indicate the execution of tasks. The gray colored boxes indicate the start delay.

execution based on the schedule defined by the static fixed-priority scheduling. The priority ordering is $T_0 > T_1 > T_2$. Task T_0 has the highest priority is scheduled first for execution. At time 2.5 ms, it preempts task T_2. Figure 2.6b shows the execution of the same tasks but considering DM scheduling. Tasks have deadlines equal to their periods. The priority ordering is defined by deadlines at the times tasks are activated. For instance, at time 0 ms task T_2 gets the highest priority, because it has an earlier deadline than T_0 and T_1.

Preemptive vs. Non-preemptive—In terms of multitasking, scheduling algorithms are classified in *preemptive* and *non-preemptive*. In *preemptive* scheduling, higher priority tasks preempt lower priority tasks. The context of the preempted task is stored and the context of the new running task is loaded. Preemption guarantees deadlines for urgent tasks. A straightforward priority assignment of tasks is often through fixed heuristics such as RM and DM. RM assigns priorities based on periods of tasks, i.e., jobs with a frequent arrival are served first and often have a shorter response time. The DM, assigns priorities based on the deadline, such that tasks with the smallest deadline take the highest priority. *Non-preemptive* scheduling was developed as an approach to control data stability problems caused by preemption and to reduce the overheads caused by a high number of preemptions. Shifting the preemption in time protects critical data from instability state. Furthermore, non-preemptive regions decrease the number of preemptions and thus the context-switching overheads. Another benefit is that it avoids starvation of low priority tasks, which in fully preemptive scheduling suffer frequent deadline misses. The number and position of preemption points is defined at design time. They have to be optimal in such a way that high priority tasks meet their deadlines and the schedule is valid [34, 35]. Non-preemptive scheduling is known in the research community as limited preemption scheduling [36, 37].

Figure 2.7 shows an example of two tasks scheduled by preemptive and non-preemptive fixed-priority scheduling. Figure 2.7a shows the execution of tasks T_0 and T_1 (running on the same core) defined by preemptive scheduling. Task T_1 has the highest priority and preempts task T_0 at time 2 ms. After termination of T_1, task T_0 resumes on the same core and continues execution until termination. Figure 2.7b shows the schedule of the same tasks as of Figure 2.7a but considering non-preemptive scheduling. Despite of the higher priority, task T_1 does not preempt task T_0 at time 2 ms but waits until its termination.

Time- vs. Event-Triggered—Scheduling is further classified in TT and ET approaches based on how certain system activities are handled. In the TT approach, task activation and scheduling is initiated at fixed points in time, that are planned before the system is deployed. In the TT scheduling, the schedule is planned at design time and the start and resume times of jobs are stored in a schedule table. Based on this table the scheduler takes decisions such as when to start, preempt, or resume a job. In ET approach, tasks are activated based on the occurrence of events and are scheduled on-line. In ET scheduling, the schedule is defined, for instance, based on priorities that are configured at design time. But its the occurrence of events that triggers the scheduler to take scheduling decisions based on defined priorities. An example of such events is the periodic event occurring for activating a periodic task.

A detailed comparison of ET and TT approaches is given in [38, 39]. Table 2.1 lists the essential attributes of both scheduling approaches based on the summarized work in [38, 39]. TT has several advantages over ET, such as deterministic execution of tasks,

	Time-Triggered	Event-Triggered
Predictability	✓	○
Fault Tolerance	✓	○
Planning	✓	○
Analyzability	✓	✓
Verification	✓	✓
Load distribution	○	✓
Resource Utilization	○	✓
Extensibility	✓	✓
Efficiency & Flexibility	✗	✓

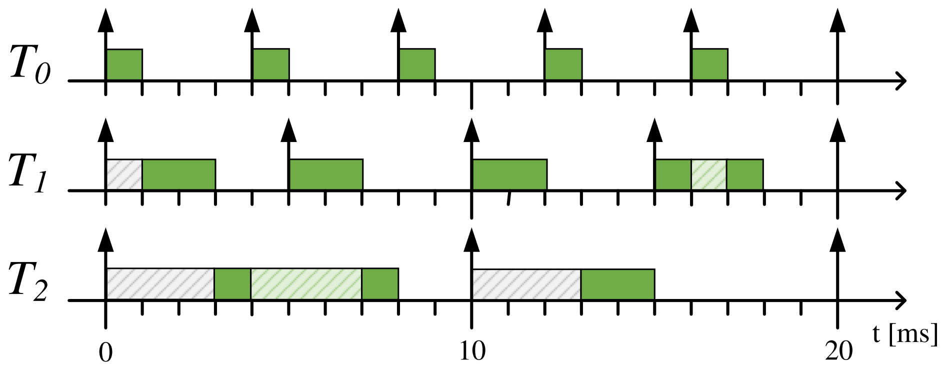
Table 2.1: Qualitative comparison given in [38, 39] of time-triggered and event-triggered scheduling approaches.
Legend: satisfied (✓), partially satisfied (○), or unsatisfied (✗).

simplified verification of timing requirements, control of scheduling delays, flexibility in software functionality extensions, and fault tolerance. In contrast, the execution behavior of ET scheduling is more efficient and flexible in terms of resource utilization, as it is able to dynamically handle unpredictable events such as sporadic and a-periodic events. Hybrid scheduling algorithms, that combine TT and ET approaches, are proposed to unify the benefits of both approaches. Examples such as slot shifting algorithm [40], sporadic server tasks and slack stealing algorithms [41] intend to schedule firm/soft sporadic and a-periodic tasks in a TT schedule. These algorithms search for available slots in which these tasks can execute. The slot shifting algorithm, for instance, is not practical for safety systems because it changes the schedule at run-time, which has gone at design time through the process of certification and intensive evaluation of all functional and non-functional requirements. Nevertheless, multi-core processors and future many-core processors, together with the hypervisor technology [20], offer the capacity to isolate both worlds in the same hardware and simultaneously get benefits of both paradigms without involving complex and resource consuming algorithms. The following sections describe the TT and ET scheduling approaches that are the focus of this work.

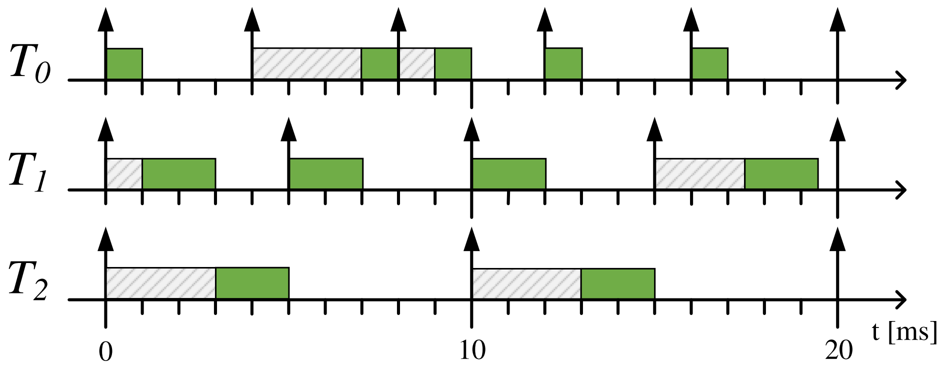
2.1.3.2 Focus of Scheduling

Scheduling algorithms found in practice and literature are a combination of scheduling classes described in the previous section. These algorithms are extensively studied in the research community and several approaches are proposed [42, 43], which either assume an independent system or are impractical for multi-core in-vehicle systems. Scheduling is often viewed as a mechanism to exploit the benefits of multi-core processors, such that the system requires as little redesign as possible, timing and data-age requirements are fulfilled, cores are sufficiently utilized, and scheduling overheads are kept minimal. An optimal scheduling algorithm must guarantee a feasible schedule and ensure robust execution of all tasks, taking into account unpredictable inputs from the environment at run-time. On the other hand, a deterministic schedule must be predictable and reproducible, both in design phase and target execution. Knowing the advantages of TT and ET scheduling approaches, this work focuses on *Time-Triggered Scheduling (TTS)* and *Fixed-Priority Scheduling (FPS)*. They are both partitioned, static, and preemptive scheduling strategies. FPS handles system changes better at run-time and TTS guarantees predictable execution of tasks. The FPS defines the order of execution of tasks running on the same core based on priorities assigned to every task. These priorities are defined statically at design time and do not change during execution. In TTS, tasks are scheduled by progression of time based on time occurrences defined in a schedule table. The duration of the schedule table is the HP of all periodic tasks. The activities (start and resume times of jobs) defined in the schedule table are repeated at a period with the same length as its duration.

Figure 2.8 shows an example of three tasks running on the same core. The execution of tasks in FPS follows this pattern: as long as a task is active, it is chosen by the scheduler for execution and if another task with lower priority is running on the core, it is preempted and the new task starts the execution. Figure 2.8a shows task execution using FPS. Task T_0 preempts task T_2, and T_2 resumes for execution after both active higher priority tasks T_0 and T_1 have finished execution. If priorities are not changed, the start and resume time of T_2 cannot be shifted to an earlier or later



(a) The order of execution between tasks is defined by the priority order $T_0 > T_1 > T_2$. The preemption of T_1 and T_2 is not avoidable for this priority assignment.



(b) The order of execution between tasks is defined in the schedule table. The schedule is constructed to have zero preemptions.

Figure 2.8: Example of **(a)** FPS and **(b)** TTS scheduling. The execution of tasks T_0, T_1, and T_2, defined by FPS and TTS scheduling approaches, is shown up to the hyper-period duration ($20ms$). The black arrows indicate the release of tasks. The deadlines are equal to the periods for all tasks. The solid green boxes indicate the execution of tasks. The light green boxes indicate the preemption time due to execution of higher priority tasks. The gray colored boxes indicate the start delay.

time, as it can be in the TTS approach. Figure 2.8b shows task execution using TTS. Tasks are not scheduled by a fixed order between them, but by start times constructed before their execution. By planning statically the TTS schedule, the execution of tasks can be constructed mainly non-preemptive such that preemption delays are reduced. As shown in Figure 2.8b, the preemptions of task T_2 are avoided in TTS by delaying the execution of tasks T_0 and T_1. In this way, the context-switching overheads are reduced.

2.1.3.3 Scheduling Overheads

Activation, terminations, and preemptions of tasks induce run-time overheads that impact their response times. These overheads are classified according to the activities that the timer and the OS takes to handle operations on tasks and consider them during schedule generation for minimizing their impact in the execution's performance of the application. These operations and their impact are described as follows. At every task's activation, the OS places the released job in the list of active tasks, allocates the stack and updates the task information in the process control block. These operations take an amount of execution time and execute in the context of the OS's timer interrupt. After tasks are activated, their active jobs are selected by the scheduler for execution. Each time a job is loaded for execution, the OS loads the context of the task to *Control Processing Unit (CPU)* registers. The context consists of the state of CPU registers, such as the state of *program counter*, the state of *stack pointer* and the state of *general* and *special* registers. During a task's preemption, the context information is stored for resuming the task's execution at the point of preemption. At every task preemption, the OS saves the context of the preempted task and loads the context of the new running task. In general, the context-switching run-time at every preemption is divided in the run-time required to *save the context* of the preempted task and in the run-time required to *load the context* of the new running task. The context-switching run-time is constant at every preemption point and it depends on the hardware architecture and the amount of registers that have to be saved/loaded. However, in several OS implementations it consists as well on operations such as for instance rescheduling decisions by the scheduler algorithm, updating of the control block of the process, and updating respective memory data structures.

Task preemptions are computationally expensive. They result in an increase of the processor time and memory size. High number of preemptions cause high context-switching overheads, which decrease execution performance of the system and cause unnecessary delays that increase the response time of tasks. Additionally, cache invocations occur each time tasks are preempted. The missing cache lines are reloaded from memory to cache. The *Cache-Related Preemption Delay (CRPD)* defines the time required to refill the cache at a task resume. It is assumed that CRPD is included in the calculation of the WCET and do not handle it explicitly during schedule generation. The higher the amount of tasks that are *preempted* at the same time the higher is the memory demand to store their stacks. Therefore, a high number of preemption must be avoided during schedule construction. Finally, when a job finishes its execution, the OS removes its information from the memory. The termination overhead consist of the time that the OS takes to terminate tasks.

2.2 AUTomotive Open System ARchitecture

AUTOSAR [1] is an open standard established in 2003 by several vehicle manufacturers. Its purpose is to handle the complexity of automotive Electronic/Electric systems by separating the application software from the ECU infrastructure. Through the concept of *Software Component (SWC)s* [44], its layered software architecture [45], and its well-defined communication semantics [46, 47], application functionalities are isolated and their integration into an ECU system is simplified. A SWC can be ported to different ECUs without affecting existing functionalities. Therefore, software providers reuse software parts and decrease the costs of developing systems for different vehicle manufacturers. AUTOSAR improves the interoperability of different development tools through its standardized model-based approach of describing the system.

2.2.1 Layered Software Architecture

AUTOSAR offers independence between software applications and the ECU platform through its layered architecture. AUTOSAR is divided in the following layers [45].

- ▷ The *Software Application Layer* [44] describes the application, sensor and actuator SWCs, and their interaction behavior.
- ▷ The *Runtime Environment (RTE)* [46] is the physical implementation of the *Virtual Functional Bus (VFB)*. It provides *Application Programming Interface (API)s* that handle the communication between SWCs and *Basic Software (BSW)* services [47]. These APIs are generated during ECU configuration, based on communication information defined in VFB in early development phases. The VFB describes all communication mechanisms between SWC*s* and their interfaces to the BSW services. Each ECU has its specific RTE implementation.
- ▷ The BSW [47] provides services such as memory, resource management, distributed communication, and I/O services to application SWCs through dedicated interfaces. It serves as a layer between RTE and the hardware platform. It is divided in *Services Layer*, *ECU Abstraction Layer*, *Microcontroller Abstraction Layer (MCAL)*, and *Complex Drivers Layer*. The Service Layer provides OS functionalities, network communication services, memory, diagnostic, ECU state,

[1]https://www.autosar.org/

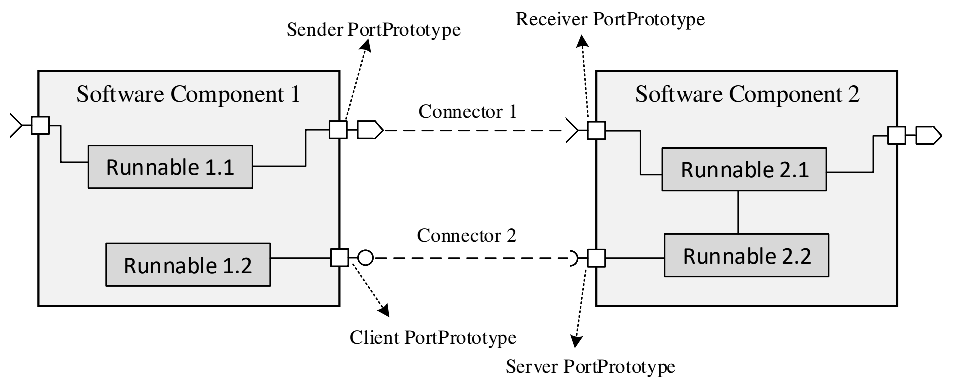


Figure 2.9: SWC elements.

and management services. MCAL provides independence of upper layers of AUTOSAR from the micro-controller. The ECU Abstraction Layer offers interfaces for accessing peripherals and devices. The Complex Drivers provide device driver functionalities that are not specified in AUTOSAR.

2.2.1.1 Software Application Concepts

Components—An AUTOSAR software application is composed by several SWCs. A SWC describes the functionality of a software application. Each SWC has well defined ports, which are used to enable the communication with other SWCs and BSW services. The internal behavior of SWCs is described by runnable entities, which have access to the interfaces of the SWC they belong to. An AUTOSAR RunnableEntity, i.e., runnable implements the algorithm or functionalities of the SWC they belong to. A SWC contains one or multiple runnables.
Different types of SWCs are specified in AUTOSAR. The *application* SWCs contain software application functionalities. It interacts with other types of SWCs and uses BSW services through dedicated ports. Further, the *sensor-actuator* SWCs contain specifics of a sensor and/or actuator. The *composition* SWCs is a form of application software component that groups logically several SWCs. It is used to hide the complexity of multiple SWCs and their communication and to simplify the architectural design of the software development. Figure 2.9 shows an example of two application SWCs. The Software Component 1 and 2 contain each two runnables, which exchange data through ports and interfaces.

Ports and Interfaces—In AUTOSAR, *ports* are communication interfaces between SWCs. They receive and provide data via *port interfaces*. Ports also have other purposes such as, for instance, to trigger runnables or mode switches. AUTOSAR distinguishes between application port interfaces and service port interfaces. The SWCs access BSW services via dedicated service port interfaces. The access to the hardware is enabled by BSW through AUTOSAR *Interfaces*. The following main port interfaces are defined:

▷ *Client-Server interface*, in which the server SWC provides an operation to one or multiple client SWCs. The client SWCs icall the operation. Through this interface ports receive and provide operations.

▷ *Sender-Receiver interface*, in which the sender SWC provides data and the receiver SWC receives data through the interface. Through this interface ports read and write data.

▷ *Parameter interface* enables SWCs to access constant or calibration data.

▷ *Trigger interface* allows SWCs to trigger other SWCs for activation.

▷ *Mode Switch interface* is used to notify a SWC for a mode switch.

The port of a SWC is connected to the port of another SWC via *connectors*. The data element is made available to the receiver port only if a connector exists within SWCs. Similarly, in Client-Server communication the server SWC receives a request by the client SWC only if their ports are connected.
The application SWCs of Figure 2.9 exchange data using one Sender-Receiver and one Client-Server interface. The output of Runnable 1.1 of Software Component 1 is provided to Runnable 2.1 of Software Component 2 through the Sender PortPrototype. The Runnable 2.1 receives the output of Runnable 1.1 through Receiver PortPrototype. Similarly, Runnable 1.2 invokes an operation request and receives the results of Runnable 2.2 via the Client PortPrototype. The Sender PortPrototype is connected with Receiver PortPrototype via Connecter 1. The Client PortPrototype is connected with Server PortPrototype via Connecter 2.

Runnables and Triggers—AUTOSAR runnables are activated by RTE [46] through the concept of *RTEEvents*. These events activate or wake up dedicated runnables at run-time. It is the responsibility of the RTE to manage the triggering of events. Different activation patters are defined for runnables in AUTOSAR. *TimingEvents* are used for triggering cyclic activation for runnables, in which a periodic event is assigned to occur periodically. In Sender-Receiver communication, runnables are activated by *DataReceiveEvent* as a consequence of receiving a new value for a referenced data element. Similarly, after the successful sending of a referenced data element, the runnable is activated via the *DataSendEvent*. A RunnableEntity is activated during mode-switching via the *ModeSwitchEvent* (mode switch is initiated) or via *ModeSwitchAckEvent* (mode switch is successfully acknowledged). In this case, the triggered RunnableEntity is executing in the new mode. In Client-Server communication, a runnable is triggered by a client request. The server runnable, that provides the output to the client runnable, is executed either in the context of the client runnable or in the context of another task. This work focuses on the communication of periodically triggered runnables only.

Tasks and Triggers—Runnables are executed in the context of tasks or other runnables (in case of server runnables). There are two types of tasks in AUTOSAR OS [19], the *extended* and *basic* tasks. The *extended* tasks are activated once and never terminate. They switch to waiting mode after they finish the execution of their runnables. They wake up if an event is set to execute any of their runnables. The *basic* tasks activate periodically and terminate after they finish their execution. The states of the tasks, such as ready, running, waiting, and suspended, and their transitions are described in detail in [19]. Runnables are mapped to extended or basic tasks during ECU configuration. The activation of runnables is implemented within the body of the task by RTE generators through dedicated events (controlled in *if..else* conditions), which are handled by RTE. In AUTOSAR, tasks have their period defined by periods of their runnables. They are activated either by *counters* [19, p. 97], *alarms* [19, p. 100], or *schedule tables* [19, p. 40]. Counters are mapped to alarms and schedule tables. They are incremented and reset during the run-time. A counter can be hardware or software based and is derived from a timer. Counters are mapped and manipulated by one core only. *Alarms* activate tasks, set events, or increment counters. An alarm is associated to one counter. When an alarm expires, a task is activated or an event is set.

The *schedule table* is a time-triggered approach of activating runnables/tasks and events. It is defined by a set of expiry points, an initial offset, and duration in ticks. At the occurrence of an expiry point, the activation of tasks or the setting of events takes place. The time of occurrence of an expiry point is defined by a unique offset. The schedule table is configured to be either repeating or single-shot. The former one is useful for triggering, for instance, initial tasks that are executed only once at the system start up. In the repeating schedule table, the expiry points are repeated identically after the duration of the schedule table is reached. In this work, the concept of scheduling table is used to activate tasks, as it provides the possibility to optimize the use of the hyper-period time interval. This is especially the case when FPS is used to schedule tasks. Moreover, AUTOSAR provides the possibility to synchronize different schedule tables and improve the end-to-end delays between different ECUs. AUTOSAR tasks are scheduled using the FPS approach. They are configured during ECU configuration to run preemptive or non-preemptive.

2.2.2 Communication Paradigms

In AUTOSAR, the communication between software components is enabled by two major paradigms: *Sender-Receiver* and *Client-Server communication*. The communication between runnables of the same SWC is handled through *Inter-Runnable communication*, which is a special form of Sender-Receiver communication. The RTE provides APIs to enable the communication between runnables for any of the aforementioned paradigms. Because communication is only possible through port interfaces, respective Sender-Receiver and Client-Server port types are created for each communication paradigm. An example of Sender-Recever, Client-Server and Inter-

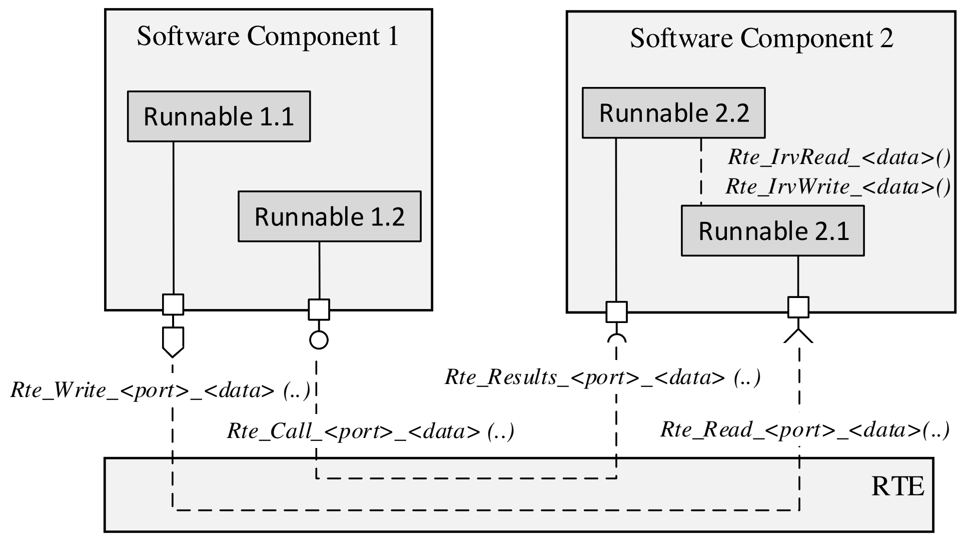


Figure 2.10: Example of AUTOSAR Communication Paradigms.

Runnable communication is shown in Figure 2.10. The sender Runnable 1.1 of Software Component 1 writes the data via *Rte_Write_<port>_<data>()* API. The receiver Runnable 2.1 of Software Component 2 reads the data via *Rte_Read_<port>_<data>()* API. Similarly, the client Runnable 1.2 of Software Component 1 invokes an operation via *Rte_Call_<port>_<data>()* API, and uses the asynchronous operation via *Rte_Results_<port>_<data>()* API. In Software Component 2, Runnable 2.1 writes via *Rte_IrvWrite_<port>_<data>()* API an inter-runnable variable that is read by Runnable 2.2 via *Rte_IrvWrite_<port>_<data>()* API. The semantics of these APIs are defined in the RTE specification document [46].

Client-Server communication—is a communication paradigm where SWCs request and provide operations among them. A *client* runnable enforces a call or uses an operation that is implemented by a *server* runnable. These operations are defined at design time. The Client-Server communication has two forms of transmissions: *synchronous* and *asynchronous* communication. In synchronous transmission the *client* RunnableEntity is blocked during the call, i.e., waits for the *server* RunnableEntity to finish and then returns to the context of the client RunnableEntity. In asynchronous call, the client RunnableEntity is not blocked and can execute other operations while the server RunnableEntity executes in parallel.

Sender-Receiver communication—is a paradigm that enables communication based on data elements, in which the sender SWC provides the data and the receiver SWC consumes the data. The exchange of data is asynchronous because the sender SWC does not wait for the receiver SWC to receive the data. AUTOSAR defines two forms of Sender-Receiver communication: *implicit* and *explicit* communication. In *explicit* communication, runnables have a direct read or write access to the data elements, i.e., to a version of data that is globally visible to all runnables that exchange this data.

Runnables have access on the data through RTE APIs, which provide the latest atomic version of a data element. In *implicit* communication, runnables have read or write access to a buffered data element. This form of communication is standardized in AUTOSAR to avoid data stability issues caused by preemption or by parallel read and write accesses between runnables executing on different cores. The RTE provides to runnables a copy of the data. Runnables have exclusive access on the provided copy. This work focuses on the implicit communication for LET and describes a buffering mechanism for this type of communication that satisfies the semantics of LET.

Inter-Runnable communication—uses InterRunnableVariables to handle the asynchronous communication between runnables of the same SWC. Runnables have a read or write access to these variables. As in Sender-Receiver communication, AUTOSAR distinguishes between implicit and explicit Inter-Runnable communication. However, the queued explicit pattern is not allowed in Inter-Runnable communication.

AUTOSAR communication paradigms are designed to be applied for the *Inter-* and *Intra-ECU* communication. In the *Intra-ECU* case, the send and receive operations of the *Sender-Receiver communication* are read and write operations to the shared data elements. In the *Inter-ECU* case, the communication is handled through communication services of BSW and distributed bus communication protocols. This work focuses only on the *Intra-ECU* communication.

2.3 Deterministic Multi-Core Systems

The following sections describe the impact of multi-core technology on the functional and timing requirements of embedded automotive systems and present the LET as a way to reduce this impact and ensure deterministic execution of these systems.

2.3.1 Multi-Core Effects

Multi-core processors offer for embedded applications advantages such as higher computing capacity and improved response times. Parallel execution of application functions on different cores reduces the time required to deliver produced outputs. In this way, functions deliver their outputs faster but also use concurrently the same processor resources such as shared memory and crossbars/buses. Concurrent accesses to these resources can cause bus and memory arbitration delays, leading to non-determinism and unpredictable timing [48]. In parallel executing applications with frequent data accesses, these interferences can degrade the performance and make the multi-core technology inefficient. Another form of multi-core interference arises due to concurrent, parallel accesses to shared resources such as semaphores or spin-locks [19].

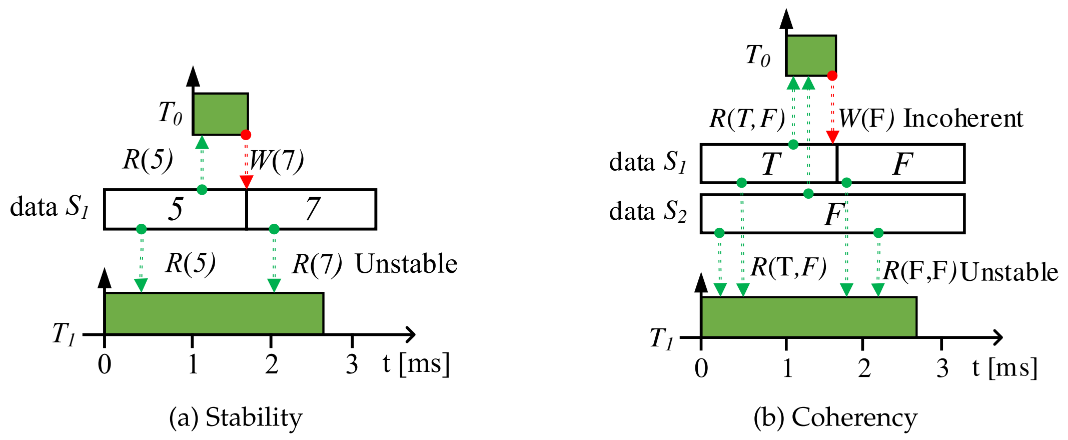


Figure 2.11: Example of **(a)** data stability and **(b)** coherency problem. Task T_0 and T_1 run on different cores. In **(a)**, task T_0 modifies the value of data element S_1 while task T_1 is executing on the other core. After some time, task T_1 reads another value of data element S_1 compared to the first read. In **(b)**, data elements S_1 and S_2 are not updated coherently by task T_0, leading to incoherent reads by task T_1. Values F and T indicate Boolean values False and True.

Locks over these resources are prone of deadlocks, which in multi-core processors are hard to avoid using single-core based protocols such as PCP [23], unless the application is designed such that coinciding requests occur in isolated time slots during execution. To avoid deadlocks in multi-core processors, several locking protocols are proposed [5, 49, 50]. Nevertheless, the delays and the core load added due to waiting for the same resource, yet impacts the performance and task's response times, and in certain situations they can cause deadline violations. As a result, the temporal isolation of high criticality functions is compromised by interferences between cores due to accesses to shared resources such as memory, bus, and semaphores.

Parallel execution of tasks in multi-core processors impacts the way data elements are accessed. Problems with data *stability*, *consistency*, and *coherency* can occur if parallel executing tasks access simultaneously the same data elements. Data stability and consistency issues arise if writer tasks update data elements that are simultaneously and concurrently consumed by reader tasks on other cores. Data stability [51] is satisfied if a task reads the same value of a data element throughout its execution. Figure 2.11a shows an example of two tasks running on two different cores. Task T_0 causes an unstable read of data element S_1 to task T_1. Between time 0 ms to 1 ms, task T_1 reads value 5 of data element S_1. At time 1 ms, task T_0 starts execution on the other core and changes the value of S_1 from 5 to 7. At time 2 ms task T_1 reads value 7, which is different from the first read value of S_1.
Data consistency refers to the consistent and correct state of data. An inconsistent value of data occurs when the data is updated concurrently by multiple tasks during a non-atomic read or write operation.

In single-core systems, preemption also leads to data stability and consistency issues. This happens when a task is interrupted during a read or write operation. During the time the task is in the preempted state, another task updates the data, and when the task resumes its execution, it reads a different value of the data. To avoid data consistency issues, the AUTOSAR OS provides the possibility to disable and enable interrupts before and after non-atomic read and write operations. The interrupt enabling and disabling for each data access adds delays and extra load to the system. In multi-core systems, shared resources are needed to preserve atomic data operations, i.e, consistency, which increase system's load due to the access and waiting delays on these shared resources. To avoid data stability issues in single-core processors, non-preemptive scheduling is used, in which the preemption is shifted to certain points in time [35], such that the data stability is maintained. Non-preemptive scheduling is not suitable for every application due to its complex design and unnecessary calls of the scheduler. In multi-core systems, data stability cannot be guaranteed by scheduling, because non-preemptive regions within tasks cannot avoid concurrent, parallel read and write accesses to the same data between functions running on different cores. In AUTOSAR, the *implicit* Sender-Receiver communication is used to ensure data stability for both single- and multi-core processors.

Data coherency [51] issues occur when a group of related data elements is not updated instantly. Examples of data elements that must be updated coherently and instantly are the ones of *structs* and *arrays* types. Figure 2.11b shows an example of two tasks running on different cores. Data elements S_1 and S_2 are of type Boolean. They must have complementary values and must be updated jointly, i.e., they must not have at the same time value True or False. However, task T_0 changes the value of S_1 incoherently (without changing S_2) from False to True while T_1 reads value False of S_2 concurrently at two different time instants, causing an incoherent read operation of task T_1. Another form of data coherency is caused by cache memory, which causes multiple tasks running on different cores to operate on different versions of data. Cache coherence protocols are developed to keep the data coherent.

2.3.2 Timing and Dataflow Determinism

A key non-functional requirement for correct and safe execution of embedded system is *determinism*. Determinism implies that the results of a task's execution must be *predictable* and *reproducible* during design and target execution. In multi-core processors, determinism is affected by issues such as high delays due to task interferences for shared resources, unpredictable communication, deadlocks, time starvation, resource deprivation, and data consistency and stability issues [52].
Two aspects of determinism are important to be guaranteed for embedded in-vehicle systems: the *dataflow* and *time predictability*. In single-core processors, *dataflow* is ensured through the sequential execution of functions and tasks. It is defined and impacted by design decisions such as scheduling and mapping of functions to tasks. In multi-core processors, dataflow is influenced, in addition to scheduling, by parallel execution of tasks. *Time predictability* refers to the regularity of the delivery time

of outputs by tasks. In embedded applications, tasks have a producer – consumer communication relation and the time of produced outputs is critical for a correct end-to-end dataflow, e.g. from sensor to actuator [53]. The *deadline requirement* is defined to indicate that task's outputs are produced within an expected time interval, such that they are either usable by consumer tasks or sent to actuators without latency. The time of produced results is influenced by scheduling decisions and the multi-core platform. Scheduling influences this time by defining the time that tasks execute and terminate. For instance, higher-priority tasks produce results faster than lower-priority ones, and the opposite holds for delayed or preempted tasks. In terms of multi-core platform, the time of produced results is increased due delays caused by concurrent accesses on shared resources. These delays are dynamic, highly nondeterministic, and hard to precisely estimate at design time. Therefore, ensuring end-to-end delay requirements and dataflow correctness results in higher design and development effort for multi-core systems.

This work applies the LET paradigm [7] to ensure data stability, time and dataflow determinism. Fully deterministic multi-core systems are hard to build, because multi-core processors are unpredictable in nature and embedded automotive applications have frequent irregular external inputs. LET paradigm increases determinism and reduces development effort of these systems. A detailed description of LET and its advantages is given in the following section.

2.3.3 The Logical Execution Time (LET)

LET is a programming paradigm [6] introduced for TT systems with the Giotto programming language in [7, 54] and for ET systems with the xGiotto language in [55] to provide time and dataflow determinism of an embedded real-time system. LET semantics and benefits are described in the following sections.

2.3.3.1 Semantics of LET

In the LET paradigm, each periodic task is bound to a specific logical time interval, called the LET interval. LET specifies fixed time points at which periodic tasks exchange data with each other. Tasks read input data at the beginning and write outputs at the end of their LET intervals with a logical execution time of zero. The actual scheduling and execution of tasks occur within their LET intervals. A task starts execution after its inputs are read and terminates before its outputs are written. Outputs are provided exactly at the end of LET intervals, even if tasks finish their execution earlier than this time. In this way, LET provides time, value, and dataflow determinism of data exchanges between tasks.

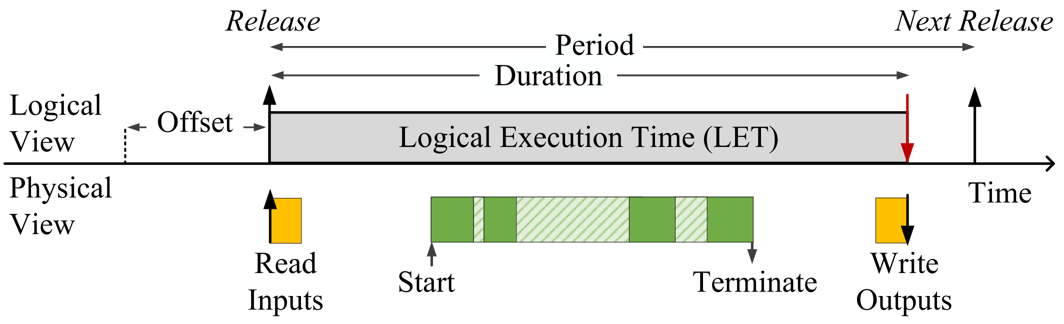


Figure 2.12: The LET task model. The periodic task is bound to a LET interval. In the logical level, tasks read their inputs at the beginning of their LET and write the outputs at the end of their LET at zero-execution time. In the physical level, the data exchange at LET boundaries occurs taking some execution time. The physical data exchange is depicted by yellow boxes. Tasks execute at any time within their LET intervals. The green boxes indicate execution time of the task. The LET interval is depicted by the gray box. The light-green pattern colored boxes indicate the preemption time of the task. The red arrow indicates the end time of the LET interval. The design of this figure is inspired by the classic abstraction of LET given in [55–57].

Figure 2.12 depicts the structure of a periodic task defined by LET. The release time of a task corresponds to the release, i.e., start of its LET interval. This time is determined by the *offset* and the *period* of the LET interval, which are identical with the offset and the period of the task. Because the release time and the end time of a LET interval is the same in any platform, the data exchange times are independent from the platform. In this way, LET maintains the time and dataflow between functional components of the application also when the platform changes. The termination, i.e., end time of a LET interval is defined by its release time and the duration of the interval. It corresponds to the absolute deadline of the task. To guarantee that tasks do not execute beyond the end times of their LET intervals, their respective WCETs must not exceed the duration of their LET intervals.

In LET, the time of reading inputs and writing outputs is known and is identical in the logical and physical view. In Giotto [7], data communication is implemented through ports defined in each LET task. The drivers, referred to as communication tasks, transfer the data between ports of different tasks. The assumption of zero execution time of data exchanges at the boundaries of LET is not realistic in practical systems. Depending on the data exchange mechanism used to integrate the semantics of LET into these systems, the amount of time for reading inputs and writing outputs in the physical view varies between tasks and can be greater than zero. The actual execution and scheduling of the task is known only in the physical view. Scheduling must guarantee that tasks finish their execution before the end of their respective LET intervals. The LET task model is suitable for any scheduling scheme because dataflow between LET tasks is independent from the scheduling mechanism. Therefore, LET task model can be applied to various approaches such as FPS or TTS.

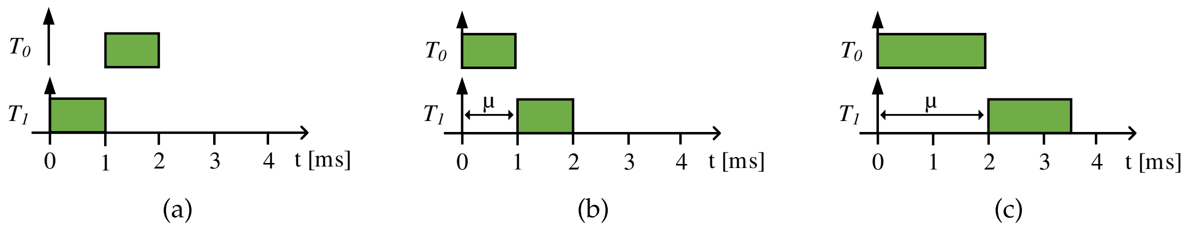


Figure 2.13: Example of delivering data without LET paradigm. The data delivery time depends on task-to-task interferences such as scheduling and task allocation decision. The green boxes indicate the running time of tasks. Figure 2.13a shows the execution of T_0 and T_1. Task T_1 has the highest priority and delivers the outputs at time 1 ms. In Figure 2.13b, task T_1 has lower priority than T_0 and delivers the outputs at time 2 ms with a start delay μ of 1 ms. In Figure 2.13c, tasks T_0 and T_1 execute on a processor with lower frequency, in which task T_1 delivers the outputs at time 3.5 ms with a start delay μ of 2 ms.

2.3.3.2 Fundamentals of LET

The LET paradigm provides a variety of benefits for multi-core development of embedded real-time systems. Due to its concept of fixed data exchange points between tasks, it reduces the effort of designing such systems for different multi-core processors. Although LET has the highest benefits for multi-core systems, it can also be used in single-core systems. A selection of benefits are discussed in the following paragraphs.

Deterministic communication – LET provides deterministic functional behavior of software applications by means of *time*, *value*, and *dataflow determinism*. In LET, tasks exchange data at the boundaries of LET intervals. The time of exchanging the data is deterministic in the sense of known communication times and end-to-end delays. This time is known, verifiable, and identical for different platforms. LET eliminates jitters of produced outputs and prevents unpredictability in timing. Dataflow between tasks is independent of their execution. Hence, the timing and order of data exchanges between tasks running on the same or different cores is predictable and unaffected by design decisions such as task-to-core allocation and scheduling. A significant advantage of the LET paradigm is platform independence. Platform refers here to the OS and the hardware processor. In LET, the actual execution of tasks does not affect the delivery time of data unless a deadline violation has occurred. In non-LET systems, the time of delivering outputs is determined by the actual execution of tasks. Figure 2.13 and Figure 2.14 illustrate an example of delivering outputs without and with LET, respectively.

Data stability – In LET, preemption and parallel execution of tasks does not effect data stability. Exchanging the data at the boundaries of LET intervals ensures a stable version of data for consuming tasks throughout their respective LET intervals. The consuming tasks read stable data at any time independently if they get preempted during execution or if other parallel tasks modify simultaneously the same data.

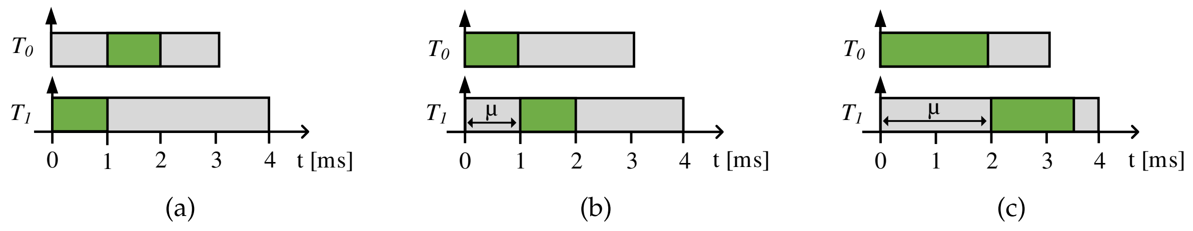


Figure 2.14: Example of delivering data with LET paradigm. Task T_1 produces the data at any time during its execution, but it is made available at time 4 ms, that corresponds to the end time of its LET interval. The green boxes indicate the running time of tasks and the gray boxes represent the LET interval. Figure 2.14a shows the execution of T_0 and T_1. Task T_1 has the highest priority and delivers the outputs at time 4 ms, which corresponds with the end of its LET interval. In Figure 2.14b, task T_1 has lower priority than T_0 and delivers the outputs at time 4 ms independent of start delay μ of 1 ms. In Figure 2.14c, tasks T_0 and T_1 execute on a processor with lower frequency. Task T_1 delivers the outputs at time 4 ms independent of start delay μ of 2 ms.

Isolation of memory contentions – The realization of LET semantics depends on the data exchange mechanism used to enable data exchange at the boundaries of LET intervals. For lock-based buffer mechanisms such as PTP, physical data exchange operations take place at the boundaries of LET intervals. In multi-core systems that use PTP, isolation of read and write accesses to shared memory at the boundaries of LET intervals decreases the unpredictability coming from arbitration and blocking delays of concurrent access to shared memory during task execution. In this case, interferences occur only during the data exchange operations. Isolation of these operations can also be accomplished, for example, by disabling their parallel execution and scheduling them to execute sequentially and uninterruptedly among cores [58].

Multi-core platform migration – LET reduces the effort of migrating embedded applications to multi-core platform. Migration of these systems to new platforms involves re-implementation and re-design effort for the specific platform. If necessary, new design decisions are taken such as re-mapping of functions to tasks, reallocation of tasks to cores, or regeneration of the schedule. Because LET provides platform independence, the redesign decisions on the new platform do not directly impact the data exchange time and dataflow between functions. LET supports parallel execution of the software application on multiple cores and enables functional correctness without the need to explicitly adjust the execution order between runnables of different cores, e.g., by offsets or scheduling. In LET, the dataflow between tasks is defined by the dataflow between their LET intervals. Therefore, if the design of LET intervals is not changed during the migration, the dataflow and timing is guaranteed, regardless of scheduling and on which core tasks are mapped to execute.

Simplified verification of the dataflow and timing requirements – In systems that apply LET semantics, verification of the dataflow and end-to-end delays is straightforward due to the time-based structure of data exchange points. In this way, the verification effort of the dataflow and timing requirements is significantly reduced compared to when LET is not applied.

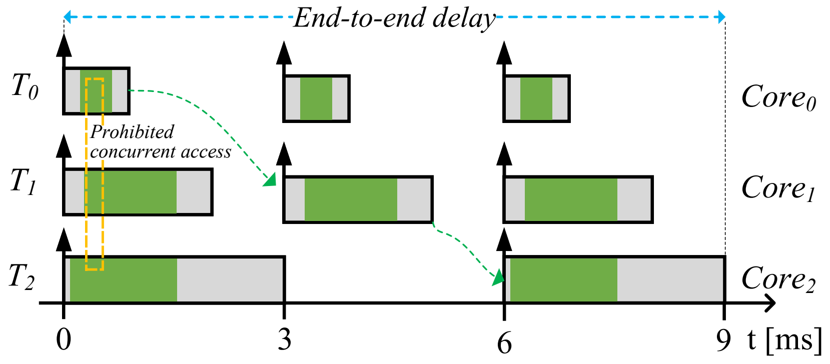
Extensibility – Extending the system with new functionalities, i.e., adding new functions or tasks does not impact the communication behavior and determinism. However, it has an impact on decisions such as reassigning tasks to cores, regenerating the schedule, or synthesizing buffers for the new functions or tasks.

Co-development – The cooperation between control and software integrator teams is more efficient with LET because timing is understood in the same way [7].

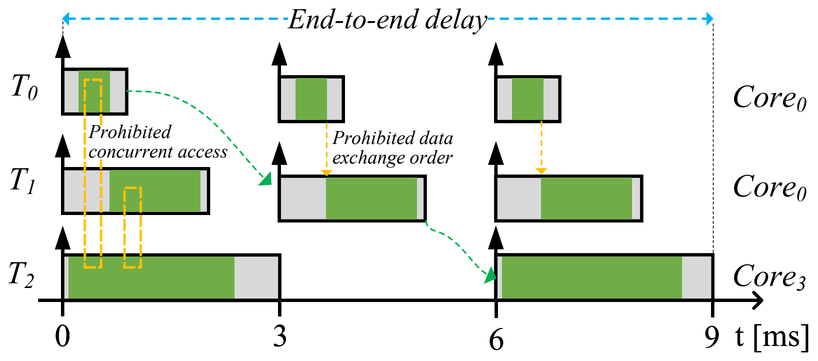
An example showing the described benefits of LET is given in Figure 2.15. A dataflow consists between tasks T_0, T_1, and T_2. In the configuration of Figure 2.15a, tasks are allocated to execute on different cores. Because data exchanges occur at the boundaries of their LET intervals, concurrent data accesses on the same data elements are avoided in overlapping LET intervals for tasks running in parallel on different cores. Hence, data stability, the dataflow and end-to-end delays are ensured. Figure 2.15b shows the same application but assuming that new functionalities are added to task T_2, indicated by a longer execution time, and the allocation of task T_2 is changed to $Core_0$. In this case, two aspects have changed: the platform, indicated by the new task-to-core allocation and scheduling, and the system is extended with new functionalities. Because data exchanges between LET intervals occurs at the boundaries of LET intervals, the execution order between T_1 and T_2, defined by scheduling, does not affect the dataflow between these tasks. Similarly, concurrent accesses between T_0 and T_2 and T_1 and T_2 are also avoided by LET. Hence, the dataflow and end-to-end delays are not affected by these changes. In both designs, the end-to-end delay is 9 ms. LET enforces that end-to-end delays are constant regardless of the system's design configurations, which simplifies their verification and reduces the effort required to ensure that timing requirements are met.

2.3.3.3 Specification of LET in AUTOSAR

LET is specified in AUTOSAR since release 4.4.0 in *Timing Extensions (TIMEX)* [59]. LET intervals are described via event chains and periodic, offset, and latency timing constraints [59]. In AUTOSAR, unlike the original definition of LET [7], a LET interval is associated with a group of runnables instead of a single task. Therefore, the implicit data accesses of the Sender-Receiver communication of the runnables mapped to LET intervals are performed using LET semantics. The software integrator must ensure that runnables of LET intervals with different timing information such as offset, period, and duration are not mapped to the same task. This is because there is no guarantee that the data exchange operations of different LET intervals occur exactly at their boundaries, unless it is ensured that all runnables of the task terminate their execution before the earliest end time of the LET intervals of the task. In this case, the complexity



(a) Tasks run all on different cores. The dataflow and end-to-end delays defined by LET intervals of tasks T_0, T_1, and T_2. The dataflow is $T_0 \rightarrow T_1 \rightarrow T_2$. The concurrent accesses in overlapping LET intervals of tasks T_0, T_1, and T_2 running on different cores are avoided by LET.



(b) Tasks T_0 and T_1 run on the same core and task T_2 is extended with new functionalities. The dataflow and end-to-end delay defined by LET intervals of tasks T_0, T_1, and T_2 are ensured. The dataflow is $T_0 \rightarrow T_1 \rightarrow T_2$. The concurrent accesses in overlapping LET intervals between tasks T_0 and T_2 and T_1 and T_2 running on different cores are avoided by LET. Tasks T_0 and T_1 have a fixed execution order defined by scheduling, but no data exchange occurs between them in overlapping LET intervals.

Figure 2.15: Example of benefits provided by LET. The green boxes indicate the running time of tasks and the gray boxes represent LET intervals. The dataflow between LET intervals is shown by the green dashed arrows.

of the scheduling increases significantly. Therefore, this work assumes that runnables mapped to one LET interval are all mapped to the same task.

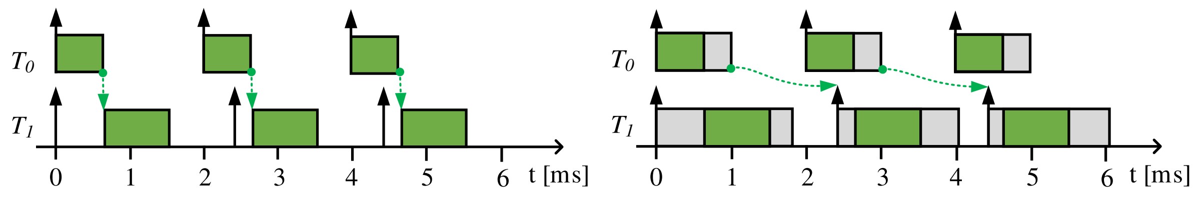
3 | Inter-Task Communication Design

This chapter introduces buffering techniques that reduce memory requirements and data-synchronization overheads induced by *Logical Execution Time (LET)*. A description of the problems solved in this chapter and the overview of several real-time programming paradigms, that are used as baseline for LET, are given in Section 3.1. The research literature and the review of techniques that apply LET paradigm in embedded applications are provided in Section 3.2. The proposed *Static Buffering Protocol (SBP)*, which is designed to apply LET in automotive applications w.r.t plausible memory and run-time overheads, is described in Section 3.4. The *Point-to-Point Protocol (PTP)* is an alternative buffering protocol described in this work to preserve LET's semantic. The PTP is not designed to minimize the memory and buffering run-time overheads but to handle the dynamic execution behavior of the application. A brief description of PTP is given in Section 3.3. A comprehensive evaluation of memory and synchronization run-time costs of PTP and SBP is conducted in Section 3.5.

3.1 Introduction

The widely used real-time programming model in embedded applications is the *Bounded-Execution Time (BET)* [6]. In BET, the execution of a task is divided into an input, a computation, and an output part. The input part corresponds to reading of the input data consumed by the task, the outputs are computed in the computation part and are written in the output part. In BET, tasks exchange data, i.e., they communicate with other tasks mainly at the boundaries of their physical execution times. Typically, this communication is handled through expensive lock-based synchronization mechanisms such as semaphores or through lock-free communication protocols. For each task, a lower and an upper bound on the execution time are defined, denoted as *Best-Case Execution Time (BCET)* and *Worst-Case Execution Time (WCET)*, respectively, which are bounded by the deadline. Hence, in BET the execution of tasks is correct if the tasks complete their execution before their next release or before their deadlines.

BET lacks time and value determinism. Thus, changing the platform or extending the application with new functionalities such as adding or removing tasks changes the time when tasks exchange data. This drawback is resolved by LET [7]. The LET model handles better system changes than BET because it separates the input and outputs parts from the actual execution of the task. In LET, outputs are not written as soon as they are available as it is in BET, but at the end of LET intervals. In LET, the reading of



(a) Data exchange in BET. (b) Data exchange in LET.

Figure 3.1: In BET, data is consumed at the boundaries of execution time or at any time during execution. In LET, data is exchanged at LET boundaries. In BET task T_1 consumes data produced by the parallel instance of task T_0. In LET, task T_1 reads the produced data by the previous instance of task T_0. The solid green filled boxes indicate the execution of tasks. The gray boxes indicate the LET intervals. The dotted lined arrows demonstrate the direction of exchanged data.

inputs and writing of outputs occur instantaneously in zero time and at the boundaries of LET intervals. The LET paradigm is a derivative of both BET and *Zero-Execution Time (ZET)* programming models [6]. The ZET paradigm is based on the semantics of synchronous-reactive programs [60, 61] and assumes that computations are logically fast and data exchanges occur in zero execution time.

Figure 3.1 shows an example of two tasks exchanging data at the boundaries of their BET and LET. In Figure 3.1a, task T_1 consumes outputs produced by task T_0 directly after T_0 finishes its execution. In Figure 3.1b, task T_1 consumes the provided outputs with a time delay, defined by the end of the LET duration of T_0. In LET, T_1 reads at every instance only the data produced by the previous instance of T_0, and not the freshest value produced by the parallel active task's instance, as it happens in BET.

The LET paradigm, just as BET and ZET, is only applicable to processor platforms in which the WCET can be estimated. In multi-core systems, the WCET is unpredictable and hard to estimate due to delays caused by interferences on shared hardware resources. These interferences are hard to estimate because of the dynamic arbitration of shared resources and nondeterministic blocking delays caused by concurrent accesses to bus and shared memory [62] . To properly define the LET duration for LET tasks, a safe upper-bound of WCET, usually pessimistic, must be calculated such that the application executes without LET interval violations. This work assumes that WCET is pessimistic and lower than the LET duration.

Several challenges emerge when integrating LET into automotive applications. LET decreases the control quality due to the increase of the end-to-end duration of the control flow between tasks [8]. This drawback of LET is influenced by how LET intervals are designed. Because outputs are made available for other tasks at the end of task's LET interval, reducing the duration of LET intervals reduces as well the end-to-end duration of the control flow.
The assumption of zero execution time for reading of inputs and writing of outputs at the boundaries of LET intervals is not valid for buffering protocols such as PTP

[8]. PTP causes run-time overheads due to buffering operations at the boundaries of LET intervals. These overheads increase not only the demands for processing power [58], but also the duration of LET intervals, because tasks must not execute beyond the duration of their LETs. The buffering overheads are differently referred to as *Worst-Case Communication Time (WCCT)*. Hence, the duration of a LET interval must be greater than the sum of WCCT and task's WCET.
An increase of the LET interval's duration means longer end-to-end delays, which makes the LET paradigm impractical for certain control applications. Although communication overheads are also present in BET model, they do not impact the control flow between functions of the application in the same way as LET, because in BET outputs are made available during or after tasks finish their execution. An example of buffering in BET model is the implicit Sender-Receiver communication [46] of *AUTomotive Open System ARchitecture (AUTOSAR)*, in which outputs are made available after each runnable's execution. Therefore, the buffering overheads are distributed over the execution of the task, and the output delays are defined by the actual scheduling and WCET of the task. It shall be noted that although LET is designed to provide constant end-to-end delays, in PTP it is not the case because jitters of produced results occur due to scheduling of multiple buffering operations at boundaries of LET intervals [63].
Finally, to ensure the semantics of LET, additional data variables are needed, thus increasing the memory capacity demands. For example, in PTP global variables, i.e., buffers are added to store the version of data in which LET tasks read and write during their execution within their LET intervals.

To integrate LET into automotive systems, a buffering protocol must be designed to enable data exchanges between tasks at the boundaries of their LET intervals. To achieve a practical and efficient integration of LET for automotive systems, the design of such protocol must address the impact of LET on end-to-end delays and the required memory and processing resources. In this work, SBP is proposed to efficiently integrate LET into automotive applications such that memory demands are reduced, buffering overheads and jitters on data exchanges between tasks are eliminated, and determinism is fully ensured. By eliminating buffering overheads at the boundaries of LET intervals, SBP provides capacity to reduce end-to-end delays and increase schedulability of the application.

3.2 Related Work and Problem Analysis

The following sections provide an overview of the buffering mechanisms that are designed for inter-task communication of embedded applications.

3.2.1 Data Stability and Integrity

This section describes buffering protocols that are designed to guarantee *Synchronous-Reactive (SR)* semantics and data stability of BET applications. These protocols are not designed for LET applications, but are used in this work and in the research community to derive approaches that fulfill LET semantics.

3.2.1.1 Circular Buffering Protocols

Kopetz et al. [24] propose the *Non-Blocking Write (NBW)* protocol as a *lock-free* approach to handle data stability issues that arise from concurrent read-write operations on data shared between one writer task and multiple reader tasks. The initial version of the protocol avoids data stability issues by enforcing a concurrency check at the end of the read operation of each reader task. The concurrency check observes if a concurrent write operation has occurred during the read operation. If an interference has occurred, then the read operation is repeated until no interference occurs. The drawback of this approach is the unpredictable amount of retries, which makes the protocol inefficient [24]. Hence, the authors improve the NBW by enforcing the concept of multiple buffers of the same data and the *wait-free* circular buffering behavior. In the cyclic NBW, writers use buffer elements cyclically and have exclusive access on them. Readers consume the recent produced data stored in a buffer element in which the writer task does not concurrently write until the read operation is performed. The number of read interferences is used to define the amount of buffers required to handle the stability of data operations.

Ntaryamira et al. [64] apply the circular buffering methodology of NBW to avoid data stability issues that occur due to preemption or concurrent operations on shared data between reader and writer tasks. In their circular buffering approach, a *First In - First Out (FIFO)* buffer is used, in which read and write accesses are granted circularly until the end of the buffer is reached. The read accesses are assumed to take place at the activation of reader tasks and the writes at the termination time of the writer task. Differently from the approach in [64], in the classical FIFO protocol, reader tasks receive all outputs, stored in a FIFO queue, produced by a writer task. The queue is emptied by the reader task after the processing of outputs takes place. An example of the classical FIFO protocol is the queued Sender-Receiver communication [46] of AUTOSAR. Ntaryamira et al. [64] define constraints that describe a functional relation of inputs and outputs between multiple tasks and define the buffer size considering the functional relation between reader and writer tasks. They define the buffer size based on the timing characteristics, i.e., period and response times of the reader and writer tasks, and on their functional-chain relation [64]. In the case of single-writer multiple-readers communication, the buffer size is defined as $\lceil \frac{R_r}{P_w} \rceil$, where R_r is the largest response time among all reader tasks and P_w is the period of the writer. In their approach, it is ensured that a reader task consumes the recent produced value, among multiple produced ones, by calculating the number of produced outputs between two occurrences of a reader task. This number is calculated on-line at the end of writer's execution and is referred by the authors as the *sub-sampling-rate*. The sub-sampling-rate is incremented by one at each execution of the writer task. Hence, the reader consumes the element of the buffer that is previous to the sub-sampling-rate. Ntaryamira et al. [65] extend the FIFO circular buffering with mechanisms that enforce reader tasks to read the version of data that is consumed by the slowest reader task for tasks that are part of an event chain.

3.2.1.2 Dynamic Buffering Protocol (DBP)

Sofronis et al. [10] propose the *Dynamic Buffering Protocol (DBP)* as an optimal *wait-free* approach to preserve data stability and the SR semantics of preemptive tasks, such that memory requirements for storing buffer elements are kept minimal. DBP is designed for single-writer multiple-reader communication for tasks that are scheduled based on priorities and execute on single-core processors. DBP uses *wait-free* communication scheme and is designed to optimize the memory demands required to preserve the data flow and data stability between tasks. DBP is dynamic, i.e., the access decisions to buffer elements are taken during run-time at every task release, and centralized, i.e., buffers are global and all shared among tasks. Buffer elements are occupied by reader and writer tasks during complete execution up to the next release of the task. Similar to NBW [24], DBP ensures that reader and writer tasks do not operate concurrently on the same buffer elements at the same time. It is shown to be optimal [10] in terms of buffer usage and the number of buffer elements required to preserve the semantics. In DBP, a *buffer* is an array that contains different versions of the same data element, which are accessed by tasks in different points in time.

In DBP, buffer elements are assigned to reader and writer tasks based on their priority ordering. The protocol distinguishes between higher-priority and lower-priority reader tasks and the writer task of a data element S_s. Lower-priority tasks are reader tasks that have a lower priority than the writer task. Similarly, higher-priority tasks are reader tasks with higher priority than the writer task. A higher-priority reader task accesses only the data produced by the previous instance of the writer task and a lower-priority reader task accesses the current produced data by the writer task or the previous one, depending if the lower-priority reader task starts its execution before the writer task. Sofronis et al. [10] defines the maximal amount of buffers required for a data element S_s equal to $|LR| + 2$, where LR defines the amount of lower-priority reader tasks. The LR is divided in lower-priority tasks without a unit delay LR^1 and with a unit delay LR^2 [10]. The two more buffer elements are required because one is needed to store the value used by the higher-priority tasks and one to store the current value written by the writer task.

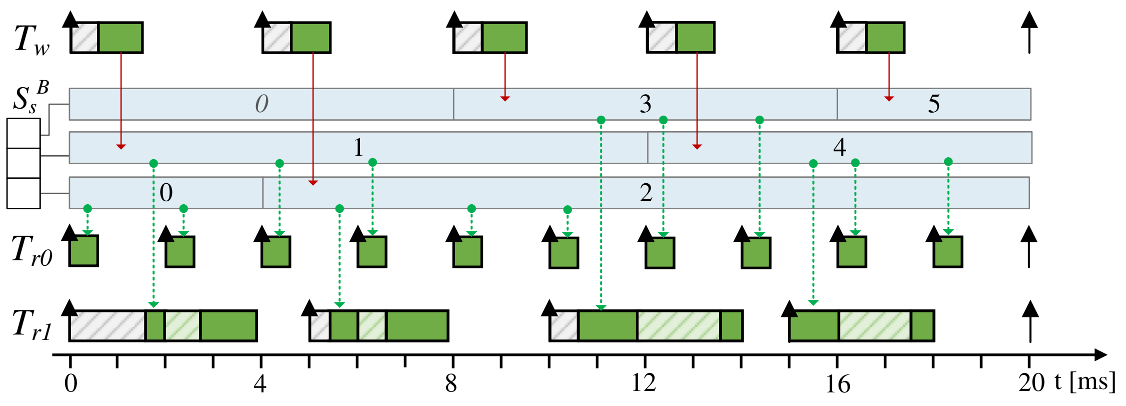


Figure 3.2: Example of buffer accesses with DBP for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The buffer S_s^B has three elements. A buffer element changes its value each time it is written by the writer task T_w. The write operations are depicted by the red solid arrows and the read operations by the green dashed arrows. The first instance of the higher-priority task T_{r0} reads the initial data. The green marked blocks indicate execution time of tasks. The gray boxes indicate the start delay of tasks and the light-green boxes the preemption time. Task T_{r1} is interrupted in every instance wither by T_w or T_{r0}. Priorities have the ordering $\pi_{r1} < \pi_w < \pi_{r0}$. The visualization of buffers is inspired by figures in [13].

The protocol tracks and grants accesses to tasks using *pointer* data structures. The *curr* pointer is used to record the last written value by the writer task. It points to the buffer element, in which the writer tasks writes during its execution. The *prev* pointer is used to record the previous produced value by the writer task before the *curr*. An array of pointers is kept to store the reference of buffer elements that are assigned to reader tasks. These pointers are used by reader tasks during their execution. Buffer elements are assigned to reader and writer tasks at their respective release times. A read access on a buffer element is granted to reader tasks only if the writer task is not concurrently writing at the same point of time. Multiple reader tasks are allowed to read simultaneously the same buffer element. Higher-priority tasks have read access only to buffer elements produced by the last finished writer instance, which is referenced by the *prev*. A lower-priority task gets access to the buffer element produced by the latest active instance, which is referenced by the *curr*, if it is in the list of tasks with a unit delay LR^1. These accesses are valid under the assumption that the writer task terminates before the start of the lower-priority reader task. Otherwise, if the lower-priority task belongs to the set of tasks without a unit delay LR^2, then it gets access to the buffer element pointed by *prev*. When a reader task finishes its execution, the granted access to a buffer element is removed. At the release time of the writer task, the pointer *prev* gets the value of *curr* and *curr* is assigned to the next free element of the buffer, in which no reader task has an access at the same point of time.

Figure 3.2 shows an example of three tasks sharing one global data element S_s. The writer task T_w writes the data every 4 ms, and the reader tasks T_{r0} and T_{r1} read the data every 2 ms and 5 ms, respectively. The priority ordering is defined as $\pi_{r1} < \pi_w < \pi_{r0}$. The π_{r1}, π_w, and π_{r0} define the priorities of T_{r1}, T_w, and T_{r0}, respectively. The accesses to the buffer are shown for all task instances up to the *Hyper-Period (HP)*. In the worst-case, three buffer elements are required, but because of oversampling not all elements of S_s^B are used at the same point of time. Task T_{r0} has two consequent read operations on the same buffer elements, e.g., the first and the second instance access buffer element $B_{s1}[0]$. Task T_{r1} is preempted by T_{r0} and T_w, and reads after resume the same buffer element as at start time.

The advantages of DBP are the ability to handle at run-time variations of the execution times of tasks and the reduced needs for memory capacity to store global buffers. DBP is designed to handle only fully preemptive tasks. But, in practice applications contain subsets of non-preemptive tasks [35], which are designed to reduce the effects that come from preemption. In order to apply DBP for these applications, special handling of this use-case is required. An extension of the protocol is to treat non-preemptive reader tasks as tasks with higher priority than the writer task. In this case, the non-preemptive reader tasks consume data produced by the previous instance of the writer task. DBP cannot be applied for multi-core processors because concurrent read-write and write-write operations of parallel running tasks can cause unstable data operations. This occurs because buffering decisions are taken in DBP based on task priorities, which have no effect on the execution of tasks among cores.

3.2.1.3 Temporal Concurrency Control Protocol (TCCP)

Wang et al. [66, 67] provide different implementations of DBP for applications that run on an *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (OSEK)* operating system. They propose the *Temporal Concurrency Control Protocol (TCCP)* as an alternative approach of DBP, such that a constant time for searching a free buffer element to assign to a writer task is achieved. TCCP is a non-blocking buffering mechanisms as DBP but it has a different buffer assignment behavior compared to DBP. TCCP uses the circular approach of NBW to assign buffer elements to a writer task. The next buffer element is assigned to the writer task by increasing the *curr* pointer. If the end of the buffer is reached, then the head of the buffer is used as the next element. Compared to the FIFO circular buffering described in [64], the TCCP assigns read accesses to reader tasks based on the DBP semantics.
Wang et al. [66] define an upper bound of the buffer size for DBP in case of multiple active instances of writer tasks. In this case, k elements are necessary to store values of *prev* active instances of writer tasks up to a time delay. The upper bound of the buffer size is calculated as $|LR| + k + 1$, where one element is required to keep the value that all higher-priority reader tasks read. Compared to DBP, in TCCP the upper bounds of the buffer size is defined by the number of instances of the writer task that overlap with the instances of the slowest reader tasks. They define the upper bound of the buffer size of TCCP, considering multiple active writer instances, as $\lceil \frac{P_r + P_w}{P_w} \rceil + k$, where P_w is the period of the writer, P_r is the period of the slowest reader, and k is the number of stored *prev* pointers up to a time delay.

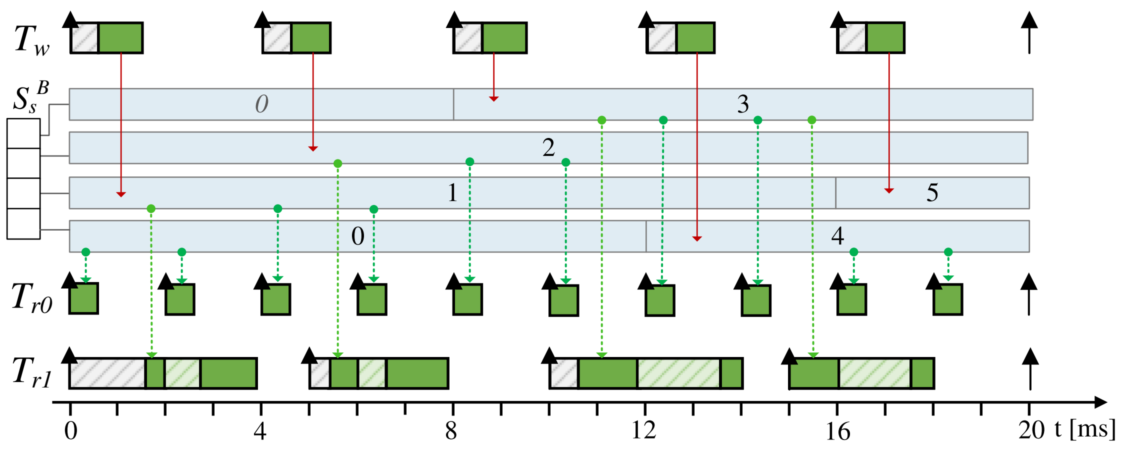


Figure 3.3: Example of buffer accesses with TCCP for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The buffer S_s^B has three elements. A buffer element changes its value each time it is written by the writer task T_w. The write operations are depicted by the red solid arrows and the read operations by the green dashed arrows. The first instance of the higher-priority task T_{r0} reads the initial data. The green marked blocks indicate execution time of tasks. The gray boxes indicate the start delay of tasks and the light-green boxes the preemption time. Task T_{r1} is interrupted in every instance wither by T_w or T_{r0}. Priorities have the ordering $\pi_{r1} < \pi_w < \pi_{r0}$. The visualization of buffers is inspired by figures in [13].

Natale et al. [68] and Wang et al. [69] improve the buffer size calculation of TCCP. Wang et al. [66] show that TCCP requires less memory to store auxiliary variables and takes less time to find a free element than DBP, but needs more buffer elements to preserve the SR semantics. The buffer size in TCCP is defined based on periods of the writer and reader tasks and not their priorities.

Figure 3.3 shows an example of the buffer schedule created by the TCCP for three tasks sharing one global data element S_s. The writer task T_w writes the data every 4 ms, and the reader tasks T_{r0} and T_{r1} read the data every 2 ms and 5 ms, respectively. The priority ordering is defined as $\pi_{r1} < \pi_w < \pi_{r0}$. The accesses to the buffer are shown for all task instances up to the HP. Task T_w accesses circularly the buffer elements starting from the first element to the fourth. In this example, TCCP requires four buffer elements to preserve the semantics. As shown in Figure 3.2, DBP requires only three buffer elements to ensure the semantics for this example.

Because DBP outperforms TCCP in terms of buffer size, this work use DBP approach to design SBP and integrate LET semantics. Although, DBP can be extended to handle LET semantics in multi-core processors, as shown in [66], it is complex to implement, it requires additional memory capacity to store auxiliary data and due to its dynamic nature accesses to the buffer schedule are unpredictable and hard to verify. Extensive evaluation of buffer accesses is necessary to identify at run-time potential violations of the time- and value-correctness. Therefore, SBP was designed as a static and deterministic approach to implement LET semantics for multi-core applications, optimize the memory requirements, and avoid the aforementioned drawbacks.

3.2.1.4 Data Stability in AUTOSAR Systems

Zeng et al. [70] provide an overview of mechanisms used to ensure data stability and dataflow of AUTOSAR applications running in a multi-core platform. They describe the explicit synchronization between the writer and reader tasks running on different cores, such that data is accessed by tasks in the order write-before-read. This approach is not practical because it must be ensured by scheduling, synchronization of cores and task activation, and by timing properties of tasks such as offsets and periods. In addition to the lock-based mechanisms [70], they describe the multi-core case of DBP. In their approach, the dataflow and concurrency between writer and reader tasks that execute on different cores are handled by synchronizing the activation of the writer and reader tasks or by sending interrupt signals to the core on which the reader tasks execute [70]. In this way, reader and writer tasks do not accesses concurrently the same buffer elements and the read accesses occurs after the write. This approach is not practical because it requires multiple synchronization interrupts between cores, which increase the preemption overheads and reduce the execution performance.

Zeng et al. [71] propose optimization techniques for minimizing the memory requirements of AUTOSAR applications. They provide methods to chose between data stability mechanisms described in [70] and to reduce the amount of preemptions by setting a preemption threshold for tasks, such that the usage of RAM and stack memory is reduced. Compared to Zeng et al. [71], this work focuses on reducing memory requirements by implementing a static buffering protocol that is independent of scheduling and that offers dataflow determinism by means of LET.

In AUTOSAR, data stability is ensured by the implicit Sender-Receiver communication [72], which sustains data stability through local buffering at runnable level. The *Runtime Environment (RTE)* of AUTOSAR implements this communication and manages the accesses to buffer elements. Local buffers are created for every runnable and for every data element that they read or write. Runnables operate during execution on the local buffers. The input data are read at runnable's start time and outputs are made available at runnable's termination time.

3.2.2 Temporal Determinism

Henzinger et al. [7, 54] propose LET as a real-time programming abstraction to guarantee deterministic data exchange between embedded applications. Only recently, the industry has recognized LET as a mechanism to increase the robustness, time and dataflow determinism of multi-core applications. The following sections provide an overview of buffering mechanisms proposed to guarantee LET semantics and the related work of LET regarding its applicability in automotive systems.

3.2.2.1 Related work on LET

Hennig et al. [73] explore LET for potential benefits in migrating an existing legacy powertrain system to multi-core platform, such that the single-core functional behavior of the system remains unchanged and the application executes in parallel. They group runnables to parallel tasks such that the LET size is decreased and the control delay is minimized. They apply the TDL modeling platform [56] to generate LET tasks and deploy them to cores. Migrating existing embedded applications to LET semantics means defining the LET interval duration of each periodic task and implementing a buffering protocol for data exchange between tasks at their LET boundaries.
Bradatsch et al. [74] propose an approach of assigning the duration of the LET interval of producer tasks to be equal to task's *Worst-Case Response Time (WCRT)*, such that the duration of end-to-end event chains is minimized. The produced data becomes available earlier and the duration of the control flow is minimized. This approach does not scale well with application changes. Deploying the system to different platforms or changing the schedule and task-to-core allocation requires re-evaluation of WCRT and re-assignment of LET duration. The calculations of WCET must be precise enough such that LET determinism is not violated at run-time due to LET overruns.

Becker et al. [75] provide analysis of end-to-end event chains for multi-rate applications considering LET paradigm. They show that the data age increases when LET is used. Martinez et al. [53] focus on minimizing the end-to-end latency of event chains for tasks that use LET semantics, by aligning period overlaps between tasks. They propose a heuristic to assign offset of tasks for reducing the latency and improving task's WCRT.

Biondi et al. [58] analyze methods to map LET buffers to memory components for the *Engine Management System (EMS)* model provided in the FMTV challenge [76]. They optimize the normalized response times by minimizing the memory accessing time. For that they use a genetic algorithm and *Mixed Integer Linear Programming (MILP)* formulation to allocate variables to memories such that memory accessing times are reduced. They provide response time calculation considering bounds of the worst-case memory access delays by concurrent executing tasks. They study two approaches of integrating LET in AUTOSAR, one as part of the RTE and the other as part of application software in form of tasks.

Brandberg et al. [77] compare the LET with the implicit and explicit Sender-Receiver communication of AUTOSAR and data consistency buffering at task level in terms of task response time. Farcas et al. [56] generate the schedule of data accesses in distributed communication considering LET semantics. They suppress sending of messages that are not used by any consumer task. Ernst et al. [78] describe extensions of LET for system-level communication between multiple *Electronic Control Unit (ECU)*s. This work focuses on LET communication within one ECU.

3.2.2.2 Buffering Mechanisms for LET

In the classic definition of the Giotto language [7], the data exchange between tasks takes place at the boundaries of their LETs through a mechanism that resembles the PTP buffering protocol. In PTP, private global variables, referred otherwise as local buffers, are necessary to store the version of data in which LET tasks read and write during the LET interval. The filling and flushing of values from the private variables to the global data elements, and vice versa, causes significant communication overheads. During data exchange at the beginning and end of LET intervals, the concurrent accesses to the same variables are protected by software resources such as semaphores or spin-locks [19], which increase further the communication overheads due to resource consumption and blocking delays. Due to the fact that filling and flushing operations are isolated at certain points in time, parallel data synchronization between cores can occur, which leads to higher interference delays caused by arbitration and usage of shared memory and bus.

Resmerita et al. [8] apply the LET semantics by using PTP for a legacy automotive systems, where they focus on reducing the computational costs coming from LET. They evaluate the optimization potential regarding the amount of buffers required to guarantee LET semantics and show that the memory requirements for buffering could be minimized if the amount of data elements that are exchanged via LET is reduced. Hence, they configure implicit Sender-Receiver communication [46] for data elements that are accessed in tasks with non-overlapping LET intervals and PTP communication for the rest of them. Because LET is only applicable between periodic tasks, the data exchange between sporadic and periodic tasks requires dedicated handling. They handle this by flagging data elements computed and consumed by LET periodic and non-LET sporadic tasks. The flagging is necessary to not discard the output of sporadic tasks, which can be overwritten by LET tasks at the end of their LET. If the output of sporadic tasks is flagged, then the content of data element is not overwritten. Finally, through simulation they estimate the WCRT used to assign as LET interval to tasks for reducing the control delays caused by LET.

Resmerita et al. [9] extend buffer analysis of [8] for legacy systems running in multi-core processors. They apply their buffering analysis at two levels of design. In the first step, they analyze the software application to collect data accesses that must be buffered according to LET semantics. In the second step, they relax the LET rules and use platform information such as scheduling and task-to-core mapping to prune away buffers of data elements that are not required to be buffered. For instance, the data accessed by tasks that execute sequentially and at higher priorities are not buffered as per LET semantics, but accessed accordingly and preserve the LET constraint. They consider event-triggered tasks to exchange data with time-triggered tasks and handle its effects in their analysis. The challenge of their approach is that it does not scale well with system extensions and future platform migrations. The buffering information has to be recalculated, considering as well that the data elements that are not buffered would require again analysis to evaluate if they are value-safe. In contrast to [8, 9], this work optimizes buffering requirements for LET through a wait-free protocol that minimizes these overheads by design instead of software configurations.

Kehr et al. [4] propose the *Timed Implicit Communication Protocol (TICP)* for executing an automotive system in parallel in multi-core processors. The TICP protocol is compliant to the LET semantics and it differs from the PTP in the way how buffers are stored and handled. In TICP global buffers are used to implement LET instead of local buffers, in which the accesses to the buffer are performed at run-time based on time-stamps attached to buffer elements. Tasks access the version of data in the buffer that corresponds to an earlier time instant relative to the time the access is performed. The drawback of the TICP is that it can suffer buffer overflows and requires verification for evaluating whether the communication is time- and value-deterministic. TICP is similar to our proposed SBP compared to the fact that both are global and centralized buffering protocols. TICP uses timestamps to take buffering decisions and SBP uses, as in DBP, the *curr* and *prev* pointers, which indirectly derive the time of produced outputs. Compared to TICP, SBP is static, deterministic and planned during the design time. The authors do not provide a description of the TICP algorithm. Therefore, additional differences to SBP cannot be identified.

Kluge et al. [79] extends a multi-core operating system to support LET semantics. They use the message passing communication on a network-on-chip (NoC) with time-division multiplexing (TDM) arbitration. For LET communication they implement a cyclic buffering approach, similar to FIFO [64], at the receiver node. A queue of received messages is maintained in the receiving nodes. A buffer element in the buffer queue stores the received data and is identified by the logical available time. In this way, the receiving task consumes only the data with a logical time that correspond to the start of its LET interval. The buffer queue stores all the data received during the LET interval of the reader task and as well the last data received up to the start of the LET interval of the reader task. Hence, the buffer size in the receiver node is defined as $\lceil \frac{P_r}{P_w} \rceil + 1$, where P_r is the period of the receiving, i.e, reader task and the P_w is the period of the sender, i.e., writer task. The circular buffer in [79] differs from the proposed SBP in several aspects. SBP is not designed for a message passing communication, but is defined to integrate semantics for tasks that use a shared memory to exchange data. SBP requires less buffers to preserve the LET semantics because buffers are not created for each reader task, but are shared in different points of time among reader and writer tasks. Hence, buffers are reused among tasks.

Beckert et al. [80] propose the *double-buffering protocol* for implementing the inter-task communication between tasks that use the LET paradigm. This protocol works as follows. Two versions of each data element are stored in a global buffer. One of the elements is used by reader tasks and one by the writer task. Tasks access buffer elements via dedicated pointers. The read pointer points to the data element that is accessed by the reader tasks and the write pointer points to the data element that is accessed by the writer task. The pointers are swapped at the end of writer's LET such that the readers can read the latest produced value at the beginning of their LETs. A copy pointer for all reader tasks is maintained and is updated with the copy of the reader pointer at the release of each reader's LET interval. This protocol provides invalid inter-task communication when the end of writer's LET interval lays within the LET interval of any reader task. In this case, the pointers could swap during the time that the reader task is executing, which not only violates LET semantics but also can lead to data stability issues and malfunctioning of the application software. Therefore, the double-buffering protocol is applicable only for tasks in which the end of writer's LET corresponds to the end or start of the LET intervals of all readers. In SBP, similarly to the double-buffering protocol, a global buffer is used to preserve LET semantics. The difference is that SBP is not constrained by the harmony of periods to provide a correct buffering behavior and, depending on the application, one to several buffer elements are required in SBP to preserve LET semantics. Another difference is that the double-buffering protocol assigns buffers at run-time by means of pointer swapping and SBP assigns buffers statically at design time. Finally, the double-buffering protocol cannot handle multiple writers of the same data element.

Beckert et al. [81] improves the drawbacks of the double-buffering protocol by enforcing the behavior of a ring-buffer, in which data is written in consecutive buffer elements in a circular way as described in FIFO [64], NBW [24], or TCCP [66] protocols. The double-buffering protocol with the ring-buffer behavior is referred in the following as ring-buffering protocol. The authors do not describe how the double-buffering ensures the ring-buffer behavior and how buffers are assigned to reader and writer tasks. In our understanding, the buffer assignment is performed at run-time by the LET Event handler, a concept provided in [81], similarly to the FIFO approach described by Ntaryamira et al. [64]. Because buffers in the double-buffering protocol are assigned to tasks at run-time, we infer that buffer assignment takes place as well at run-time when the ring-buffer behavior is enforced.
The notion of global buffers is identical in the ring-buffering protocol as in the proposed SBP originally published in [12, 13]. However, the SBP approach differs with the ring-buffering protocol in several aspects. SBP assigns buffers to the writer and reader tasks, not circularly, but based on the semantics of DBP. SBP is designed to reduce the amount of buffers required to preserve the LET semantics. Unlike the circular buffer assignment of the ring-buffering, SBP does not need a next buffer element to assign to a writer task, but it reuses any previous buffer element not used by any reader task during the LET time interval of the writer. Hence, SBP does not assign the next buffer element to writer tasks as it happens in a circular buffering approach, but it assigns a buffer element that has no read accesses during the writer's LET interval. If an available element is not found, then a new buffer element is assigned to the writer task. Therefore, SBP requires less buffers, i.e., less memory to preserve LET semantics than the ring-buffering. Furthermore, SBP reduces the buffer size by suppressing the unnecessary writes, e.g., in case of under-sampling and in case of data age constraints, which the ring-buffering cannot handle. Unlike the ring-buffering,

buffers are assigned to tasks statically at design time to avoid the run-time overhead for buffer assignment at the boundaries of LET.

Ogawa et al. [82] apply the double-buffering protocol [80] to integrate LET for powertrain applications running on a multi-ECU platform. In their version of the double-buffering protocol, reader tasks operate on local variables and a copy-in operation, as in PTP, copies the data from the read pointer to the local variables of each reader task. The writer task operates on the writer pointer and the swap of read and writer pointers is done as described in [80]. The authors describe the synchronized version of the double-buffering protocol, in which the swapping of buffer pointers and copy-in operations are distributed to multiple *Control Processing Unit (CPU)*s and are synchronized to occur in the required order. As the authors state, the synchronization is time consuming and causes buffering overheads. Therefore, in their proposed asynchronous approach the swapping of pointers is avoided by enforcing tasks and runnables to decide based on the current time and LET interval which of the pointers to use. The extensions of the double-buffering protocol given in [82] improve the drawbacks of its initial definition described in [80].

Biondi et al. [83] propose an approach of implementing PTP for synchronous tasks with implicit deadlines. They allocate the copy-in and copy-out operations for execution at the beginning of LET and ensure that the copy-out operations take place before the copy-in. This approach works well for tasks with implicit deadlines and synchronous activation of tasks. But for tasks with constrained deadlines, special handling is required for executing copy-out operations. They suppress the unnecessary writes by avoiding writing of the output to global data element for writer task instances that have no reader instance. To avoid memory contentions during data exchange, they synchronize copy-in and copy-out operations among cores. That means, no parallel copy-in and copy-out operations occur in their approach. Nevertheless, this can lead to an increase of processing time due to synchronization, to an inefficient use of processor utilization, and to an increase of tasks response times.

A buffering approach that focuses on reducing the memory and run-time overheads of LET is introduced in [84]. The approach is based on dedicated local buffers for reader and writer task. Outputs are written at the end of writer's LET interval to the local buffers of the reader tasks rather than to a global data element as in PTP. To suppress unnecessary writes, the authors describe an approach to reduce the number of copy operations at the end of LET intervals by avoiding publishing outputs that are never consumed. Unlike PTP, copy operations take place only at the end of writer's LET interval. This approach is valid only for tasks with harmonic periods and when the period equals the duration of the LET interval. Their approach of suppressing writes is also effective for PTP. Therefore, the authors apply it in PTP to handle the communication between tasks with non-harmonic periods. The copy operations are mapped to execute in the context of computation tasks instead of dedicated tasks. A major advantage of their buffering strategy is the explicit synchronization of exchanging outputs at the end of LET intervals in case tasks have bidirectional consumer-producer relation on different data elements. Like PTP, their buffering strategy requires more memory space and causes run-time overheads at the boundaries of LET intervals than SBP.

3.2.3 Summary of Related Work

A summary of the related work is given in Table 3.1. Only approaches that target buffering mechanisms are compared in this table. Buffering protocols such as the NBW [24], DBP [10], TCCP [66], and FIFO [64] use a global buffering methodology to ensure SR semantics and data stability. They are designed to handle the single-writer to multiple-reader communication of non-LET systems. The implicit Sender-Receiver communication [46] of AUTOSAR enforces a local buffering approach to ensure data stability. It defines rules that inquire buffering at the boundaries of runnable's BET or at the beginning and the end of the non-preemptive sections. The frequent buffering operations induce run-time overheads, which increase the overall system's utilization. Less overheads are expected in the aforementioned global protocols because buffer assignment occurs only at the beginning of task release times and not in multiple locations within tasks' execution. Except of FIFO [64], the rest of SR preserving buffering protocols handle only the inter-task communication between tasks running on single-core processors. Zeng et al. [70] provides an extension of DBP for the multi-core case. However, the proposed approach is impractical due to scheduling and high synchronization effort between cores. None of the aforementioned protocols fulfills LET semantics. However, they are used in the related work to derive new buffering protocols that ensure LET semantics. DBP overcomes the rest of SR preserving and data stability buffering protocols regarding the number of buffer elements required to guarantee the SR semantics. Therefore, the proposed SBP is based on the DBP.

Related buffering approaches that support LET semantics, with the exception of the work in [80–82], support multiple writer tasks. The main advantage of the proposed SBP protocol compared to related works is that it provides zero communication time at the boundaries of LET intervals, reduces memory demands for buffering, and causes zero jitters on data sampling time. These jitters refer to the time difference between the end time of LET intervals and the actual writing of output data. SBP uses global buffers to preserve the semantics and to reduce the overheads. It ensures zero communication time, i.e., zero buffering overheads by avoiding physical data exchange and run-time buffer assignment decision at the boundaries of LET intervals. In SBP, buffers are assigned to tasks at design time such that reader and writer tasks have exclusive access on dedicated buffer elements throughout their LET intervals. This is not the case for the approaches found in the related work.

The ring-buffering approach in [81] is most similar to SBP. The main differences are that SBP requires less buffers to ensure LET semantics and uses the DBP approach to assign buffers at design time, rather than circularly at run-time like the ring-buffer

	LET Semantics	*Buffering*	*Multiple writers*	*Scalable*	*Memory Optimization*	*Overhead Optimization*	*Multi-Core*
Contribution	✓	G	✓	✓	✓	✓	✓
Kopetz et al. [24]	✗	G	✗	✗	✓	○	?
Sofronis et al. [10]	✗	G	✗	○	✓	✓	✗
Wang et al. [66]	✗	G	✗	✗	○	○	✗
Ntaryamira et al. [64]	✗	G	✗	✗	○	?	✓
Zeng et al. [70]	✗	G	✗	✗	✓	?	✓
AUTOSAR et al. [46]	✗	L	✓	✗	✗	✗	✓
Beckert et al. [80]	✓	G	✗	✗	✗	○	✓
Beckert et al. [81]	✓	G	✗	✓	✗	○	✓
Kehr et al. [4]	✓	G	?	○	✗	?	✓
Ogawa et al. [82]	✓	M	✗	○	○	○	✓
Kluge et al. [79]	✓	L	?	✗	✗	✗	✓
Resmerita et al. [8]	✓	L	✓	✗	✓	○	✓
Resmerita et al. [9]	✓	L	✓	✗	✓	○	✓
Biondi et al. [83]	✓	L	✓	✗	✓	✗	✓
Haefele et al. [84]	✓	L	?	○	✓	○	✓

Table 3.1: Qualitative comparison of buffering mechanisms.
Legend: satisfied (✓), partially satisfied (○), unsatisfied (✗), unknown (?), Global (G), Local (L), or Mixed (M).

protocol. Furthermore, SBP reduces the amount of buffers by suppressing unnecessary writes or writes that meet data age constraints. The former ones are used to provide to reader tasks data of a certain age and to suppress the unnecessary writes for writer jobs with unused outputs. The ring-buffering assigns buffers circularly at run-time, similar to NBW [24], TCCP [66], and FIFO [64], which require more buffers than DBP.

Related works in [8, 9, 83] apply the PTP protocol to satisfy LET semantics. Unlike SBP, PTP uses local buffers to integrate LET and requires more memory capacity to store all variables because one global variable is still held in the shared memory to store the latest produced output. In SBP, buffers are shared, except that tasks only access the buffer element that satisfies the semantics of LET. Therefore, the overlapping of LET intervals and timing parameters such as periods and offsets can reduce further the amount of needed buffers in SBP. PTP is easy to integrate into automotive systems due to its similarity to the implicit Sender-Receiver communication and offers advantages such as handling of data exchanges between periodic LET and periodic, sporadic, and a-periodic non-LET tasks without special handling of buffers. For this reason, PTP is described and evaluated in this work as an alternative approach to integrate LET into automotive systems.

3.3 Point-to-Point Protocol (PTP)

This section provides a detailed description of the PTP protocol, commonly used in the related work as an approach to implement LET semantics in automotive applications [9, 83]. This work applies the elemental design principles of PTP found in the related work and enhances it with aspects that address system's schedulability, scalability, and stability of data exchanges. The characteristics and methodology of PTP are described in the following sections.

3.3.1 Overview

PTP is a *lock-based* and a *decentralized* buffering protocol that uses local buffers to preserve LET semantics. In PTP, every *initial* task $T_i \in \tau$ that consumes or produces a global data element S_s gets a corresponding local copy of S_s, referred to as local buffer S_s^b, which task T_i reads or writes during execution. These buffers are filled at the beginning of a task's LET and flushed at the end of its LET. Hence, the value of the global data element is copied to the local buffer at the start of a task's LET interval. Task T_i operates during execution on the local buffer and the produced data is copied from the local buffer to the global data element before the end time of task's LET interval. In this work, the filling and flushing of the local buffers are referred to as *copy-in* and *copy-out* operations, respectively. For every task $T_i \in \tau$ one dedicated copy-in and copy-out operation is defined for every accessed data element. A copy-in

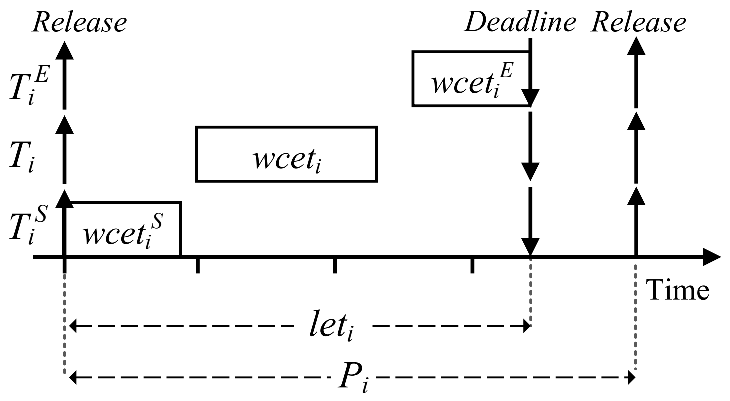


Figure 3.4: Composition of tasks into LET Start T_i^S, LET End T_i^E and computation T_i. Communication tasks T_i^S and T_i^E take each $wcet_i^S$ and $wcet_i^E$ time to execute. The end of let_i of T_i corresponds to the end of deadline D_i. For tasks with LET interval let_i equal to their period P_i the write operation takes place before the release of the next job.

operation of a data element $S_s \in S$ consists of two data accesses: one read of the global data element S_s and one write of the buffer S_s^b. Similarly, a copy-out operation of a data element $S_s \in S$ consist of one read of the buffer S_s^b and one write of the global data element S_s. If a task T_i reads and writes the same data element S_s, then two buffers are created, one for the reading operation and one for the writing. It is assumed that reading the intermediate results stored in the write buffer at multiple points within the task is a decision of the function developer and is independent of the PTP buffering behavior and the number of buffers created. Hence, for each data element S_s with N_R reader tasks and N_W writer tasks a total of $2 * (N_R + N_W)$ data accesses are added in copy operations, in addition to the $N_R + N_W$ existing data accesses occurring in computation tasks.

The PTP described in [9] assigns LET copy-in/-out operations of every task T_i to functions, which are on the other hand mapped to driver tasks. This work uses a similar concept, with the difference that copy-in operations are not mapped to the same driver task that contain copy-out operations. They are mapped to different tasks for improving the schedulability of the system and for better load balancing between cores. Resmerita et al. [9] use one dedicated driver task for every core. The execution time of such task can get significantly large due to many copy-in and copy-out operations, which makes it difficult to balance the utilization of cores by means of task-to-core allocation during migration to different processors.

Figure 3.4 shows the composition of tasks after the PTP buffering is synthesized. The computation task $T_i \in \tau$ is associated with the LET Start T_i^S and LET End T_i^E tasks. Both T_i^S and T_i^E are driver tasks and are alternatively referred to as communication tasks. In PTP, the task T_i remains unchanged, except that it has access only to local buffers. The task T_i^S has a WCET of $wcet_i^S$, which corresponds to the time required to read input data, i.e., the run-time of copy-in operations. Task T_i^E has a WCET of $wcet_i^E$, which corresponds to the time required to write outputs, i.e., the run-time of copy-out operations. These times are also referred to as WCCT. In this work, the WCCT considers as well the time to guarantee atomic read/write operations, i.e, consistency using lock-based mechanisms. Tasks T_i^S and T_i^E have the same timing attributes as T_i, such as the period P_i, the deadline D_i, the LET duration let_i, and periodic offset O_i.

An example of a PTP implementation is given in Algorithm 1. Both the global data element *data* and the local buffer $data\_buffer$ are typically declared as global variables. The LET copy-in function *copyInData* copies the value of the global data element to the local buffer and the copy-out function *copyOutData* does the opposite. The computation function *computation* operates on the local buffer.

Algorithm 1: Example implementation of PTP.

```
Input:
data – The global data element,
data_buffer – The buffer element of data
1 Function copyInData(data, data_buffer):
2     $data\_buffer \leftarrow data$ ;
3 Function copyOutData(data, data_buffer):
4     $data \leftarrow data\_buffer$ ;
5 Function computation(data_buffer):
6     $data\_buffer \leftarrow data\_buffer + 1$;
```

The mechanism of PTP is similar to the buffering approach of the *implicit* Sender-Receiver communication of AUTOSAR, but with the difference that buffers in PTP have a lifetime equal to the duration of LET intervals rather than the execution time of computation runnables.

To fulfill LET semantics, driver tasks T_i^S and T_i^E must execute close to the boundaries of LET intervals and must finish before the end of LET. Task T_i^S must execute close to the start of task's LET and must finish before the start of T_i. The computation task T_i must execute at any time between the end of T_i^S and the start of T_i^E. Task T_i must terminate before the start of LET End T_i^E task. Hence, the execution order of driver tasks T_i^S and T_i^E and computation task T_i is, as indicated in Figure 3.4, constrained as $T_i^S \rightarrow T_i \rightarrow T_i^E$. For tasks with coinciding LET intervals, LET Start tasks must finish execution before the start of computation tasks, which must finish all before the start of LET End tasks. Biondi et al. [83] orders tasks T_i^S, T_i^E, and T_i as: $T_i^E \rightarrow T_i^S \rightarrow T_i$, where the T_i^E copies out the results of the previous job to the global buffer before the reading of the current job takes place. This execution order is scalable only if the LET duration is equal to the period. Although, the authors state that this approach isolates memory contentions and controls memory traffic at the beginning of the period when synchronized between cores, this is not optimal if all these operations are handled by one driver task running on one core, because the duration of the copy-in and copy-out operations can delay the execution of tasks running on other cores. Furthermore, the memory contentions to shared memories are

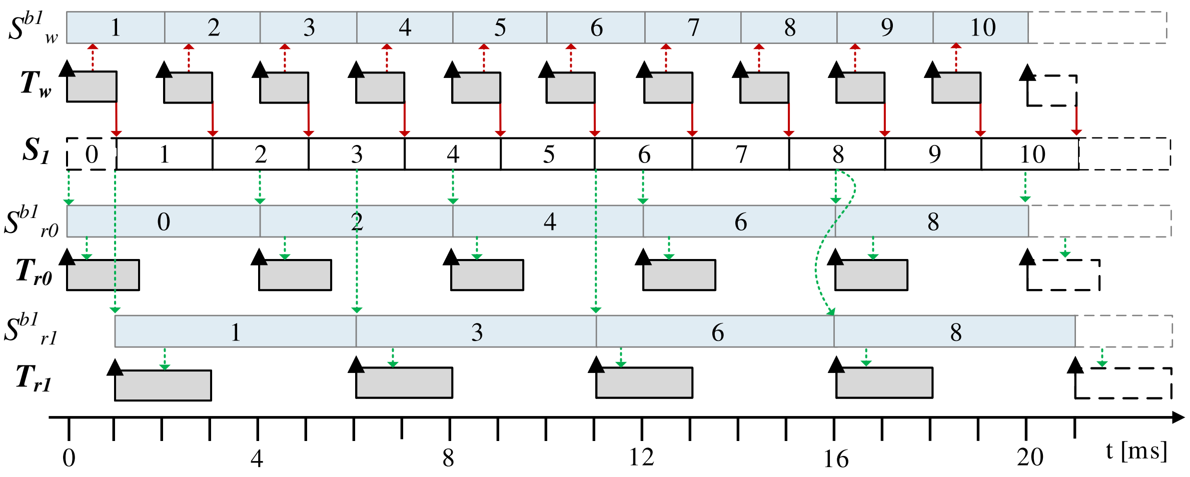


Figure 3.5: Example of buffer accesses using PTP for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The gray boxes represent the LET intervals of each job. The green and red arrows indicate *read* and *write* accesses, respectively. The white boxes represent the global data element S_1 and the values it stores in different time intervals. The light blue boxes represent the local buffers S^{b1}_{w}, S^{b1}_{r0}, and S^{b1}_{r1} of T_w, T_{r0}, and T_{r1}, respectively. The dashed white boxes indicate the LET intervals and buffers of the first task instances of the next HP interval.

not fully avoided because memories are typically shared among all cores. Therefore, to increase the schedulability, these operations are allocated to different tasks where each computation task has its dedicated *driver* tasks. It should be noted that a LET interval is associated with one computation task.

An example of data exchange using PTP protocol is visualized in Figure 3.5. During execution, the writer task T_w writes and reads the local buffer S^{b1}_{w}. The value of the global data element S_1 is updated at every instance of T_w right before the end of its LET. The reader tasks T_{r0} and T_{r1} read during execution only their respective local buffers S^{b1}_{r0} and S^{b1}_{r1}, which are updated with the recent value of S_1 at every task's release. The task's release corresponds in this case, with the start of the task's LET. The first instance of T_{r0} reads the initial value of S_1 and the first instance of T_{r1} reads the first produced output by T_w. The following instances of T_{r0} and T_{r1} read the output produced by the preceding instances of T_w that have the end of LET smaller or equal to the start of the reader's LETs. For simplification, the composition of tasks into driver and computation tasks are not shown in this example.

In PTP, the buffer size B_s of each signal $S_s \in S$ is as

$$B_s = N_R + N_W + 1, \tag{3.1}$$

where N_R and N_W are the number of reader and writer tasks, respectively, and "1" represents the global data element. Equation (3.1) considers the case of multiple writers of signal S_s. It is assumed that reader tasks, that are also writers of S_s, always read the version of the global data element S_s and not the locally produced value.

Equation (3.1) includes the global data element as part of the buffer size to estimate the total memory needed to fulfill the LET semantics.

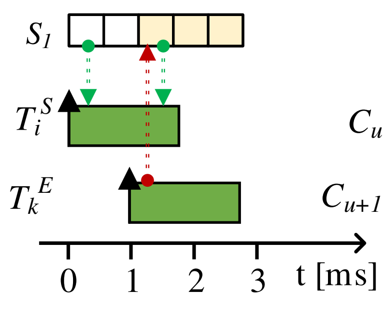
3.3.2 Consistency of Data Synchronizations

The copy-in operations execute in the context of LET Start tasks and the copy-out in the context of LET End tasks. When executing LET Start and End tasks in parallel on different cores, concurrent reads and writes of the same global data element could occur. Hence, concurrent execution of copy-in/-out operations cause inconsistent values of data elements. This means that during a copy-in operation, the LET Start task reads a value that is concurrently written, i.e., the write operation is unfinished, by a LET End task that is running on the other core. This is an issue only for data element types for which writing operations are not atomic, i.e., writing takes multiple processor cycles and does not occur instantaneously. The same applies if multiple concurrent write operations occur on the same data element.

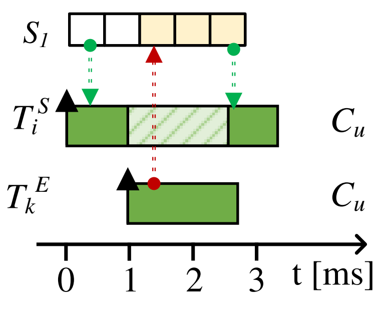
Inconsistent values of global data elements are caused as well by preemption of the LET Start and End tasks. If a LET Start task is preempted by higher priority interrupts during a non-atomic copy-in operation of a data element, then it could read, after the resume, a different value of the data compared to the version before the preemption. This occurs due to the fact that during the task's preemption, the data is modified by parallel executing LET End tasks or by LET End tasks with a higher priority than of the LET Start task. If a LET End task is preempted during a non-atomic copy-in operation of a data element, the parallel running LET Start task could read during the copy-in operation an inconsistent value of the data.

An example of driver tasks T_i^S and T_k^E that operate concurrently on the same data element S_1 is shown in Figure 3.6. In Figure 3.6a, tasks T_i^S and T_k^E execute in parallel on different cores and read/write concurrently the data element S_1. After 1 ms, T_i^S reads an incorrect value of S_1 because T_k^E modifies the rest of the bytes of S_1. In Figure 3.6b, tasks T_i^S and T_k^E execute on same core. Task T_k^E has a higher priority than T_i^S and preempts T_i^S at time 1 ms. After the resume, task T_i^S reads an incorrect value of S_1 because T_k^E modified the rest of S_1's bytes while T_i^S was in preemption state.

This work ensures *instantaneous* and *atomic* copy-in/-out operations by enclosing each operation within one critical section. In a typical embedded *Operating System (OS)*, these sections are implemented by disabling interrupts and by using OS resources, i.e., spin-locks or semaphores [19]. The protected copy-in/-out operations at the boundaries of LET are denoted as *data-synchronizations*. Although critical sections guarantee consistent data-synchronization at the boundaries of LET, they increase the run-time of LET Start and End tasks. Disabling of interrupts and requests/releases of resources take a considerable amount of time because the OS has to synchronize among cores and check which resources are being used. Additionally, the accesses to



(a) Non-atomic parallel operations.



(b) Non-atomic preempted operations.

Figure 3.6: Data consistency issues of copy-in/-out operations of non-atomic data elements. The data element S_1 is not atomic and requires multiple processor cycles per read and write. Task T_i^S executes the copy-in operation of data element S_1 and T_k^E performs the copy-out operation of data element S_1. The green arrow indicates a *read* access and the red arrow a *write* access.

resources such as spin-locks are not always granted immediately. In case of parallel copy-in and copy-out operations of the same data element, execution delays in form of active waiting times occur if the resource is used by other tasks on other cores. If the resource is not available, the task spins actively for the lock until it is available. These delays can increase with the increase of the amount of parallel executing copy-in and copy-out operations of the same data element.

In this work, OS resources are used to protect parallel read-write and write-write operations of the same data elements. Spin-locks are applied to every data element that is accessed by at least two LET communication tasks executing on two different cores. Spin-locks are not added to every copy operation but only to the ones that could have potential concurrent read-write and write-write operation of the same data element on different cores. This optimization decreases the run-time overheads caused by accesses to these resources at certain degree. The upper bound of the number of spin-locks added for this purpose equals the number of data elements. This is defined under the assumption that a concurrent, i.e., parallel copy operation exists for every data element. In each copy operation two spin-lock accesses occur, one for requesting and one for releasing a spin-lock. Hence, for each data element S_s with N_R reader tasks and N_W writer tasks a maximum of $2 * (N_R + N_W)$ spin-lock accesses are added to preserve consistency of all accesses of S_s occurring in copy operations.

Because LET communication tasks are typically configured to execute non-preemptively, interrupts are not disabled and enabled before and after copy operations. This optimization is used under the assumption that in case LET communication tasks are preempted by urgent interrupts they are interrupted shortly or between release and request of different spin-locks. Additionally, in order to avoid deadlocks, nested spin-locks are not allowed. Otherwise, long preemption times could cause long waiting times for the spin-lock to be available to the parallel running tasks on other cores. This time is equally increasing to the time that the preempted tasks remains preempted, and in the worst-case it can lead to deadline violations of the parallel running tasks on other cores and degrade execution's performance of the system. Therefore, in such circumstances the disabling of interrupts is mandatory. Another way to control further the overheads involved during data-synchronization is to avoid the preemptions and the number of parallel occurring copy-in and copy-out operations. One way is through *Time-Triggered Scheduling (TTS)*, which allows to control task's execution at design and target execution. Hence, the schedule of tasks for all cores can be constructed such that communication tasks with concurrent copy-in and copy-out operations of the same data element do not execute in parallel. However, this would result in an under-utilization of cores and long start delays for computations tasks.

3.3.3 Jitters of Data Synchronizations

In LET, the communication between tasks is deterministic in the sense of known dataflow, communication delays, and end-to-end delays. These times are predictable and identical in any running platform. Nevertheless, this statement do not entirely hold in PTP because PTP does not fully eliminate the jitter of produced data. This occurs due to the fact that the time of data exchanges that occur at the boundaries of LET intervals is not zero as assumed in LET. Additionally, extra delays occur in the time of produced results because multiple copy-in/-out operations are scheduled and executed at the boundaries of LET intervals. The *data-jitter* defines the time between the end of producer's LET interval and the actual time that outputs are made available. Although jitters could occur as well at the beginning of the LET, they are irrelevant for computation tasks as long as copy-in operations finish execution before computation tasks start running. However, the advantage of PTP in automotive applications, compared to when LET is not applied, is that it provides bounds of communication and end-to-end delays, which are upper bounded by the end time of the LET interval. These bounds depend on the platform and must be evaluated each time the system's design changes. This drawback of PTP is addressed by SBP.

3.3.4 Run-time Overheads

In PTP, the buffering run-time overheads, annotated as O_{ptp}, consist of the processor's utilization of copy-in and copy-out operations executed at the boundaries of LET intervals. It is defined as

$$O_{ptp} = \sum_{\forall T_i \in \tau} \frac{wcet_i^S + wcet_i^E}{P_i}, \tag{3.2}$$

where T_i is the computation task, and $wcet_i^S$ and $wcet_i^E$ are the WCETs of driver tasks T_i^S and T_i^E of T_i, respectively.

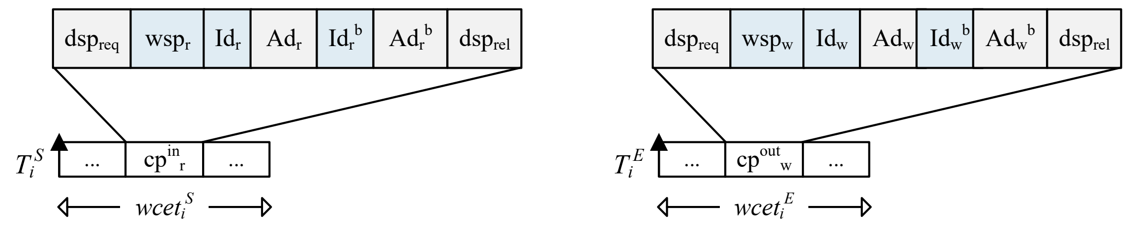
To formally define the buffering run-time overheads of PTP, the following definitions are given. Let $S_i^R = \{S_r | r \in \mathbb{N}^+\}$ and $S_i^W = \{S_w | w \in \mathbb{N}^+\}$ denote respectively the set of global data elements that are read and written by task T_i. For each data element $S_r \in S_i^R$ and $S_w \in S_i^W$ accessed in T_i, their respective buffers S_r^b and S_w^b are defined. If a task T_i reads and writes the same data element, then S_r and S_w indicate the same data element. However, in this case buffers S_r^b and S_w^b are different data elements and are still required to store two versions of the same data. Unique copy-in operations are defined for each $S_r \in S_i^R$ and unique copy-out operations are defined for each $S_w \in S_i^W$. The cp_r^{in} annotates the copy-in operation of data element $S_r \in S_i^R$ and cp_w^{out} the copy-out operation of $S_w \in S_i^W$.

The execution time duration of a copy operation is defined by several time delays. The definition of these delays is a formal description of the elements affecting the WCETs of the copying operations only, and not an exact calculation of the WCET values. For every data element $S_r \in S_i^R$ read by task T_i, the time required to read S_r is defined as a *memory access delay* Ad_r and the time delay occurring due to bus and memory interferences during the read operation of S_r is defined as an *interference delay* Id_r. Similarly, for every data element $S_w \in S_i^W$ written by task T_i, the memory access delay Ad_w and the interference delay Id_w are defined. Interference delays depend on the bus scheduling and the memory traffic caused by simultaneous accesses between tasks running in parallel on different cores. The read and write memory access delays are assumed constant and are calculated based on the data size of the data element and on the amount of processor cycles it takes to read or write data of a given size. These delays vary among processors and memory types. For every buffer S_r^b and S_w^b, the respective access delays Ad_r^b and Ad_w^b and interference delays Id_r^b and Id_w^b are defined.

If consistency of copy operations is required to be ensured, then one unique spin-lock is created for every global data element. The time delay for requesting and releasing a spin-lock is assumed constant and is defined as dsp_{req} and dsp_{rel}, respectively. In every copy operation, a spin-lock is requested before and released after the operation. If a spin-lock is used by another task during the execution of a copy operation, a concurrent request of the same spin-lock leads to a time delay in the copy operation of the requesting task. For each data element S_r accessed in task T_i, the wsp_r defines the *waiting-for-spin-lock* time delay occurring in the copy-in operation cp_r^{in}. The wsp_r is the waiting time for the spin-lock to be released by any occupying task. The wsp_w defines the spin-lock's waiting time delay in the copy-out operation cp_w^{out} of the element S_w.

Figure 3.7 depicts the formal definition of $wcet_i^S$ and $wcet_i^E$ of every task T_i decomposed into different time delays. The amount of copy operations that execute in the context of tasks T_i^S and T_i^E is defined by the number of global data elements that a task T_i reads and writes. Therefore, the $wcet_i^S$ and $wcet_i^E$ of every task T_i are defined as

$$wcet_i^S = \sum_{\forall S_r \in S_i^R} (Ad_r + Id_r + Ad_r^b + Id_r^b + dsp_{req} + dsp_{rel} + wsp_r), \tag{3.3}$$



(a) Execution time of copy-in operations. (b) Execution time of copy-in operations.

Figure 3.7: Formal decomposition of time delays in copy-in and copy-out operations of tasks T_i^S and T_i^E in PTP.

$$wcet_i^E = \sum_{\forall S_w \in S_i^W} (Ad_w + Id_w + Ad_w^b + Id_w^b + dsp_{req} + dsp_{rel} + wsp_w). \tag{3.4}$$

In case protection of a copy operation is not required, then the time delay for spin-lock usage and waiting is not added in the calculation of $wcet_i^S$ and $wcet_i^E$. Similarly, if bus and memory interferences do not occur during a data access then interference delays are zero. The time delays wsp_r and wsp_w are greater than zero if the requested spin-lock is occupied by another task running in parallel on another core. These delays are variable and depend on the occurrence of concurrent operations and on the occupation duration of the spin-lock by occupying tasks. They are zero unless concurrent copy-in and copy-out operations of the same data element occur. Considering zero interference delays and zero spin-lock access delays, the $wcet_i^S$ and $wcet_i^E$ consist of only the time required to perform a read and write operation.

3.4 Static Buffering Protocol (SBP)

This section describes the SBP buffering protocol as a resource efficient data exchange mechanism to implement LET semantics for multi-core processors under plausible memory requirements, zero communication time, and zero jitter of sampling data. It should be noted that SBP is platform independent and can be used for single- and multi-core systems using any scheduling mechanism. SBP is a derivative of DBP [10] and it is designed to handle buffer accesses statically and according to LET semantics. It is *deterministic*, a quality it derives from the LET paradigm and its static nature.

3.4.1 Overview

SBP is a *static* and *centralized* buffering protocol [12]. It creates for every data element S_s a global array of elements, which is shared among several tasks. The buffer of S_s is notated as S_s^B. The reader and writer tasks have exclusive access to dedicated buffer elements at different points in time. SBP ensures that writer and reader tasks do not access the same buffer element at the same time instant. Simultaneous accesses to

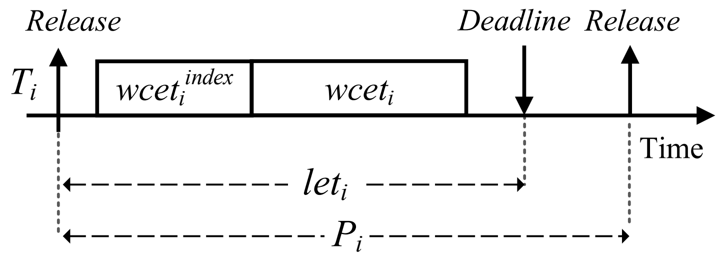


Figure 3.8: LET task composition in SBP. The computation task T_i has period P_i and the duration of LET interval let_i. The LET interval of T_i has a duration let_i equal to the deadline D_i. The $wcet_i^{index}$ is the WCET for initializing the buffer indexes at the beginning of execution. The $wcet_i$ defines the WCET of T_i.

the same buffer elements are only allowed among reader tasks because data stability is in this case not violated. SBP is platform *independent*, i.e, buffer decisions are not taken based on priorities as in DBP but based on task periods, offsets, and LET interval duration such that writing and reading operations in the buffer are performed according to LET semantics. Hence, reader tasks of any priority and executing on any core, read the version of data produced by a previous completed LET instance of the writer task. Accesses to buffer elements are allocated by SBP at design time and are stored in the *buffer schedule*. The schedule contains accesses to dedicated buffer elements for all reader and writer jobs released within the HP interval. Task execution and buffer accessing is repeated in identical order in each HP interval iteration.

SBP does not affect the original task composition of an application (defined in Section 2.1.2.3). As Figure 3.8 shows, driver tasks are not necessary in SBP because in SBP data exchanges do not physically occur at the boundaries of LET intervals as in PTP, but it is handled through non-overlapping buffer accesses that are calculated at design time. To align the task terminology of SBP with PTP, the task $T_i \in \tau$ is referred in SBP as computation task. SBP is designed to handle *multiple writers* of a data element. Every writer job has its dedicated buffer element to access during its LET interval. Jobs of the reader tasks read, in different LET intervals, outputs produced by jobs of writer tasks that have the end of LET right before or equal to the start of reader job's LET. In case multiple write jobs have coinciding LET ends, the system designer decides during the synthesis of the buffer schedule which output the reader jobs must consume. Multiple writers of the same data element are not common in single-core systems, but may occur in multi-core systems due to the allocation of runnables to multiple cores.

Although SBP can be integrated in application's code in various ways, this work adopts an approach that reduces the amount of global auxiliary data required to manage SBP semantics. The accesses to buffer elements are granted to tasks via indexes, which store the position in the buffer array that jobs can access during their LET interval. At the beginning of a task's execution, thus, before the computation starts, buffer indexes are initialized. The initialization operations take an amount of WCET, notated for each task T_i as $wcet_i^{index}$, which is considered a buffering utilization overhead. The size of $wcet_i^{index}$ is defined by the amount of index accesses. An index access refers to a write access on an index variable. Hence, for each data element S_s with N_R reader tasks and N_W writer tasks a total of $N_R + N_W$ index variables and a total of $N_R + N_W$ index accesses are added, in addition to the $N_R + N_W$ existing data accesses occurring in computation tasks. In SBP, the WCET of task T_i is the sum of $wcet_i^{index}$ and $wcet_i$.

An alternative approach implementing SBP, instead of indexes, is to define the buffer schedule as a global array variable instead of integrating it at the task's code. Accesses to buffer elements are enabled via *set* and *get* functions. The advantage of the first method is that the schedule is stored in the stack instead of the global memory. Furthermore, set and get functions take an amount of time to search on the buffer schedule for the correct element to assign to the requesting task to read and write. Although the initialization of indexes takes also execution time, it is more predictable and easily upper-bounded. The use of the indexes isolates the buffering overheads at the beginning of the task execution and simplifies their evaluation.

There are two ways to implement the granting of buffer accesses via indexes: the *local* and the *global* programming styles. In the *local* programming style, the intermediate produced value is stored in a local variable before it is written to the global buffer. In the *global* programming style, the intermediate value is written directly to the global buffer. In the local programming style, additional reusable memory, i.e., stack is required to store local variables. In this work, it is used to implement suppression of unnecessary writes to reduce the memory capacity required to store global buffers. An example describing the difference between programming styles is shown in Algorithm 2. Lines 12 to 14 show the writing operation of the global buffer in the local style and Line 15 show the writing operation using the global programming style. Line 2 to Line 11 show the initialization of buffer indexes. The task *let_comp* reads and writes the global buffer *data* using the corresponding indexes *b_index_r* and *b_index_w*, which are initialized at the beginning of the task's execution. Values of *b_index_r* and *b_index_w* are determined using the *job* counter, which is incremented at each task execution. *job* is reinitialized at the end of each HP interval with the maximum number of jobs *max_jobs* in a HP interval.

3.4.2 Buffering Algorithm

SBP uses the timing information of tasks such as periods, offsets, and duration of LET intervals to assign buffers to all reader and writer jobs that are released in the HP interval. Specifically, in SBP, buffering decisions are taken at every release time $r_{i,j}$ and absolute deadline time $d_{i,j}$ of each computation job $J_{i,j}$ of every task $T_i \in \tau$. A generic and formal description of SBP is given in Algorithm 3. The buffer schedule is constructed for every data element S_s. The buffer of S_s is defined as S_s^B. Algorithm 3 generates for each data S_s the buffer schedule $B_s = \{b_{i,j}^s | \forall J_{i,j}, \forall T_i \in \tau\}$, where $b_{i,j}^s$

Algorithm 2: Example of programming styles in SBP.

```
Input:
$job$ – The current job, $max\_jobs$ – the total number of jobs in the HP interval.
Output:
$job$ – The next job, $data[idx]$ – the $idx^{\text{th}}$ buffer element
Data: The global buffer $data[1,..,B_s]$
TASK let_comp (job, max_jobs):
    $b\_index\_r \leftarrow 0$;
    $b\_index\_w \leftarrow 0$;
    // initialize b_index variables
    switch job do
        case 1 do $b\_index\_r \leftarrow 0; b\_index\_w \leftarrow 1$ ;
        case 2 do $b\_index\_r \leftarrow 3; b\_index\_w \leftarrow 2$ ;
        case 3 do $b\_index\_r \leftarrow 5; b\_index\_w \leftarrow 4$ ;
    if $job = max\_jobs$ then
        $job \leftarrow 0$
    else
        $job \leftarrow job + 1$;
    // Option 1.  Local programming style
    $tmp\_data \leftarrow data[b\_index\_r]$;
    $tmp\_data \leftarrow tmp\_data + 5$;
    $data[b\_index\_w] \leftarrow tmp\_data$;
    // Option 2.  Global programming style
    $data[b\_index\_w] \leftarrow data[b\_index\_r] + 5$;
    return $job, data[b\_index\_w]$;
```

Algorithm 3: Static Buffering Protocol (SBP).

```
Input:
$U_{times}$ – The set of all unique ordered times,
$R_{hp}$ – The set of unique ordered release times,
$E_{hp}$ – The set of unique ordered end times,
$J^R_{hp}$ – The set of reader computation jobs released in one HP interval,
$J^W_{hp}$ – The set of writer computation jobs released in one HP interval,
$B^B_s$ – The buffer array of data element $S_s$
Output:
$B_s = \{b^s_{i,j} | \forall J_{i,j}\}$ – The buffer schedule of the data element $S_s$

Function sbp($U_{times}$,$R_{hp}$, $E_h$, $J^R_{hp}$, $J^W_{hp}$,$B^B_s$):
    $curr \leftarrow \{\}$ // The set of buffer indexes used by writers
    $uses \leftarrow \{\}$ // The set of buffer indexes used by readers
    $prev \leftarrow S^B_s[0]$ // The index of the last produced output

    foreach $time\ t \in U_{times}$ do
        // Release all used buffers before reassingment
        // STEP 1:  release buffer elements
        if $time\ t \in E_{hp}$ then
            // LET end of writer tasks
            foreach $job\ J_{i,j} \in J^W_{hp}\ with\ d_{i,j} = t$ do
                $prev \leftarrow curr^w_{i,j}$
                $curr \leftarrow curr \setminus \{curr^w_{i,j}\}$
            // LET end of reader tasks
            foreach $job\ J_{i,j} \in J^R_{hp}\ with\ d_{i,j} = t$ do
                $uses \leftarrow uses \setminus \{(J_{i,j}, b^s_{i,j})\}$
        // STEP 2:  assign buffer elements
        if $time\ t \in R_{hp}$ then
            // LET start of reader tasks
            foreach $job\ J_{i,j} \in J^R_{hp}\ with\ r_{i,j} = t$ do
                $b^s_{i,j} \leftarrow prev$
                $uses \leftarrow uses \cup \{(J_{i,j}, b^s_{i,j})\}$
            // LET start of writer tasks
            foreach $job\ J_{i,j} \in J^W_{hp}\ with\ r_{i,j} = t$ do
                $curr^w_{i,j} \leftarrow$ findFree ($J^R_{hp}, J^W_{hp}, J_{i,j}, uses, curr, prev, B^B_s$)
                $curr \leftarrow curr \cup \{curr^w_{i,j}\}$
                $b^s_{i,j} \leftarrow curr^w_{i,j}$
    return $B_s$
```

identifies the index of the buffer element of S_s^B that a job $J_{i,j}$ accesses during its LET interval. Algorithm 3 gets as input the set of unique timestamps U_{times}, the set of release times R_{hp}, and the set of end times E_{hp} and uses them to construct the buffer schedule B_s.

Definition 3.1: *Let U_{times} be the set of ordered unique times, and R_{hp} and E_{hp} be the set of all ordered release times and LET end times of all jobs that read or write data element S_s, respectively, and that are released in the HP interval of duration hp. The U_{times}, R_{hp}, and E_{hp} are defined as*

$$R_{hp} = \{r_{i,j} | \forall J_{i,j} \wedge j \in [1,n_i], \forall T_i \in \tau\}, \tag{3.5}$$

$$E_{hp} = \{d_{i,j} | \forall J_{i,j} \wedge j \in [1,n_i], \forall T_i \in \tau\}, \tag{3.6}$$

$$U_{times} = \{R_{hp} \cup E_{hp}\}. \tag{3.7}$$

SBP uses *prev* and *curr* pointers to generate the accesses to buffer elements for every job, a concept inherited by DBP. A set of *curr* variables, notated as $curr = \{curr_{i,j}^{w} | \forall J_{i,j}\}$, is recorded for every active job of the writer task to identify the buffer elements that are assigned to writer jobs. The $curr_{i,j}^{w}$ indicates the buffer index assigned to the writer job $J_{i,j}$. The *prev* variable stores the buffer's index of last produced value by any of the writer jobs. It points the buffer element of the last produced output and it is updated at every LET end of writer jobs. A set of used buffer elements is maintained for reader jobs, notated as *uses*, to ensure exclusive access on them and to disallow the algorithm to assign these elements to concurrently executing writer jobs. In Line 5, *prev* is initialized with the index of the initial value of the buffer. For each unique time t in the timestamps U_{times}, buffer elements are assigned and unassigned for every job $J_{i,j}$ of $T_i \in \tau$ that has release time $r_{i,j}$ or LET end $d_{i,j}$ equal to t. Hence, SBP involves two main steps for generating the buffer schedule, which are described as follows.

Step1: Unlock buffer elements – Before assigning buffer elements to the next released jobs, all elements that are used by the jobs that finish their LETs at time t are unlocked, such that they can be assigned to the next released reader and writer jobs. Hence, the condition in Line 8 must take place before the condition in Line 14. In the first iteration of the loop in Line 7, the time t corresponds to the the smallest release time, which means that before this time there are no jobs using any of the buffer elements. Therefore, in this iteration no buffers are released in Line 13. In the next iterations of the loop in Line 7, it is checked in Line 8 whether time t corresponds to the end time of any LET interval, i.e., if t is in the set of LET end times E_{hp}, then all reader jobs that have the end time equal to t are iterated in Line 12 and their buffer elements are unlocked in Line 13. Similarly, all writer jobs that have an end time equal to t are iterated in Line 9 and their buffer elements are unlocked in Line 11. When t is equal to a writer's deadline $d_{i,j}$, i.e., the LET end of writer job $J_{i,j}$, the *prev* variable takes in Line 10 the value of $curr_{i,j}^{w}$, i.e, the current value produced by the writer job. This value is assigned to reader jobs, released at time t or later, in Line 16. The value of *prev* is overwritten by every writer job and the last value of *prev* belongs to the output produced by the writer job with the latest LET end time. In case t equals the LET end of any reader job, in Line 13 the used buffer elements are removed from the *uses* to make them available and assign them to writer jobs in Line 19. Because the uses of a buffer element are stored for every job, Line 13 ensures that the uses by other tasks are still in *uses* also when the uses of some reader jobs are removed from the *uses*.

Step2: Assign buffer elements – In Line 5, *prev* is initialized with the index of the initial buffer element. In the first iteration of the loop in Line 7, time t is equal to the release time of the first released jobs. In Line 18, buffer elements are assigned to writer jobs if time t corresponds to the release time of any writer job. In case the time t corresponds to the release time of any reader job, the *prev* is assigned to all released jobs, which in the first iteration of the loop in Line 7 the initial buffer element is assigned to these jobs. In Line 17, each reader job and their accesses to the buffer elements are added to *uses* to avoid assigning the index of the *prev* element to writer jobs. In the next iterations of the loop in Line 7, if t is in the set of release times R_{hp}, which is checked in Line 14, then all jobs that have the release time equal to t are iterated in Line 15 and Line 18 and their buffer elements are assigned. When t is equal to the release time of a writer job $J_{i,j}$, an available element for writing is searched in Line 19 via the $findFree$ algorithm and is assigned to the $curr^w_{i,j}$ variable. The writer job $J_{i,j}$ writes on the buffer element with index $curr^w_{i,j}$ until the $d_{i,j}$ is reached. When t is equal to the release time of a reader job $J_{i,j}$, *prev* is assigned to $J_{i,j}$ for reading until the end of its LET interval is reached. In Line 17, the job $J_{i,j}$ and its assigned index $b^s_{i,j}$ are added in the list of *uses*. The algorithm ends after all times in U_{times} are iterated.

The $findFree$ function is defined in Algorithm 4. It assigns to a writer job $J_{i,j}$ a buffer element that is not used by any reader or writer job during its LET interval defined as $[r_{i,j}, d_{i,j}]$. Algorithm 4 reduces the amount of buffers by reusing buffer elements that are never read by any reader job. This is enabled by assigning *prev* to a job $J_{i,j}$ if it is never read by any upcoming reader job. Line 4 checks if *prev* is already assigned to a reader task. In this case, *prev* cannot be reassigned and an existing unused buffer element is searched in Line 5 and Line 6. The condition in Line 6 checks for buffer indexes that are not in *uses* or in *curr* and that are different from *prev*. If an unused element is found, then the index is returned in Line 7. If *prev* is not assigned to any reader task up to time $r_{i,j}$, then it is checked if it can be assigned to the current job $J_{i,j}$. Between Line 9 to Line 10, the Algorithm 4 searches for any writer job that produces an output between the LET start time $r_{i,j}$ and end time $d_{i,j}$ of $J_{i,j}$. If an output is produced by another writer job in this interval, then only reader jobs that have their LET start in $[r_{i,j}, d_{i,j})$ are of interest. If such reader jobs exist, then *prev* cannot be used because it is an output that is assigned in later iterations of Algorithm 3 to at least one reader job. Otherwise, the *prev* is reassigned in Line 11, but only if there are no other writer jobs with their deadline in the LET interval of $J_{i,j}$. In case an available buffer element is not found, in Line 12 a new element *be* is added to the buffer B^B_s and its index is assigned to the writer job in Line 11. The size of B^B_s is increased each time the Algorithm 4 does not find an unused buffer element to assign to a job $J_{i,j}$. Note that in the first call of

Algorithm 4: Searching for a free buffer element to assign to a writer job $J_{i,j}$. A buffer element is assigned to a job $J_{i,j}$ if it is not used by any reader or writer task and if it does not store an output that is consumed by any future reader job. The *prev* is reassigned if there are no reader jobs that can read the value of *prev*.

```
Input:
$J^R_{hp}$ – The set of reader computation jobs released in one HP interval,
$J^W_{hp}$ – The set of writer computation jobs released in one HP interval,
$J_{i,j}$ – The current writer job to assign a buffer element,
$uses$ – The set of indexes used by reader jobs,
$curr$ – The set of indexes assigned to writer jobs,
$prev$ – The buffer's index of storing the last produced output,
$B^B_s$ – The buffer array of signal $S_s$
Output: The buffer index of an available element to assign to $J_{i,j}$

Function findFree($J^R_{hp}, J^W_{hp}, J_{i,j}, uses, curr, prev, B^B_s$):
    $bs \leftarrow |B^B_s| - 1$ // The maximal buffer index of $B^B_s$ at time $r_{i,j}$
    if $prev \in uses$ then
        // Search for an existing unused buffer element
        foreach buffer index $i \in [0, bs]$ do
            if $i \notin uses \wedge i \neq prev \wedge i \notin curr$ then
                return $i$
    else
        if $prev \notin curr$ then
            if $(\nexists J_{k,l} \in J^R_{hp}$ with $r_{k,l} \in [r_{i,j}, d_{i,j})) \wedge (\nexists J_{m,n} \in J^W_{hp}$ with $d_{m,n} \in (r_{i,j}, d_{i,j}))$ then
                return $prev$ // Reasign $prev$
    // Otherwise, allocate $be$ as a new buffer element
    $B^B_s \leftarrow B^B_s \cup \{be \leftarrow \phi\}$
    return $bs + 1$
```

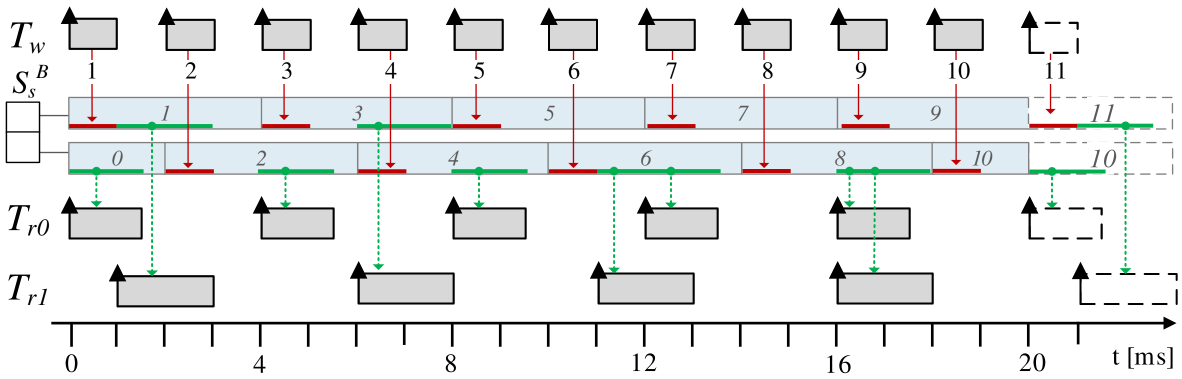


Figure 3.9: Example of buffer accesses using SBP for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The gray boxes represent the LET interval of each task instance. The green arrow indicates a *read* access and the red arrow a *write* access. The light blue boxes represent the lifetime of the global buffer S_B^s used interchangeably by T_w, T_{r0}, and T_{r1} in different time-intervals. The buffer content changes every time a new value is written. The horizontal red and green bold lines show the usage lifetime of a buffer element. A green horizontal line indicates that a buffer element is read by a reader job for the time interval indicated by the length of the line. Similarly, a red horizontal line indicates that the buffer element is written by a writer job. The dashed white boxes indicate the LET intervals and buffers of the first task instances of the next HP. The visualization of buffers is inspired by figures in [13].

the $findFree$ function, the buffer array B_s^B contains only the initial element. After the schedule is generated for the data element S_s, the number of elements in B_s^B defines the buffer size of S_s in SBP.

An example of tasks exchanging data using SBP is shown in Figure 3.9. The example shows one writer task T_w and two reader tasks T_{r0} and T_{r1}. Tasks write and read the data element S_s, which in SBP is transformed to the array S_s^B. The first job of reader task T_{r0} reads the initial value stored in $S_s^B[0]$. At the beginning of writer's LET, SBP assigns to writer task T_w the next available buffer element, in which no reader or other writer task has a parallel access. Hence, the first job of T_w has access to $S_s^B[1]$ during its LET interval. The red horizontal line between 0 ms and 1 ms indicates that the buffer element $S_s^B[0]$ is assigned to the first job of T_w. Furthermore, it indicates that task T_w writes value "1" at any point of time within $[0\,\text{ms}, 1\,\text{ms}]$. The duration of $[0\,\text{ms}, 1\,\text{ms}]$ corresponds to the LET interval duration of the first job of T_w. The lock of $S_s^B[1]$ is released in LET end of T_w, which corresponds to the end of the red line between 0 ms and 1 ms. Accesses to buffer elements are assigned for reader tasks in a similar way. Because the release time of the first job of reader task T_{r1} corresponds to the end of the first job of the writer task T_w, the first job of T_{r1} reads the output produced by the first job of T_w. The green line between 1 ms and 3 ms indicates that the buffer element $S_s^B[0]$ is assigned to the first job of T_{r1}. During this time, T_{r1} reads the value "1" at any time within its LET interval. As the Figure 3.9 depicts, except for reader tasks, no concurrent accesses occur on the same buffer elements between reader and writer jobs. This is as well indicated by the non-overlapping red and green horizontal lines. The usage lifetime of buffer elements by a reader and a writer task equals the LET interval duration of the task. In applications with tasks whose LET interval duration is less than the period, some time intervals may exist in which buffer elements are not used by any task, as indicated in Figure 3.9 by the empty spaces between green and red horizontal lines.

In each iteration of the HP interval, accesses of jobs to buffer elements are repeated in the same order as planned in the buffer schedule. At the start of the next HP interval, all jobs that read the initial value must read the last produced value of the previous HP interval. But, the last produced value might not always be stored in the initial element of the buffer. This means that if the last writer job in the previous HP interval writes the output to any buffer element and not to the element where the initial value is stored, then the first reader job of the next HP interval would not read this output but any old output generated in the previous HP interval. To overcome this situation, an *Interrupt Service Routine (ISR)*, referred to as *initialize* ISR, copies the value of the last produced output to the initial buffer element. The *initialize* ISR executes in the initialization phase of the next HP interval. It copies the last produced values to the initial buffer element for all data elements that require such data transfers. These copy operations are constructed statically during the buffer allocations and only for data elements that have the last value written in any buffer element except of the initial one. In Figure 3.9, this copy operation is not necessary because the last produced output of the previous HP interval is already stored in the initial buffer element.
In the worst-case, i.e., when initial buffers of all data elements must be refilled, two additional accesses, i.e., a read and a write access take place in the *initialize* ISR for each data element, independent of how many tasks accesses them. Hence, for each data element S_s with N_R reader tasks and N_W writer tasks a total of 2 data accesses are added in the aforementioned copy operation, in addition to the $N_R + N_W$ existing data accesses occurring in computation tasks. Let $wcet_{init}$ be the WCET of the *initialize* ISR. The $wcet_{init}$ represents the buffering overhead for re-initialization of buffer elements and is defined by the total memory accessing time of the aforementioned added data accesses. The *initialize* ISR, annotated as IR^{init}, is executed between iterations of HP interval. Its period is set equal to the HP duration.

The IR^{init} executes only in one core and handles buffer initialization of data elements that are exchanged between tasks running in any core. Buffer initialization operations in IR^{init} must lead to a stable state of data stored in buffer elements. Hence, IR^{init} must not execute while computation tasks are operating on buffer elements. To avoid data racing between IR^{init} and computation jobs of the same core, to IR^{init} is assigned a priority higher than those of computation tasks. Nevertheless, the priority does not avoid data racing of IR^{init} and computation jobs running on other cores. To handle such a situation, several approaches are available. If tasks are scheduled via TTS, then the isolation of IR^{init} is ensured by scheduling, i.e., by assigning a time-frame to IR^{init} in which it executes without interfering with the execution of computation jobs running on the same or on other cores. Additionally, computation jobs running on any core are scheduled such that they only execute after the time-frame of IR^{init} has finished. This approach is not applicable if tasks are scheduled via *Fixed-Priority*

Scheduling (FPS). In this case, the core on which IR^{init} executes must be synchronized with other cores until IR^{init} finishes the execution. One approach is to create for each core a dedicated high-priority task, which wait actively for IR^{init} to set an OS event at the end of its execution. These tasks are configured such that they execute in a loop until the event that indicates IR^{init}'s termination is set. If SBP is applied to asynchronous tasks, then in order to preserve correct execution between IR^{init} and computation jobs, all computation jobs of the previous HP interval must finish their execution before the start of the next HP interval. This is possible by constraining the offsets and LET duration of computation tasks as

$$\forall T_i \in \tau, O_i \in [0, P_i) \Longleftrightarrow let_i \leq P_i - O_i \land P_i \neq O_i. \tag{3.8}$$

Hence, SBP is feasible for periodic *synchronous* and *asynchronous* tasks with *implicit* and *constrained* deadlines that fulfill Equation (3.8). The assignment of offsets to values less then the periods is done such that the expected dataflow between tasks is ensured.

SBP is designed for communication between LET tasks with purely periodic activation. Due to its static nature, SBP cannot handle dynamic changes of the system at run-time such as the mixed communication between periodic and sporadic or a-periodic tasks. SBP requires that the behavior of a task's execution is known at design time and defined by attributes such as period, offset, and LET duration of tasks, which is not the case for sporadic and a-periodic tasks. The correctness of SBP is highly sensitive to unpredictable activation jitters of LET intervals and tasks. SBP is applicable for event-driven AUTOSAR applications as long as these jitters are predictable, upper-bounded, and considered during buffer schedule synthesis.

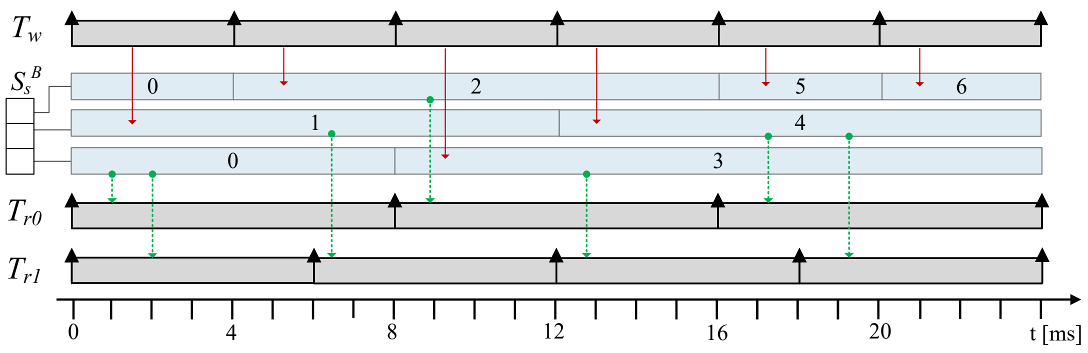
The static nature of SBP offers convenient verification of time and value correctness of the exchanged data. This is because SBP has zero communication time and zero jitters on data sampling at the boundaries of LET intervals, which ensures that the delay between the writer's output and the reader's input is the time difference of the writer's LET end time and the beginning of the reader's LET. At run-time, if tasks violate the end of their LET intervals, then the buffer schedule of SBP is violated, which leads to accesses of incorrect values by other tasks. One way to avoid incorrect SBP buffering behavior in case of LET overruns is to forcibly terminate the task causing the violation. LET interval overruns can occur in case of software malfunctions, poor scheduling design, or when scheduling is generated under unrealistic bounds of WCET. It should be noted that LET overruns violate the dataflow and timing determinism of LET, regardless of whether SBP or PTP is used as the buffering mechanism.

The SBP's buffering decisions are taken statically using Algorithm 3. The purpose of calculating buffering decisions statically is to avoid execution overheads that are caused by running the Algorithm 3 at run-time. This approach offers also the benefit of calculating the exact buffer size and estimating precisely at design time the memory requirements to effectively deploy LET systems. Nevertheless, SBP can be applied to take buffering decisions dynamically by executing Algorithm 3 at the beginning and at the end of a task's LET, such that buffers are assigned to jobs before they start execution and are released at the end of their LETs after they finish their execution. In this case, buffering overheads occur at LET boundaries for searching, assigning, and releasing buffer elements. The *LET Dynamic Buffering Protocol (LDBP)* refers to the dynamic SBP to distinguish it from the DBP (described in Section 3.2.1.2), which takes buffering decisions differently from SBP. In LDBP, offsets, and LETs of tasks are not required to be constrained by Equation (3.8) because buffer decisions are taken dynamically and the *initialize* ISR is not required to reinitialize buffers.

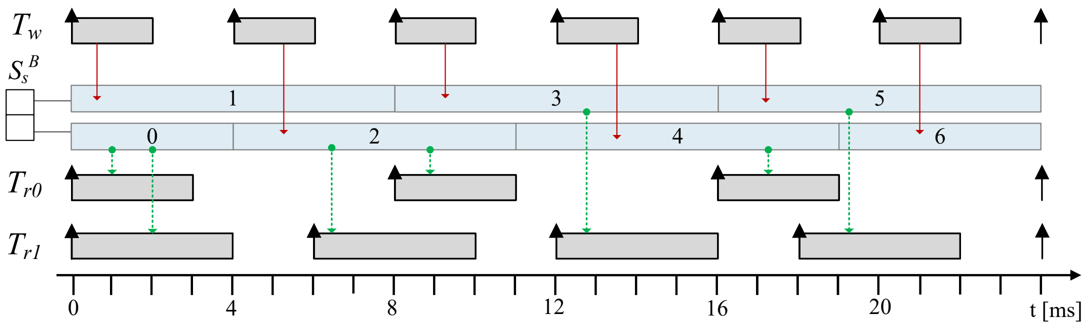
3.4.3 Buffer Size

The size of the global memory required to store SBP buffers of a data element S_s is defined as a multiple of the data size of S_s and the number of buffer elements required to preserve LET semantics for S_s. Unlike PTP, for SBP the exact number of buffers is defined by the overlaps between LET intervals of reader and writer tasks and is calculated by the Algorithm 3. The overlaps are defined by periods, offsets, and LET duration of these jobs. For instance, if two computation tasks, one reader and one writer of the same data element have non-overlapping jobs, then only one buffer element is required for both, which is used interchangeably between them at different time intervals. These overlaps can be reduced by changing the offsets or the duration of LET intervals of tasks.

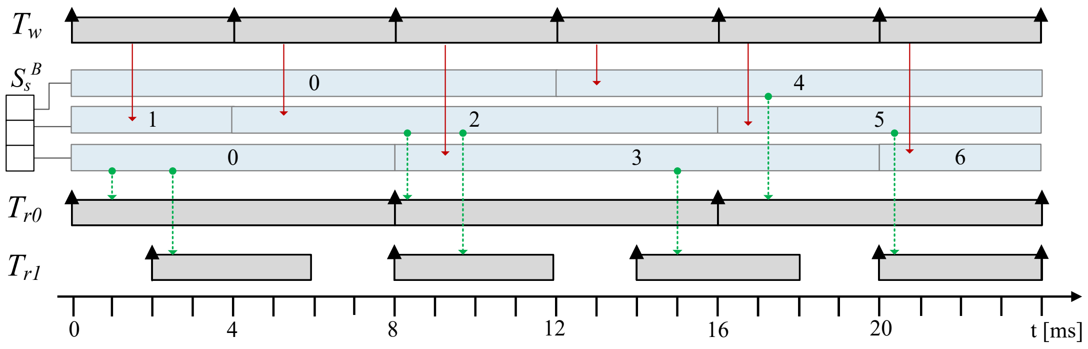
Figure 3.10 shows an example of three tasks and their accesses to buffer elements as defined by Algorithm 3. In Figure 3.10a, tasks have synchronous offsets and LET duration equal to their period. To preserve LET semantics, three buffer elements are required. In Figure 3.10b, the LET duration is decreased, which reduces the overlaps between jobs and results in a buffer size of two. Another benefit of reducing LET interval duration is the decrease of data ages and end-to-end delays. As shown in Figure 3.10b, the second job of T_{r1} reads the output produced by the second job of T_w instead of the output produced by the first job of T_w. In Figure 3.10c, the same task set is shown but by adding an offset to task T_{r1} and reducing its LET duration. In this case, three buffer elements are needed. This example shows the sensitivity of end-to-end delays and of the buffer size of SBP to timing parameters of tasks such as offsets, periods, and LET duration. This aspect of SBP provides the ability to control and reduce the memory capacity required to preserve LET semantics, e.g., by the optimal assignment of task timing parameters. In PTP, the buffer size is independent on the timing parameters of tasks, but is defined by the amount of reader and writer tasks. The buffer size for the example shown in Figure 3.10 is "4" for PTP.



(a) Synchronous offsets. LET interval duration is equal to the task's period.

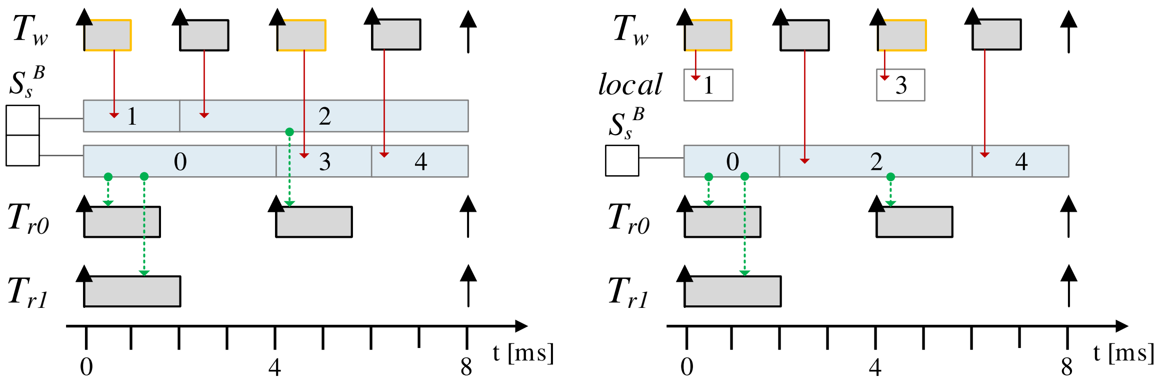


(b) Synchronous offsets. LET interval duration is less than the task's period.



(c) Asynchronous offsets. LET interval duration is less or equal to the task's period.

Figure 3.10: Example of reducing the buffer size and end-to-end delays in SBP through changes of activation offsets and LET interval duration for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The gray boxes represent the LET interval of each task instance. The green arrow indicates a *read* access and the red arrow a *write* access. The light blue boxes represent the global buffer S_B^s used by T_w, T_{r0}, and T_{r1} in different time intervals. The visualization of buffers is inspired by figures in [13].



(a) Before suppressing writes.

(b) After suppressing writes.

Figure 3.11: Example of suppressing unnecessary writes in SBP for one writer task T_w and two reader tasks T_{r0} and T_{r1}. The gray boxes represent the LET interval of each task instance. The green arrow indicates a *read* access and the red arrow a *write* access. The light blue boxes represent the global buffer S_B^s used by T_w, T_{r0}, and T_{r1} in different time-intervals. Jobs of writer task T_w with LET interval marked in yellow produce an output that is not consumed by any reader job of T_{r0} and T_{r1}. The writing of these jobs is suppressed such that they only write to the local variable named *local*, which is indicated by the white boxes. The *local* variable has a lifetime equal to the execution time of the writer job. The visualization of buffers is inspired by figures in [13].

3.4.4 Memory Optimizations

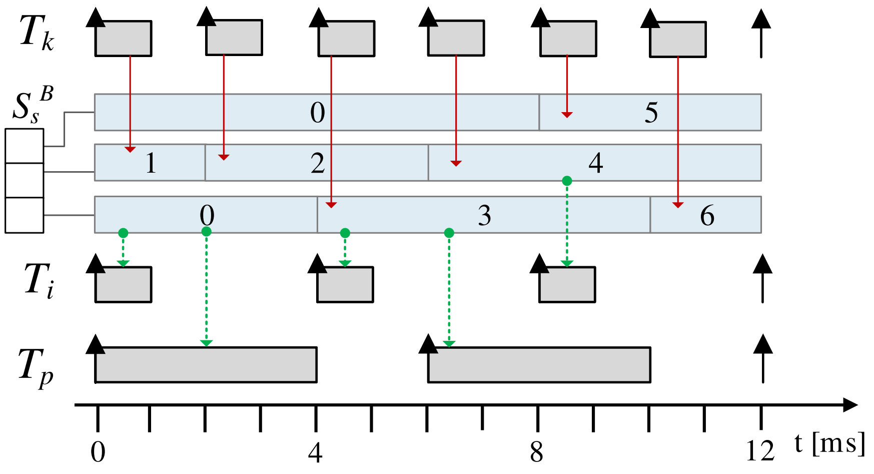
SBP is designed to require less memory for buffers than PTP. This work introduces an additional approach to further reduce the amount of memory required to store global buffers that the SBP needs to preserve the communication between tasks according to LET semantics. The amount of global buffers is reduced by suppressing unnecessary writes of dedicated writer jobs. A produced output by a writer job is marked as an "unnecessary write" when it is overwritten by another job of the same or of a different task and is never consumed by any reader job, or when the system designer defines through *data age* constraints that a dedicated task reads an older value of data instead of the most recently produced one. In this work, writes are suppressed for both cases.

The suppression of unnecessary writes is performed during generation of the buffer schedule. Suppressing a write operation in a writer job means that the job does not write to any of the buffer elements but to a dedicated local variable. Therefore, this optimization is only applicable when the *local* programming approach is used. Due to local temporal variables, the local programming approach has higher stack memory demands compared to the *global* programming model. Therefore, the local programming approach is applied for data elements for which the suppression of writes is possible and for which the memory required to store global buffers decreases.

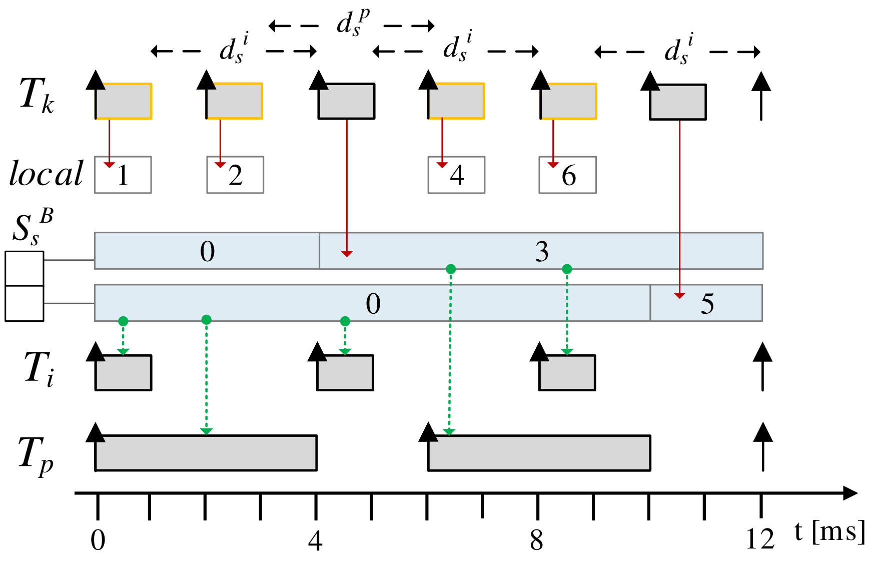
An example of suppressing writes in case of under-sampling is shown in Figure 3.11. The data element S_s is produced by the writer task T_w every 2 ms and consumed by tasks T_{r0} and T_{r1} every 4 ms and 8 ms, respectively. Figure 3.11a shows the buffer schedule before suppressing the writes of jobs released at times 0 ms, 4 ms, 8 ms and 12 ms. Figure 3.11b shows the buffer schedule after suppressing the writes. As Figure 3.11b depicts, the jobs with suppressed writes write no longer to the global buffer but to the local variable named *local*. The number of global buffer elements is reduced to one from two compared to when buffers are not suppressed. In terms of memory capacity, assuming that S_s has the size of 8 bits, the global buffer's memory is 16 bits without suppressing the writes and 8 bits with suppressing the writes. But in the second case, 8 bits are additionally required to store the local variable. The difference is that the local variable is stored in the stack of the task, which is used only during task's run-time and is shared with other tasks at different executing times. In this case, by suppressing writes the global memory for storing buffers is reduced, but the stack size dedicated to these tasks is relatively increased. This approach is a trade-off between the reusable memory at run-time, i.e., stack and the global memory.

In addition to the identified unused writes, the proposed SBP algorithm uses *data age* constraints [59] to reduce the memory required for global buffers. Data age constraints are defined in the *Timing Extensions* document of AUTOSAR [59], but in the context of this work their definition and interpretation is adapted as follows. The data age constraints of every data element S_s and reader task T_i is denoted as d_s^i and defines that each job $J_{i,j}$ of T_i reads any value of S_s produced up to d_s^i time relative to the release of the job even if its not the most recently produced value. For instance, a data age of 4 ms of reader task T_i means that every job $J_{i,j}$ of T_i reads any value produced between the time interval $[r_{i,j} - 4\,\text{ms}, r_{i,j}]$. After all defined data age constraints are resolved, then the resulted unused writes of S_s are suppressed. Such a design affects the data flow between tasks and is applicable only if functional requirements are fulfilled and if it is approved by the system's designer. An example of suppressing writes using data age constraints is shown in Figure 3.12. The buffer size for data element S_S is decreased from three to two elements after suppressing of the writes using the data age constraint with duration of 3 ms for both reader tasks T_i and T_p.

The suppressing of writes through *data age* constraints does not always guarantee that the buffer size is decreased. This is because some overlaps of LET intervals between reader and writer jobs are unavoidable also when data age constraints suppress the writes in some writer jobs. An example of such a situation is shown in Figure 3.13, where the suppressing of the writes does not decrease the number of buffer elements, compared to non-suppressing of the writes. This occurs because the first job of the reader task T_i reads the initial value, which can never be suppressed. During the time interval of this job, the first writer job of task T_k requires an additional buffer element. The output of this job is not suppressed because it is required by the second job of T_i executing in the next HP interval. However, the impact of suppressing unused writes using data age constraints is higher in asynchronous tasks or task sets with more than one reader task and more than two buffer elements. This work assumes that data ages are valid and lead to some degree of buffer size minimization.



(a) Before suppressing writes.



(b) After suppressing writes.

Figure 3.12: Example of suppressing unnecessary writes in SBP for one writer task T_k and two reader tasks T_i and T_p. The suppressing of writes is based on data age constraints d_s^i and d_s^p, which have a duration of 3 ms. The gray boxes represent the LET interval of each task instance. The green arrow indicates a *read* access and the red arrow a *write* access. The light blue boxes represent the global buffer S_B^s used by T_k, T_i, and T_p in different time-intervals. Jobs of writer task T_k with LET interval marked in yellow produce an output that is not consumed by any reader job of T_i and T_p. The writing of these jobs is suppressed such that they only write to the local variable named *local*, which is indicated by the white boxes. The *local* variable has a lifetime equal to the execution time of the writer job. The visualization of buffers is inspired by figures in [13].

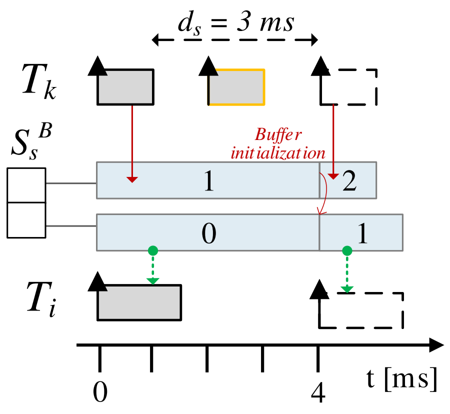


Figure 3.13: Example of suppressing unnecessary writes in SBP for one writer task T_k and one reader tasks T_i. The suppressing of writes is based on the data age constraint d_s^i, which has a duration of 3 ms. The gray boxes represent the LET interval of each task instance. The green arrow indicates a *read* access and the red arrow a *write* access. The light blue boxes represent the global buffer S_B^s used by T_k and T_i in different time-intervals. The second job of the writer task T_k, with LET interval marked in yellow, is suppressed considering the data age constraint d_s^i. The job writes during execution on the local variable named *local*, which is indicated by the white boxes. The suppressing of the write does not decrease the number of buffers. The *local* variable has a lifetime equal to the execution time of the writer job. The dashed white boxes represent the LET intervals of the first task instances of the next HP interval. The red line between the first and the second buffer element of S_B^s indicates the buffer initialization at the start of the next HP interval. The visualization of buffers is inspired by figures in [13].

The suppressing of unnecessary writes without data ages is an additional optimization to the one described in the *findFree* function, defined in Algorithm 4, in which the buffer index of a *prev* pointer is reassigned to a writer task if no upcoming reader jobs exist that read the output stored in the buffer element pointed by *prev*. A detailed description of the approach of suppressing the writes with and without data ages in SBP is given in [12]. If for a data element a write is suppressed in a writer task T_i, then one local variable is created for T_i. The size of a local variable is the same as the size of the data element for which the write is suppressed. Therefore, the amount of local variables resulting from suppressing of the writes is defined by the number of writer tasks that have suppressed writes and the amount of data elements that are suppressed. For each data element S_s with N_R reader tasks, N_W writer tasks, and $N_W^{'}$ writer tasks with suppressed writes a total of $2 * N_W^{'}$ data accesses are added in copy operations of local variables, in addition to the $N_R + N_W$ existing data accesses occurring in computation tasks. A copy operation of a local variable consists of a read access on a local variable and a write access of a buffer element. Copy operations occur at the end of writer task's execution.

3.4.5 Run-time Overheads

In SBP, the buffering run-time overheads consist of the processor's utilization for initializing indexes at the beginning of each task's execution, refilling of initial buffers at the beginning of each HP interval, and copy operations on local variables in case of suppressed writes. The $wcet_i^{index}$ denotes the WCET of initializing indexes at the start of a task T_i and the $wcet_{init}$ defines the WCET of buffer initialization at the beginning of the HP interval with duration hp. The value of $wcet_i^{index}$ is influenced by the length of the buffer schedule, i.e, the amount of jobs released within one HP interval and the number of data elements that are read and written by the task. The higher the number of jobs of a task, the more buffers must be initialized inside the task and, hence, the higher gets the run-time for initializing indexes. Note that buffers are initialized for all data elements that a task accesses during its execution. Therefore, the more data elements are read and written by a task T_i, the higher $wcet_i^{index}$ gets. The $wcet_i^{local}$ is the WCET of copy operations for the local programming approach. If a task T_i does not have suppressed writes, then the $wcet_i^{local}$ is zero.

The buffer run-time overhead O_{sbp} is defined as

$$O_{sbp} = \sum_{\forall T_i \in \tau} \left(\frac{wcet_i^{index} + wcet_i^{local}}{P_i}\right) + \frac{wcet_{init}}{hp}. \tag{3.9}$$

In the example of Algorithm 2, the $wcet_i^{index}$ is the execution time of instructions between Line 2 and Line 11. The $wcet_i^{local}$ is the execution time for reading the local data element in Line 14 and copying of the global data to the local element in Line 12.

The following definitions of $wcet_i^{index}$ and $wcet_i^{local}$ of each task T_i and of the $wcet_{init}$ are only a formal description of the elements affecting buffering overheads of SBP, and not an exact calculation of the WCET values.

Let $S_i^r = \{S_r | r \in \mathbb{N}^+\}$ and $S_i^w = \{S_w | w \in \mathbb{N}^+\}$ denote, respectively, the set of global data elements that are read and written by task T_i. For every data element $S_r \in S_i^r$ and $S_w \in S_i^w$ of each task T_i, the local variables for representing indexes are defined as Idx_r and Idx_w, respectively. The time required to write an index variable Idx_r is defined as a memory access delay WAd_r^{idx} and the time delay occurring due to bus and memory interferences during the write operation is defined as an interference delay Id_r^{idx}. Similarly, the write memory access delay WAd_w^{idx} and the interference delay Id_w^{idx} of an index variable Idx_w are defined. The $wcet_i^{index}$ of each task T_i is defined as

$$wcet_i^{index} = \sum_{\forall S_r \in S_i^r} (WAd_r^{idx} + Id_r^{idx}) + \sum_{\forall S_w \in S_i^w} (WAd_w^{idx} + Id_w^{idx}) + c, \tag{3.10}$$

where c is the worst-case execution time of the rest of operations during buffer initialization. In the evaluation results of this work, c is assumed as zero.

The $S_{init} = \{S_f | f \in \mathbb{N}^+\}$ denotes the set of data elements that must be reinitialized at the beginning of the HP interval. For every data element $S_f \in S_{init}$, the time required to read S_f is defined as a memory access delay RAd_f. Similarly, the WAd_f defines the memory access delay for writing a data element S_f. The time delay occurring due to the bus and memory interferences during a read and write operation is assumed zero because buffer initialization occurs on one core and other cores wait actively until the initialization finishes. Therefore, concurrent accesses to buffer elements do not occur. The $wcet_{init}$ is defined as

$$wcet_{init} = \sum_{\forall S_f \in S_{init}} (RAd_f + WAd_f) + t_{event}, \tag{3.11}$$

where the t_{event} defines the WCET for setting and clearing of events required to synchronize cores during buffer initialization. If the TTS is used to schedule tasks, then the event synchronization is not required because the schedule can be planned in a way that other cores do not execute any task during buffer initialization. In this case, the t_{event} is zero. The write and read accesses during buffer initialization and write accesses of index variables are not protected by spin-locks. The index variables are local in the task context and are assumed to require only one processor cycle to read and write. Furthermore, no concurrent accesses to buffer elements take place during buffer initialization. Therefore, data stability issues do not occur in both cases.

The $S_i^{sw} = \{S_{sw} | sw \in \mathbb{N}^+\}$ denotes the set of data elements that have suppressed writes in task T_i. Let RAd_{sw} and WAd_{sw} be the memory access delay for reading and writing a data element $S_{sw} \in S_i^{sw}$. The interference delays during a read and write operation are defined as RId_{sw} and WId_{sw}, respectively. A local variable S_{sw}^{local} is defined for each $S_{sw} \in S_i^{sw}$. The memory accesses and interference delays for the read and write of each local variable S_{sw}^{local} are defined as RAd_{sw}^{local}, WAd_{sw}^{local}, RId_{sw}^{local} and WId_{sw}^{local}. If bus and memory inferences do not occur during the read and the write of S_{sw}^{local}, then RId_{sw}^{local} and WId_{sw}^{local} are zero. The same applies for RId_{sw} and WId_{sw}. The $wcet_i^{local}$ of every task T_i with suppressed writes is defined as

$$wcet_i^{local} = wcet_i^{local}(s) + wcet_i^{local}(e) \tag{3.12}$$

where $wcet_i^{local}(s)$ and $wcet_i^{local}(e)$ are the WCETs of copy operations at the start and the end of task T_i, respectively. They are defined as

$$wcet_i^{local}(s) = \sum_{\forall S_{sw} \in S_i^{sw}} (RAd_{sw} + RId_{sw} + WAd_{sw}^{local} + WId_{sw}^{local}), \tag{3.13}$$

$$wcet_i^{local}(e) = \sum_{\forall S_{sw} \in S_i^{sw}} (RAd_{sw}^{local} + RId_{sw}^{local} + WAd_{sw} + WId_{sw}). \tag{3.14}$$

The access and interference delays of a data element $S_{sw} \in S_i^{sw}$ are considered in the calculation of $wcet_i^{local}(s)$ if the task T_i reads and writes S_{sw}. Otherwise, if T_i has only writes S_{sw}, then the delays are not considered. Copy operations in the context of writer tasks with suppressed writes are not needed to be protected via spin-locks because concurrent accesses do not occur for local variables and buffer elements. Local variables are private variables for each task and are created to apply the local programming approach. Additionally, the update at the end of task's execution occurs exclusively to a buffer element that is assigned by SBP.

3.4.5.1 Heuristic of Data-to-Memory Allocation

In typical multi-core memory architectures, the accessing delays of local memories are lower when data are accessed by the local core of the memory and higher when accessed by other cores. Similarly, the delays of accessing shared memory is higher than accessing the local memory of the core. Therefore, to reduce memory access delays caused by reading and writing a data element, the *smallest-accessed-memory-delay* heuristic to allocate data elements to memory components is used. The heuristic allocates data elements to memory components as follows. Every data element S_s accessed by any core is allocated to the memory component in which it causes the lowest access utilization. The access utilization of a data element S_s is calculated for every core and every memory component because accessing delays of S_s by tasks running on different cores is different depending on where the data element is allocated to. A data element can be accessed by several tasks at a different accessing rate and frequency, i.e., period. For instance, task T_i accesses data element S_s three times during its execution (the accessing rate) and every 1 ms (the period). Tasks are mapped for execution to different cores. Let $U^s_{u,m}$ be the access utilization of data element S_s, which is mapped to memory M_m and accessed by all tasks running on core C_u, then

$$U^s_{u,m} = \sum_{\forall T_i \in T^s_u} \frac{rate_i * delay^s_m}{P_i}, \tag{3.15}$$

where T^s_u is the set of tasks that run on core C_u and access data element S_s, $rate_i$ is the amount of times that S_s is accessed by task T_i, and $delay^s_m$ is the memory accessing time of S_s. The value of $delay^s_m$ depends on the size of data element S_s and the accessing delay of memory M_m. The heuristic allocates data elements to memories such that the lowest access utilization is caused by accessing the data elements by tasks of all the cores. Hence, the data element S_s is allocated to memory M_m if the total access utilization $\sum_{\forall C_u} U^s_{u,m}$ is the smallest among total access utilization values of all memory components. If M_m has no capacity, then the next memory with the smallest total access utilization is considered to allocate the data element. Data elements with the highest access utilization are allocated first.

Memory and bus interferences occur when multiple tasks attempt to access concurrently the same memory and bus component, causing in this way interference delays to each-other. These delays are not considered by this heuristic because concurrent accesses cannot be evaluated without exact definition of every task's instruction, which is not possible when WCET is used to represent the total task's instructions. However, this heuristic offers the possibility to reduce the memory accessing times of SBP buffers and the memory accessing time of data elements by LET driver tasks of PTP protocol to an extend. Therefore, it is applied after the SBP and PTP buffer synthesis.

3.5 Evaluation

This section describes the evaluation of the introduced LET buffering protocols for automotive applications. The performance of both protocols is evaluated using synthetic benchmarks and an industrial EMS model provided in the FMTV challenge [76]. The synthetic benchmarks are performed using models with characteristics of industrial applications. The following aspects of the PTP and SBP protocols are evaluated.

1) The global and the stack memory size required to store data and buffer elements created by each protocol is evaluated. The global memory size refers to the global memory capacity required to store global buffers, including global data elements for PTP. The stack memory size refers to the stack capacity required to store *index* and local variables.

2) The run-time overheads caused by buffering decisions for each buffering protocol to preserve LET semantics are observed. In PTP, these overheads consist of the processor's utilization of copy-in and copy-out operations at the boundaries of LET and the execution utilization of data-protection operations used to preserve stability of buffering. In SBP, they consist of the processor's utilization of initializing indexes at a task's initialization and refilling of initial buffers at the beginning of each hyper-period. In case of suppressed writes with local programming approach, the run-time overheads include the time required to copy inputs to local variables and copy outputs to the assigned buffer element. The formal definition of overheads for PTP and SBP is given in Section 3.3.4 and Section 3.4.5, respectively. In this evaluation, these overheads are estimated using simulation results. This means that the actual overheads are reported by the simulation and not by WCET analysis. To improve the run-time delays caused by memory accesses, the global data elements and buffers are allocated to memories by using the *smallest-accessed-memory-delay* heuristic.

This evaluation is based on the simulation of application's buffering and execution behavior using the TA.Simulation option of TA Tool Suite [85] of Vector Informatik GmbH [1], which is an industry proven tool for model-based simulation of embedded applications. The TA Tool Suite was extended to synthesize LET buffering of automotive applications and to simulate them using the TTS and FPS strategy.

[1]www.vector.com

Period (ms)	**Chassis (# tasks)**	**EMS (# tasks)**
1	4	2
2	1	4
2.5	1	.
5	9	8
10	7	2
20	.	1
50	.	1
200	.	2
1,000	.	2

Table 3.2: Parameters of tasks. The EMS models have a hyper-period of 1 s and the Chassis models have a hyper-period of 10 ms.

Attributes	**Values**
Data Elements (#)	100 – 5,000
Data Accesses (#)	500 – 25,000
Data Size (bit)	8 – 256

Table 3.3: Parameters of data elements.

3.5.1 Synthetic Benchmarks

The main goal of the synthetic benchmarks is to identify the memory and the utilization costs of SBP and PTP considering several characteristics of software applications. Because the amount of data elements and data exchanges between tasks have the highest impact on the buffering run-time and memory overheads, several synthetic application models are generated that differ on the amount of data elements, data accesses, and task periods. Both protocols are evaluated using application models with attributes that characterize the ones of EMS and Chassis domains such as task periods, the amount of tasks, the amount of global data elements, and data accesses. For evaluation, applications of EMS and Chassis domains have been chosen for exposing the impact that the granularity of the system has on the performance of both protocols.

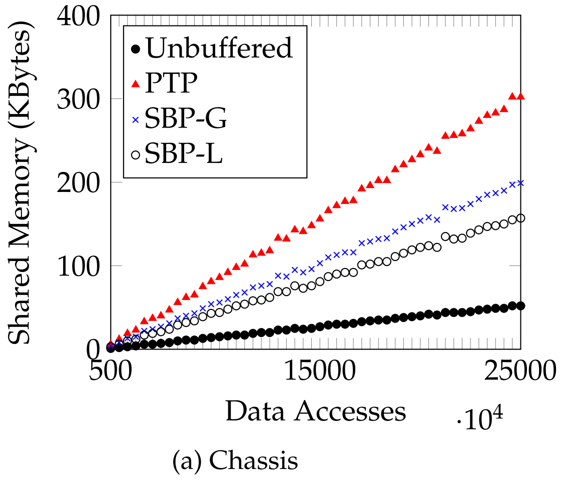
Models are generated randomly based on the uniform distribution of the parameters depicted in Table 3.2 and Table 3.3. Fifty synthetic application models are generated for each configuration. To isolate the effects of other application attributes, a fixed amount of tasks and fixed periods for all tasks have been configured. Each application has 22 tasks. In general, the number of tasks configured in an application is highly dependent on the design and optimization aspects applied by engineers during software integration phase. The offsets of all tasks are zero and the LET interval duration is equal to the period of the task. Although not included in this study, it is expected that applications with offsets greater than zero and LET interval duration less than the period produce similar results, with the difference that in PTP, due to the reduced LET interval duration and the added buffering load, not all applications would be schedulable. Tasks are scheduled using TTS and the schedule is constructed using the *Earliest Deadline First (EDF)* heuristic.

The amount of global data elements for each model is in the range of $100 - 5{,}000$ and the amount of data accesses is in the range of $500 - 25{,}000$. The typical communication schemes observed in industrial applications are the *single-writer single-reader* and *single-writer multiple-readers* communication. To increase the complexity to a certain extent, models are generated to consider two writer runnables and three reader runnables for every data element. Runnables are mapped to any task. A task can read and write the same data element but in different runnables. Furthermore, a data element can be read multiple times within the same task, i.e., in the context of different runnables. Similarly, a data element can be written multiple times in the same task. Therefore, each data element is written by $1 - 2$ writer tasks and read by $1 - 3$ reader tasks.

This evaluation considers data types of the size between 8 bit and 256 bit. The size of each index variable of SBP is assumed as 8 bit. Unnecessary writes are suppressed in SBP using also data age constraints. In this evaluation, 40 % to 52 % of data elements have data age constraints of duration $3 - 7$ times the period of the writer task. It is assumed that enforcing data ages with higher duration that the defined ranges cause to LET tasks to read intolerably old version of data.

Applications are simulated considering a hexa-core processor with execution frequency of 200 MHz for each core. A processor with six cores is chosen based on the assumption that the application is highly paralellized, i.e., tasks are distributed to different cores and the energy consumption is kept low without loss of the performance. Furthermore, due to higher parallel execution of tasks, the multi-core effects on LET communication can be captured in this evaluation. The processor has one shared memory, which is shared among multiple cores, and one local memory for each core. The data size of the memory and bus is 64 bit. The accessing time of a data element with size up to 64 bit takes 5 ns if the data is allocated to the local memory of the accessing core. Otherwise, 10 ns accessing time is involved if the data element of size up to 64 bit is allocated in the shared memory or in the local memory of other cores.

The execution time for requesting and releasing a spin-lock is 200 ns. The execution times of spin-lock accesses is implementation dependent and can vary among different OSs. In this case study, due to the high amount of data accesses in a model, an



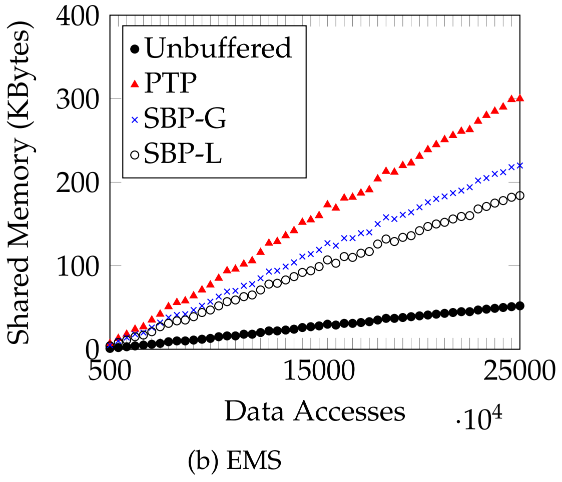


Figure 3.14: Shared memory capacity of the Unbuffered, PTP, SBP-G and SBP-L models for **(a)** Chassis and **(b)** EMS applications.

optimized implementation of spin-lock handling is assumed. A pessimistic execution time for accessing spin-locks causes in these models simulation results that have deadline violations. If deadline violations occur, then the impact of spin-locks on the buffering overheads cannot be properly evaluated.
A 44 % computation load is generated for every model, such that after the buffer synthesis enough processing capacity is present for scheduling the system without deadline violations. The computation load does not include the memory accessing time of data elements. The load resulting from the memory accessing times is added during the simulation.

The *Unbuffered* models contain the system without applying any buffering strategy and considering that the global data is accessed at any time during a task's execution. Three models are created after buffer and schedule synthesis for every *Unbuffered* model. The *PTP* models contains the application with the PTP buffer information. The *SBP-G* and *SBP-L* models contain the buffered application using SBP with global and local programming style, respectively. Models are simulated with a duration of three times the HP.

3.5.1.1 Memory Evaluation

This section describes the evaluation results of the memory capacity requirements of each buffering protocol. The results of this evaluation are also given in Section A.1.0.1 of Appendix A.

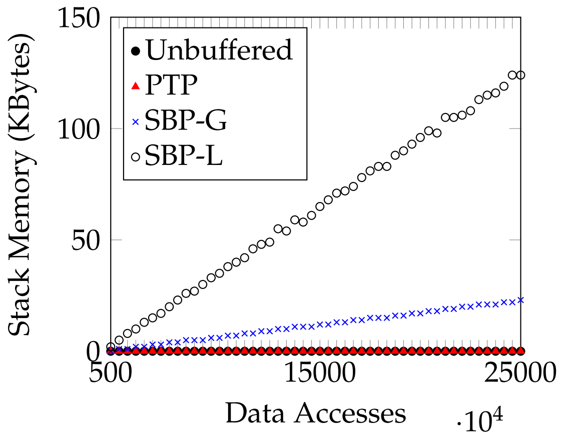
Evaluation results for global memory—Figure 3.14 shows the global memory capacity required for SBP and PTP. The graphs depict the memory in KBytes required to store global data elements and buffers for each protocol. The x-axis shows the number of unique data accesses of the initial, i.e, unbuffered models. Considering the configuration of 5 runnables accessing a data element, i.e., 2 writers and 3 readers, the data accesses depicted in this graph shows five times the number of data elements of the unbuffered models. The number of data elements differs between models by 100.

Which global memory needs has each protocol? The global memory capacity is impacted by the data size and the number of data elements or buffers that result after buffer synthesis. In PTP, assuming that reader and writer runnables of a data element are mapped to different tasks, the number of data elements after buffering is defined as $(N_R + N_W + 1) * data = 6 * data$, where $data$ defines the number of data elements before buffering, i.e., of the unbuffered models, and N_R and N_W define the number of reader and writer tasks, respectively. If the writer runnables of a data element are all assigned to the same task, only one buffer element is required for the writer task. The same holds for reader runnables. In this case study, reader and writer runnables of the same data element are mapped to any task. Consequently, N_R has any value in $1 - 3$ and N_W in $1 - 2$. Therefore, the actual amount of data elements in PTP is less than $6 * data$. In SBP-G, the exact amount of buffers is defined by Algorithm 3. In SBP-L, the amount of buffers is influenced by the amount of writes that can be suppressed.

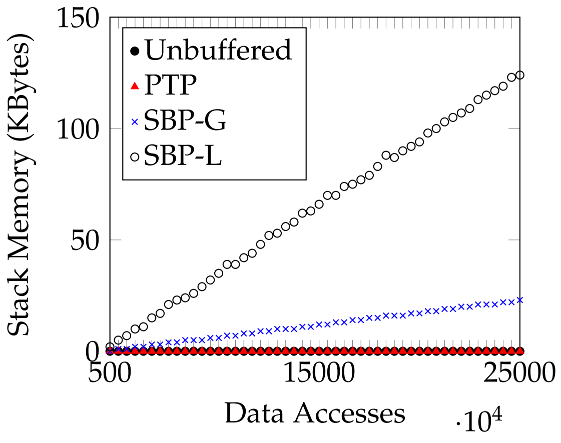
Thus, in Chassis models, PTP, SBP-G, and SBP-L have on average 5.8, 3.5, and 3 times more data elements than the unbuffered models, respectively. Therefore, PTP, SBP-G, and SBP-L require on average 5.8, 3.8, and 3 times more memory capacity than the unbuffered models, respectively. In EMS models, PTP, SBP-G, and SBP-L have on average 5.8, 4.29, and 4.57 times more data elements than the unbuffered models, respectively. Therefore, PTP, SBP-G, and SBP-L require on average 5.8, 4.3, and 3.6 times more memory capacity than the unbuffered models, respectively. The increase in memory size between the buffered and unbuffered models is not equal to the increase in the number of data elements because the number of buffers is not identical for all data elements and the data elements have different data sizes.

The results show that PTP requires more memory to store buffers than SBP. This occurs because SBP allows sharing of buffer elements between tasks and defines the buffer size based on the overlapping between LET intervals, rather than on the number of reader and writer tasks. By suppressing the unneccesary writes or writes that satisfy data age constraints, the memory demands are further reduced. Therefore, in Chassis models, SBP-G requires on average 34 % less memory than PTP. SBP-L requires on average 48 % and 21 % less memory than PTP and SBP-G, respectively. In EMS models, SBP-G requires on average 27 % less memory than PTP. SBP-L requires on average 39 % and 17 % less memory than PTP and SBP-G, respectively.

What is the impact of increasing data elements and data accesses? These results show that the linear increase of data elements and their accesses in the unbuffered models increases linearly the number of data elements after buffer synthesis. Therefore, the memory size required to store all data elements after buffering is also linearly increased. The slight increase of the memory for unbuffered models is due to the linear increase of



(a) Chassis



(b) EMS

Figure 3.15: Stack memory capacity of the Unbuffered, PTP, SBP-G and SBP-L models for **(a)** Chassis and **(b)** EMS applications.

data accesses among models and the uniform selection of the data size between 8 bit and 256 bit. The effect of data accesses and data size selection is reflected also in the results of PTP, SBP-G, and SBP-L models. The slight drops of the memory size among models, despite of the increased amount of data elements, occur because of different data sizes assigned to data elements.

Evaluation results for stack memory—Figure 3.15a and Figure 3.15b show the stack memory in KBytes required to store local variables for the Chassis and EMS models. The depicted stack size refers only to the memory required to store local data related to LET buffering. In SBP-G, local variables are the indexes initialized at the beginning of task's execution, which are used for accessing the buffers. In SBP-L, the local variables include the indexes and the local variables that are used by writer jobs with suppressed writes to write during execution. Although index variables have a life time equal to the execution time of the task, the local variables of SBP-L, that are used to suppress writes, have only the life time of the task's runnables that operate on these data. Therefore, the stack memory of SBP-L shown in the graphs is the worst-case scenario, i.e., the case when all local variables are allocated at the same time in stack. In PTP, local variables for the purpose of buffering are not required. Therefore, the stack size is zero.

In SBP, the stack memory size required to store indexes is defined by the data size of index variables and their amount created for each task. In this case study, the size of each index variable is 8 bit. As shown throughout this chapter, the amount of index variables in every task is defined by the number of global data elements that a task reads and writes during its execution. The maximal amount of index variables is $(N_R + N_W) * data = 5 * data$, where *data* defines the amount of data elements of the unbuffered models, and N_R and N_W define the number of reader and writer tasks. If a task has runnables that read and write the same data element, then two index variables are created. If a task has more than two runnables that write the same data element, then one index variable is created for both runnables because they must write on the same buffer element. The same holds for reader runnables. In this case study, reader and writer runnables of the same data element are mapped to any task. Consequently, N_R has any value in $1 - 3$ and N_W in $1 - 2$. Therefore, the actual number of index variables is less than $5 * data$.

In SBP-L, in addition to the memory needed for index variables, the memory for local variables that are created to suppress writes is shown in both graphs. If for a data element a write is suppressed in a writer task, then one local variable is created. The size of a local variable is the same as the size of the data element for which the write is suppressed. Therefore, the amount of local variables resulting from suppressing of the writes is defined by the number of writer tasks that have suppressed writes and the amount of data elements that are suppressed. The amount of suppressed data elements is defined by the data age constraints and the overlapping between tasks.

The results show a linear increase of the stack memory with the increasing number of data accesses. The slight drops of the stack among models, despite of the increased amount of data elements, occur because of different data sizes assigned to data elements.

3.5.1.2 Buffering Run-Time Evaluation

To evaluate the impact of data protection in buffering, the run-time overheads of PTP, PTP-WSP and PTP-WOSP models are defined to distinguish between simulation results of PTP with and without spin-locks. In PTP-WOSP models, the accesses to spin-locks are not simulated and, hence, the time of accessing and waiting for spin-locks is zero. The PTP-WOSP models show the performance of PTP under ideal circumstances, in which no protection of data reads and writes is involved during execution of copy-in and copy-out operations. The PTP-WOSP is not common for applications with data elements shared between multiple cores. However, in this case study the aim is to emphasize the extend at which data protection overheads influence the performance of PTP. The SBP-G and SBP-L models contain the application buffered using SBP with global and local programming styles, respectively. The buffering run-time overheads are transformed to utilization in (%) and are referred to as *buffering utilization*. The detailed results of this evaluation are given in Section A.1.0.2 of Appendix A.
Although briefly described in this chapter, to understand these results, the influencing application parameters of the buffering overheads are summarized as follows.

Which application parameters define the buffering run-time overheads? The size of data elements and the amount of their accesses influence in PTP-WSP and PTP-WOSP the run-time of copy-in/-out operations that occur at the boundaries of LET intervals. In SBP-G and SBP-L they impact the run-time of buffer initialization at the start of the HP interval and at the boundaries of task's execution, for both indexes and local data operations. The data size defines the amount of time it takes for a task to read and write a data element from the memory. This time is also determined by the memory component to which the data is allocated. For instance, as described in the configuration of this benchmark, for a data element with size up to 64 bit a minimal memory accessing time of 5 ns occurs if it is allocated to the local memory of the core and 10 ns if it is allocated to the shared memory or the local memory of the other cores. This time is higher for bigger data sizes and is also increased by bus and memory interference delays. In this evaluation, the allocation of data elements to memories is not identical in all buffering configurations and models because the heuristic in Section 3.4.5.1 is applied. Hence, the total memory accessing time differs among models and buffering configurations. The periods of computation tasks define the accessing frequency of data accesses for both protocols. In SBP, the period of the HP interval defines the frequency of data accesses at the start of the HP.

As described in this chapter, the number of data accesses added for buffering purposes for each data element with N_R reader tasks and N_W writer tasks is defined as follows. In this evaluation, N_R and N_W for each data element are in the ranges $1-3$ and $1-2$, respectively. Hence, for each data element,

- ▷ PTP has at most $2*(N_R+N_W)=10$ data accesses in copy operations,
- ▷ SBP-G has at most $N_R+N_W=5$ write accesses on index variables and 2 accesses at the start of each HP interval, and
- ▷ SBP-L has at most $N_R+N_W=5$ accesses on index variables, 2 accesses at the start of each HP interval, and $2*N_W^{'}$ on local variables. $N_W^{'}$ defines the number of writer tasks with suppressed writes. In this evaluation, forty to fifty-two percent of data elements are suppressed by data age constraints and $N_W^{'}$ is defined by Algorithm 3 when writes are suppressed. Therefore, the run-time of copy operations in SBP-L does not exceed the run-time of copy operations of PTP.

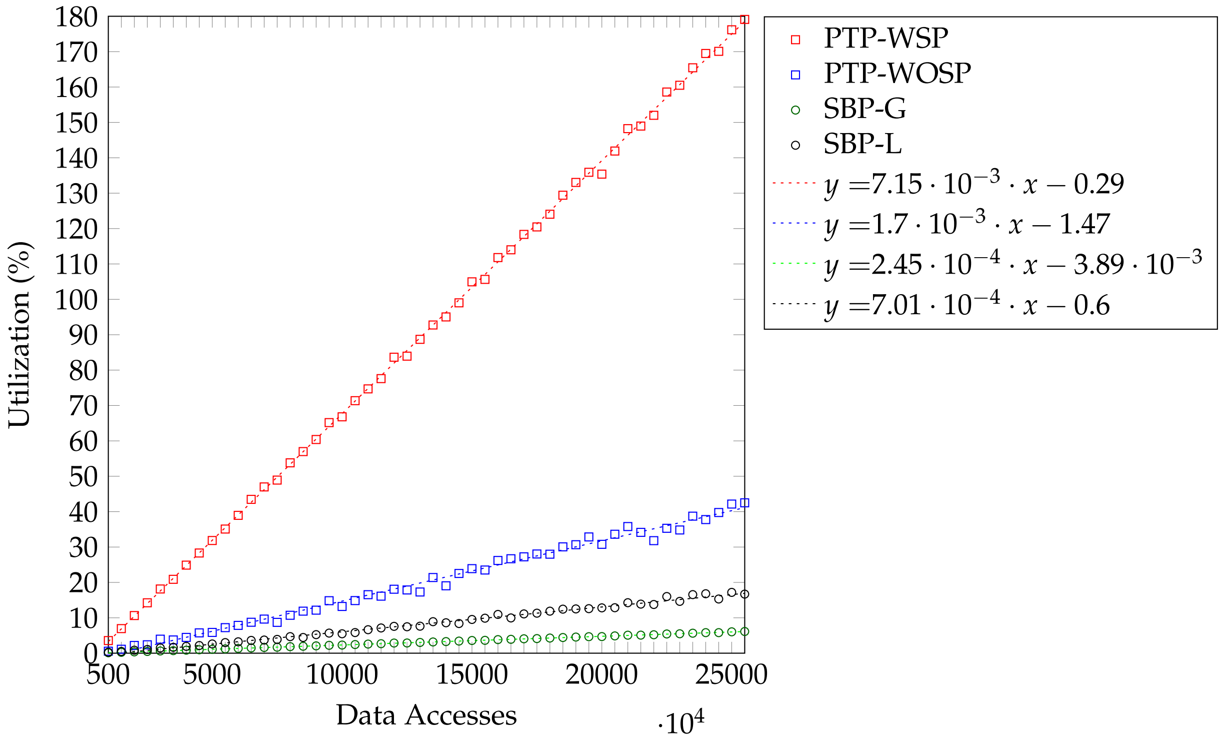
In SBP-G and SBP-L, the accesses at the start of the HP interval occur only if the last produced output is not stored in the initial buffer element. Their number may be different between SBP-G and SBP-L because the buffer schedule is not identical in both protocols and the last output is not necessarily stored in the same buffer element. The write accesses on index variables differ from the rest of additional accesses because indexes are allocated in the stack and are assumed to have a data size of 8 bit.

Thus far, only the effect of data elements and their accesses in buffering utilization is described. In PTP-WSP, extra buffering utilization comes due to the usage of spin-locks. For each data element accessed in any computation task, one unique spin-lock is used to ensure data stability during concurrent accesses in copy operations. A maximal amount of $2*(N_R+N_W)$ spin-lock accesses are added for each data element with N_R reader tasks and N_W writer tasks. In this evaluation, at most 10 spin-lock accesses are added for each data element. Except of the fixed run-time for spin-lock's usage, the waiting time for blocked spin-locks increases further the run-time of copy operations.

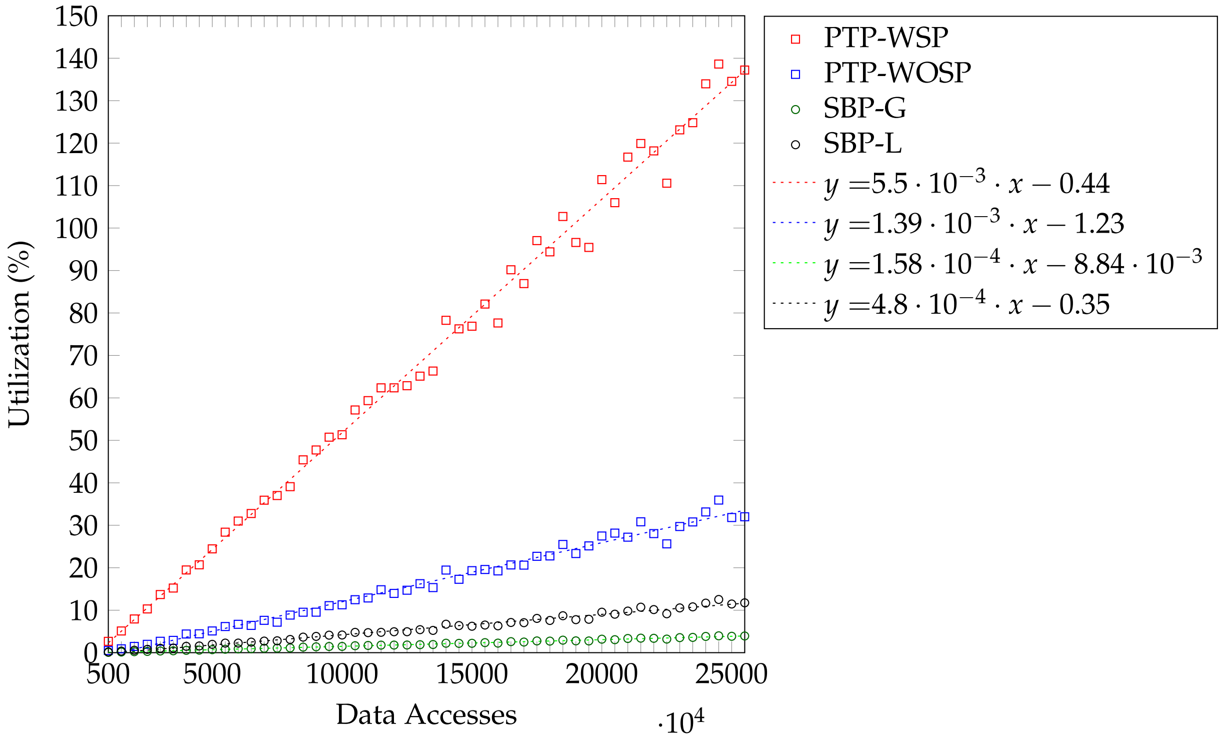
Evaluation results—The buffering run-time overheads of SBP and PTP protocols for Chassis and EMS models are shown in Figure 3.16a and Figure 3.16b, respectively. The graphs show the total utilization (in %) caused by SBP and PTP buffering operations for applications with data accesses ranging between 500 and 25,000. The x-axis shows the amount of unique data accesses of the initial, i.e, unbuffered models. Considering the configuration of 5 runnables accessing a data element, i.e., 2 writers and 3 readers, the data accesses depicted in this graph shows five times the number of data elements in each model. The number of data elements differs between models by 100. Specifically, the graphs show the buffering utilization of PTP with spin-locks (PTP-WSP) and without spin-locks (PTP-WOSP) and the buffering utilization of SBP with global (SBP-G) and local (SBP-L) programming styles.

Which buffering protocol performs better in terms of buffering overhead? The results show that SBP performs better than PTP in terms of buffering overhead in both programming cases, for all models and in all cases (with or without spin-locks). This happens because in PTP, compared to SBP, more data accesses occur due to buffering operations. Also when spin-locks are not used, the performance of PTP does not exceed the one of SBP. These results indicate that physical data exchanges at the boundaries of LET, i.e, by means of copy operations in PTP, have an enormous impact on the performance of LET regarding buffering run-time overheads. This occurs because a large number of data transfers, protected by expensive resources such as spin-locks, occur between different memory locations and memory components.
The effect of physical data exchanges is observed also in SBP-L, in which buffering utilization is increased due to the suppressing of writes. This occurs because in tasks with suppressed writes physical data exchanges occur at the boundaries of runnables execution, i.e, filling and flushing of local variables. Although the overall global memory demands are reduced, choosing SBP-L over SBP-G to integrate LET is a trade-off between the increased stack size, the reduced demands for global memory's space, and the increase of buffering utilization. Thus, in Chassis models, SBP-G has on average 29.1, 6.2, and 2.6 times less buffering overhead than PTP-WSP, PTP-WOSP, and SBP-L, respectively. In EMS models, SBP-G has on average 34.7, 7.9, and 2.8 times less buffering overhead than PTP-WSP, PTP-WOSP, and SBP-L, respectively.

Which impact has the increase of data elements and data accesses? The results show that the buffer utilization of each protocol increases linearly with the linear increase in data accesses and data elements. As previously described, this occurs because more data and spin-lock accesses occur in the system after buffering is applied. The sudden drops of utilization between models of the same protocol, despite of the increasing number of data accesses, occur due to the distribution of data accesses to tasks of different periods and the size (in bits) of data elements. In this evaluation, the distribution of data accesses to tasks is generated randomly and the size of data elements is generated



(a) Chassis models



(b) EMS models.

Figure 3.16: Buffering utilization of PTP-WSP, PTP-WOSP, SBP-G and SBP-L for **(a)** Chassis and **(b)** EMS models. The dotted lines represent the linear regression. The utilization shows the total buffering load of 6 processor cores.

using a uniform distribution. This means that tasks of equal periods that belong to different models can have different amount of data accesses and access data elements of different data size, and hence, different buffering utilization occurs. Additionally, concurrent data accesses among tasks running on different cores can cause different memory and bus interferences for different models. Furthermore, the execution times added by the usage of spin-locks shift the position at which concurrent data access operations occur, thus causing a different amount of memory access and interference times when spin-locks are not used. These effects explain the scattered distribution of buffering utilization of PTP-WSP and PTP-WOSP models with different data accesses.

The differences of PTP-WSP models are higher for applications of Figure 3.16b than for applications of Figure 3.16a because in EMS models 6 out of 22 tasks have periods in the range of 20 ms − 1000 ms. In Chassis models, all tasks have periods lower than 20 ms. The same behavior is observed also for SBP-G and SBP-L, but the differences in buffering utilization between models are not as significant as for PTP-WSP and PTP-WOSP because memory accessing times are lower in SBP-G and SBP-L.

What is the impact of spin-lock usage in buffering overheads of PTP? Although not explicitly shown in Figure 3.16, the usage of spin-locks (PTP-WSP) causes in PTP approximately 4 − 7 times more buffering utilization compared to when spin-locks are not used (PTP-WOSP). This shows that data protection of copy operations via spin-locks highly impacts the performance of PTP and increases the demands of PTP for processor resources. In PTP-WSP, for Chassis applications with up to 5,000 data elements and 25,000 data accesses approximately 30 % of each core's load is used for data transfers, i.e., copy operations, which is highly inefficient in practice. For applications using PTP for LET buffering with up to 1,000 data elements, the buffering utilization per core is up to 6 %. In PTP-WSP, doubling the number of data elements and their accesses approximately doubles the buffering utilization per core. If the amount of spin-locks is reduced, in the best case, i.e., in PTP-WOSP, the PTP's buffering performance is greatly improved. For instance, in Chassis applications with up to 5,000 data elements and 25,000 data accesses the buffering utilization is for each core approximately 7 %. It is possible to reduce the amount of used spin-locks in PTP through design decisions such as task-to-core allocation or TTS scheduling. In TTS, spin-locks can be completely avoided by scheduling copy operations to execute sequentially between cores.

3.5.2 Industrial Case Study

In this case study, SBP and PTP protocols are evaluated using the EMS system published in the FMTV challenge [76]. The original periodic tasks are transformed according to the LET semantics and are assigned to have the LET interval duration equal to their periods. The original model has utilization higher than 100 %. Therefore, the amount of instructions to tasks is decreased until a schedulable system is found. The list of tasks and their attributes are shown in Table 3.4. Tasks are distributed to

	Period (*ms*)	**Priority**	**Utilization (%)**	**Core**
Task_1	1	10	30.69	0
Task_2	2	9	15.26	3
Task_5	5	9	12.92	0
Task_10	10	7	53.74	2
Task_20	20	6	31.22	1
Task_50	50	8	4.11	1
Task_100	100	5	6.67	1
Task_200	200	4	0.06	1
Task_1000	1,000	3	0.13	1

Table 3.4: Modified EMS parameters (hyper-period of 1*s*). The *Priority* attribute defines the priority of the task. The *Core* attribute contains the index of the core in which a task is mapped to execute. *Utilization* attribute defines the computation load.

different cores such that they execute without deadline violations. They are mapped to a quad-core processor with cores indexed by labels $0 - 3$. Each processor's core has a frequency of 200 MHz and has access to its dedicated local memory via a crossbar with a delay of 10 ns and to the shared global memory and local memories of other cores via a crossbar with a delay of 40 ns. The processor has four local memories of size 128 kB each and one global memory of size 256 kB. All memories have 32 bit data width and bus contentions are handled via first-come first-serve bus scheduling strategy.

To ensure that LET communication tasks execute uninterruptedly and at the boundaries of LET intervals, the following configurations are done. Firstly, LET communication tasks are configured as non-preemptive and with higher priorities than LET computation tasks. Secondly, offsets are assigned to LET End tasks such that they execute after LET Start tasks and before the end of their LET intervals. This evaluation considers only periodic tasks and assumes that sporadic tasks and interrupts execute on dedicated cores. Because LET is applied for data exchanges between periodic tasks, only data elements that are entirely exchanged between periodic tasks are considered. Therefore, data elements that are exchanged between periodic tasks and sporadic interrupts are not part of this evaluation and are removed from the system model in order to avoid their impact on the results.

The application has 8,181 data elements, where 3,364 of those are constants and 4,817 are global data. Constant data types are not buffered and are accessed explicitly by tasks. In 4,817 global data elements 3,758 of them have one writer and one reader task, 89 have one writer and two reader tasks, 3 have one writer and three reader tasks, 915

	Unbuffered	PTP-WOSP	PTP-WSP	SBP-G
Buffering Utilization (%)	0.0	7.07	69.63	1.51
Global Memory Size (KBytes)	13.6	38.5	38.5	24.9
Stack Memory Size (KBytes)	0.0	0.0	0.0	9.7

Table 3.5: Results of EMS model. The global memory size is the memory capacity for global data elements including buffers. The stack memory size is the memory capacity for *index* variables.

are not consumed by any task, and 52 are never written. In total, the application has 4,112 read accesses and 4,765 write accesses to global data elements. Data elements have a size between 8 bit and 1024 bit. The size of SBP buffer indexes is 8 bit. Data elements and buffers are re-mapped to memory components during buffer synthesis using the *shortest-accessed-memory-delay* heuristic for reducing memory access delays. The original mapping of constants to memories is not changed.

To measure the overheads of PTP considering an industrial OS, the 750 ns delay is used as the maximal time of requesting and releasing a spin-lock. This time is based on measurements of these operations in an existing industrial OS. It is assumed that LET communication tasks are not interrupted by urgent interrupts because the former ones are assumed to run on dedicated cores. Therefore, overheads caused by enabling and disabling of interrupts in copy operations of PTP are not considered in this evaluation.

In this case study, PTP and SBP with global programming style is evaluated. No writer task was identified in the model for which writes could be suppressed, and since the model does not contain any data age constraints that can be applied to suppress writes, buffer generation using SBP with local programming style is not applied. Four models are created after buffer synthesis. The *Unbuffered* model contains the application without applying any buffering strategy. Hence, for this model it is assumed that global data elements are explicitly accessed at any time during a task's execution. The *PTP-WSP* and *PTP-WOSP* models contain the application with the PTP buffer information with and without spin-locks, respectively. The *SBP-G* model contains the buffered application using SBP with global programming style. Each model is simulated after buffering generation.

Evaluation results—The results of this evaluation are shown in Table 3.5. The global memory size (in KBytes) refers to the global memory capacity required to store global data elements including buffers. The stack memory size (in KBytes) refers to the stack capacity required to store *index* variables. The memory required to store global data elements and buffers is in PTP, i.e., in PTP-WSP and PTP-WOSP models, approximately three times more than the memory needed for the Unbuffered model. The SBP-G requires nearly twice the global memory of the Unbuffered model. As

already described in this chapter, the memory demands of each protocol are defined based on the data size of the global data elements and the amount of data elements or buffers that result after buffer synthesis. Hence, in PTP the amount of data elements after buffering is 13,579 and in SBP the amount of buffers is 8,749. In PTP, the number of data elements after buffering is calculated as $3 * 3{,}758 + 4 * 89 + 5 * 3 + 2 * 915 + 2 * 52$, where 3,758 data elements have one reader and one writer task, 89 have two readers and one writer, 3 have three readers and one writer, 915 have only one writer, and 52 have only one reader task. In SBP-G, the exact amount of buffers is defined by Algorithm 3.
The results show that SBP-G needs 35 % less global memory than PTP and 10 % less total memory (including the stack) than PTP. The memory improvement of SBP-G considering the stack is not satisfying because the overlapping between periods and LET intervals of tasks is high. As previously mentioned, the stack size represents the worst-case situation, in which all tasks are active and execute at the same time.

The buffering run-time overhead is converted in (%) and is referred to otherwise as *buffering utilization*. The buffering utilization of SBP-G is 98 % less than the utilization of PTP-WSP and 79 % less than the utilization of PTP-WOSP. SBP-G requires an utilization of 1.51 % for buffering operations, where 1.5 % of it is due to initialization of indexes at the beginning of each task's execution and 0.1 % due to initialization of buffers at the start of the hyper-period. PTP causes an increase of 69.63 % of the total utilization if spin-locks are used (PTP-WSP model) and 7.07 % utilization if spin-locks are not used (PTP-WOSP model). This evaluation demonstrates that the highest buffering utilization (approximately 62.55 %) is caused by the usage of spin-locks for ensuring data stability of copy-in and copy-out operations.

3.5.3 Conclusions

The conducted case studies showed that SBP significantly outperforms PTP in terms of memory requirements and buffer run-time overheads for applications with purely periodic communication. The results showed that SBP requires less global memory than PTP to preserve the semantics of LET but increases the memory demands for the stack. Therefore, choosing between PTP and SBP to integrate LET semantics using memory requirements as quantifier is a trade-off between global and stack memory requirements. The main advantage of SBP is that, compared to PTP, it causes insignificant increase of the application's utilization, while ensuring deterministic data exchange between tasks. The poor performance of PTP is caused by buffering operations that take place at the boundaries of LET intervals and by spin-locks that are used to ensure data stability of the buffering operations. Also when spin-locks are not used, the performance of PTP does not exceed that of SBP. Therefore, integrating LET into such applications using the PTP requires high processing resources to ensure LET semantics and all timing requirements. Using the PTP is particularly inefficient when LET is applied for a high number of data accesses and elements. Thus, SBP provides

scope to add new functionality to the application without changing the hardware platform due to additional memory and processing resources requirements.
Another disadvantage of PTP is the *sampling jitters* of data exchanges at the boundaries of LET intervals. In PTP, the points in time at which the produced results are made available to other tasks depends on the scheduling decisions and the run-time of copy-in and copy-out operations. Scheduling defines the execution order of copy-in and copy-out operations, which are executed in the context of driver tasks, as well as their distance to the start and the end of their LET intervals. Hence, the results are not made available exactly at the end of the LET interval, but some time before. This challenges the evaluation of LET semantics and end-to-end delay requirements, which are, in PTP, not precisely *time deterministic* as presumed in the LET paradigm.

PTP has the advantage of handling a *hybrid communication* between tasks. Due to its dynamic nature and the local buffering approach, the data exchange between LET and non-LET tasks works in PTP without changes in the communication mechanisms that are used by the non-LET tasks. The non-LET tasks can be periodic, sporadic or a-periodic and they access the global data elements independently of the buffers used by LET periodic tasks. This hybrid communication is not possible in SBP without changes of the non-LET communication mechanisms used by non-LET tasks, because in SBP multiple buffers of the same data element exist and the non-LET tasks cannot choose among them without the knowledge of the produced time and whether the data is being accessed concurrently by other LET tasks. Therefore, SBP is applicable for data elements that are only exchanged between periodic tasks that communicate through the LET paradigm. Finally, a semi-optimal integration design of LET for automotive applications is possible using both SBP and PTP. Applying both protocols for the same application incorporates both LET and non-LET tasks, and balances the memory and buffering run-time overheads caused by LET semantics in PTP.

4 | Scheduling Design

This chapter describes approaches of constructing the schedule of *Logical Execution Time (LET)* tasks for *Time-Triggered Scheduling (TTS)* and *Fixed-Priority Scheduling (FPS)* mechanisms. The proposed schedule synthesis algorithms generate the schedule of LET tasks considering non-functional requirements such as timing, LET communication semantics, and resource requirements and optimize their schedule in terms of overheads caused by preemption and *Operating System (OS)* operations. Although schedule synthesis approaches targeted in this work are mainly intended for LET tasks that exchange data via *Point-to-Point Protocol (PTP)* and *Static Buffering Protocol (SBP)*, they are independent of the communication patterns and can be applied for non-LET tasks that use any data exchange methods.

This chapter is organized as follows. An introduction and the motivation of an automatic schedule synthesis for LET tasks is given in Section 4.1. The research literature and the review of approaches that generate the schedule of embedded applications are provided in Section 4.2. Next, the description and constraints of the application model scheduled by TTS and FPS are provided in Section 4.3. Thereafter, in Section 4.4, the overall methodology of schedule synthesis applied for TTS and FPS scheduling strategies is provided. The proposed schedule synthesis algorithms for the TTS and FPS approaches are described in Section 4.5 and Section 4.6, respectively. Finally, Section 4.7 describes a comprehensive evaluation performed for the TTS and FPS synthesis regarding scheduling overheads, performance, and efficiency.

4.1 Introduction

The schedule of embedded real-time systems must be feasible independently if LET or the *Bounded-Execution Time (BET)* mechanism is used for data exchange. Scheduling design of LET tasks is a necessary step during integration of LET paradigm in automotive applications for the following reasons. If PTP is used to apply LET, then scheduling must guarantee LET semantics in terms of time and dataflow determinism. The PTP protocol on its own does not fully satisfy the semantics of LET. Hence, scheduling must not only guarantee that tasks execute within their LET intervals but also in a fixed order. Otherwise, LET semantics are violated. For instance, the communication tasks that execute copy-in operations at the beginning of LET intervals must execute exactly or close to the beginning of their LET intervals and before their corresponding computation tasks. Similarly, the communication tasks that execute

copy-out operations must be scheduled to execute after their corresponding computation tasks and exactly or close to the end of their LET intervals. Although SBP does not impose any extra requirement that must be considered in scheduling, LET interval violations must be avoided at any cost. Otherwise, read and write accesses on shared buffers could overlap among different tasks, thus leading to dataflow and time determinism violations and an incorrect functional behavior of the software. To avoid this situation, scheduling of LET tasks must take into account the impact of the *timer interrupt* and context-switching overheads caused by preemption, such that timing differences between the planned schedule at design time and the one occurring on real target are avoided to an extent. The determinism of LET and functional behavior is violated also in PTP if outputs are provided by communication tasks after the end of their LETs.
Scheduling of LET tasks increases memory and processing time demands due to activation and preemption of tasks. PTP needs higher hardware resources than SBP because of more buffering and scheduling overheads. Applications that use PTP have a higher number of tasks compared to SBP. The time required to activate and schedule the additional tasks increases the requirements for memory and processor capacity further. Therefore, planning and optimizing the schedule of LET tasks at design time assists on hardware resource assessment during deployment of a LET system.

In this chapter, an approach to automatically synthesize and optimize the schedule of LET tasks for TTS and FPS scheduling techniques is described. This work focuses on these two scheduling techniques because of several reasons. The FPS scheduling is the widely used approach in industrial in-vehicle applications and this work aims to apply the proposed LET approaches for industrial applications. Additionally, FPS is flexible and handles dynamic changes of the application at run-time and is, therefore, the favorable approach for highly event-based applications.
Compared to FPS, TTS offers benefits such as controlling of overheads, better load distribution, higher schedulability, deterministic execution of LET computation and communication tasks, extensibility, and scalability. TTS is a convenient way to increase the deterministic execution of existing automotive applications. Recently, the automotive domain is evolving towards ways that handle software extensibility and online updates efficiently. For instance, the *AUTomotive Open System ARchitecture (AUTOSAR)* consortia introduced the software cluster concept [86] to simplify the integration and the update of different application parts provided by several vendors. Hence, TTS is viewed as a way to guarantee a feasible schedule execution also if an application cluster is updated at run-time, such that the execution of tasks of other application clusters remains unaffected after an update. Furthermore, TTS offers the benefit of planning the processor's time for future software extensions. Therefore, this work exploits the possibility to schedule LET tasks by TTS and explores potential optimization capabilities in terms of scheduling overheads.

The scheduling synthesis targeted in this work considers strict *timing*, *communication*, and *performance*, i.e., resource requirements. Each requirement is fulfilled during scheduling synthesis and is described as follows.

Predictability and Timing - As aforementioned, LET demands that the execution of tasks is predictable and reproducible at design and implementation of the software application. Although "correct by construction" ensures *execution determinism* of tasks due to static nature of the selected scheduling strategies, in FPS, the reproducibility of task execution flow is harder to predict because scheduling decisions are taken at run-time based on priorities. Hence, an under- or overestimation of *Worst-Case Execution Time (WCET)*, which usually differs from the actual execution time of tasks, impacts the execution flow and interferences of tasks. This is not a critical issue as long as tasks finish their execution before the end of their LET interval. If WCET is not pessimistically defined, then task overruns occur and LET intervals are violated. This is because it is hard to precisely estimate WCET for multi-core processors [87]. Note that also when WCETs cannot be estimated, approaches of defining them as early time-budgets, before the functions are implemented, are proposed in [88]. This is a reasonable approach to estimate WCETs during the first iteration of designing LET intervals and scheduling.
The execution determinism is not an issue for TTS when WCETs are overestimated because the execution order of tasks defined at design time is identical at run-time. In both scheduling strategies, the schedule synthesis must generate the schedule of tasks such that timing requirements are fulfilled, i.e., all tasks finish their execution before their end of LET intervals or deadlines. Otherwise, the application produces erroneous behavior and violates LET requirements. Additionally, the schedule of tasks must be constructed in such a way that dropping of any task's job is strictly avoided.

Communication - In PTP, data exchanges occur "*close*" to LET boundaries because copy-in and copy-out operations do not take zero-time to execute and several of these operations could occur at the same time instant, which delay each other due to scheduling decisions. Scheduling defines when and in which order LET tasks are executed. For instance, in applications with multiple LETs released at the same time instant, the scheduling decides which of the communication tasks are executed first. The later a communication task is scheduled, the larger is the distance between the production time of its outputs and the end of its LET interval. As described in the previous chapter, every LET computation task is in PTP associated with a LET Start and a LET End task. To schedule LET tasks of PTP correctly, the Requirement 4.1 is addressed during schedule generation.

Requirement 4.1: *LET communication tasks must execute uninterruptedly, isolated and free from interference. LET communication tasks must not preempt each-other nor could be preempted by computation tasks, otherwise data consistency issues can occur and dataflow determinism is violated. The only preemptions allowed are by the timer interrupt for handling task's activation, by urgent interrupts, or by the OS. After the preempting interrupts finish their executions, the preempted task resumes its execution and no other ready tasks. Communication tasks are mapped onto the same core as their respective computation tasks. The LET Start and End tasks must execute before and, respectively, after their related computation task. For coinciding LET intervals, all LET Start tasks* *must execute before and LET End tasks after computation tasks for reducing their distance to the boundaries of their LET intervals.*

In terms of data exchange, SBP handles LET semantics by design rather than implementation. The order of execution between LET tasks in SBP is not important, unless *LET Dynamic Buffering Protocol (LDBP)* is used. In LDBP, buffers are assigned and released at the boundaries of LET interval. Therefore, the task that handles these operations must execute before computation tasks and at the end of LET interval. In this case, the scheduling requirements for LDBP are similar to the PTP. If SBP is used, then scheduling must guarantee that the dedicated interrupt defined to initialize buffers at the beginning of each *Hyper-Period (HP)* iteration is executed before computation tasks start their execution.

Resource - In this work, LET tasks are activated by a *high-resolution timer*, which causes less task activation jitters and task preemptions compared to a periodic timer. Nevertheless, activation and preemption delays cannot be fully avoided also when the high-resolution timer is used, because the timer interrupt takes a certain amount of time to execute and has a higher priority than any LET task. Similarly, preemption delays are as well caused by the execution of urgent LET tasks. For instance, in PTP, communication tasks often delay and preempt computation tasks. In order to ensure a feasible schedule, computation tasks are allowed to preempt each-other.
In applications with many preemptions, the response times of LET tasks can increase beyond their LET interval duration. Furthermore, preemptions increase a system's load due to processing of context switches and can often degrade the execution's performance of the application. In case of hierarchical preemption, which for some scheduling mechanisms, e.g., FPS is unavoidable, a higher stack memory capacity is required to store the contexts of preempted tasks. Preemption cannot be avoided entirely. Otherwise, if tasks are scheduled fully non-preemptively, finding a feasible schedule is not always possible. Nevertheless, the number of preemptions can be controlled and reduced by designing the schedule such that execution overlaps of tasks are kept as low as possible.
In this work, the *resource requirement* is addressed by schedule synthesis as a way to reduce the demands on hardware resources that are required to deploy a LET system, e.g, by minimizing the amount of preemptions. If activation and preemption delays caused by the timer interrupt and urgent LET tasks are not considered during schedule design, then the planned task execution can significantly deviate from the reality and the hardware resources that are required to deploy the application cannot be precisely estimated. Furthermore, inclusion of preemption and context-switching delays during schedule construction of LET tasks is crucial in order to ensure that the execution behavior of tasks on target approximates the one estimated at design phase. To design a schedule with a limited amount of preemptions, the preemption run-time overheads are important to estimate the extent to which preemption overheads can be reduced. Limiting only the number of preemptions [89] without taking into account

preemption run-time overheads does not explicitly show how much system load is saved by optimizing the schedule in terms of preemptions.

4.2 Related Work and Problem Analysis

The following sections provide an overview of schedule synthesis approaches proposed in the research community regarding TTS and FPS scheduling mechanisms.

4.2.1 Time-Triggered Scheduling

The TTS approach is typically used in the avionic domain [18] to schedule time partitions in highly deterministic fashion. In the automotive domain, TTS is described in OSEKTime [90] to schedule tasks and in FlexRay protocol [91] as well as in the *802.1Qbv shaper* of *Time-Sensitive Networking (TSN)* [92] to schedule bus messages. The schedule synthesis of TTS is extensively studied in the research community. Due to the complexity of the scheduling problem, most of the proposed approaches, along with this work, apply mathematical programming techniques [93] to generate a feasible and optimal time-triggered schedule. The efficiency of these techniques depends on the formalization and decomposition of the problem. Therefore, in contrast to the majority of the related works, our approach focuses additionally on ensuring time efficiency of the schedule synthesis. To our knowledge, related work does not address such approach, semantics of LET, optimization of resource requirements, and the delays caused by the execution of the timer interrupt and OS operations.

The majority of publications target the non-preemptive scheduling [94–110] and a few the preemptive scheduling [111–119]. Because scheduling affects end-to-end delays in non-LET systems, most of these works focus on ensuring the control performance and quality by synchronizing the schedule of tasks and messages. Additionally, to maintain the dataflow between tasks that affect the end-to-end delays of a critical path, the execution order constraints are considered during schedule synthesis. In our work, LET is used to ensure deterministic dataflow and to enforce zero jitters on end-to-end delays and, hence, ensure the control quality. Note that in LET, the dataflow between tasks is independent from the actual scheduling of computation tasks.

To ensure control performance and quality, the approaches in [94–97] synthesize schedules for distributed systems during the design of controllers, which consists of the definition of the sampling periods. They solve the synthesis of tasks and messages in the same step such that end-to-end delays are fulfilled and the recent data values are shared among sensor, controller, and actuator components. Samii et al [94] propose a *Constraint Logic Programming (CLP)* formulation to construct the schedule. Schneider et al. [95] and Goswami et al. [96] propose a *Constraint Satisfaction Problem (CSP)* and an *Integer Linear Programming (ILP)* formulation, respectively, for schedule synthesis of FlexRay bus messages and tasks scheduled by the non-preemptive scheduler of the eCos OS [120]. Roy et al. [97] focus on the trade-off optimization of the two optimization objectives such as control performance and resource consumption of the bus. The approaches discussed thus far describe similar formalization of the scheduling problem.

Voss et al. [100] construct the schedule of tasks and messages in the same step as the generation of task-to-core allocations. They consider several safety requirements [17] and solve the schedule using *Satisfiability Modulo Theories (SMT)* [93]. They fulfill task execution order constraints during schedule generation by ordering tasks in the order that the distributed messages are sent. Zwerlov et al. [101, 102] extend the synthesis problems [100] with design exploration techniques such that multiple optimization goals are handled during the search for optimal solutions.
Igna et al. [103] evaluate the approach defined in [100] with an industrial avionic application. They extend the approach to speedup the search of optimal solutions and define a customized "preprocessing phase" [103, p. 5], in which schedulability tests are used to check if two tasks can execute non-preemptively by means of different offsets. If a possible preemption relation exists, then tasks are mapped for execution to different cores. Otherwise, they are allocated to the same core and their offsets are defined using an *Integer Programming (IP)* formulation similar to the approach in [98]. In our work, offsets are assigned only to LET End tasks to ensure Requirement 4.1.

Sagstetter et al. [104, 105] propose an approach to integrate the non-preemptive schedule of tasks and messages of different *Electronic Control Unit (ECU)*s "into a global schedule" [104, p. 1]. They use the ILP formulation proposed in [116] to generate the schedule of each ECU independently. The purpose of this approach, as the authors state, is to "reduce the integration complexity" [104, p. 1]. They integrate schedules of different ECUs by assigning an offset to them using a SMT solver. These offsets shift the start times of tasks and messages by a fixed amount of time. In the evaluation case study, the authors show that the ILP based approach of [116] requires, for a realistic system, more than 24 h to provide results. This time is reduced when local schedules are constructed separately and later integrated using their integration algorithm. These results show that solving the schedule of tasks and messages for all ECUs in one step is impractical for complex industrial systems and does not scale well with the increase of system's size. To overcome this performance and scalability issues, Darbandi et al. [106] solve scheduling of tasks and messages in two different steps using *Mixed Integer Linear Programming (MILP)* and "minimize the number of used slots" [106, p. 3] and the slack time between the deadline and finish time of jobs. For the same reason, Hu et al. [109] avoid constraint programming and construct the non-preemptive schedule of tasks and messages using the *list scheduling* approach described in [121]. Their algorithm generates, in iterative fashion, the schedule of tasks and messages in the same step and if no feasible schedule is found, then offsets of tasks are reassigned to reduce the interference between tasks. The algorithm backtracks until a valid schedule is found. Hu et al. [110] propose a similar algorithm for safety applications and construct the schedule using the list scheduling approach described in [122]. These algorithms lack the flexibility to optimize the schedule due to their heuristic-based synthesis approach.

Minaeva et al. [107] focus on optimizing the control performance and propose a heuristic search based algorithm to improve the solving performance of the non-preemptive schedule synthesis considering zero-jitters of end-to-end delays by means of LET. The authors consider the zero-time assumption of LET for exchanging data and do not consider explicit semantics of LET in the schedule synthesis, except that for tasks that use LET the execution order constraints are not applied. In their approach, the start time of each job is defined equal to the start time of its respective previous job multiplied by a fixed amount of time. They use a schedulability test, similar to the one presented in [103], to validate the feasibility of the schedule. In our work, we handle explicit LET requirements during the schedule synthesis and allow any start time for jobs in order to construct a feasible schedule also for cases in which a fully non-preemptive schedule does not exist.

Wang et al. [108] describe the generation of the non-preemptive schedule for multi-core AUTOSAR systems [72]. The start times of runnables are generated considering the execution order between dependent runnables, which is essential because they are mapped to multi-core processors and the correct data flow must be preserved. They constructs the schedule table, which is defined in AUTOSAR to activate runnables by means of expiry points [19]. In this case, runnables are still scheduled by FPS but because the schedule is constructed non-preemptively, they execute as being scheduled by TTS. Eisenbrand et al. [98] propose an approach to generate the schedule of tasks for aircraft applications using IP. Similar to [103, 108], they avoid preemption of tasks by assigning different activation offsets to tasks such that all jobs are released in unique times. Blikstad et al. [99] describe a *Mixed Integer Programming (MIP)* approach to generate the non-preemptive schedule of aircraft applications focusing on forecasting the extensibility of the software and on the execution order constraints and delays between tasks that have dataflow dependencies. They plan extensibility in the schedule of each task by defining a "maximum idle time between tasks" [99, p. 4]. In this way, new functionalities can be added into a task without reconstruction of the schedule. In their approach, the schedule of tasks is defined in task-level and is solved together with the schedule of bus messages.

Although non-preemptive TTS scheduling offers several benefits, it cannot always guarantee a feasible schedule, especially for complex industrial automotive applications. The complexity of constructing the schedule of preemptive compared to non-preemptive TTS is higher. Hilbrich et al. [114, 115] propose a *Constraint Programming (CP)* formulation to generate the schedule of tasks for aircraft applications. The authors handle preemption between tasks by using the slicing approach, in which the WCET of tasks is divided into slices of fixed size before the schedule is constructed. These time-slices are allocated to the HP interval during synthesis. Although this approach is valid, it does not always guarantee a feasible schedule, does not scale well in terms of resource usage, and depends on the optimal definition of the slice size.

Zheng et al. [111] and Zeng et al. [112, 113] describe similar MILP formulations to generate a preemptive schedule of tasks. They construct the schedule of tasks in the same step as the schedule of bus messages distributed on a FlexRay network [91]. In [112], the arrival times and the release times of each job are generated during schedule synthesis. In case of activation jitters, the arrival time is different from the release times. In this work, these times are calculated outside of the constraint solver for improving the execution performance of the synthesis algorithm. In addition to the formalization for the non-preemptive scheduler of eCos OS [120] given in [96], Lukasiewycz et al. [116] describe an ILP formulation for preemptive schedule generation of tasks scheduled by *Last In - First Out (LIFO)* of OSEKTime [90]. Their schedule generation of tasks is performed in task level and not in job level as in our work. They define the start time of jobs as an increment of the task's period and the start time of the first job released in the hyper-period. Nevertheless, to calculate a maximal end time of tasks, which is required to validate the schedule feasibility, the authors generate the actual end times of all jobs released in one HP. Although this approach reduces the exploration space and minimizes the solving time, it can lead to infeasible solutions faster also when a feasible schedule can be found. Therefore, in our work, the start times of jobs can take any value within the boundaries of their LET intervals.

McLean et al. [119] defines the TTS schedule of ADAS applications using *Earliest Deadline First (EDF)* as part of a simulated annealing algorithm, which they propose to allocate tasks to cores and assign offsets and deadlines of tasks. Although EDF is a fast way to assign priorities, it does not always provide an optimal solution. Zhou et al. [117] propose an approach of generating the time-triggered schedule of safety partitions of ARINC 635 [18] considering the context-switching time that occurs between partitions. The authors focus on reducing the number of context switches between partitions and the duration of the schedule. They treat partitions as server tasks with constant attributes such as period and WCET and use the EDF heuristic to construct their schedule. They propose "a greedy algorithm to reduce the number of preemptions" [117, p. 3]. In their algorithm, periods and WCETs of server tasks are recalculated until a feasible schedule is found. The schedule of tasks within the partition is not handled in [117]. In our TTS schedule synthesis, time partitions are not considered, periods and WCETs of tasks are not changed during synthesis, and the context-switching between tasks is handled and optimized using the CP approach. In [117], the optimization of preemptions provides only one solution, which is not necessarily the optimal one.

Han et al. [118] propose a genetic algorithm to generate the schedule of safety partitions of ARINC 635 and reducing resource usage of the processor by reducing preemptions among partitions. In their approach, the schedule is simulated in order to validate the schedulability. In scheduling problems, genetic algorithms tend to fail in providing solutions of sufficient quality, and often an unpredictable number of generations is required to produce a feasible schedule.

Theis et al. [123] propose an EDF heuristic and a search based algorithm to construct the schedule of tasks for applications with two criticality modes. A review on mixed criticality systems is given in [124]. As described in [123], because the WCET is hard to precisely estimate [125], more pessimistic upper bounds of WCET are defined for high criticality tasks in order to certify the schedule accordingly to safety standards. Tasks with overly pessimistic WCETs run on high mode and tasks with the assumption of reasonable upper bound of WCET run on a low mode. It is not in the focus of this work to construct the schedule of tasks for high and low modes. Additionally, this work assumes a pessimistic upper-bounding of task WCETs such that LET overruns do not occur at run-time.

A summary of the discussed related work is given in Table 4.1. Approaches in [111–113] relate the most to this work. They are similar to the approach proposed in this work in the way preemption and execution time-frames of jobs are calculated in the constraint-based formalization. The differences are as following. In this work, LET is used for data exchanges and, hence, execution order constraints are not needed to ensure dataflow between tasks. Our approach constructs the schedule of tasks independently of the schedule of bus messages to improve run-time performance, and handles optimization of resource requirements and the preemption and start delays caused by the timer's execution and OS routines. A schedule defined without the impact of the timer and OS overheads leads to an inaccurate execution behavior of tasks and an unrealistic planning of hardware resources. In terms of the proposed mathematical formulation, our work conducts an improvement of the work in [111].

	Methodology	LET Semantics	Preemptive Scheduling	Overheads	Resource Optimization	Schedulability	Task Scheduling	Message Scheduling
Contribution	CP/Heuristic	✓	✓	✓	✓	TB	✓	✗
Eisenbrand et al. [98]	IP	✗	✗	✗	✗	?	✓	✗
Steiner et al. [126, 127]	SMT	✗	✗	✗	✗	?	✗	✓
Schneider et al. [95]	CP	✗	✗	✗	✗	TB	✓	✓
Goswami et al. [96]	ILP	✗	✗	✗	✗	TB	✓	✓
Voss et al. [100]	SMT	✗	✗	✗	✗	TB	✓	✓

Igna et al. [103]	SMT	✗	✗	✗	✗	ST	✓	✓
Samii et al. [94]	CLP	✗	✗	✗	✗	TB	✓	✓
Roy et al. [97]	CP	✗	✗	✗	✗	TB	✓	✓
Darbandi et al. [106]	MILP	✗	✗	✗	✗	TS	✓	✓
Wang et al. [108]	MILP	✗	✗	✗	✗	TS	✓	✗
Hu et al. [109, 110]	Heuristic	✗	✗	✗	✗	?	✓	✓
Sagstetter et al. [104, 105]	SMT	✗	✗	✗	✗	TB	✓	✓
Minaeva et al. [107]	Heuristic	○	✗	✗	✗	ST	✓	✓
Blikstad et al. [99]	MIP	✗	✗	✗	✗	TB	✓	✓
Theis et al. [123]	Heuristic	✗	?	✗	✗	?	✓	✗
Zhou et al. [117]	Heuristic	✗	○	✗	○	?	✗	✗
Han et al. [118]	Genetic	✗	○	✗	○	?	✗	✗
Hilbrich et al. [114, 115]	CP	✗	✓	✗	✗	TS	✓	✗
Lukasiewycz et al. [116]	ILP	✗	✓	✗	✗	TB	✓	✓
Zheng et al. [111]	MILP	✗	✓	✗	✗	TB	✓	✓
Zeng et al. [112, 113]	MILP	✗	✓	✗	✗	TB	✓	✓
McLean et al. [119]	EDF	✗	✓	✗	✗	SIM	✓	✗

Table 4.1: Qualitative comparison of approaches for TTS synthesis problem.
Legend: satisfied (✓), partially satisfied (○), unsatisfied (✗), or unknown (?), TS (Time-Slicing) or TB (Time-Budgeting), ST (Schedulability Tests), Simulated Annealing (SA), SIM (Simulation).

4.2.2 Fixed-Priority Scheduling

As TTS, the FPS approach is used in the automotive [90] and avionic [18] domain to schedule tasks and bus messages. FPS is one the most studied scheduling mechanisms. The related work targets several aspects of FPS [128], such as definition of efficient schedulability tests [27, 33, 129], priority assignment synthesis, techniques of limiting preemption [37, 130], and methods of applying FPS in multi-processors [29]. This work focuses on the schedule synthesis problem. Therefore, only the related work targeting priority assignment for the preemptive FPS is addressed in this section.

A considerable amount of publications describe heuristics, meta-heuristics, and mathematical programming formulations to solve the preemptive FPS synthesis problem. This work applies constraint programming techniques [93] combined with a heuristic to generate a feasible and optimal task priority assignment. Due to the complexity of validating schedule feasibility, most of the proposed approaches use schedulability tests to validate if a set of priorities leads to a feasible schedule. Pazzaglia et al. [129] describe several sufficient tests for FPS under different assumptions.
In this work, schedule feasibility is validated not based on *Worst-Case Response Time (WCRT)* analysis, as widely performed in the related work, but by calculating during priority assignment the exact execution of all jobs released in the HP interval. The integration of WCRT calculations into mathematical formulations significantly increases the complexity of the scheduling problem and the solving times of schedule synthesis, because the WCRTs of tasks are calculated iteratively. Our approach offers not only reasonable solving times, but also the possibility for optimizations such as minimization of preemption overheads and actual response times. It supports synchronous and asynchronous tasks with implicit and explicit deadlines without formulating these use cases in the WCRT calculation. Reducing the number of preemptions, does not only reduce preemption overheads, defined by context-switching delays and *Cache-Related Preemption Delay (CRPD)*s, but also improves the overall response times [131]. Although not explicitly described in our formalization, this approach offers as well the possibility to consider job based execution times instead of a WCET on task level. To our knowledge, related work does not address the semantics of LET, the optimization of resource requirements, the overhead of context switching, and the delays caused by the execution of timer interrupt and OS operations.

In single-core processors, the *Rate Monotonic (RM)* and *Deadline Monotonic (DM)* heuristics are often used to assign priorities to tasks. Davis et al. [132] provide a summary of the related work regarding these heuristics. Because RM and DM do not always provide optimal priority ordering, they are not sufficient for applications with dependent and asynchronous tasks that run on a multi-core platform. Audsley et al. [133] propose the *Optimal Priority Assigment (OPA)* algorithm to assign task priorities. Davis et al. [134] provide an extension of OPA to ensure robustness and Grenier et al. [135] apply the OPA to decrease the complexity of offset free systems [136]. They use OPA to reduce the search space of their offset assignment algorithm. Garibay-Martinez et al. [137] apply OPA for distributed systems considering single and multi-threaded tasks running in a multi-core platform. They propose an extension of OPA to assign priorities to tasks and bus messages during the allocation of tasks to processors. They show that OPA performs better than DM also in distributed systems. Similarly, Huang and Cheng et al. [138] adopt OPA to find an optimal priority ordering for a set of sporadic self-suspending tasks. OPA assigns priorities based on schedulability test.

Despite of their fast way to assign priorities, heuristic based approaches, including OPA, are not sufficient for industrial applications because they cannot be used to assign priorities by focusing, in addition to the deadlines, on optimization goals such as preemption overheads, end-to-end delays, and execution order between tasks. Garcia

and Harbour et al. [139] propose the *Heuristic Optimized Priority Assignment (HOPA)* algorithm, which assigns priorities using DM based on "local deadlines" [139, p. 2]. These deadlines are calculated in each iteration of the algorithm considering as well end-to-end delays. The authors show that HOPA provides a faster and better priority assignment than the simulated annealing. They assess the quality of the solution by the laxity, i.e., the time between the deadline and WCRT of a task.
Zhu et al. [140, 141] apply and extend HOPA to assign priorities considering extensibility. Although HOPA provides quick priority ordering and better solutions than the simulated annealing and DM per se, it does not always guarantee an optimal assignment. Azketa et al. [142] showed that their proposed genetic algorithm provides a better priority assignment than HOPA. Nevertheless, to gain its benefits, they apply HOPA to generate the initial population, which provides already a set of solutions with feasible schedule.

Several authors apply meta-heuristics to assign priorities to tasks. Hamann et al. [143] apply in their genetic algorithm the concept of "traffic shaping" [143, p. 3] to lead the search towards feasible solutions in case high jitters cause overload of events. Samii et al [94] assign priorities to tasks during control design and Sayuti et al. [144] assign priorities in the same step as task allocation considering end-to-end delays and NoC communication model. Bouaziz et al. [145] reduce the amount of preemptions by solving the problem of mapping functions to tasks such that the overlapping between tasks is reduced. They apply RM to assign priorities and a genetic algorithm for the mapping problem. Bate et al. [146] propose a simulated annealing algorithm to allocate tasks to cores and to assign priorities to tasks and bus messages. They perform sensitivity analysis [147] to define a safe upper-bound of task execution times such that available capacity for future functionality extensions is efficiently planned during the aforementioned designs. As previously mentioned, genetic algorithms tend to fail in providing solutions of sufficient quality, and often an unpredictable number of generations is required to produce a feasible schedule.

To distinguish the benefits of mathematical programming techniques and of genetic algorithms, Mehiaoui et al. [148] describe a MILP formulation and a genetic algorithm to assign task priorities and to allocate functions to tasks and ECUs and messages to bus. Their optimization goals are end-to-end delays and extensibility. To validate the schedulability, they adopt WCRT analysis on function level. Wozniak et al. [149] solve the aforementioned problems in one step using a genetic algorithm and a MILP formulation as in [148] but considering as well optimizations of the memory and run-time overheads that emerge from data protection mechanisms. Compared to [149], they do not assume a zero communication time for tasks mapped to the same ECU. To validate the schedule feasibility, they adopt WCRT analysis of Zhu et al. [140] on function level. In our work, critical sections are only part of LET communication tasks and the run-time of accessing the resource is part of task's WCET. Wozniak at al. [149] denote that due to a high number of variables out-of-memory issues occur, which causes the MILP solver to fail even for small applications. This occurs because, in addition to solving multiple problems together, the MILP formulations in both works [148, 149] apply schedulability analysis on function level and assign priorities to functions instead of tasks. Note that realistic industrial applications are composed of thousands of runnables. Therefore, Mehiaoui et al. [148] describe an approach to solve the above problems in two different steps.

In the related work, the majority of approaches that apply mathematical formulations solve the scheduling problem in the same step with other design problems, such as the allocation of tasks to cores or bus message scheduling. Despite the influence that task scheduling has on other design problems and vice versa, it is not practical for industrial applications to solve them together due to the high complexity and long solving times of the synthesis problem. These effects are demonstrated also in the following works. Metzner et al. [150] formalize the impact of scheduling during solving of task-to-core allocation problem. They assign priorities via DM and do not focus explicitly on priority assignment but verify the schedulability by integrating WCRT analysis in the SAT-CP formulation of task-to-core allocation. Zheng et al. [151] and Zhu et al. [152] propose comparable MILP formulations to assign task priorities in the same step as the generation of task-to-core allocations and scheduling of bus messages. To validate the feasibility of the schedule, the authors incorporate the formulation of Metzner et al. [150] to calculate the WCRTs of tasks, which, as they also state, it increases the complexity of the formulation and leads to high solving times. Therefore, the authors describe an approach based on simulated annealing that solves these problems step-wise. Similarly, Schlatow et al. [153] propose a MILP formulation to assign priorities in the same step as task-to-core allocations and activation offsets. They also incorporate WCRT calculation in the ILP formulation.

Wieder et al. [154] consider in their ILP formulation the effects and access delays of shared resources, i.e., spin-locks. The authors solve priority assignment and task-to-core allocation of a set of sporadic tasks with constrained deadlines and include the aforementioned delays in their WCRT formulation. Again, due to the complexity of constraints and high solving times for realistic applications, the authors propose a heuristic to solve the above problems. Their heuristic adopts OPA to assign priorities. To reduce the high solving times caused by the complexity of WCRT calculations in ILP formulations, Zeng et al. [155] define an optimistic formulation of WCRT, which validates the feasibility of only the first job of each task. In this case, the assigned priorities are not always optimal. The complexity of validating schedulability in ILP formulations using WCRT is also discussed in Zhao et al. [156]. The authors propose the "concept of unschedulability core" [156, p. 1] to validate schedulability. In our work, to improve the performance and reduce complexity, the FPS scheduling is synthesized as a dedicated step and an efficient approach is proposed to validate schedule feasibility.

Approaches targeting the scheduling of LET applications are described as follows. Farcas et al. [56] use EDF to schedule LET tasks. Derler et al. [57] increase the schedulability of tasks by loosening up the constraints of LET. They define release times of LET tasks such that tasks can execute outside the boundaries of their LET intervals. This

approach is not practical for realistic applications because it creates a strong dependency between a task's implementation code and the actual scheduling. To decrease the preemption interference and high response times of low-priority tasks, Beckert et al. [80] use the slack-stealing approach, in which the execution of higher-priority tasks is delayed until the next synchronization point. They achieve this by boosting the priority of low-priority tasks during their non-preemptive regions, which are defined between two synchronization points.
Igarashi et al. [157] apply LET for multi-rate applications and describe several design aspects regarding LET. The authors provide an approach to allocate LET communication tasks to cores and define their activation schedule such that their execution does not overlap and memory contentions are avoided. As in this work, they prioritize LET Start tasks higher than LET End tasks and define their schedule in different steps. In [157], priorities of tasks are assigned based on the *lowest-laxity-first* heuristic, in which tasks with the lowest laxity take the highest priority. Finally, to fulfill end-to-end delay requirements, the authors reduce the duration of LET intervals to a proportion of WCETs until these requirements are ensured. Yano et al. [158] follow up on the work published in [157] for multiple computer clusters and apply the sub-scheduling approach of [82] to schedule LET tasks.

Although not explicitly targeted in this work, the number of preemptions in low-priority tasks can be reduced by means of limited preemptive scheduling [36, 37]. Several publications target priority assignment in the same step as the design of limiting the preemption considering any of the methods described in [36]. For instance, Zeng et al. [159] propose an algorithm to assign priorities and the preemption threshold [130] based on DM. Their optimization goal is the minimization of the stack usage caused by preemption. In general, the maximal stack usage depends on the preemption point within the task. Therefore, defining preemption points between runnables, as shown in [159], the required maximal stack memory size is reduced. In AUTOSAR systems, the cooperative FPS [19, 160] is employed to limit preemption of low priority tasks. Reducing the number of preemptions by scheduling tasks cooperatively means not only defining which tasks can run non-preemptively, but also assigning preemption points, i.e., by means of explicit scheduler calls within a task. An explicit invocation of the scheduler can increase the overhead for context switching and rescheduling, especially in situations when no active task with higher priority is waiting in the ready queue to be scheduled for execution. These design aspects are not targeted in this work and are part of our future work together with the optimization of the stack memory.

A summary of the related work for preemptive FPS is given in Table 4.2.

	Methodology	LET Semantics	Preemptive Scheduling	Scheduling Overheads	Preemption Optimization	Schedulability Check
Contribution	CP/Heuristic	✓	✓	✓	✓	TB
Audsley et al. [133]	Heuristic	✗	✓	✗	✗	WCRT
Davis et al. [134]	Heuristic	✗	✓	✗	✗	WCRT
Garibay-Martinez et al. [137]	Heuristic	✗	✓	✗	✗	WCRT
Garcia and Harbour et al. [139]	Heuristic	✗	✓	✗	✗	WCRT
Huang and Cheng et al. [138]	Heuristic	✗	✓	✗	✗	WCRT
Zhu et al. [140, 141]	Heuristic	✗	✓	✗	✗	WCRT
Metzner et al. [150]	CP	✗	✓	✗	✗	WCRT
Bate et al. [146]	SA	✗	✓	✗	✗	WCRT
Hamann et al. [143]	GA	✗	✓	✗	✗	(?)
Samii et al [94]	GA	✗	✓	✗	✗	WCRT
Azketa et al. [142]	GA	✗	✓	✗	✗	WCRT
Mohd et al. [144]	GA	✗	✓	✗	✗	WCRT
Mehiaoui et al. [148]	MILP/GA	✗	✗	✗	✗	WCRT
Wozniak et al. [149]	MILP/GA	✗	✗	✗	✗	WCRT
Wieder et al. [154]	ILP/Heuristic	✗	✓	✗	✗	WCRT
Bouaziz et al. [145]	Heuristic/GA	✗	✓	✗	✓	SIM
Zeng et al. [155]	MILP	✗	✓	✗	✗	WCRT
Zeng et al. [112, 113]	MILP	✗	✓	✗	✗	WCRT
Zheng et al. [151]	MILP	✗	✓	✗	✗	WCRT
Zhu et al. [152]	MILP	✗	✓	✗	✗	WCRT
Schlatow et al. [153]	ILP	✗	✓	✗	✗	WCRT

Zhao et al. [156]	ILP/Heuristic	✗	✗	✗	✗	UC
Farcas et al. [56]	EDF	✓	✗	✗	✗	(?)
Derler et al. [57]	(?)	✓	✗	✗	✗	(?)
Beckert et al. [80]	(?)	✓	✓	✗	(○)	WCRT
Igarashi et al. [157]	Heuristic	✓	(?)	✗	✗	(?)
Yano et al. [158]	Heuristic	✓	(?)	✗	✗	(?)

Table 4.2: Qualitative comparison of approaches for the FPS synthesis problem.
Legend: satisfied (✓), partially satisfied (○), unsatisfied (✗), or unknown (?), TS (Time-Slicing), TB (Time-Budgeting), SA (Simulated Annealing), GA (Genetic Algorithm), SIM (Simulation), UC (Unschedulability Core).

4.3 System Representation

The schedule synthesis described in this work is applied to applications of each core separately. The following sections describe the application and the overhead model.

4.3.1 Application Model

The software application consists of $n \in \mathbb{N}$ periodic tasks $\tau = \{T_1, ..., T_n\}$. Each task of τ is mapped for execution to any of the processing units, i.e., processor cores, defined as $C = \{C_1, ..., C_m\}$. Let $\tau^u_{cp} = \{T_1, ..., T_{n^u}\}$ be the set of $n^u \in \mathbb{N}$ tasks mapped for execution to a core $C_u \in C$, where $1 \leq u \leq m$ and $m \in \mathbb{N}$. A task $T_i \in \tau^u_{cp}$ is referred to as computation task, where $1 \leq i \leq n^u$.

The number of tasks changes after the integration of LET. In PTP, each computation task T_i is associated with LET Start T^S_i and LET End T^E_i tasks. They are called communication tasks and are responsible for buffering operations. Tasks T^S_i and T^E_i have the same timing attributes as T_i, such as the period P_i, the deadline D_i, the LET duration let_i, and the periodic offset O_i. Let $\tau^u_S = \{T^S_1, ..., T^S_{n^u}\}$ and $\tau^u_E = \{T^E_1, ..., T^E_{n^u}\}$ be the set of LET Start and End tasks mapped to a core $C_u \in C$, respectively. Hence, the set $\tau^u_{cc} = \tau^u_S \cup \tau^u_E$ defines the set of communication tasks on core $C_u \in C$.
In SBP, the IR^{init} is defined to initialize buffers at the beginning of each *global* HP. The duration of the *global* HP, notated as $hp \in \mathbb{N}$, is defined as the *Least Common Multiple (LCM)* of periods of tasks mapped to all cores. The duration of the *local* HP, notated as $hp^u \in \mathbb{N}$, is defined as the LCM of periods of tasks in τ^u_{cp}, where $hp^u \leq hp$. The period P_{init} of IR^{init} is equal to hp and its offset O_{init} is assigned to zero. Although IR^{init}

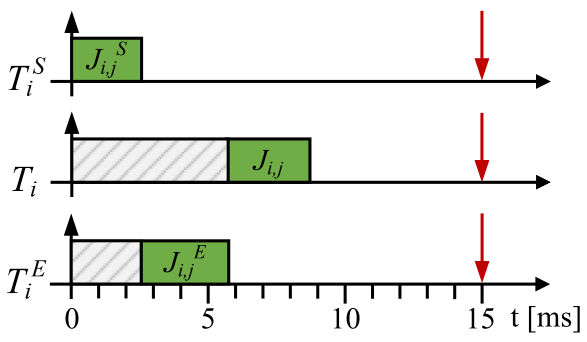
initializes the buffers with default values at the first occurrence of the HP interval, assigning O_{init} equal to zero reduces the amount of memory required to store the schedule table. Otherwise, if O_{init} is assigned equal to hp, a schedule with a duration of $2 * hp$ must be generated and stored in the memory.

The WCET of each task T_i^S, T_i^E, and T_i are defined as $wcet_i^S$, $wcet_i^E$, and $wcet_i$, respectively. The $wcet_{init}$ defines the WCET of IR^{init}. In SBP, for each task T_i the $wcet_i^{index}$ defines the WCET of initializing indexes at the start of task's execution and $wcet_i^{local}$ defines the WCET of copy operations in case of *suppressed writes*. To simplify the formalization in schedule synthesis, in case of SBP, $wcet_i^{index}$ and $wcet_i^{local}$ are assumed to be part of $wcet_i$. Furthermore, although the $wcet_i$ of T_i is different between SBP and PTP, the $wcet_i$ is not explicitly annotated for each protocol to simplify the description of the formalization shown in this chapter.

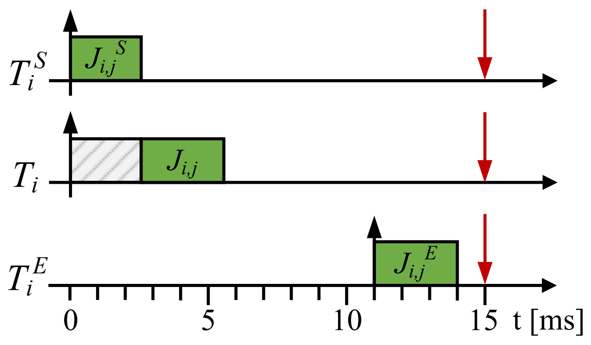
In one HP interval, each task $T_i^S \in \tau_S^u$, $T_i^E \in \tau_E^u$, and $T_i \in \tau_{cp}^u$ is instantiated into $n_i^u \in \mathbb{N}$ number of jobs and IR^{init} into one job. In SBP, the schedule has a duration of hp because IR^{init} has its period equal to hp and $hp^u = hp$ and the number of jobs is defined as $n_i^u = \frac{hp}{P_i}$. In PTP, the schedule has a duration of hp^u and $n_i^u = \frac{hp^u}{P_i}$. Depending on the periods of tasks, hp^u can equal hp also in PTP. Let $J_{i,j}^S$, $J_{i,j}^E$ and $J_{i,j}$ define the j^{th} jobs of tasks $T_i^S \in \tau_S^u$, $T_i^E \in \tau_E^u$, and $T_i \in \tau_{cp}^u$, respectively, where $1 \leq j \leq n_i^u$.

4.3.1.1 Task Activation

Tasks can be activated by the occurrence of events or by the progression of time. In AUTOSAR, the schedule table concept is defined to activate tasks by the progression of time, in which activation offsets in form of *expiry points* are defined to activate each job released in one HP interval. This work uses the schedule table concept [19] of AUTOSAR to activate all LET tasks for two main reasons. Firstly, schedule table offers the flexibility to plan the activation of LET tasks and improve the schedulability. Secondly, AUTOSAR describes means to synchronize schedule tables of different ECUs for the purpose of guaranteeing end-to-end delay requirements across multiple ECUs. This is highly important in LET to ensure time determinism of data exchanged between LET tasks integrated in different ECUs. In order to distinguish the aforementioned table from the schedule table that is used by TTS to schedule jobs, the table that stores activation offsets is referred in this chapter as *activation table*. The duration of the activation table equals the duration of the HP interval, which depending on the LET buffering protocol can be equal to hp or hp^u. The activation table is configured to repeat infinitely after each iteration. Definition 4.1 describes the elements of the activation table.



(a) Equal activation times.



(b) Different activation times.

Figure 4.1: Task execution order in FPS influenced by activation times of LET jobs. Priority ordering is $T_i^S > T_i^E > T_i$. The red arrows indicate the absolute deadlines of jobs. The gray and green boxes indicate the start delays and the execution of jobs, respectively.

Definition 4.1: *The activation table O of all LET jobs released in one HP is defined as*

$$O = O^S \cup O^E \cup O^C, \tag{4.1}$$

where O^S, O^E, and O^C are the set of activation offsets of LET Start, End, and computation jobs, respectively. They are defined as

$$O^S = \{o_{i,j}^S | \forall J_{i,j}^S, \forall T_i^S \in \tau_S^u, i \in [1, n^u], j \in [1, n_i^u]\}, \tag{4.2}$$

$$O^E = \{o_{i,j}^E | \forall J_{i,j}^E, \forall T_i^E \in \tau_E^u, i \in [1, n^u], j \in [1, n_i^u]\}, \tag{4.3}$$

$$O^C = \{o_{i,j} | \forall J_{i,j}, \forall T_i \in \tau_{cp}^u, i \in [1, n^u], j \in [1, n_i^u]\}, \tag{4.4}$$

where $o_{i,j}^S$, $o_{i,j}^E$, and $o_{i,j}$ define activation offsets of respective jobs $J_{i,j}^S$, $J_{i,j}^E$, and $J_{i,j}$.

In FPS, assigning higher priority to communication tasks than to computation tasks is necessary but not sufficient to ensure the communication requirement of PTP. Hence, in FPS, different from TTS, the LET requirements cannot be ensured by scheduling alone, but by considering as well task activation times.

Figure 4.1 shows an example of the execution order between jobs $J_{i,j}^S$, $J_{i,j}^E$, and $J_{i,j}$. In Figure 4.1a, jobs are activated at the same time. Job $J_{i,j}^E$ executes before job $J_{i,j}$ because the priority of task T_i^E is higher than the priority of task T_i. According to LET semantics, the execution of job $J_{i,j}^E$ must occur right before the end of its LET, i.e., at time 15 ms and after the termination of job $J_{i,j}$. In this case, the semantics of LET are not fulfilled. As shown in Figure 4.1b, by changing the activation time of job $J_{i,j}^E$ at a later time, the execution order is ensured and Requirement 4.1 is fulfilled. The time when jobs are activated impacts not only the communication requirement of LET but as well the overall schedule feasibility. Activation offsets must be assigned such that all jobs finish their execution before their deadlines. Job $J^E_{i,j}$, shown in Figure 4.1b, misses its deadline if the activation offset is assigned to 13 ms or 14 ms.

The activation offset $o^S_{i,j}$ of each job $J^S_{i,j}$ of each task $T^S_i \in \tau^u_S$ and the activation offset $o_{i,j}$ of each job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$ are defined based on the period P_i and periodic offsets O_i, as

$$o^S_{i,j} = O_i + (j-1) * P_i \wedge r^S_{i,j} = o^S_{i,j}, \tag{4.5}$$

$$o_{i,j} = O_i + (j-1) * P_i \wedge r_{i,j} = o_{i,j}, \tag{4.6}$$

where $r^S_{i,j}$ and $r_{i,j}$ define the release times of $J^S_{i,j}$ and $J_{i,j}$, respectively.

Activation offsets of LET End jobs are defined during FPS and TTS schedule synthesis such that the communication requirement is fulfilled and a feasible schedule is constructed. The activation offset $o^E_{i,j}$ of each job $J^E_{i,j}$ of each task T^E_i is bounded as

$$o_{i,j} \leq o^E_{i,j} \leq d_{i,j} \wedge r^E_{i,j} = o^E_{i,j}, \tag{4.7}$$

where the $d_{i,j}$ is the absolute deadline defined as $d_{i,j} = r^S_{i,j} + D_i$.

A key benefit of assigning activation offsets to LET End jobs rather than one periodic offset to their respective LET End task is that the schedule capacity can be utilized better and LET End jobs can execute closer to the end of their LET intervals. The offset assigned to a LET End task shifts the activation of its LET End jobs by a fixed amount of time in each periodic occurrence. Whereas, in case of activation table, different activation times can be defined for different jobs of the same LET End task. In addition, the problem of assigning offsets is simplified when activation tables are used.

4.3.1.2 Constraints for SBP Semantics

Periodic offsets assigned to LET tasks shift the occurrence of jobs and the occurrence of their LET intervals. In case of asynchronous offsets, the LET end time of the last jobs can exceed the boundaries of a HP interval. This means that the last job of a LET task could execute in the next HP interval. In SBP, jobs instantiated in one HP interval must not execute in the next HP interval because at the start time of each HP interval, the IR^{init} task must execute before the start of all LET jobs released in the current HP interval. If LET jobs of the previous HP interval have not finished execution, then the job of IR^{init} released in the current HP interval can preempt these jobs, which leads to data stability issues and wrong buffering behavior. The priority of IR^{init} cannot be defined lower than the priority of LET tasks because buffer initialization must occur before task execution. Figure 4.2 shows an example of three asynchronously activated tasks. LET intervals of jobs $J_{k,2}$ and $J_{p,1}$ exceed the HP duration of 10 ms. Hence, the execution of job $J_{p,1}$ spans in the next HP interval. At time 10 ms, jobs $J_{i,3}$ and IR^{init}_1 are released. To preserve the buffering behavior, job IR^{init}_1 must execute before $J_{i,3}$ and

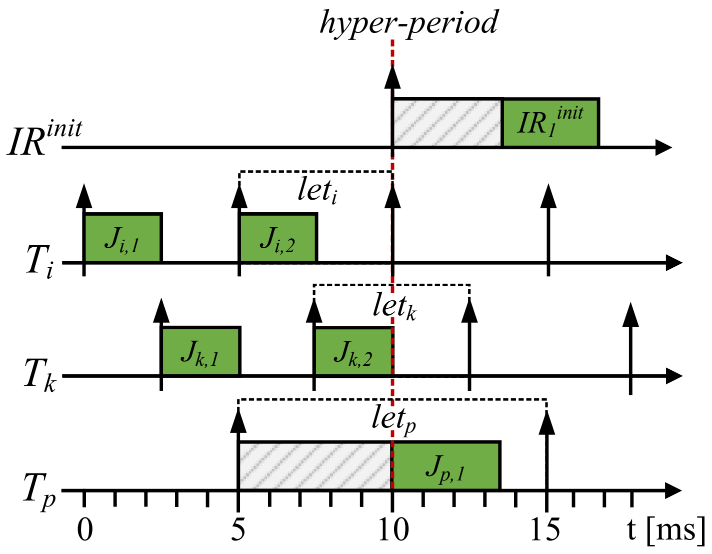


Figure 4.2: Validity of Task Attributes. The gray and green boxes indicate the start delays and the execution of jobs, respectively.

after job $J_{p,1}$ has finished its execution. While this situation can be handled in TTS by delaying the execution of the IR^{init} job and the computation jobs released in the current HP interval until all jobs in the previous HP interval have terminated, this is not possible in FPS because computation tasks are scheduled fully preemptively and can be preempted by IR^{init}.

Because the schedulability of jobs in the current HP interval is affected by the execution of jobs of the previous HP interval, the schedule must be built in both scheduling mechanisms for a duration that is twice as long as the duration hp of the global HP. This can increase the time to synthesize the schedule and the duration of the schedule table. Hence, SBP is applicable for tasks with periodic offsets and LET interval duration that satisfy the condition in Equation (3.8) defined in Section 3.4.2. Equation (3.8) enforces that all jobs released within one HP interval have the end time of their respective LET intervals less than the start of the next HP interval. If the periodic offset of a task equals the period, then the entire LET interval of the last job occurs in the next HP interval. Also in this case, the buffer schedule must have a duration of $2 * hp$ and the period of IR^{init} must equal $2 * hp$, which increases the memory demands to store the buffers and the schedule table. The condition in Equation (3.8) is not a limitation of the schedule synthesis algorithms proposed in this work, but a constraint for correct buffering behavior of SBP.

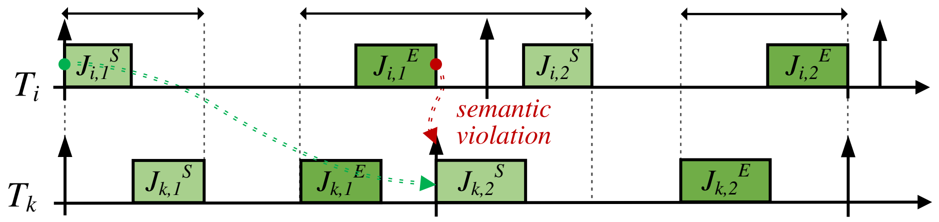
4.3.1.3 Constraints for PTP Semantics

The dataflow and value determinism of LET means that consumer jobs receive outputs produced by dedicated producer jobs exactly at the start and end time of their LET intervals. In the scheduling design phase, there are two key aspects that are essential in PTP regarding dataflow and value determinism, as described below.

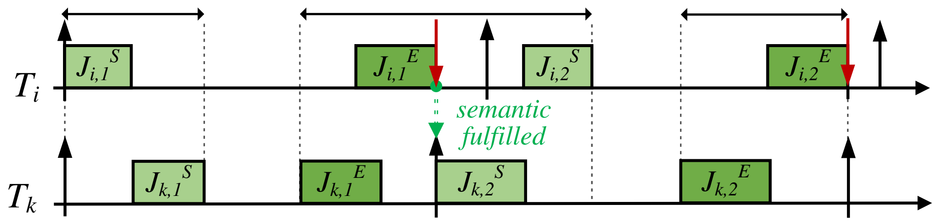
1) The Requirement 4.1 of PTP must be ensured by scheduling. The execution order among simultaneously active LET Start jobs of different tasks is not essential in terms of determinism. Nevertheless, they must be scheduled to execute all before their computation and LET End jobs. Hence, if several LET Start jobs of different tasks are active at the same time instant, then they are scheduled to execute as a group first before the start of any of their respective computation jobs. Similarly, the execution order of simultaneously active LET End jobs of different tasks is not important among them as long as they execute all before the end of their LET intervals and after their respective computation jobs have finished their execution.

2) Simultaneously active LET Start and LET End jobs of different LET tasks that have a consumer–producer relation must execute such that the dataflow and value determinism is ensured. For instance, in coinciding LET intervals, a LET Start job of the consuming LET interval overlaps with the execution of a LET End job of the producing LET interval. In this case, the LET Start job may or may not read the outputs produced by the LET End job. This depends on the job that is scheduled to execute first. This form of LET interval overlapping occurs for instance between LET intervals with non-harmonic duration and for non-harmonically and asynchronously activated LET tasks. Hence, if LET Start and End jobs of two different LET intervals have coinciding execution, then according to LET semantics the LET Start job must always execute before the LET End job, because the LET Start job must read the output of the previous occurrence of producer's LET interval. A schedule cannot always be found under such constraint. Therefore, this form of overlapping between LET intervals must be avoided during the design of LET intervals.

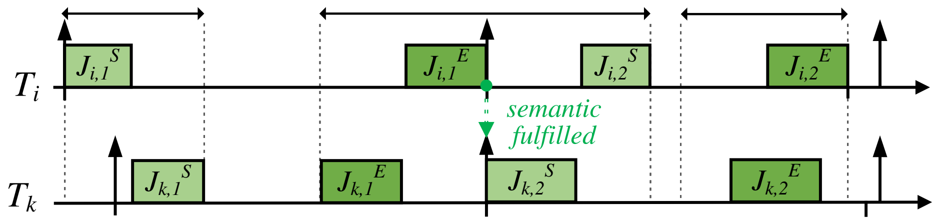
An example describing the non-harmonic overlapping between LET Start and End jobs is given in Figure 4.3a. According to LET semantics, job $J^S_{k,2}$ must not read the output of job $J^S_{i,1}$ but the initial value, which in this case is the previous output. To avoid this situation, $J^S_{i,1}$ can be scheduled to execute after $J^S_{k,2}$, which is not possible because the deadline of $J^S_{k,2}$ cannot be fulfilled. A feasible schedule is only found if the initial semantic of LET is relaxed and jobs $J^E_{i,1}$ is scheduled to execute before jobs $J^S_{i,2}$ and $J^S_{k,2}$. However, this is against the main purpose of using LET because dataflow and value determinism is partially fulfilled. One way of solving this problem and provide determinism is by changing the duration of LET intervals as shown in Figure 4.3b or by changing the periodic offset as shown in Figure 4.3c. The design of LET attributes, such as the duration and periodic offset, is not the focus of this work. Therefore, the proposed schedule synthesis is applicable under the assumption that the end of one LET interval does not overlap with the beginning of another LET interval, as is the case of Figure 4.3a.



(a) Coinciding LET Start and End jobs. LET semantics are violated for jobs $J_{i,1}^E$ and $J_{k,2}^S$. LET interval duration is equal to period for both tasks T_i and T_k.



(b) Changing the duration of producer's LET interval. LET interval duration is equal to the period for T_k and less than the period for T_i.



(c) Changing the periodic offset of consumer's LET interval. LET interval duration is equal to period for both tasks T_i and T_k.

Figure 4.3: Value and dataflow determinism in PTP for coinciding LET intervals.

4.3.2 Overheads Model

The schedule synthesis of this work considers the impact of different OS operations. Three types of overheads are defined and considered: the timer interrupt overhead, the task termination overhead, and the preemption overhead. The *timer* interrupt overhead refers to the time required by the timer to activate a job. The *preemption* overhead consists of the context-switching time caused by preemption. Whenever a task is assigned for execution, the running job is preempted, including the *idle* job, and the context change occurs before the preempting job starts its execution. The preemption overhead is otherwise referred to as context-switching overhead. The *termination* overhead defines the time required by the OS to terminate a job. Further OS operations such as for example the activation of tasks by explicit call of ActivateTask(), or SetEvent() and WaitEvent() functions [19] are not considered. However, if such functions are called in the context of a task, their run-time overheads are assumed as part of task's WCET.

The timer interrupt is instantiated each time one or multiple tasks are released for activation. The average execution time of the timer interrupt is notated as ov_a. It includes the time to activate one task and the time involved in counter activities such as for example assigning the next compare value of the counter. Let I^u_{tr} denote the timer interrupt that handles the activation of tasks allocated for execution on core C_u. This work assumes that if an occurrence of I^u_{tr} activates more than one task at the time of its release, then the execution time of this occurrence is a multiple of ov_a and the number of released jobs. Note that when a task is activated, a job is released.

The handling of context switches and task termination varies among different OS implementations and configurations. Typically, the context-switching operation is not a standalone step but is part of different OS operations. For example, because AUTOSAR uses a FPS scheduler to schedule tasks, the context-switching operation is associated with the execution of the scheduler, which takes scheduling decisions such as assigning for execution the next active job. The operation of saving and restoring of task contexts is atomic and occurs non-preemptively. Therefore, the run-times of enabling and disabling interrupts is as well associated with the context-switching operation. Additionally, if timing protection is enabled [19], then a switch from user to OS mode is performed and timing and memory protection activities take place. Hence, a context-switch involves in addition to saving and restoring the contexts of jobs as well operations such as for instance enabling and disabling of interrupts, setting up timing protection activities, management of OS data structures, saving of the stack pointer in the control block, switching to the system's stack pointer, and scheduler execution. This happens as well because in FPS a preemption takes place when a task or interrupt is activated or if an OS *Application Programming Interface (API)* function is called in a task's context. In TTS, this is not always the case because a preemption can happen as well when the execution time interval assigned to a task ends and the time interval of another task begins. Considering only the hardware context-switching time, i.e., the time to save and restore registers is not enough to address the impact of the OS in the schedule construction. Therefore, in this work, the context-switching overhead includes the hardware context-switching time and the execution time of the scheduler and of any required OS operation occurring during a context change.

The termination operation is called in the end of a task's execution. It includes several operations such as removing the task from the schedule queue, changing the context between the terminating job and the next running job, checking if resources are released by the terminating task, and the call of the scheduler. Because the termination operation is explicitly called in the context of a task, its run-time is often included in WCET calculations. In this work, the run-time of the termination and context-switching operations are abstracted outside of task WCETs. Let ov_t and ov_{cs} define the execution time that is involved at the termination of a job and the execution time of one context-switch, respectively. The hardware based context-switch is included in ov_t. Because the context-switch and terminate operation include several activities, in practice they are not fixed for each preemption or termination point. In this work, they are assumed as constant to simplify their handling during schedule synthesis. This work assumes that estimations of activate, terminate, and context-switching run-times are defined before schedule synthesis either by WCET calculations or by measuring these operations using commercial tracing and debugging tools. A dedicated interrupt I^u_{cs} is defined to handle the context switch between two tasks on core C_u. The I^u_{cs} is invoked at the start, preemption, and termination of every LET task to perform the necessary context change. At the termination of a job, the I^u_{cs} has an execution time of ov_t. At the start and preemption of a job, I^u_{cs} has the execution time of ov_{cs}. The I^u_{cs} is an abstraction defined to isolate ov_t and ov_{cs} operations.

Figure 4.4 shows an example of two tasks scheduled by FPS. Task T_1 has higher priority than task T_2. Tasks execute on the same core C_u and their activation is handled by the timer interrupt I^u_{tr}, which is triggered for activation at times 0 ms, 5 ms, and 10 ms. In each occurrence of I^u_{cs}, a context-switch operation takes place before the start of I^u_{cs}. After I^u_{tr} terminates, a context switch takes place. At time interval 1.5 ms − 2 ms the context switch occurs between I^u_{tr} and T_1. The actual activation times of T_1 and T_2 are not the same and differ from the start time of the timer interrupt because they are set in active state during its execution. In this example, the WCET are as follows: $wcet_1 = 1\,\text{ms}$, $wcet_2 = 2.5\,\text{ms}$, $wcet^u_{tr} = 1\,\text{ms}$, and $ov_{cs} = ov_t = 0.5\,\text{ms}$. Hence, the utilization of T_1, T_2, I^u_{cs}, and I^u_{tr} are 30 %, 25 %, 35 %, and 20 %, respectively. By considering timer and context-switching overheads the system is fully utilized. In case these overheads are not addressed during schedule synthesis, then the schedule is planned assuming that the system has 55 % of utilization.

The *start* and *preemption delays* caused by the execution of the timer interrupt, by higher priority tasks, and by the described overheads define the actual start and end times of tasks. These delays are considered during schedule generation of both TTS and FPS. In TTS, overheads are added during the full construction of the schedule table. In FPS,

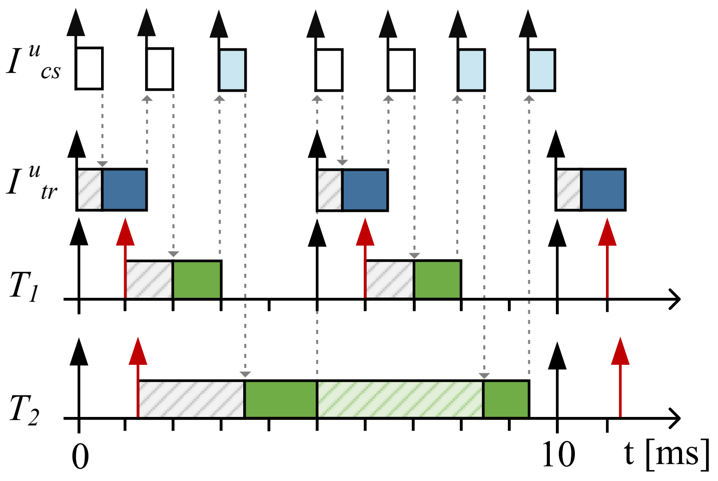


Figure 4.4: Example of task activation and context-switching. Tasks T_1 and T_2 have respective periods 5 ms and 10 ms. The timer interrupt I^u_{tr} is triggered at times 0 ms, 5 ms, and 10 ms. The context-switching occurs before and after execution of the timer interrupt. The timing information is: $wcet_1 = 1$ ms, $wcet_2 = 2.5$ ms, $wcet^u_{tr} = 1$ ms, $ov_{cs} = ov_t = 0.5$ ms. The actual activation of jobs, indicated by the red arrows, occurs during the execution of the timer interrupt. The activation jitter of every job is the time distance between occurrence of the black and red arrows. The white and light blue boxes in the I^u_{cs} indicate the ov_{cs} and ov_t, respectively. The green boxes indicate the execution of tasks and the dark blue boxes the execution time of I^u_{tr}. The green boxes and the gray boxes indicate the preemption and start delays, respectively.

they are added during schedule verification, such that the assigned priorities to tasks lead to a valid schedule.

4.4 Schedule Synthesis Approach

This section describes the methodology of constructing the schedule of LET tasks for both FPS and TTS scheduling mechanisms.

4.4.1 Methodology

The schedule synthesis methodology and workflow is described in Figure 4.5. Although TTS and FPS are conceptually different scheduling strategies, the proposed methodology of constructing the schedule of LET tasks is applied for both of them. To improve the run-time performance, the schedule is constructed step-wise and for each core separately. Hence, if the software application has tasks distributed to multiple cores, then the schedule is generated for tasks of each core individually. The approach described here offers the possibility to construct, although not optimal, the schedule of

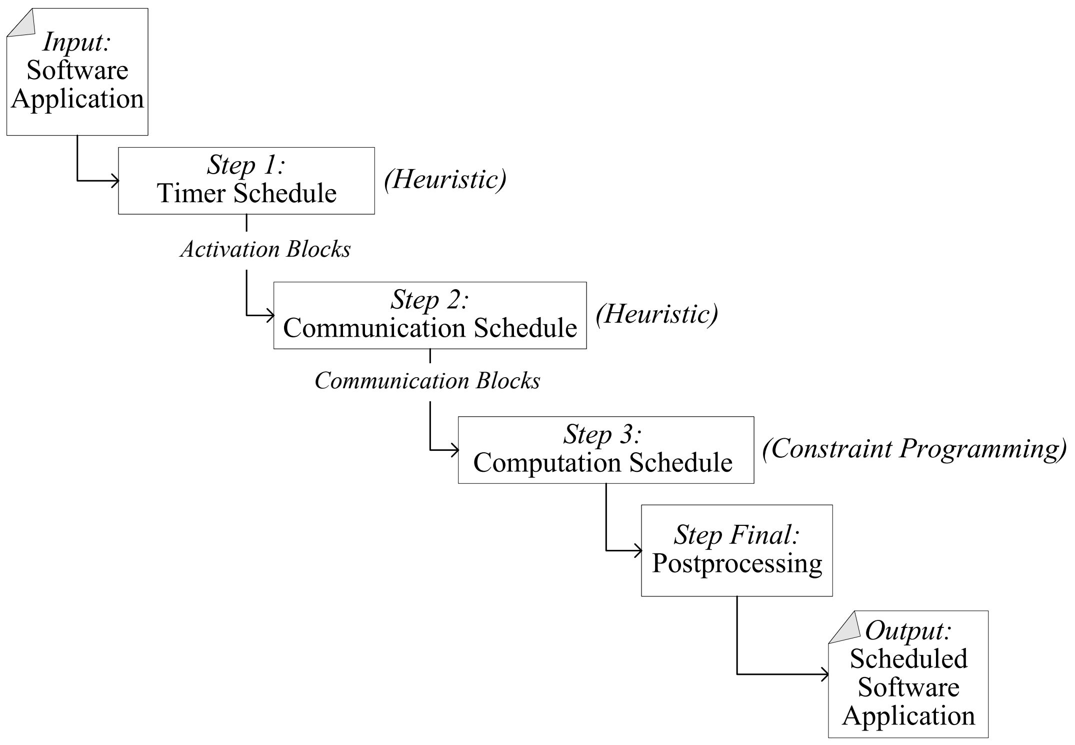

Input: Software Application
Step 1: Timer Schedule
(Heuristic)
Activation Blocks
Step 2: Communication Schedule
(Heuristic)
Communication Blocks
Step 3: Computation Schedule
(Constraint Programming)
Step Final: Postprocessing
Output: Scheduled Software Application

Figure 4.5: Schedule Synthesis Workflow and Methodology.

communication tasks running on different cores such that inter-core communication effects and overheads caused by data stability mechanisms are fully eliminated.

The schedule of LET tasks is constructed in three main steps. In *step 1*, the schedule of the timer interrupt is generated for all its occurrences released in the defined HP interval to activate computation and LET Start tasks. Because the timer interrupt executes with higher priority than LET tasks, start and end times of all its occurrences are generated to define start and preemption delays that they cause to communication and computation jobs. The schedule of the timer interrupt is used as input for the next steps. In *step 2* and *step 3*, the schedule of communication and computation tasks is constructed. To improve the performance, the proposed TTS and FPS schedule synthesis algorithms generate the schedule of communication and computation tasks in two separate steps. The schedule of communication tasks is constructed first because these tasks execute at higher priority than computation tasks and must satisfy Requirement 4.1. The schedule of computation tasks is constructed after the schedule of communication tasks is generated and their execution is allocated by the algorithm to the remaining time intervals that are not occupied by communication jobs. In the *final step*, the resulting schedule is post-processed and exported to a file with the *Best Trace Format (BTF)* format [161] for further evaluation of the resulting solution. Although the step-wise approach reduces greatly the schedule synthesis performance, it enforces certain limitations and assumptions, which are described throughout this chapter.

To avoid inconsistent buffering operations and reduce resource requirements for stack memory and processing time caused by context switches, communication tasks are not allowed to be preempted either by other communication tasks or by computation tasks. Only the timer interrupt or high-priority interrupts can preempt these tasks. Start delays of the timer interrupt caused by the execution of LET tasks cannot be fully avoided. These delays occur when LET tasks use during their execution the *suspend interrupt locks* [19] of OS to ensure atomic operation on shared data. Constructing the schedule based on WCET, which is highly necessary, reduces the possibility to know precisely in which position of task's execution time suspending and resuming of interrupts occurs. Therefore, during schedule generation, either the pessimistic assumption is taken that at preemption time of a job the timer interrupt is always delayed by a fixed amount of time, or its execution is always shifted to the termination time of any running job. The full shift of timer interrupt's execution would be valid in terms of scheduling as long as the timer interrupt starts the execution at the termination of the job before the start of any other active job. However, in practice the timer interrupt has a higher priority than any OS task. Therefore, the preemption of LET jobs occurs as soon as the *resume interrupt lock* operation is called by these jobs. The preemption caused by the timer interrupt cannot be fully avoided for computation tasks because activation of tasks must take place at any condition. Otherwise, if the execution of the timer interrupt is postponed for to long, then the activation of tasks or setting of specific OS events are also delayed and the activation times are no longer deterministic. This work assumes that only communication tasks use *suspend interrupt locks*. Hence, the schedule of the timer defined in *step 1* is adjusted in *step 2* by a fixed time delay defined as ov_{res}.

Computation tasks are allowed to preempt and delay each-other or be preempted and delayed by communication tasks. Applications targeted in this work are highly utilized especially because of the increased load caused by LET semantics. If preemptions are fully disabled, then a valid schedule is unlikely to be found for such applications. Therefore, the schedule of computation tasks is constructed in a way that preemptions are enforced by the constraint solver whenever necessary. Thus, the schedule synthesis algorithm provides a valid, fully non-preemptive schedule for computation tasks, if one exists. In SBP, the algorithm of *step 2* is used to build the schedule of the IR^{init}.

A straightforward algorithm based on DM heuristic is described in *step 2* to construct the schedule of communication jobs, while satisfying *timing*, *communication*, and *resource* requirements. A CP approach is not applied for these tasks, although possible, because the desired solution is achieved faster without optimization strategies provided by CP algorithms. The proposed algorithm ensures that communication jobs execute generally non-preemptively, unless preempted by the timer interrupt, and close to the boundaries of their LET intervals. Preemptions caused by the timer interrupt are avoided for LET End jobs by assigning their activation offsets, during generation of their schedule, such that their execution does not overlap with the execution of any timer's job.

The schedule of computation tasks is constructed in *step 3* via CP techniques because multiple constraints and requirements must be solved while a feasible schedule is searched. These techniques are known to solve multiple constraints and reduce the complexity of the problem, which is not possible in heuristic-based algorithms. Furthermore, CP techniques offer the possibility to efficiently optimize the schedule of computation tasks regarding preemption overheads while providing a feasible schedule in a reasonable amount of time. The efficiency of CP techniques for scheduling problems is already showed in Section 4.2.

4.4.2 Start and Preemption Delays

The execution of the timer interrupt I^u_{tr} and the occurrence of context-switching and terminate overheads causes *start* and *preemption delays* to jobs of LET tasks. Similarly, LET tasks cause start and preemption delays to each-other. The *start delay* of a job defines the duration between its activation, i.e., release time and the start of execution. The *preemption delay* caused by execution of the timer or other LET tasks defines the total time duration in which a task is in preemption state. Start and preemption delays increase the response time of tasks and in certain cases they can lead to deadline violations. Therefore, they must be addressed and minimized during schedule synthesis. This work distinguishes between the *preemption delays* and *preemption costs*. The former ones refer to the time and memory overheads caused by the context-switching between jobs. To handle preemption and start delays during FPS and TTS schedule synthesis, the concept of *activation* and *communication blocks* is defined, as described in Sections 4.4.2.1 and 4.4.2.2.

4.4.2.1 Activation Blocks

Activation blocks are defined to simplify the management of activation and preemption delays in schedule synthesis of LET tasks. They are defined to isolate the execution of the timer interrupt from the execution of LET tasks. An activation block defines a time interval in which the execution of one or multiple instances of the timer interrupt I^u_{tr} take place. Definition 4.2 annotates elements of an activation block.

Definition 4.2 (Activation Block): *Let A^u be the set of activation blocks on core $C_u \in C$ calculated based on release times of all jobs executing on core C_u, where $u \in \mathbb{N}$. The a^u_s denotes the s-th activation block, where $s \in \mathbb{N}$. The a^u_s.start and a^u_s.end define the start and end time of the activation block a^u_s, respectively. The duration of each block a^u_s is constant and defined as the difference between a^u_s.end and a^u_s.start.*

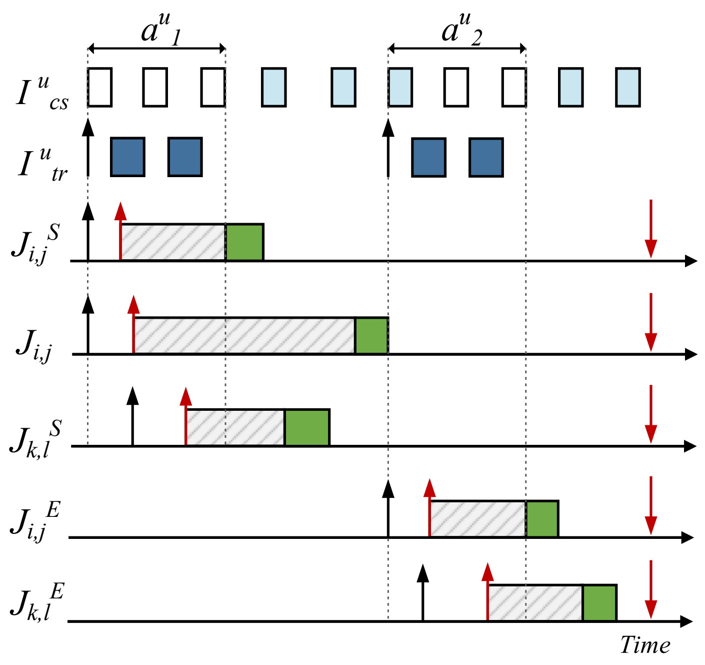
The duration of an activation block is defined the number of unique release times of jobs and the overlapping of timer interrupt instances. In each unique job's release time, one instance of the timer interrupt is released to handle the activation of all jobs released at this time. If several jobs release at the same time instant, one corresponding activation block is created, which has the duration equal to the sum of two context-switches overheads and the execution of one timer interrupt instance, defined in this case as $t * ov_a$, where t is the number of activated jobs. Part of the activation block's duration is also the context-switching overhead that is involved to switch the context between the timer interrupt and any running job, including the idle job. The minimum duration of an activation block is $2 * ov_{cs} + t * ov_a$. If multiple instances of the timer interrupt overlap, one activation block is created by merging all overlapping blocks into one. Therefore, the duration of an activation block is defined by the number of activated jobs, the number of timer instances, and the number of context switches.

Activation blocks are calculated statically before the generation of task's schedule based on unique release times of all jobs activated within one HP, defined by configured task periods and periodic offsets, and their respective overlapping. Activation blocks of LET End tasks are calculated after the assignment of their activation offsets.

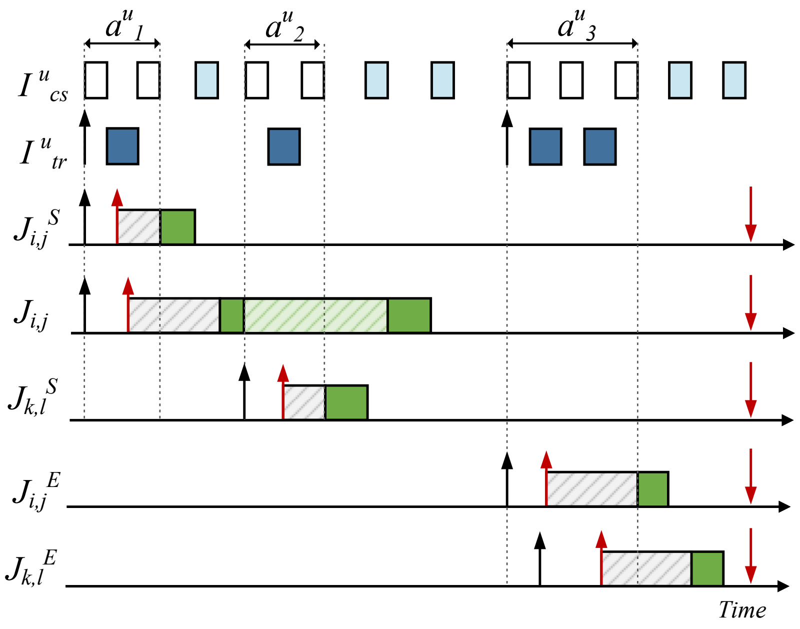
Figure 4.6 shows two examples of activation blocks defined based on the described approach. The example of Figure 4.6a depicts the start delay caused by two unique activation blocks. The first block isolates the execution run-time of two occurrences of the timer interrupt and the run-time overhead for three context-switches. The first instance of the timer interrupt in a_1^u activates jobs $J_{i,j}^S$ and $J_{i,j}$. The second timer's instance activates job $J_{k,l}^S$. A timer interrupt instance is released in each unique release time of jobs. The second activation block a_2^u isolates the activation of jobs $J_{i,j}^E$ and $J_{k,l}^E$. Similarly as in the first activation block, the third and fourth instances of the timer are activated in different times, which correspond to the release times of jobs $J_{i,j}^E$ and $J_{k,l}^E$. Figure 4.6b shows an example of preemption delays caused by activation block a_2^u to job $J_{i,j}$. The actual activation of jobs happens during the execution of the timer interrupt. In these examples, the actual activation of jobs is indicated by red arrows.

4.4.2.2 Communication Blocks

Communication tasks are scheduled with higher *urgency* than computation tasks. To improve the schedulability of the system and to ensure that communication jobs execute close to the release and terminate of their LET intervals, the preemption of computation tasks is allowed and handled during schedule synthesis of computation tasks. The notion of *communication blocks* is used to isolate the execution of communication jobs during schedule synthesis of computation jobs. By abstracting the execution of communication jobs within communication blocks, handling of start delays and preemptions during schedule synthesis of computation jobs is simplified and the time to synthesize the schedule of computation jobs is reduced. Reducing the time that it takes to synthesize and optimize the schedule of computation jobs is highly necessary especially for large applications, in which the enormous amount of jobs in one HP

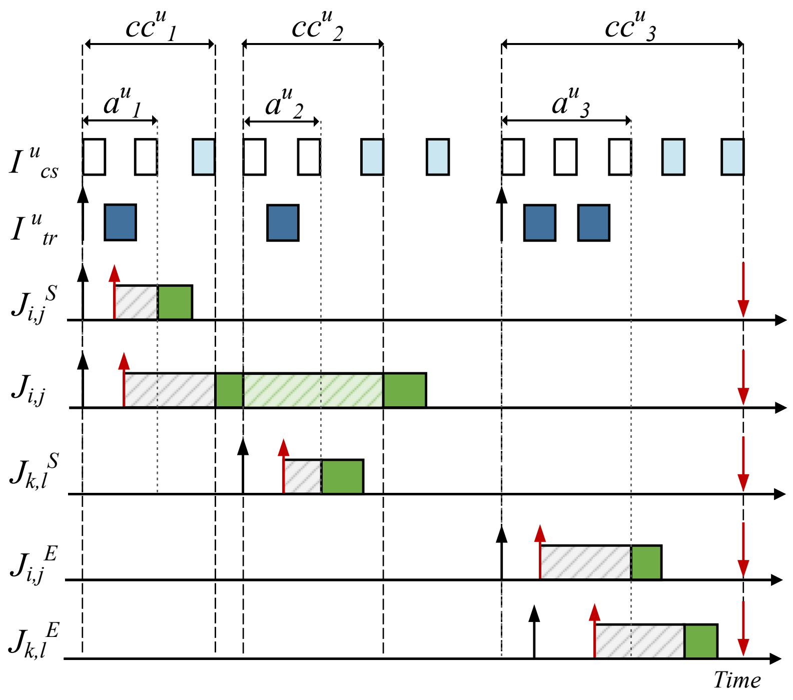


(a) Activation blocks with different releases.

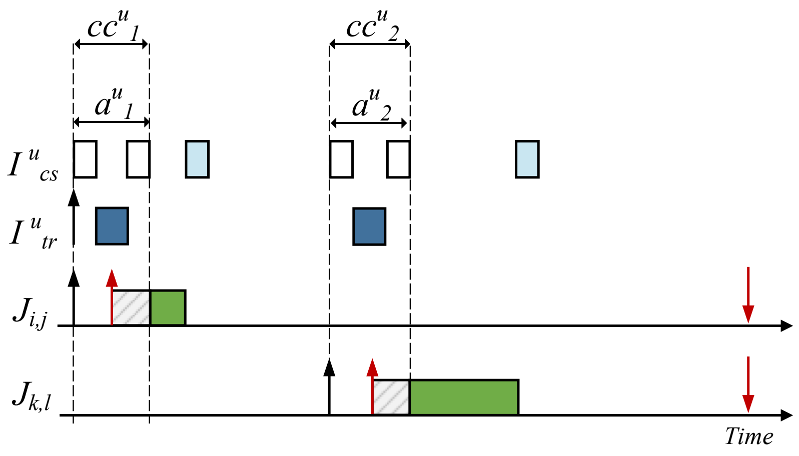


(b) Activation blocks with preemption.

Figure 4.6: Examples of activation blocks created by activation of LET jobs executing on the same core. The dark blue boxes indicate the execution time of the timer interrupt. The white boxes indicate the time of the context switch and the light blue boxes indicate the duration of the terminate operation. The green boxes indicate the execution time of jobs. The back and red arrows pointing up indicate the planned and the actual release of jobs, respectively. The red arrows pointing down indicate the absolute deadline of jobs. The start and preemption delays are indicated by the gray and green pattern-filled boxes, respectively.



(a) Composition of communication and activation blocks in PTP.



(b) Composition of communication and activation blocks in SBP.

Figure 4.7: Examples of communication blocks. The dark blue boxes indicate the execution time of the timer interrupt. The white boxes indicate the time of the context-switching and the light blue boxes indicate the duration of the terminate operation. The green boxes indicate the execution time of jobs. The back and red arrows pointing up indicate the planned and the actual release of jobs, respectively. The red arrows pointing down indicate the absolute deadline of jobs. The start and preemption delays are indicated by the gray and green pattern-filled boxes, respectively.

interval increases the quantity of variables and constraints that must be solved by the CP solver. Definition 4.3 annotates elements of a communication block.

Definition 4.3 (Communication Block): *Let cc^u be the set of communication blocks on core $C_u \in C$ defined based on the schedule of communication jobs executing on core C_u, where $u \in \mathbb{N}$. Let $cc^u_r \in cc^u$ be the r^{th} communication block, where $r \in \mathbb{N}$. The $cc^u_r.start$ defines the start time and the $cc^u_r.end$ defines the end time of cc^u_r. The duration d^u_r of cc^u_r is defined as $d^u_r = cc^u_r.end - cc^u_r.start$.*

Unique communication blocks are calculated statically after the schedule generation of communication tasks. In PTP, a communication block can have one or multiple activation blocks and the execution of at least one communication job. In SBP, except of the first communication block, the rest of communication blocks equal activation blocks because LET communication tasks are not present in SBP. The first communication block contains the execution of the IR^{init}. In SBP, computation jobs are either preempted by the timer interrupt or by other computation jobs. It should be noted that an application may have both SBP and PTP protocols integrated to support LET.

Figure 4.7a shows an example of five jobs activated and executed on core C_u. The example depicts communication block composition for the PTP case. Activation of communication job $J^S_{i,j}$ and computation job $J_{i,j}$ is handled by the first instance of the timer interrupt I^u_{tr} and is encapsulated in the activation block a^u_1. The communication block cc^u_1 consists of the duration of the activation block a^u_1, the execution of communication job $J^S_{i,j}$, and the context-switch involved at the termination of job $J^S_{i,j}$. The communication block cc^u_2 encapsulates the activation block a^u_2 and the execution of communication job $J^S_{k,l}$ followed by one context-switching operation. The communication block cc^u_3 encapsulates activation block a^u_3, the execution of communication jobs $J^E_{i,j}$ and $J^E_{k,l}$, and the context-switching overheads included in their respective termination operations. The occurrences of I^u_{cs} that correspond to the termination of computation or communication jobs contain ov_t as run-time.
Figure 4.7b shows an example of communication block composition for the SBP. The duration of communication blocks cc^u_1 and cc^u_2 consist of the duration of activation blocks a^u_1 and a^u_2, respectively, which encapsulate the activation of jobs $J_{i,j}$ and $J_{k,l}$.

The duration of each communication block includes the context-switching operation that is required to switch to the execution of the first job in the block and the context-switching operation that is required to switch the execution to the next job after the communication block. This approach simplifies the schedule synthesis of computation jobs such that context-switching overheads are not explicitly handled in case a computation job starts or terminates at the end or the beginning of a communication block. In certain cases, if the end of a communication block corresponds to the start of another communication block, then blocks are merged to one. The final set of communication blocks considered in the schedule synthesis of computation jobs does not have overlapping communication blocks.

4.5 Time-Triggered Schedule Synthesis

The TTS scheduling of this work is based on the *time-triggered* scheduling used in the avionic domain. The task set of avionic applications is basic in nature and task's preemptions are generally avoided by constructing a non-preemptive schedule. However, this is not the case for the automotive applications targeted in this work, in which preemption is a necessary mechanism used to ensure that timing and communication requirements are fulfilled. Therefore, the proposed TTS scheduling synthesis approach enables preemption of tasks by defining in the schedule table points of time in which tasks can resume their execution.

TTS is an effective way to deterministically schedule LET tasks and to ensure their *timing*, *communication*, and *resource* requirements in design time and during their target execution. In TTS, scheduling decisions are taken through the progression of time based on *start times* defined in the *schedule table* for all the jobs released in the HP duration. Whenever the time for starting a job is reached, the *scheduler* allocates the job for execution. The running task is preempted, its context is stored, and the context of the new running task is loaded. In case a task finishes its execution before the planned time, then the *idle job* executes until the start time of another job begins. After a task is preempted, it is resumed at dedicated *resume times* defined in the schedule table.

In this work, the schedule table, denoted as S^u, is constructed at design time for all periodic tasks. It contains the start and resume times of all the jobs released and executed within the HP interval. The schedule table of TTS is not the same as the AUTOSAR schedule table concept [19]. The former one is defined to activate tasks based on the progression of time but scheduling decisions are taken based on priorities by the FPS scheduler. In TTS, LET tasks are scheduled independently of how they are activated, i.e., either by events or by an AUTOSAR schedule table. Definition 4.4 describes the TTS schedule table.

Definition 4.4 (TTS schedule table): *The TTS schedule of LET tasks is a composition of start and resume times of all the jobs released in the HP interval.*
The schedule table S^u is defined as

$$S^u = S^S \cup S^E \cup S^C \cup R^S \cup R^E \cup R^C \cup \{st_1^{init}\} \cup \{rt_1^{init}\}, \tag{4.8}$$

where S^S and R^S are the set of start and resume times for all LET Start jobs, respectively. The S^E and R^E are the start and resume times of all LET End jobs. The S^C and R^C denote the set of start and resume times of all computation jobs.

$$S^S = \{st_{i,j}^S | \forall J_{i,j}^S, \forall T_i^S \in \tau_S^u, i \in [1, n^u], j \in [1, n_i^u]\} \tag{4.9}$$

$$S^E = \{st_{i,j}^E | \forall J_{i,j}^E, \forall T_i^E \in \tau_E^u, i \in [1, n^u], j \in [1, n_i^u]\} \tag{4.10}$$

$$S^C = \{st_{i,j} | \forall J_{i,j}, \forall T_i \in \tau^u_{cp}, i \in [1, n^u], j \in [1, n^u_i]\} \quad (4.11)$$

$$R^S = \{rt^S_{i,j} | \forall J^S_{i,j}, \forall T^S_i \in \tau^u_S, i \in [1, n^u], j \in [1, n^u_i]\} \quad (4.12)$$

$$R^E = \{rt^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1, n^u_i]\} \quad (4.13)$$

$$R^C = \{rt_{i,j} | \forall J_{i,j}, \forall T_i \in \tau^u_{cp}, i \in [1, n^u], j \in [1, n^u_i]\} \quad (4.14)$$

The st^{init}_1 defines the start time of the first job of IR^{init} and rt^{init}_1 defines the set of resume times of the first job of IR^{init}. The $st^S_{i,j}$, $st^E_{i,j}$ and $st_{i,j}$ denote the respective start times of jobs $J^S_{i,j}$, $J^E_{i,j}$ and $J_{i,j}$. The $n^u_i \in \mathbb{N}$ defines the number of jobs released in one HP interval.

The $rt_{i,j} = \{rt_{i,j}(p_t) | p_t \in [1, n^C_{rt}], n^C_{rt} \in \mathbb{N}\}$ defines the set of resume times for job $J_{i,j}$, and $rt_{i,j}(p_t)$ denotes the p_t-th resume time of job $J_{i,j}$.
The $rt^S_{i,j} = \{rt^S_{i,j}(p_t) | p_t \in [1, n^S_{rt}], n^S_{rt} \in \mathbb{N}\}$ defines the set of resume times for job $J^S_{i,j}$, and $rt^S_{i,j}(p_t)$ denotes the p_t-th resume time of job $J^S_{i,j}$.
The $rt^E_{i,j} = \{rt^E_{i,j}(p_t) | p_t \in [1, n^E_{rt}], n^E_{rt} \in \mathbb{N}\}$ defines the set of resume times for job $J^E_{i,j}$, and $rt^E_{i,j}(p_t)$ denotes the p_t-th resume time of job $J^E_{i,j}$.

Because a computation job can be preempted multiple times during its execution, multiple *resume* times are defined and stored in the schedule table such that the computation job is resumed again after each preemption until it completes its execution. In general, the number of resume times corresponds to the number of times that a job is preempted, under the assumption that the job takes exactly WCET time to execute. If the job finishes earlier than the planned WCET time, then the resume times occurring after this time are ignored by the scheduler. Hence, the TTS scheduler resumes for execution only jobs that are active at the time the resume time is reached. If computation jobs are not preempted, then the set of resume times R^C is empty. The same applies for communication jobs.

As described throughout this chapter, communication tasks are not allowed to be preempted by each other or by computation tasks. Instead, they can be preempted by the timer interrupt I^u_{tr} or any other software or hardware interrupts. To resume their execution, resume times are stored in the schedule table. If communication jobs are not preempted, then sets R^S and R^E are empty. In this work, the resume times are calculated statically based on the constructed schedule trace after the schedule of LET tasks is generated.

In TTS, the *schedule generation problem* refers to the construction of the schedule table S^u, such that *timing*, *communication*, and *resource* requirements are fulfilled. The schedule synthesis algorithm described in Section 4.5.1 constructs the schedule of communication tasks. This algorithm is applied when any or both PTP and SBP protocols are used to integrate LET semantics. In case of SBP, it constructs the schedule of the IR^{init}, which during synthesis is treated as a LET Start task. The schedule synthesis described in Section 4.5.2 generates the schedule of computation tasks for PTP, SBP, and a combination of both protocols for the same application.

4.5.1 Scheduling of Communication Tasks

This section describes a straightforward heuristic-based algorithm to assign start and resume times of all communication jobs. The algorithm's flow is described by three main steps.

St 1 The activation blocks that isolate the execution of the timer are calculated to handle the start and preemption delays that the timer interrupt causes to LET Start jobs. These activation blocks include only the time it takes to activate LET Start and computation jobs and the respective terminate and context-switch delays. Activation blocks that result from activation of LET End jobs are created only during offset assignment and schedule generation of these jobs.

St 2 The schedule of LET Start jobs is constructed considering preemptions and delays caused by activation blocks. The execution and activation of LET Start jobs is encapsulated in communication blocks, which are calculated at the end of their schedule synthesis and are used during schedule generation of LET End jobs.

St 3 The schedule and activation offsets of LET End jobs are generated. Activation offsets are assigned such that LET End jobs are never preempted or delayed by the occurrence of any timer job or other communication jobs.

Algorithm 5 constructs the schedule of LET Start jobs and Algorithm 6 generates the schedule and activation offsets of LET End jobs. Both algorithms assign start times to every job such that the end of a job corresponds to the start of another job or to the start of an activation block. The assignment of start times to LET Start jobs does not follow any ordering, besides the ordering defined by the time that these jobs are released. A particular ordering of LET Start jobs is not enforced because if they are released at coinciding time intervals then they must all be scheduled in any order before the execution of their respective computation and LET End jobs. The end times of the jobs are calculated during schedule synthesis to ensure that their WCETs are entirely scheduled within their deadlines.

For each communication job $J_{i,j}^S$ of each task $T_i^S \in \tau_S^u$, let the start time $st_{i,j}^S$, the end time $et_{i,j}^S$, and the preemption time $pt_{i,j}^S$ be the scheduling attributes solved by Algorithm 5. For each communication job $J_{i,j}^E$ of each task $T_i^E \in \tau_E^u$, let the start time $st_{i,j}^E$, the end time $et_{i,j}^E$, and the offset $o_{i,j}^E$ define the scheduling attributes that are solved by Algorithm 6. Although the release of a job does not occur exactly at the time of its activation offset,

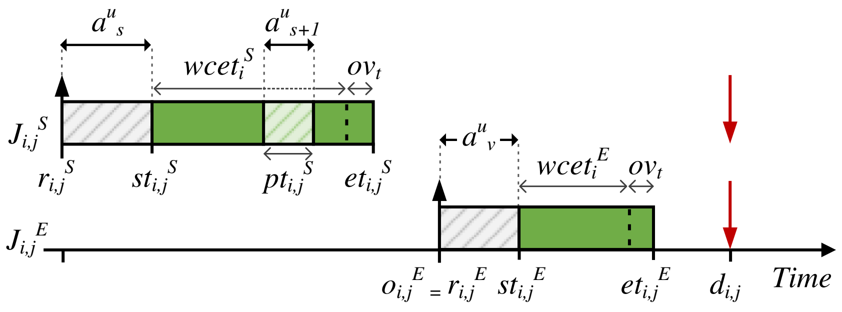


Figure 4.8: Scheduling parameters jobs $J^S_{i,j}$ and $J^E_{i,j}$ of LET tasks T^S_i and T^E_i, respectively.

for simplicity the activation offset $o^E_{i,j}$ of each job $J^E_{i,j}$ is defined equal to its release time $r^E_{i,j}$. The scheduling attributes are depicted in Figure 4.8.

4.5.1.1 Scheduling of LET Start Jobs

The execution of LET Start jobs is delayed either by the activation block in which they are released, by other activation blocks, or by the execution of other LET Start jobs. The bounds and values of the start time $st^S_{i,j}$ and the end time $et^S_{i,j}$ of every job $J^S_{i,j}$ of each task $T^S_i \in \tau^u_S$ are defined as

$$st^S_{i,j} \in [a^u_s.end, d_{i,j}) \wedge et^S_{i,j} = st^S_{i,j} + wcet^S_i + pt^S_{i,j} \wedge et^S_{i,j} < d_{i,j}. \tag{4.15}$$

The element a^u_s defines the activation block in which job $J^S_{i,j}$ is released. Because job $J^S_{i,j}$ can be preempted by the timer interrupt, which executes to activate other LET Start jobs, the time interval $[st^S_{i,j}, et^S_{i,j}]$ includes the preemption delays $pt^S_{i,j}$. The termination overhead that follows after a job finishes the execution is considered during assignment of the start time of the next executing job.

Algorithm 5 takes as input the sorted set of unique release times of all LET Start jobs U^S, the set of all activation blocks A^u, and the set J^S of (K,V) pair elements, where K is the release time and V is the set of LET Start jobs released at time K. The set of activation blocks A^u considers only the occurrences of the timer interrupt, that handle activation of LET Start and computation jobs, and their respective context-switching times at the start and end of each occurrence. Algorithm 5 generates the schedule of LET Start jobs as follows. For every release time t in the set of unique release times U^S, generate sequentially the schedule of all LET Start jobs that have release time equal to t. The assignment of start times is handled by the lst variable, which indicates the last end time of a job. The assignment of start times is done such that the start of one job corresponds to the end time of the terminate operation of the previously scheduled job. In case the end time of the previously ended job corresponds to the start of an activation block, then the start time of the next schedulable job is assigned to the end

Algorithm 5: Schedule synthesis of LET Start jobs in TTS

```
Input:
$U^S$ – the sorted set of unique release times of all LET Start jobs,
$A^u$ – the set of all activation blocks reserved for scheduling of timer jobs,
$J^S$ – the set of (K,V) pair elements, where K is the release time and V is the set of LET Start jobs released at time K
Output:
$S^S = \{st^S_{i,j} | \forall J^S_{i,j}, \forall T^S_i \in \tau^u_S, i \in [1, n^u], j \in [1, n^u_i]\}$ start times of all LET Start jobs,
$E^S = \{et^S_{i,j} | \forall J^S_{i,j}, \forall T^S_i \in \tau^u_S, i \in [1, n^u], j \in [1, n^u_i]\}$ end times of all LET Start jobs

Function ScheduleStartJobs($U^S$, $A^u$, $J^S$):
    $lst \leftarrow min(U^S)$
    foreach release time $t \in U^S$ do
        $J^S_t \leftarrow J^S[t]$ unique set of jobs released at time $t$
        if $lst \leq t$ then
            if $\exists a^u_r \in A^u$, where $t = a^u_r.start$ then
                $lst \leftarrow a^u_r.end$
        foreach job $J^S_{i,j} \in J^S_t$ do
            $st^S_{i,j} \leftarrow lst$
            $pt^S_{i,j} \leftarrow$ CalculatePreemptionTime ($A^u$, $st^S_{i,j}$, $wcet^S_i$)
            $et^S_{i,j} \leftarrow st^S_{i,j} + wcet^S_i + pt^S_{i,j}$
            $lst \leftarrow et^S_{i,j} + ov_t$
            if $lst \geq d_{i,j} - wcet^E_i - ov_t$ then
                Error: Infeasible schedule.
                return $\emptyset, \emptyset$
            if $\exists a^u_r \in A^u$, where $lst \in [a^u_r.start, a^u_r.end)$ then
                $d \leftarrow a^u_r.end - a^u_r.start$
                if $a^u_r$ starts with $ov_{cs}$ then
                    $d = d - ov_{cs}$
                $a^u_r.start \leftarrow lst$
                $a^u_r.end \leftarrow a^u_r.start + d$
    return $S^E, E^E$
```

time of the activation block. In the first iteration of time t, the start time of the first selected job is assigned to the end time of the activation block containing t.

The preemption delays of every job are calculated by the function *CalculatePreemption-Time* in Line 11, which searches recursively for all activation blocks that overlap with the time interval $[st^S_{i,j}, et^S_{i,j}]$. The time interval $[st^S_{i,j}, et^S_{i,j}]$ is increased in each iteration of the recursive function by the preemption time that is caused by activation blocks found in previous iterations of recursion. The recursive function stops if no activation blocks are found that overlap with the time interval $[st^S_{i,j}, et^S_{i,j}]$ and returns the total duration of the overlapping activation blocks. The end time of the job is then assigned in Line 12 considering the calculated preemption delay. Because the recursive function that calculates the preemption delay of a job is basic in nature, its definition is not provided. Delays of the timer interrupt due to the *suspend interrupt locks* are handled inside the recursive function as follows. The start times of timer interrupt instances, that preempt a communication job, are shifted by a fixed time delay defined as ov_{res}. This means that if an activation block a^u_r preempts a job $J^S_{i,j}$, it can preempt the job only at time $a^u_r.start + ov_{res}$. Therefore, the recursive function changes the start of a^u_r to $a^u_r.start + ov_{res}$. If a shifted interval a^u_r overlaps with another one, then the other interval is also shifted. If the end time of a shifted interval corresponds to the end time of the preempted job, then the interval is shifted further by an amount of ov_t time and if the interval starts with a context-switch, then its duration is reduced by the amount of context-switching time ov_{cs}. The shifting and duration adjustment of overlapping intervals occurs until no overlaps exists among them.

The schedule of LET Start jobs is defined as infeasible if at least one LET Start job cannot be scheduled within the time interval defined by its release time and deadline. The schedule is invalid if the start time of LET Start jobs is greater or equal to the end time of their respective LET End jobs, because it results in insufficient capacity for scheduling computation jobs. The rest of the algorithm inside the *if* check in Line 17 checks if the end of the terminate operation corresponds to the start or lays inside an activation block. In this case, if the activation block starts with a context switch operation, then its duration is reduced by the context-switching time ov_{cs} because this operation is already included at the terminate operation of the job.

4.5.1.2 Scheduling of LET End Jobs

Algorithm 6 synthesizes a non-preemptive schedule of LET End jobs. Their activation offsets are generated such that their execution does not overlap with the execution of any occurrence of the timer interrupt or with the execution of LET Start jobs. The activation offset $o^E_{i,j}$ of each job $J^E_{i,j}$ of each task $T^E_i \in \tau^u_E$ is bounded as

$$(a^u_s.end + wcet_i + wcet^S_i + 2 * ov_t) \leq o^E_{i,j} \leq (d_{i,j} - wcet^E_i - ov_t), \qquad (4.16)$$

where a_s^u represents the activation block in which jobs $J_{i,j}^S$ and $J_{i,j}$ are released. The start time $st_{i,j}^E$ and end time $et_{i,j}^E$ of every job $J_{i,j}^E$ are defined as

$$st_{i,j}^E = a_v^u.end \wedge et_{i,j}^E = st_{i,j}^E + wcet_i^E + ov_t, \tag{4.17}$$

where a_v^u is the activation block in which job $J_{i,j}^E$ is released. Algorithm 6 takes as input the sorted set of unique absolute deadlines U^E of all LET End jobs, the HP duration hp^u, the set of communication blocks cc^u calculated after the schedule of the timer and of LET Start jobs, and the set J^E of (K,V) pair elements, where K is the absolute deadline time and V is the set of LET End jobs with deadline equal to time K.

Algorithm 6 generates the schedule of LET End jobs as follows. For every absolute deadline t in the set of unique absolute deadlines U^E, generate sequentially the schedule of all LET End jobs that have absolute deadline equal to t. The assignment of start times and offsets of LET End jobs is backwards starting from the HP duration because these jobs are scheduled such that their execution happens close to the end of their LET intervals. The algorithm generates the end times first and later the start times and activation offsets. The activation offset represents as well the release time of the timer interrupt job released to activate the LET End job. In the first iteration of time t, the end time of the first selected job is assigned to the HP duration hp^u.

The assignment of end times is handled by the lst variable, which stores the activation offset of the last scheduled job. End times are assigned such that the end of one job corresponds to the activation offset of the previously scheduled job. If the activation offset of the previously assigned job corresponds to the end of an activation block (Line 8), then the end time of the next schedulable job is assigned to the start time of the activation block. If the start time of a job equals the end of a communication block (Line 21), then the activation of the job and the execution of the timer interrupt takes place before the block starts. The *while* loop in Line 13 checks for communication blocks that could overlap with the execution of a LET End job. If such a block is found, then the schedule of this job is planned before the occurrence of the block. Although this approach is not effective in terms of utilizing the HP interval, it can provide a non-preemptive execution of LET End jobs. In harmonic task sets with LET duration equal to the periods, such overlaps do not occur.

The schedule of LET End jobs is defined as infeasible if at least one LET End job cannot be scheduled within the time interval defined by the release time of its corresponding LET Start job and its absolute deadline. The schedule is invalid if the offsets of LET End jobs are smaller or equal to the end time of their respective LET Start jobs. In such a case, there is insufficient capacity for the execution of computation jobs. Validation of the schedule feasibility of computation job is handled by the schedule synthesis algorithm of computation tasks.

Algorithm 6: Schedule synthesis of LET End jobs in TTS

```
Input:
$U^E$ – the sorted set of unique absolute deadline times of all LET End jobs,
$cc^u$ – the set of communication blocks,
$J^E$ – the set of (K,V) pair elements, where K is the absolute deadline and V is
the set of LET End jobs with deadline K, $hp^u$ – the HP duration in core $C_u$
Output:
$S^E = \{st^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1, n^u_i]\}$ start times of all LET End jobs,
$E^E = \{et^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1, n^u_i]\}$ end times of all LET End jobs,
$O^E = \{ot^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1, n^u_i]\}$ offsets of all LET End jobs

Function ScheduleEndJobs($U^E$, $cc^u$, $J^E$, $hp^u$):
    $lst \leftarrow hp^u$
    foreach deadline time $t \in U^E$ do
        $J^E_t \leftarrow J^E[t]$ unique set of jobs that must finish latest at time $t$
        if $lst > t$ then
            $lst \leftarrow t$
        if $\exists cc^u_r \in cc^u$, where $lst \in [cc^u_r.start, cc^u_r.end]$ then
            $lst \leftarrow cc^u_r.start$
        $cJob \leftarrow |J^E_t|$
        foreach job $J^E_{i,j} \in J^E_t$ do
            $cJob \leftarrow cJob - 1$
            while (1) do
                $s \leftarrow lst - ov_t - wcet^E_i$
                if $\exists cc^u_r \in cc^u$, where $cc^u_r.start \leq s < cc^u_r.end$ then
                    $lst \leftarrow cc^u_r.start$
                else
                    break
            $et^E_{i,j} \leftarrow lst - ov_t$
            $st^E_{i,j} \leftarrow et^E_{i,j} - wcet^E_i$
            if $\exists cc^u_r \in cc^u$, where $cc^u_r.end = st^E_{i,j}$ then
                $ot^E_{i,j} \leftarrow cc^u_r.start - ov_a$
            else
                $ot^E_{i,j} \leftarrow st^E_{i,j} - ov_{cs} - ov_a$
            if $cJob \leq 0 \vee \nexists cc^u_r \in cc^u$, where $ot^E_{i,j} = cc^u_r.end$ then
                $ot^E_{i,j} \leftarrow ot^E_{i,j} - ov_{cs}$
            if $ot^E_{i,j} \leq r^S_{i,j} + wcet^S_i + ov_t$ then
                Error: Infeasible schedule.
                return $\emptyset,\emptyset,\emptyset$
            $lst \leftarrow ot^E_{i,j}$
    return $S^E, E^E, O^E$
```

4.5.1.3 Optimization of PTP Overheads

As shown in Chapter 3, the high buffering run-time overheads of PTP occur due to data stability of copy operations caused by the usage of spin-locks and the inter-core communication delays caused by concurrent accesses on shared resources such as bus and memory. TTS offers the possibility to reduce these overheads by enforcing a sequential execution also for tasks running on different cores. Hence, the schedule can be constructed such that execution overlaps of LET communication tasks running on different cores are reduced or fully eliminated. Two straightforward approaches are described to construct the TTS schedule such that PTP overheads are decreased.

In the *first* approach, the schedule of LET communication jobs running on different cores is constructed such that none of these jobs execute in overlapping time intervals. In this case, delays due to concurrent operations and due to spin-locks are eliminated. However, this approach decreases heavily the schedulability and the efficient utilization of core's capacity, i.e., the idle time increases at the boundaries of LET intervals, unless other non-LET tasks execute during this time. Sequential execution of copy operation running on different cores is described as well in [82, 83].

In the *second* approach, the schedule of LET communication tasks is constructed such that only concurrent LET Start and End jobs of different cores and concurrent LET End jobs of different cores execute in non-overlapping time frames. In this case, parallel execution of LET Start jobs of different cores do not cause data stability issues and, hence, spin-locks are not required. However, the inter-core communication overheads still occur. Compared to the first approach, the second one is expected to have better schedulability and higher core utilization, but slightly more buffering overheads. In both cases, finding a feasible schedule of LET tasks is less possible compared to when none of these approaches is applied.

The step-wise TTS schedule synthesis, proposed in this work, enables realization of both aforementioned approaches by combining the strategy and algorithms described in Sections 4.4.2, 4.5.1.1 and 4.5.1.2. In the *first* approach, the schedule of the timer interrupts is constructed for each core separately and the schedule of communication jobs is then performed step-wise, as described in Sections 4.5.1.1 and 4.5.1.2, but treating all tasks as they all run in one core. In the *second* approach, the schedule is generated as follows. In the first step, the schedule of the timer interrupt and of LET Start tasks is generated for each core separately using the approaches described in Sections 4.4.2 and 4.5.1.1. In the second step, communication blocks of each individual core are calculated, which are later merged to one set of communication blocks B^u. In this case, the blocks in B^u represent the time intervals occupied by occurrences of the timer interrupt and LET Start tasks running on all cores. In the final step, the schedule of LET End tasks of all cores is generated at once using the approach described in Section 4.5.1.2 and the set of communication blocks B^u. The schedule construction of LET End tasks is done treating them as they run all in one core. This approach ensures that LET End tasks of any core run not only sequential among themselves, independent on which core they run, but as well in relation to any LET Start task. The described schedule syntheses are applicable only for homogeneous processors.

4.5.2 Scheduling of Computation Tasks

The schedule synthesis algorithm of computation tasks generates *start* and *resume times* of all computation jobs such that they all meet their absolute deadlines under consideration of preemption delays caused by the execution of the timer interrupt and of communication jobs. The schedule of the timer interrupt and of communication jobs is encapsulated within communication blocks, which are treated during schedule synthesis as fixed non-schedulable time intervals that can delay or interrupt the execution of computation jobs. The schedule of computation tasks is constructed via the CP approach. Therefore, the rest of this section describes all variables and constraints proposed to construct their schedule via CP.

Definition 4.5 (Scheduling Parameters): *Let $st_{i,j}$ and $et_{i,j}$ be the start and end times of job $J_{i,j}$ of the computation task $T_i \in \tau^u_{cp}$, respectively. Their values are bounded as*

$$st_{i,j} \in [est_{i,j}, le_{i,j} - wcet_i - ov_t] \wedge et_{i,j} \in [est_{i,j} + wcet_i + ov_t, le_{i,j}], \tag{4.18}$$

where $est_{i,j}$ defines the earliest start time of $J_{i,j}$ and $le_{i,j}$ the latest end time of $J_{i,j}$, and ov_t is the termination overhead associated with completion of $J_{i,j}$.

The value of $est_{i,j}$ is defined by the end time of the communication block cc^u_r in which job $J_{i,j}$ is released, where $r \in \mathbb{N}$. If no such communication block cc^u_r exists, then $est_{i,j}$ equals the release time $r_{i,j}$. The boundaries of $est_{i,j}$ are defines as

$$est_{i,j} = cc^u_r.end \Longleftrightarrow \exists cc^u_r \rightarrow r_{i,j} \in [cc^u_r.start, cc^u_r.end]. \tag{4.19}$$

The value of $le_{i,j}$ is defined by the start time of the communication block cc^u_{r+s}, which contains the absolute deadline $d_{i,j}$ of job $J_{i,j}$, where $r + s \in \mathbb{N}$. If no such block cc^u_{r+s} exists, then $le_{i,j}$ equals the deadline $d_{i,j}$. The boundaries of $le_{i,j}$ are defines as

$$le_{i,j} = cc^u_{r+s}.start \wedge d_{i,j} \in [cc^u_{r+s}.start, cc^u_{r+s}.end]. \tag{4.20}$$

The scheduling parameters of every computation job $J_{i,j}$ of every task $T_i \in \tau^u_{cp}$ are depicted in Figure 4.9. Every job $J_{i,j}$ can be scheduled at any time during the interval $[est_{i,j}, le_{i,j}]$ as long as the end time $et_{i,j}$ is less or equal to the latest end time $le_{i,j}$. The end times of computation jobs are calculated during schedule synthesis to ensure that their complete WCET is scheduled within their LET intervals. Furthermore, the generation of end times during the assignment of start times, avoids the need to use expensive and insufficient schedulability tests, which are typically based on definition

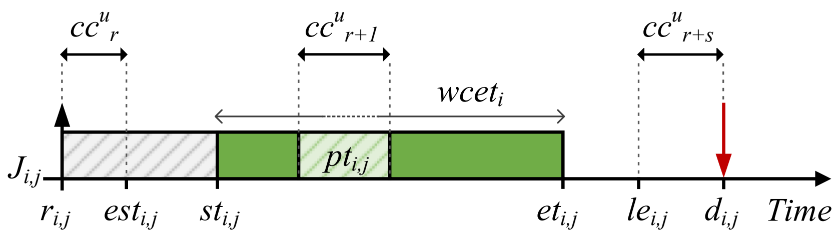


Figure 4.9: TTS scheduling parameters of job $J_{i,j}$ of computation task $T_i \in \tau^u_{cp}$.

of WCRT. Preemption delays caused by communication blocks and other computation jobs are included in the calculation of end times.

The following terminology is used in the remaining sections. For each job $J_{i,j}$ of $T_i \in \tau^u_{cp}$, $atw_{i,j}$ denotes the *active time interval* defined by $[r_{i,j}, d_{i,j}]$ and $etw_{i,j}$ denotes the *execution time interval* defined by $[st_{i,j}, et_{i,j}]$. The active and execution time intervals of each communication block $cc^u_r \in C_u$ corresponds to the interval $[cc^u_r.start, cc^u_r.end]$.

4.5.2.1 Schedule Generation

Computation jobs are allowed to be preempted more than once during their execution as long as they finish their execution within their LET intervals. They can be preempted either by other computation jobs or by communication blocks. A preemption time point of a computation job corresponds to the start time of another job or the beginning of a communication block, and a resume time point corresponds to the end of the preempting job or the end of a communication block. The resume times are constructed based on the end times of preempting jobs and after the schedule is generated. Preemptions of computation jobs are allowed in a way that the execution of preempting jobs or preempting communication blocks is completed within the execution time interval of the preempted jobs. The *preemption delay* of each job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, notated by the variable $pt_{i,j}$, is calculated during schedule synthesis and is considered in the calculation of job's end time $et_{i,j}$.

Several variables are defined to enforce an execution order between computation jobs and to calculate preemption delays. The *preemption* variables are defined to track if computation jobs are preempted by other computation jobs or by communication blocks. In this way, if preemptions occur, then preemption delays are calculated accordingly. The *non-overlapping* variables are created to track the overlapping situation of execution time intervals of computation jobs when a preemption relation does not exists. Hence, if two jobs do not preempt each-other, then their active time intervals do not overlap. The *direct left neighbor* variables are defined to handle context-switching overheads at the start of computation jobs.

The described variables are created for each unique pair of computation jobs that have an overlap of their active time intervals and for each unique pair of computation job and communication block with overlapping active time intervals. Otherwise, if two jobs have no overlapping of their active time intervals, then the aforementioned variables are not created because these jobs cannot influence each-other's execution. Similarly, a communication block cannot preempt a job if its execution does not overlap with the active time interval of the job.

Definition 4.6 (Preemption Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $ip_{i,j}^{k,l}$ indicate the Boolean variable that defines if job $J_{i,j}$ is preempted by job $J_{k,l}$ and $ip_{k,l}^{i,j}$ the Boolean variable that defines if job $J_{k,l}$ is preempted by job $J_{i,j}$. The values of these variables are defined as*

$$ip_{i,j}^{k,l} = \begin{cases} 1, & \textit{if } J_{k,l} \textit{ preempts } J_{i,j} \\ 0, & \textit{otherwise} \end{cases}, \tag{4.21}$$

$$ip_{k,l}^{i,j} = \begin{cases} 1, & \textit{if } J_{i,j} \textit{ preempts } J_{k,l} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.22}$$

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $ip_{i,j}^r$ indicate the Boolean variable that defines if job $J_{i,j}$ is preempted by the communication block cc_r^u. The values of $ip_{i,j}^r$ are defined as

$$ip_{i,j}^{r} = \begin{cases} 1, & \textit{if } cc_r^u \textit{ preempts } J_{i,j} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.23}$$

Communication blocks are non-schedulable entities and are not allowed to be preempted by computation jobs. Therefore, the preemption relation of cc_r^u by $J_{i,j}$ is not defined.

Constraint 4.1 (Preemption Constraints): *For each computation job $J_{i,j}$ of each task $T_i \in \tau_{cp}^u$, the execution order and preemption relations of $J_{i,j}$ with other computation jobs and communication blocks that have overlapping active time intervals with $J_{i,j}$, are defined as*

$$\forall T_k \in \tau_{cp}^u, \forall J_{k,l}, ip_{i,j}^{k,l} = 1 \Longleftrightarrow st_{i,j} < st_{k,l} \wedge et_{k,l} < et_{i,j}, \tag{4.24}$$

$$\forall cc_r^u \in C_u, ip_{i,j}^{r} = 1 \Longleftrightarrow st_{i,j} < cc_r^u.start \wedge cc_r^u.end < et_{i,j}. \tag{4.25}$$

The constraints defined in Constraint 4.1 enforce that if a computation job $J_{i,j}$ of a task $T_i \in \tau_{cp}^u$ is preempted by another computation job or a communication block, then their execution time interval must be included in the interval $[st_{i,j}, et_{i,j}]$ of $J_{i,j}$.

Definition 4.7 (Non-overlapping Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $nov_{i,j}^{k,l}$ be the Boolean variable* *that defines if jobs do not have an overlap of their execution time intervals. The values of $nov_{i,j}^{k,l}$ are defined as*

$$nov_{i,j}^{k,l} = \begin{cases} 1, & \textit{if } J_{k,l} \textit{ and } J_{i,j} \textit{ do not overlap} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.26}$$

An overlap of the execution time intervals of jobs $J_{i,j}$ and $J_{k,l}$ implies a preemption relation between jobs.

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $nov_{i,j}^r$ be the Boolean variable that defines the non-overlapping relation between job $J_{i,j}$ and communication block cc_r^u. The values of $nov_{i,j}^r$ are defined as

$$nov_{i,j}^{r} = \begin{cases} 1, & \textit{if } cc_r^u \textit{ and } J_{i,j} \textit{ do not overlap} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.27}$$

Constraint 4.2 (Non-overlapping Constraints): *For each unique pair of computation jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if jobs $J_{i,j}$ and $J_{k,l}$ do not overlap, then their execution must not overlap, defined as*

$$nov_{i,j}^{k,l} = 1 \Longleftrightarrow etw_{k,l} \cap etw_{i,j} = \emptyset, \tag{4.28}$$

For each unique pair of computation job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, if job $J_{i,j}$ and communication block cc_r^u do not overlap, then the execution time interval $etw_{i,j}$ of $J_{i,j}$ and the time interval $[cc_r^u.start, cc_r^u.end]$ of cc_r^u must not overlap, defined as

$$nov_{i,j}^{r} = 1 \Longleftrightarrow [cc_r^u.start, cc_r^u.end] \cap etw_{i,j} = \emptyset. \tag{4.29}$$

In TTS, the execution of concurrently active jobs does not have to be strictly sequential as in FPS. This means that after a running job terminates at time *t*, it is not mandatory that another ready computation job starts its execution at time *t*. The schedule is also correct when the start time of a job takes place after a considerable amount of time has elapsed after the termination of another job. In between these times the *idle task* can execute. This offers the opportunity to utilize the HP interval in a more balanced way, rather than causing peak loads close to the activation times of jobs. Therefore, the overlapping between jobs can be minimized and, hence, the preemption overheads are reduced. The proposed synthesis algorithm does not enforce restrictions on start times of jobs, besides that they must start and finish their execution within their LET intervals. Constraint 4.3 defines the calculation of the end time of computation jobs.

Constraint 4.3 (End Time and Preemption Delay): *The end time $et_{i,j}$ of each job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$ is defined as*

$$et_{i,j} = st_{i,j} + wcet_i + pt_{i,j} + ov_t + hcs_{i,j} * ov_{cs}, \tag{4.30}$$

where $pt_{i,j}$ represents the preemption delay, calculated as

$$pt_{i,j} = \sum_{\forall cc^u_r \in C_u} (d^u_r * ip^r_{i,j}) + \sum_{\forall J_{k,l}, \forall T_k \in \tau^u_{cp}} ((hcs_{k,l} * ov_{cs} + wcet_k + ov_t) * ip^{k,l}_{i,j}), \tag{4.31}$$

and $hcs_{i,j}$ defines if $J_{i,j}$ needs a context-switch before its start time.

As shown in Equation (4.30), the run-time of the terminate operation ov_t is included in the calculation of the end time of each computation job. In case a computation job starts its execution directly after termination of another computation job or of a communication block, then the context-switching takes place within the terminate operation. Otherwise, if a job starts some time later or if it preempts another job, including the *idle job*, then a special handling is needed to include the context-switching operation before the start of the job's execution. For this reason, a dedicated variable is defined for each computation job to track during schedule synthesis if a job starts the execution at the end time of another job or of a communication block. Hence, for each job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, the Boolean variable $hcs_{i,j}$, referred as *has-context-switch*, defines if job $J_{i,j}$ needs a context switch before the start of its execution, which takes the value "1" only if the job doesn't start directly at the end time of another job or of a communication block. As shown in Equation (4.31), the *has-context-switch* variable not only enforces a time distance of ov_{cs} between jobs, but also provides the possibility to include the context switch overhead ov_{cs} in the preemption delays of each computation job. To calculate the value of $hcs_{i,j}$, the *direct left neighbor* variables and constraints are defined in Definition 4.8 and Constraint 4.4.

Definition 4.8 (Neighbor Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $idln^{k,l}_{i,j}$ indicate the Boolean variable that defines if job $J_{k,l}$ is the direct left neighbor of $J_{i,j}$ and $idln^{i,j}_{k,l}$ the Boolean variable that defines if job $J_{i,j}$ is the direct left neighbor of $J_{k,l}$. The values of these variables are defined as*

$$idln^{k,l}_{i,j} = \begin{cases} 1, & \textit{if } J_{k,l} \textit{ is direct left neighbor of } J_{i,j} \\ 0, & \textit{otherwise} \end{cases}, \tag{4.32}$$

$$idln^{i,j}_{k,l} = \begin{cases} 1, & \textit{if } J_{i,j} \textit{ is direct left neighbor of } J_{k,l} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.33}$$

Jobs $J_{i,j}$ and $J_{k,l}$ cannot be direct left neighbors of each-other at the same time. This is constrained as

$$idln_{i,j}^{k,l} + idln_{k,l}^{i,j} = \begin{cases} 1, & \text{either } J_{i,j} \text{ or } J_{k,l} \text{ is direct left neighbor} \\ 0, & \text{neither } J_{i,j} \text{ nor } J_{k,l} \text{ is direct left neighbor} \end{cases}. \tag{4.34}$$

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $idln_{i,j}^r$ indicate the Boolean variable that defines if communication block cc_r^u is the direct left neighbor of job $J_{i,j}$. The values of $idln_{i,j}^r$ are defined as

$$idln_{i,j}^r = \begin{cases} 1, & \text{if } cc_r^u \text{ is direct left neighbor of } J_{i,j} \\ 0, & \text{otherwise} \end{cases}. \tag{4.35}$$

Because the communication block cc_r^u cannot be rescheduled and the context-switching run-time is included in the time duration d_r^u, a variable $idln_r^{i,j}$ to describe the opposite of $idln_{i,j}^r$ is not needed.

Constraint 4.4 (Neighbor Constraints): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if job $J_{k,l}$ is a direct left neighbor of $J_{i,j}$ then the start time $st_{i,j}$ of $J_{i,j}$ equals the end time $et_{k,l}$ of $J_{k,l}$, otherwise these times must be different. This constraint is defined as*

$$idln_{i,j}^{k,l} = \begin{cases} 1, & st_{i,j} = et_{k,l} \\ 0, & st_{i,j} \neq et_{k,l} \end{cases}. \tag{4.36}$$

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, if communication block cc_r^u is a direct left neighbor of $J_{i,j}$, then the start time $st_{i,j}$ of $J_{i,j}$ equals the end time $cc_r^u.end$ of cc_r^u, otherwise these times must be different. This constraint is defined as

$$idln_{i,j}^r = \begin{cases} 1, & cc_r^u.end = st_{i,j} \\ 0, & cc_r^u.end \neq st_{i,j} \end{cases}. \tag{4.37}$$

A computation job does not necessarily have a direct left neighbor. At most one computation job or communication block is a direct left neighbor of a computation job. A computation job has a context-switch before the start only if it doesn't have a direct left neighbor. Therefore, the value of $hcs_{i,j}$ of each computation jobs $J_{i,j}$ of each task $T_i \in \tau_{cp}^u$ is assigned to zero if the sum of all job's direct left neighbor variables is one. Otherwise, the job needs a context-switch before it starts executing. The Boolean type of $hcs_{i,j}$ enforces that at most one direct left neighbor can exist for $J_{i,j}$. The value of $hcs_{i,j}$ is calculated based on the sum of direct left neighbors variables of $J_{i,j}$ as shown in Constraint 4.5.

Constraint 4.5 (Context-Switch Constraints): *For each computation jobs $J_{i,j}$ of each task $T_i \in \tau_{cp}^u$, the value of $hcs_{i,j}$ is calculated considering direct left neighbor variables of $J_{i,j}$ as*

$$hcs_{i,j} = \sum_{\forall J_{k,l}, T_k \in \tau_{cp}^u} idln_{i,j}^{k,l} + \sum_{\forall cc_r^u \in C_u} idln_{i,j}^{r}. \tag{4.38}$$

The relation of *preemption*, *non-overlapping*, and *direct left neighbor* variables is given in Constraint 4.6.

Constraint 4.6 (Exclusive Disjunction Constraints): *For each unique pair of computation jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if job $J_{k,l}$ preempts job $J_{i,j}$, then the opposite is not allowed. If jobs do not preempt each-other, then they must either not overlap or a direct left neighbor relation must exist among them. These constraints are defined as*

$$ip_{i,j}^{k,l} + ip_{k,l}^{i,j} + nov_{i,j}^{k,l} + idln_{i,j}^{k,l} + idln_{k,l}^{i,j} = 1. \tag{4.39}$$

For each unique pair of computation job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, if cc_r^u preempts job $J_{i,j}$, then their execution overlaps. Otherwise, cc_r^u is a direct left neighbor of job $J_{i,j}$. This constraint is defined as

$$ip_{i,j}^{r} + nov_{i,j}^{r} + idln_{i,j}^{r} = 1. \tag{4.40}$$

Figure 4.10 shows an example of a computation schedule generated by the described approach. This example is only one of several possible solutions that the proposed schedule synthesis can provide. As the figure shows, job $J_{i,j}$ is preempted by all other jobs and by the communication block cc_{r+1}^u. Job $J_{k,l}$ requires a context-switch operation at its start time because it doesn't have a direct left neighbor and it must handle the context change between the preempted job $J_{i,j}$ and the preempting job $J_{k,l}$. Similarly, job $J_{p,q}$ requires a context-switch operation because it preempts job $J_{k,l}$ and has no direct left neighbor as well. Because job $J_{h,f}$ starts the execution directly after job $J_{k,l}$ terminates, it doesn't need a context-switch operation at its start time. The same applies for job $J_{i,j}$, which starts directly after cc_r^u. This is possible because the context-switch is included in the duration of cc_r^u and in the termination operation of the job $J_{k,l}$. The context change from the termination of $J_{h,f}$ and resume of $J_{i,j}$ is as well included in the termination operation of $J_{h,f}$. A hierarchical preemption of job $J_{i,j}$ by jobs $J_{k,l}$ and $J_{p,q}$ is allowed by the synthesis algorithm. Although hierarchical preemptions can increase the stack memory size, they are unavoidable for certain applications and are necessary to ensure a feasible schedule.

Terminate and context-switching operations must occur atomically, i.e., not preempted by any job. To allow preemption only during the actual execution time of jobs, for each job $J_{i,j}$ of each task $T_i \in \tau_{cp}^u$ the actual start time $acst_{i,j}$ and the actual end time

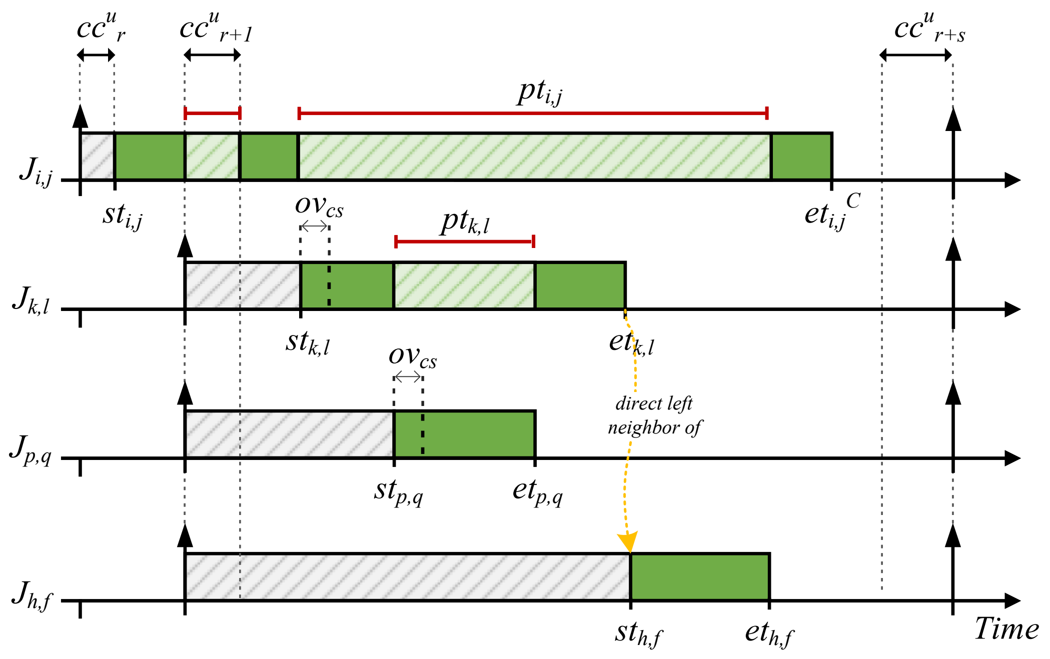


Figure 4.10: Example of TTS synthesis considering preemption overheads. Job $J_{i,j}$ has as direct left neighbor the communication block cc^u_r. Jobs $J_{k,l}$ and $J_{p,q}$ do not have direct left neighbors and they require a context-switch operation before their start times. Job $J_{h,f}$ has $J_{k,l}$ as direct left neighbor and it does not require a context-switch at its start time. Job $J_{i,j}$ is preempted by all the other jobs and by communication block cc^u_{r+1}. The dark green boxes indicate the execution of jobs. The light green and gray boxes indicate the preemption and start delays, respectively.

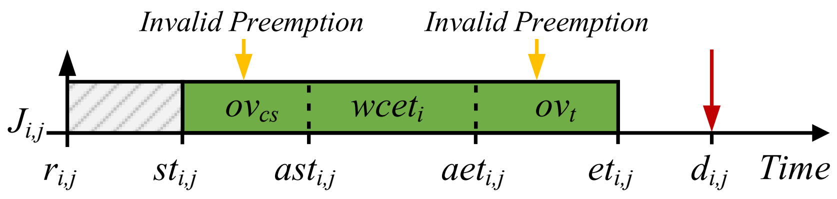


Figure 4.11: Invalid preemption points for computation jobs. The dark green and gray box indicate the execution time and the start delay of job $J_{i,j}$, respectively.

$acet_{i,j}$ are defined to determine non-preemptive regions in $etw_{i,j}$, defined as

$$acst_{i,j} = \begin{cases} st_{i,j} + ov_{cs}, & hcs_{i,j} = 1 \\ st_{i,j}, & \text{otherwise} \end{cases}, \tag{4.41}$$

$$acet_{i,j} = et_{i,j} - ov_t. \tag{4.42}$$

These regions are intervals $[st_{i,j}, ast_{i,j}]$ and $[aet_{i,j}, et_{i,j}]$. Their abstraction is shown in Figure 4.11. To enforce valid preemption points, the *disable preemption* constraints are defined in Constraint 4.7, which are redundant to the ones given in Constraint 4.1.

Constraint 4.7 (Disable Preemption): *For each computation jobs $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, the position of preemption by other computation jobs and communication blocks that have an overlap of their active time interval with the one of $J_{i,j}$ is defined as*

$$\forall T_k \in \tau^u_{cp}, \forall J_{k,l}, ip^{k,l}_{i,j} = 1 \Longleftrightarrow acst_{i,j} < st_{k,l} \wedge et_{k,l} < acet_{i,j}, \tag{4.43}$$

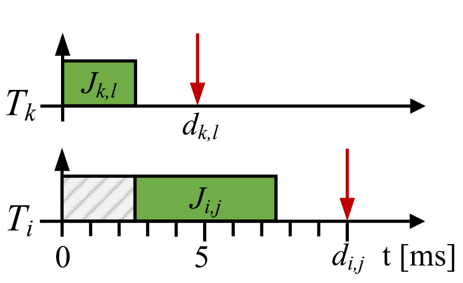
$$\forall cc^u_r \in C_u, ip^r_{i,j} = 1 \Longleftrightarrow acst_{i,j} < cc^u_r.start \wedge cc^u_r.end < aet_{i,j}. \tag{4.44}$$

4.5.2.2 Schedule Optimization

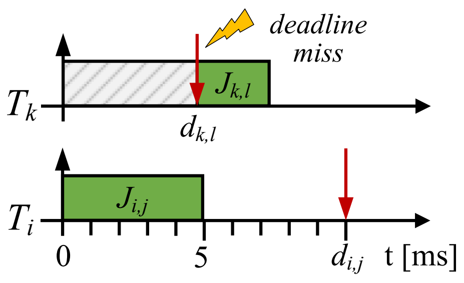
In this work, the preemption costs are optimized by minimizing the function $f^u_r(3)$, defined in Equation (4.45), which instructs the solver to produce solutions with minimal number of preemptions.

$$f^u_r(3) = \sum_{\forall J_{i,j}, \forall T_i \in \tau^u_{cp}} \Big(\sum_{\forall J_{k,j}, \forall T_k \in \tau^u_{cp}} ip^{k,l}_{i,j} + \sum_{\forall cc^u_r \in C_u} ip^r_{i,j} \Big) \tag{4.45}$$

This optimization applies only to preemptions of computation tasks. The number of preemptions for communication tasks is reduced by scheduling them non-preemptively when possible and with higher urgency than computation tasks. If the solver is not configured to optimize the schedule by means of function $f^u_r(3)$, then it provides all feasible solutions that *satisfy* the constraints described in this section.



(a) Job $J_{k,l}$ scheduled before $J_{i,j}$.



(b) Job $J_{i,j}$ scheduled before $J_{k,l}$

Figure 4.12: Scheduling of LET End tasks with equal priorities. The dark green and gray boxes indicate the execution time and the start delay of jobs, respectively.

4.6 Fixed-Priority Schedule Synthesis

The FPS scheduling is designed to efficiently handle dynamic behavior of a system at run-time. In FPS, scheduling decisions are taken *online* based on *static* priorities of tasks. The OS maintains a *ready queue* to store all active jobs. The FPS scheduler selects from the queue jobs of higher-priority tasks before the low-priority ones. The arrival of a high-priority task causes a call to the scheduler, which interrupts the execution of a running low-priority job and loads for execution the arrived higher-priority task. If the ready queue is empty, then the *idle job* is loaded for execution.

In FPS, priorities define the execution flow of tasks. Therefore, they must be defined such that all high- and low-priority tasks finish their execution before their deadlines, i.e., within their LET intervals and data exchanges occur according to LET semantics. To ensure that LET semantics are fulfilled at design and target execution phase, the execution flow of tasks must be deterministic, i.e., predictable and reproducible. Because scheduling decisions in FPS are taken online, for certain priority configurations the execution flow of tasks defined at design can result different to the one occurring on target. If multiple tasks share the same priority, then the FPS scheduler selects for execution any of their coinciding active jobs. The determinism problem arises when jobs are released at the same time instant but have different absolute deadlines. Although in certain OS implementations the *First In - First Out (FIFO)* mechanism is used by the scheduler to select from the ready queue first active jobs of the same priority, in case jobs share the same release time the FIFO mechanism is not enough to provide a predictable FPS schedule.

An example of two tasks sharing equal priorities is shown in Figure 4.12. Jobs $J_{i,j}$ and $J_{k,l}$ are released at the same time, but have different absolute deadlines. In this case, only one job for each task and only relative deadlines are shown in Figure 4.12. If the scheduler selects first job $J_{k,l}$ for execution, then the deadlines are fulfilled by both jobs $J_{i,j}$ and $J_{k,l}$, as shown in Figure 4.12a. Otherwise, as shown in Figure 4.12b, if the scheduler selects job $J_{i,j}$ first for execution, then job $J_{k,l}$ misses the deadline. In case tasks have different priorities and task T_k has higher priority than T_i, then the execution order between jobs $J_{i,j}$ and $J_{k,l}$ is identical at design and target execution time, which results in the same execution order as in Figure 4.12a. To avoid this situation and to ensure task execution flow determinism, the proposed priority synthesis approach assigns unique priorities to all LET tasks.

The *schedule generation problem* for FPS refers to the assignment of priorities to LET tasks, such that *timing*, *communication*, and *performance* requirements are fulfilled. In this work, task priorities are assigned and their schedule is validated under consideration of the timer interrupt and preemption overheads. The ordering between communication and computation tasks is guaranteed, as constrained by LET semantics, by defining priorities of LET Start tasks higher than of LET End tasks, and the priorities of computation tasks lower than of LET Start and End tasks. Furthermore, to reduce preemption delays of LET communication jobs caused by other non-LET high-priority tasks, their execution is defined as *non-preemptive*.

The *priority ranges* of LET tasks are defined for each core $C_u \in C$ considering the number of computation tasks $n^u \in \mathbb{N}$, as shown in Definition 4.9. The number of tasks in PTP and SBP is $3 * n^u$ and $n^u + 1$, respectively.

Definition 4.9 (FPS Priorities of LET Tasks): *Let π_i^S, π_i^E, and π_i denote the priorities of the LET Start $T_i^S \in \tau_S^u$, LET End $T_i^E \in \tau_E^u$, and computation task $T_i \in \tau_{cp}^u$, respectively. Their values are bounded as*

$$\pi_i^S \in \{3 * n^u,..,2 * n^u + 1\}, \tag{4.46}$$

$$\pi_i^E \in \{2 * n^u,..,n^u + 1\}, \tag{4.47}$$

$$\pi_i \in \{n^u,..,1\}, \tag{4.48}$$

where the value $3 * n^u$ defines the highest priority and 1 the lowest.

The β_i^S, β_i^E, and β_i define the preemptability of T_i^S, T_i^E, and T_i, respectively. Values of β_i^S and β_i^E are assigned to "non" to define a non-preemptive execution of LET Start T_i^S and LET End T_i^E tasks. Values of β_i are assigned to "full" to define that jobs of computation task T_i are preempted at any time during their execution.

To improve the performance, the schedule of LET tasks is constructed and verified in two different steps. The schedule synthesis algorithm described in Section 4.6.1 generates the schedule of communication tasks for PTP. This algorithm is also used in SBP to construct the schedule of IR^{init}, which is treated during synthesis as a LET Start task with the highest priority. A generic mathematical formulation is described in Section 4.6.2 as a CSP problem to generate the schedule of computation tasks. An approach to validate the schedulability of computation tasks during the synthesis of priorities is proposed. This approach replaces formal schedulability tests, which are complex, have high solving time, and are applicable under strict assumptions.

This algorithm generates the schedule of computation tasks when PTP, SBP, or a combination of both protocols is used in the same application.

4.6.1 Scheduling of Communication Tasks

Priorities of LET Start and End tasks are assigned using the DM heuristic. In DM, tasks with the earliest deadline get the highest priority. LET Start jobs with coinciding release and execution must execute, independent of their priority order, all before the start of their respective computation jobs. The order of execution of LET End jobs is essential and jobs with the earliest deadlines must execute first. Activation offsets of LET End jobs are assigned during priority assignment such that these jobs are not released at the same time or in coinciding execution time intervals and they complete their execution within their deadlines. Therefore, DM heuristic and the priority ranges defined in Definition 4.9 are a sufficient approach to assign priorities of LET communication tasks while satisfying the deadlines and the Requirement 4.1.

The methodology of assigning priorities and validating the schedule of LET communication tasks is described by three main steps.

St 1 Priorities of LET Start and End tasks are assigned using the DM heuristic and the priority ranges defined in Definition 4.9.

St 2 As in TTS, activation blocks are calculated to isolate the execution of the timer jobs that occur for activating LET Start and computation tasks. They are used during construction of the schedule trace, i.e., execution of LET Start jobs according to the assigned priorities, such that start and preemption delays caused by the timer's execution are added during the assignment of start and end times of all LET Start jobs released within one HP interval. After the schedule trace of LET Start jobs is constructed and validated, their execution and activation is enclosed in communication blocks, which are used during schedule generation of LET End jobs. Their purpose is to not only validate the schedule of LET End jobs, but to also calculate possible time intervals in which those jobs can activate, execute, and simultaneously fulfill their deadlines.

St 3 Activation offsets of LET End jobs are generated in a way that these jobs are never preempted or delayed by the occurrence of any timer job or by other communication jobs. The schedule trace is constructed and validated considering the assigned priorities and offsets.

Because the execution order of LET Start jobs is straightforward when DM is applied, the generation of the execution trace of these jobs is not described in this section.

The generation of the schedule trace and activation offsets of LET End jobs is given in Algorithm 7. Scheduling attributes of LET End tasks shown in Section 4.5.1 are also the attributes solved by Algorithm 7. Hence, the $st^E_{i,j}$, $et^E_{i,j}$, and $ot^E_{i,j}$ define the start time, the end time, and the activation offset of the LET End job $J^E_{i,j}$, respectively. Although the release of a job does not occur exactly at the time of its activation offset, for simplicity the activation offset $o^E_{i,j}$ of each job $J^E_{i,j}$ is defined equal to its release time $r^E_{i,j}$. The start and end times of LET End jobs are calculated during their offset assignment to ensure that these jobs execute close to the end of their LET intervals considering the priority ordering defined by DM and after the timer interrupt occurrences that are released to activate these job have completed execution.

Algorithm 7 takes as input the sorted set of unique absolute deadlines of all LET End jobs U^E, the HP duration hp^u of core C_u, the set of communication blocks cc^u calculated after the schedule of the timer and of LET Start jobs, and the set J^E of (K,V) pair elements, where K is the absolute deadline time and V is the set of LET End jobs with deadline equal to time K. Algorithm 7 assigns activation offsets of LET End jobs as follows. A queue of *active* jobs J^E_a stores the set of unscheduled LET End jobs that have absolute deadlines less or equal to a time lst. The jobs in J^E_a are ordered by ascending priorities and the queue is updated in several positions within the algorithm. The assignment of activation offsets is handled by the lst variable, which stores the activation offset of the last scheduled job. End times are assigned such that the end of one job corresponds to the activation offset of the previously scheduled job of a lower priority. If the activation offset of the previously assigned job corresponds to the end of a communication block, then the end time of the next schedulable job is assigned to the start time of the communication block. In the first step, lst is assigned to the HP duration hp^u and the set of active jobs is filled with the set of first jobs that have absolute deadlines less or equal to hp^u. Jobs of the queue J^E_a are scheduled within the *while* loop in Line 6, which executes as long as there is at least one job to be scheduled. The algorithm selects the job with the lowest priority to schedule from the queue (in Line 7), and after the job is scheduled, it is removed from J^E_a (in Line 27).
Before the job is scheduled, the lst is checked if it overlaps with any communication block (in Line 8) and if such a block is found then its start time, stored in variable $nlst$, is used as a starting point to schedule the next job. Hence, if such block is found then the $nlst \neq lst$. In this case, the job is not scheduled but the queue J^E_a is updated with all possible jobs that have an absolute deadline in the interval $[nlst, lst]$ and the lst is updated with $nlst$. The scheduling of the jobs takes place in the next iteration of the loop because the newly added jobs in J^E_a could have lower priority than the selected job $J^E_{i,j}$. Note that the priorities of tasks are assigned using DM based on the relative deadline of the task and not on the absolute deadline of the job. The planning of a job $J^E_{i,j}$ is scheduled from Line 14 to Line 26. If the start time $st^E_{i,j}$ corresponds to the end of a communication block cc^u_r, then the execution of the timer interrupt is planed before the start of cc^u_r. Otherwise, its execution is planned between $ot^E_{i,j}$ and the $st^E_{i,j}$. The *if* condition in Line 16 does not consider in the $ot^E_{i,j}$ assignment the context-switch operation at the beginning of the timer interrupt because if other jobs are scheduled at this time, then the context-switching is handled in the ov_t operation. This operation is

Algorithm 7: Part 1. Schedule synthesis of LET End jobs in FPS

```
Input:
$U^E$ – the sorted set of unique absolute deadline times of all LET End jobs,
$cc^u$ – the set of communication blocks, $hp^u$ – the HP duration in core $C_u$,
$J^E$ – the set of (K,V) pair elements, where K is the absolute deadline and V is
the set of LET End jobs with deadline K.
Output:
$S^E = \{st^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1,n^u_i]\}$ start times of all LET End jobs,
$E^E = \{et^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1,n^u_i]\}$ end times of all LET End jobs,
$O^E = \{ot^E_{i,j} | \forall J^E_{i,j}, \forall T^E_i \in \tau^u_E, i \in [1, n^u], j \in [1,n^u_i]\}$ offsets of all LET End jobs
1
2  Function ScheduleEndJobs($U^E$, $cc^u$, $hp^u$, $J^E$):
3      $lst \leftarrow hp$
4      $J^E_a \leftarrow \{\}$ // The set of priority-ordered active jobs
5      $lst \leftarrow$ UpdateActiveJobs($U^E, J^E, J^E_a, lst$)
6      while ($|J^E_a| > 0$) do
7          $J^E_{i,j} \leftarrow J^E_a[0]$
8          $nlst \leftarrow$ NonoverlappingLst($J^E_{i,j}, cc^u, lst$)
9          if $lst \neq nlst$ then
10             foreach deadline $t \in U^E$, where $nlst \leq t < lst$ do
11                 $J^E_a \leftarrow J^E_a \cup J^E[t]$
12             $lst \leftarrow nlst$
13         else
14             $et^E_{i,j} \leftarrow lst - ov_t$
15             $st^E_{i,j} \leftarrow et^E_{i,j} - wcet^E_i$
16             if $\exists cc^u_r \in cc^u$, where $cc^u_r.end = st^E_{i,j}$ then
17                 $ot^E_{i,j} \leftarrow cc^u_r.start - ov_a$
18             else
19                 $ot^E_{i,j} \leftarrow st^E_{i,j} - ov_{cs} - ov_a$
20             foreach deadline $t \in U^E$, where $ot^E_{i,j} \leq t \leq st^E_{i,j}$ do
21                 $J^E_a \leftarrow J^E_a \cup J^E[t]$
22             if $\nexists J^E_{k,l} \in J^E_a$ with $d_{k,l} \geq ot^E_{i,j} \vee \nexists cc^u_r \in cc^u$ with $cc^u_r.end = ot^E_{i,j}$ then
23                 $ot^E_{i,j} \leftarrow ot^E_{i,j} - ov_{cs}$
24             if $ot^E_{i,j} \leq r^S_{i,j} + wcet^S_i + wcet_i + 2 * ov_t + 2 * ov_a$ then
                   // Error: Infeasible schedule
25                 return ∅,∅,∅
26             $lst \leftarrow ot^E_{i,j}$
27             $J^E_a \leftarrow J^E_a \setminus J^E_{i,j}$
28         if $|J^E_a| = 0 \wedge lst > 0$ then
29             $lst \leftarrow$ UpdateActiveJobs($U^E, J^E, J^E_a, lst$)
30     return $S^E, E^E, O^E$
31
```

Algorithm 8: Part 2. Schedule synthesis of LET End jobs in FPS

```
Function UpdateActiveJobs($U^E$, $J^E$, $J^E_a$, $lst$):
    foreach deadline $t \in U^E$ do
        if $t \leq lst$ then
            $J^E_a \leftarrow J^E_a \cup J^E[t]$
            return $t$

Function NonoverlappingLst($J^E_{i,j}$, $cc^u$, $lst$):
    $nlst \leftarrow lst$
    while (1) do
        $s \leftarrow nlst - ov_t - wcet^E_i$
        if $\exists cc^u_r \in cc^u$, where $cc^u_r.start \leq s < cc^u_r.end$ then
            $nlst \leftarrow cc^u_r.start$
        else
            break
    return $nlst$
```

included in the calculation of $ot^E_{i,j}$ in Line 22 to handle the switch between the idle job and the timer interrupt, only if the queue J^E_a does not contain an unscheduled job with absolute deadline greater or equal to the $ot^E_{i,j}$ or if it does not exist a communication block that overlaps with $ot^E_{i,j}$. Before this operation takes place, the queue is updated in Line 20 with jobs that have deadlines in the interval $[st^E_{i,j}, ot^E_{i,j}]$. The queue J^E_a is also updated in Line 28 with the next first jobs with deadline less or equal to lst. If the queue is empty, then the algorithm terminates.

Sequential execution of communication tasks executed on different cores to reduce communication overheads cannot be guaranteed by FPS because each core has its own FPS scheduler that takes scheduling decisions independently of the schedulers on the other cores. In FPS, to avoid parallel execution of LET communication tasks of different cores, their activation offsets must be defined such that they are active and finish their execution in non-overlapping time intervals. However, in case of unpredictable activation jitters, FPS cannot guarantee non-overlapping execution of these tasks. Therefore, to avoid unpredictable data consistency situations, it is commonly recommended to use locking mechanisms in case of FPS.

4.6.2 Scheduling of Computation Tasks

The FPS schedule synthesis of computation tasks is constructed via *constraint programming* approach. The proposed algorithm generates *priorities* of these tasks such that they meet their deadlines under consideration of start and preemption delays. Unique priorities are assigned to computation tasks of each core $C_u \in C$ by using the *all different* constraint as

$$alldifferent(\{\pi_i | \forall T_i \in \tau^u_{cp}\}). \tag{4.49}$$

A priority assignment is valid if the schedule derived by these priorities and the FPS semantics is *feasible*. The feasibility of the schedule is validated during constraints solving time using the following approach, which is relevant for PTP, SBP, and a combination of both protocols. Hence, the FPS schedule is constructed and validated during priority assignment, i.e., in the constraint solver, by calculating the start and finish time of every job released in the HP interval considering start and preemptions delays caused by high-priority tasks, terminate operation of low-priority tasks, and communication blocks. The final set of unique communication blocks is constructed after the schedule of communication blocks is generated. These blocks are treated during synthesis of computation jobs as scheduled intervals of high-priority that can delay or interrupt the execution of computation jobs.

Synthesizing the execution of jobs during constraint solving is highly challenging because, contrary to TTS, in FPS jobs must follow the fixed execution order that is defined by the assigned priorities and semantics of FPS. Therefore, different variables and constraints are defined and described throughout this section. To track the execution of jobs released in the HP interval, variables such as the *activation time*, the *start time*, and the *end time* of each job are defined. By calculating the end times of computation jobs during priority assignment, the usage of expensive schedulability tests and insufficient classical response time analysis such as calculation of WCRT are avoided. In the proposed approach, the actual response time of each job is calculated as the time between the release and end time of the job. The formalization described in this section is published in [15].

Definition 4.10 (Scheduling Parameters): *Let $st_{i,j}$ and $et_{i,j}$ be the start and end times of job $J_{i,j}$ of task T_i, respectively. Their values are bounded as*

$$st_{i,j} \in [est_{i,j}, le_{i,j} - wcet_i - ov_t] \wedge et_{i,j} \in [est_{i,j} + wcet_i + ov_t, le_{i,j}], \tag{4.50}$$

where $est_{i,j}$ defines the earliest start time of $J_{i,j}$, $le_{i,j}$ defines the latest end time of $J_{i,j}$, and ov_t is the termination overhead associated with the completion of job $J_{i,j}$.

The value of $est_{i,j}$ is defined by the end time of the communication block cc^u_r in which the job $J_{i,j}$ is released. If no such communication block cc^u_r exists, then $est_{i,j}$ equals the release time $r_{i,j}$. The boundaries of $est_{i,j}$ are defines as

$$est_{i,j} = cc^u_r.end \Longleftrightarrow \exists cc^u_r \in C_u \rightarrow r_{i,j} \in [cc^u_r.start, cc^u_r.end]. \tag{4.51}$$

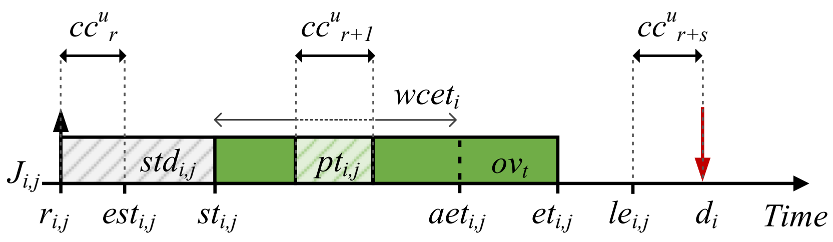


Figure 4.13: Fixed-priority schedule verification parameters of job $J_{i,j}$ of computation task T_i. The time difference between the release time $r_{i,j}$ and the start time $st_{i,j}$ defines the start delay $std_{i,j}$, depicted by the gray box. The time duration of communication block cc^u_{r+1} defines the preemption delay $pt_{i,j}$, marked by the light green box. The dark green box indicates the execution of $J_{i,j}$. Reprinted from [15].

The value of $le_{i,j}$ is defined by the start time of the communication block cc^u_{r+s} in which the respective LET End job $J^E_{i,j}$ of $J_{i,j}$ is contained. Note that the absolute deadline $d_{i,j}$ of job $J_{i,j}$ is either contained in cc^u_{r+s} or occurs some time later. This depends on the schedule synthesis of the communication jobs. If no such communication block cc^u_{r+s} exists, then $le_{i,j}$ equals the deadline $d_{i,j}$. The boundaries of $le_{i,j}$ are defines as

$$le_{i,j} = cc^u_{r+s}.start. \tag{4.52}$$

The earliest start and latest end times are fixed static values and are only used to define the boundaries of start and end time variables. While the release time $r_{i,j}$ of every job $J_{i,j}$ is fixed, the start $st_{i,j}$ and end time $et_{i,j}$ vary and depend on the execution of communication blocks and other computation jobs. The start time $st_{i,j}$ is necessary to track the actual start of the job considering start delays caused by the occurrence of communication blocks, by the execution of higher-priority jobs, and by the execution of the terminate operation of low-priority jobs. Start delays coming from the timer interrupt and high-priority communication jobs are encapsulated in the communication blocks and are included in the calculation of start delays in the same way as for computation jobs. The end time $et_{i,j}$ defines the time that job $J_{i,j}$ of task T_i terminates under the assigned priority π_i. It is defined to validate if job $J_{i,j}$ finishes its execution before its absolute deadline and is calculated considering the start time $st_{i,j}$ and the preemption delay $pt_{i,j}$. The start time $st_{i,j}$ and end time $et_{i,j}$ of every job $J_{i,j}$ are calculated in Constraint 4.8.

Constraint 4.8 (Start and End times): *The actual start time $st_{i,j}$ of each computation job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$ is defined as*

$$st_{i,j} = r_{i,j} + std_{i,j}, \tag{4.53}$$

where the variable $std_{i,j}$ defines the start delay of $J_{i,j}$. The end time of $J_{i,j}$ is defined as

$$et_{i,j} = st_{i,j} + wcet_i + ov_t + pt_{i,j}, \tag{4.54}$$

where $pt_{i,j}$ is the preemption delay, ov_t is the run-time of the terminate operation, and $wcet_i$ is the WCET of task T_i. The $aet_{i,j}$ specifies the start time of the terminate operation of $J_{i,j}$.

The scheduling attributes of each job $J_{i,j}$ of every task T_i are depicted in Figure 4.13. Each job $J_{i,j}$ can be scheduled at any time during the interval $[est_{i,j}, le_{i,j}]$ as long as the end time $et_{i,j}$ is less or equal to the latest end time $le_{i,j}$. In FPS, start delays must be explicitly calculated for all computation jobs because the execution order between active jobs must follow the FPS semantics, which define that if a job is active at a given time then it is executed as soon as it has the highest priority among active jobs.

The terminate operation run-time ov_t is included in the calculation of the end time of each job. Because the terminate operation includes the context-switching operation, the time distance between a terminating job and the next starting job does not require a special handling. In FPS, different to TTS, the tracking of *left* neighborhood relations between computation jobs is not required to track the context-switch operation at the start of a job, because the execution of jobs is strictly sequential, as defined by priorities, and computation jobs start either after the termination of another computation job or after the end of a communication block, which corresponds to the end of a communication job or the timer interrupt. In the former case, the context-switching operation is integrated in the duration of the communication block. In FPS, after the termination of a job the next active job with the highest priority starts executing. If the schedule queue has no active jobs, then the *idle job* resumes its execution. Therefore, no special handling is required for the context-switching at the start of a job.

Terminate operations must occur atomically, i.e., non-preemptively. Therefore, preemption of low-priority jobs by high-priority ones is not allowed during the execution of terminate operations. In this case, the high-priority jobs are delayed until the low-priority jobs finish their execution. However, because communication blocks are treated during schedule synthesis of computation tasks as non-schedulable entities, preemption of the terminate operation of computation jobs is allowed by such blocks to simplify the formalization. In a real situation, all jobs that execute within the communication block must be delayed until the terminate operation finishes. But, enabling the preemption of such operations in the formalization gives the possibility to find a feasible schedule, if one exists, which is not be possible if constrained otherwise. This is an effect of the proposed step-wise approach of constructing the schedule of communication and computation jobs separately.

If the algorithm provides a solution with a preemption of the terminate operation, then the following actions can be taken. If the preempting communication blocks contain only occurrences of the timer interrupt, then the communication blocks, and hence, the execution of timer interrupt occurrences is shifted to the end of ov_t operation in a post-processing step. In this case, the utilization of the HP interval is unaffected and the execution order of other jobs does not change. This case is typically applied for SBP. An example of such shift is shown in Figure 4.14. If the communication block contains

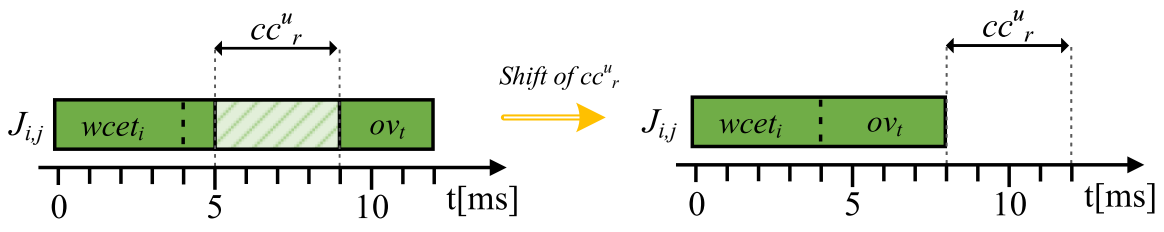


Figure 4.14: Shifting the execution of a communication block cc^u_r after terminate operation of a job $J_{i,j}$ finishes the execution. The dark green and light green boxes indicate the execution time and the preemption delay of job $J_{i,j}$, respectively.

jobs of LET Start and End tasks, then the delay caused by the terminate operation must be planned in their schedule construction and the schedule is regenerated. This means planning the execution of LET Start jobs with a delay of ov_t and shifting activation offsets of LET End jobs by an amount of ov_t to the left. The delay ov_t for LET Start jobs is planned only if they are contained in the preempting communication block. The same applies for LET End jobs. If the regeneration of the schedule returns a feasible schedule with the same priority ordering but without the preemption of the terminate operation, then the assignment is valid. This case is applied for PTP.

In FPS, the start of each communication block corresponds to the start of the context-switch operation that takes place before the occurrence of the timer interrupt, which is released to activate one or more LET tasks. If the proposed synthesis algorithm provides a priority ordering solution, in which the end time of a computation job equals the start time of a communication block, then this solution is considered valid and does not mean an over-utilization of the HP interval with context-switching overheads. This is because during and at the end of a terminate operation, there is no knowledge about which and when the next jobs are released. However, when the terminate operation is preempted by a communication block, which is, e.g., shifted at post-processing step, the context-switch operation is considered twice, at the terminate operation and at the start of the communication block. In this case, the second context-switch operation is not required. Because the preemption of the terminate operation by a communication block is expected to occur occasionally, allowing the context-switch operation twice is an implicit assumption of this approach.

The following terminology is used in the remaining sections. For each job $J_{i,j}$ of $T_i \in \tau^u_{cp}$, $atw_{i,j}$ denotes the *active time interval* defined by $[r_{i,j}, d_{i,j}]$, $etw_{i,j}$ denotes the *execution time interval* defined by $[st_{i,j}, et_{i,j}]$, and $rtw_{i,j}$ denotes the *response time interval* defined by $[r_{i,j}, et_{i,j}]$. The active time interval of each communication block $cc^u_r \in C_u$ corresponds to the interval $[cc^u_r.start, cc^u_r.end]$.

4.6.2.1 Start and Preemption Delays

Several variables are defined to ensure the FPS semantics and to calculate start and preemption delays caused by the execution of computation tasks and communication blocks. The *delay* and *preemption* variables are defined to track if computation jobs are delayed or preempted by other computation jobs or by communication blocks. In this way, if delays or preemptions occur, then start and preemption delays are calculated accordingly. A computation job is allowed to be preempted by other computation jobs only during its computation time. The terminate operation is not allowed to be preempted by these jobs, but only by communication blocks. A computation job is delayed either by communication blocks, by high-priority computation jobs, or by the terminate operation of low-priority jobs. The *non-overlapping* variables are defined to track the overlapping situation of active time intervals of computation jobs when a preemption or delay relation does not exists. Hence, if two jobs do not delay or preempt each-other, then their active time intervals do not overlap. The *right-neighbor* variables are defined to ensure FPS semantics in case of start delay situations.

The described types of variables are defined for each unique pair of computation jobs and for each unique pair of computation job and communication block that have an overlap of their activate time intervals. They are defined as follows.

Definition 4.11 (Delay Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $id_{i,j}^{k,l}$ indicate the Boolean variable that defines if job $J_{i,j}$ is delayed by job $J_{k,l}$ and $id_{k,l}^{i,j}$ the Boolean variable that defines if job $J_{k,l}$ is delayed by job $J_{i,j}$. The values of these variables are defined as*

$$id_{i,j}^{k,l} = \begin{cases} 1, & \text{if } J_{k,l} \text{ delays } J_{i,j} \\ 0, & \text{otherwise} \end{cases}, \tag{4.55}$$

$$id_{k,l}^{i,j} = \begin{cases} 1, & \text{if } J_{i,j} \text{ delays } J_{k,l} \\ 0, & \text{otherwise} \end{cases}. \tag{4.56}$$

Let $idhp_{i,j}^{k,l}$ be the Boolean variable that indicates if $J_{k,l}$ delays $J_{i,j}$ because T_k has higher priority than T_i ($\pi_i < \pi_k$) and $idnp_{i,j}^{k,l}$ define the Boolean variable that indicates if $J_{k,l}$ delays $J_{i,j}$ during the non-preemptive terminate operation when T_k has lower priority than T_i ($\pi_i > \pi_k$). Variables $idhp_{k,l}^{i,j}$ and $idnp_{k,l}^{i,j}$ describe the opposite relation between $J_{i,j}$ and $J_{k,l}$. The values of delay variables $id_{i,j}^{k,l}$ and $id_{k,l}^{i,j}$ are as

$$id_{i,j}^{k,l} = idhp_{i,j}^{k,l} + idnp_{i,j}^{k,l}, \tag{4.57}$$

$$id_{k,l}^{i,j} = idhp_{k,l}^{i,j} + idnp_{k,l}^{i,j}. \tag{4.58}$$

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $id_{i,j}^r$ indicate the Boolean variable that defines if job $J_{i,j}$ is delayed by communication block cc_r^u. The values of $id_{i,j}^r$ are defined as

$$id_{i,j}^r = \begin{cases} 1, & \textit{if } cc_r^u \textit{ delays } J_{i,j} \\ 0, & \textit{otherwise} \end{cases} . \tag{4.59}$$

Communication blocks are non-schedulable entities and are not delayed by computation jobs. Therefore, the opposite delay relation between $J_{i,j}$ and cc_r^u is not defined.

Constraint 4.9 (Delay Constraints): *For each computation job $J_{i,j}$ of each task T_i, the priority ordering and start delay relations of $J_{i,j}$ with other computation jobs and communication blocks that have overlapping active time intervals with $J_{i,j}$, are defined by the constraints given in Equations* (4.60) *to* (4.62).

$$\forall T_k, \forall J_{k,l}, idhp_{i,j}^{k,l} = 1 \Longleftrightarrow (\pi_i < \pi_k) \wedge (r_{i,j} < et_{k,l}) \wedge (et_{k,l} \leq st_{i,j}) \tag{4.60}$$

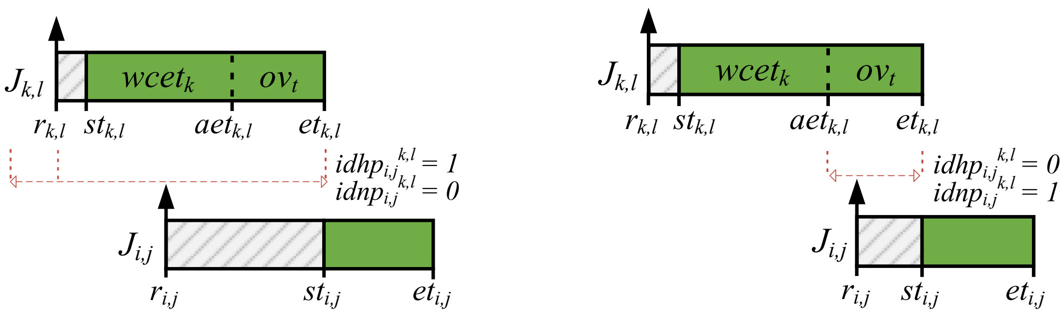
$$\forall T_k, \forall J_{k,l}, idnp_{i,j}^{k,l} = 1 \Longleftrightarrow (\pi_i > \pi_k) \wedge (aet_{k,l} < r_{i,j} < et_{k,l}) \wedge (et_{k,l} \leq st_{i,j}) \tag{4.61}$$

$$\forall cc_r^u \in C_u, id_{i,j}^r = 1 \Longleftrightarrow (r_{i,j} < cc_r^u.end) \wedge (cc_r^u.end \leq st_{i,j}) \tag{4.62}$$

The constraint in Equation (4.60) implies that a job $J_{k,l}$ delays job $J_{i,j}$ when $idhp_{i,j}^{k,l} = 1$ only if task T_k has higher priority than T_i. In this case, the delayed job $J_{i,j}$ is released at any time during the active time interval of the delaying job $J_{k,l}$ or before the release of $J_{k,l}$. The delay situation occurs only if job $J_{i,j}$ starts after job $J_{k,l}$ ends the execution. Therefore, the end time of $J_{k,l}$ and the start time of $J_{i,j}$ are constrained as $et_{k,l} \leq st_{i,j}$.

The constraint in Equation (4.61) implies that if a job $J_{k,l}$ delays job $J_{i,j}$ during the terminate operation, then task T_k must have lower priority than T_i and the release time of job $J_{i,j}$ must lay in the time interval $[aet_{k,l}, et_{k,l}]$, which defines the execution time interval of the terminate operation. In this case, the delay situation occurs only if job $J_{i,j}$ starts after job $J_{k,l}$ ends the execution. The constraint in Equation (4.62) implies that a communication block cc_r^u delays job $J_{i,j}$ only if it occurs before job $J_{i,j}$ starts the execution and after it is released.

An example of the delay relation between two jobs $J_{k,l}$ and $J_{i,j}$ is shown in Figure 4.15. The delay of $J_{i,j}$ by job $J_{k,l}$ when the priority of T_k is higher than T_i is shown in Figure 4.15a. In this case, the release time of $J_{i,j}$ is not constrained and can take place at any time in the time interval $[r_{k,l}, et_{k,l}]$ or at any time before the release time $r_{k,l}$. In the former case, job $J_{i,j}$ is delayed by $J_{k,l}$ only if it has not started execution until $r_{k,l}$ point of time. Figure 4.15b shows the case when $J_{i,j}$ is delayed by $J_{k,l}$ during the terminate overhead operation. This case happens only if the release time of $J_{i,j}$ occurs during $[aet_{k,l}, et_{k,l}]$ time interval, otherwise if $r_{i,j}$ occurs during $[st_{k,l}, aet_{k,l}]$, then $J_{i,j}$ preempts $J_{k,l}$ because T_i has higher priority than T_k.



(a) Delay due to high priority $\pi_k > \pi_i$.

(b) Delay due to low priority $\pi_k < \pi_i$.

Figure 4.15: Delay of job's execution due to priority ordering and non-preemptive section of the terminate operation. The gray boxes show the start delay intervals and the dark green boxes the execution of jobs.

Definition 4.12 (Preemption Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $ip_{i,j}^{k,l}$ indicate the Boolean variable that defines if job $J_{i,j}$ is preempted by job $J_{k,l}$ and $ip_{k,l}^{i,j}$ the Boolean variable that defines if job $J_{k,l}$ is preempted by job $J_{i,j}$. The values of these variables are defined as*

$$ip_{i,j}^{k,l} = \begin{cases} 1, & \text{if } J_{k,l} \text{ preempts } J_{i,j} \\ 0, & \text{otherwise} \end{cases}, \tag{4.63}$$

$$ip_{k,l}^{i,j} = \begin{cases} 1, & \text{if } J_{i,j} \text{ preempts } J_{k,l} \\ 0, & \text{otherwise} \end{cases}. \tag{4.64}$$

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $ip_{i,j}^r$ indicate the Boolean variable that defines if job $J_{i,j}$ is preempted by the communication block cc_r^u. The values of $ip_{i,j}^r$ are defined as

$$ip_{i,j}^{r} = \begin{cases} 1, & \text{if } cc_r^u \text{ preempts } J_{i,j} \\ 0, & \text{otherwise} \end{cases}. \tag{4.65}$$

Let $ipc_{i,j}^r$ define the Boolean variable that indicates if job $J_{i,j}$ is preempted by the communication block cc_r^u during computation time and $ipt_{i,j}^r$ the Boolean variable that indicates if job $J_{i,j}$ is preempted by cc_r^u during terminate overhead operation. The value of $ip_{i,j}^r$ is defined based on variables $ipc_{i,j}^r$ and $ipt_{i,j}^r$ by the constraint

$$ip_{i,j}^r = ipc_{i,j}^r + ipt_{i,j}^r. \tag{4.66}$$

Communication blocks are non-schedulable entities and are not preempted by computation jobs. Therefore, the opposite preemption relation between $J_{i,j}$ and cc_r^u is not specified.

Constraint 4.10 (Preemption Constraints): *For each computation job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, the priority ordering and preemption relations of $J_{i,j}$ with other computation jobs and communication blocks that have overlapping active time intervals with $J_{i,j}$, are defined by the constraints in Equations* (4.67) *to* (4.70).

$$\forall T_k, \forall J_{k,l}, ip^{k,l}_{i,j} = 1 \Longleftrightarrow \pi_i < \pi_k \wedge (st_{i,j} < r_{k,l} \leq aet_{i,j}) \wedge (st_{i,j} < st_{k,l}) \wedge (et_{k,l} \leq aet_{i,j}) \tag{4.67}$$

$$\forall cc^u_r \in C_u, ipc^r_{i,j} = 1 \Longleftrightarrow st_{i,j} < cc^u_r.start \wedge cc^u_r.end \leq aet_{i,j} \tag{4.68}$$

$$\forall cc^u_r \in C_u, ipt^r_{i,j} = 1 \Longleftrightarrow aet_{i,j} < cc^u_r.start \wedge cc^u_r.end < et_{i,j} \tag{4.69}$$

$$aet_{i,j} = et_{i,j} - ov_t - \sum_{\forall cc^u_r \in C_u} (d^u_r * ipt^r_{i,j}) \tag{4.70}$$

The constraint in Equation (4.67) implies that a job $J_{k,l}$ preempts job $J_{i,j}$ only if task T_k has higher priority than T_i. Furthermore, the preemption of job $J_{i,j}$ by another computation job $J_{k,l}$ is allowed only if $J_{k,l}$ is released and started after the start time of $J_{i,j}$ and only if $J_{k,l}$ ends the execution before or at the start time $aet_{i,j}$ of the terminate operation of $J_{i,j}$. If the release time $r_{k,l}$ and the end time $et_{k,l}$ equals $aet_{i,j}$, then in the *postprocessing* step of the computation schedule, i.e., during generation of the BTF, job $J_{k,l}$ does not preempt $J_{i,j}$, but $J_{i,j}$ terminates before the start of $J_{k,l}$ and the terminate operation of $J_{i,j}$ executes after the terminate operation of $J_{k,l}$.

The constraint in Equation (4.68) implies that a block cc^u_r preempts job $J_{i,j}$ during the computation time only if the execution interval of cc^u_r lays within the interval $[st_{i,j}, aet_{i,j}]$ of job $J_{i,j}$. Similarly, the constraint in Equation (4.69) implies that a communication block cc^u_r preempts job $J_{i,j}$ during the terminate operation only if the execution interval of cc^u_r lays within the interval $[aet_{i,j}, et_{i,j}]$ of job $J_{i,j}$. The distinction between variables $ipc^r_{i,j}$ and $ipt^r_{i,j}$ is required to calculate the start time $aet_{i,j}$ of the terminate operation of job $J_{i,j}$. Hence, the constraint in Equation (4.70) defines the value of $aet_{i,j}$, which can be any value between $st_{i,j}$ and $et_{i,j}$, but considering the terminate overhead ov_t and preemption delays that occur during the terminate operation.

An example of the preemption relation between two jobs $J_{k,l}$ and $J_{i,j}$ and job $J_{k,l}$ and communication block cc^u_r is shown in Figure 4.16a. The preemption of job $J_{k,l}$ by job $J_{i,j}$ occurs because job $J_{i,j}$ is released during $[st_{k,l}, aet_{k,l}]$ and the priority ordering is $\pi_i > \pi_k$. Similarly, the communication block cc^u_r preempts $J_{k,l}$ because cc^u_r occurs in the time interval $[st_{k,l}, aet_{k,l}]$. Figure 4.16b shows the preemption of a job $J_{k,l}$ by the communication block cc^u_r during the terminate operation. This preemption occurs because the execution time interval of cc^u_r lays in the interval $[aet_{k,l}, et_{k,l}]$. As shown earlier, job $J_{i,j}$ cannot preempt job $J_{k,l}$ during the terminate operation despite of the priority ordering $\pi_i > \pi_k$. Therefore, job $J_{i,j}$ is delayed by $J_{k,l}$.

Definition 4.13 (Non-overlapping Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $nov^{k,l}_{i,j}$ be the Boolean variable*

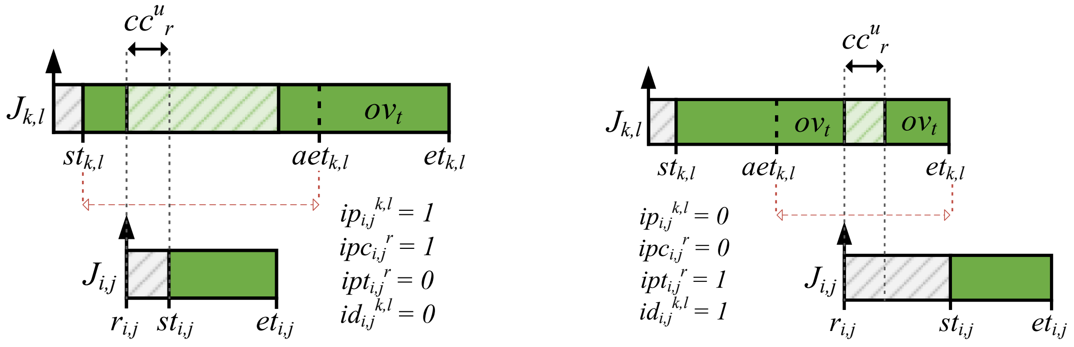


Figure 4.16: Preemption of job's execution and of terminate operation. The gray boxes show the start delay intervals and the dark green boxes the execution of jobs. The light green boxes show the preemption time of jobs.

that indicates if jobs do not have an overlap of their response time intervals. The values of $nov_{i,j}^{k,l}$ are defined as

$$nov_{i,j}^{k,l} = \begin{cases} 1, & \textit{if } J_{k,l} \textit{ and } J_{i,j} \textit{ do not overlap} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.71}$$

An overlap of the response time intervals of jobs $J_{i,j}$ and $J_{k,l}$ implies a delay or a preemption relation between jobs.

For each unique pair of job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, let $nov_{i,j}^r$ be the Boolean variable that defines the non-overlapping relation between job $J_{i,j}$ and communication block cc_r^u. The values of $nov_{i,j}^r$ are defined as

$$nov_{i,j}^{r} = \begin{cases} 1, & \textit{if } cc_r^u \textit{ and } J_{i,j} \textit{ do not overlap} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.72}$$

An example of jobs that have an overlapping of their active time intervals, but not an overlap in their response time intervals, is when for instance a job $J_{k,l}$ is released during the active time interval of a job $J_{i,j}$ and after $J_{i,j}$ has finished its execution. In this case, the constraint solving has to ensure that these jobs do not have either start delay or preemption relation, but instead they have a non-overlapping relation.

Constraint 4.11 (Non-overlapping Constraints): *For each unique pair of computation jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if jobs $J_{i,j}$ and $J_{k,l}$ do not overlap, then their response time intervals must not overlap, defined as*

$$nov_{k,l}^{i,j} = 1 \Longleftrightarrow rtw_{k,l} \cap rtw_{i,j} = \emptyset. \tag{4.73}$$

For each unique pair of computation job $J_{i,j}$ and communication block cc^u_r with overlapping active time intervals, if a job $J_{i,j}$ and a communication block cc^u_r do not overlap, then the response time interval $rtw_{i,j}$ of $J_{i,j}$ and the time interval of cc^u_r must not overlap, defined as

$$nov^r_{i,j} = 1 \Longleftrightarrow [cc^u_r.start, cc^u_r.end] \cap rtw_{i,j} = \emptyset. \tag{4.74}$$

Constraint 4.12 (Exclusive Disjunction Constraints): *For each unique pair of computation jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if job $J_{k,l}$ preempts job $J_{i,j}$, then the opposite is not allowed. If jobs do not preempt and do not delay each-other, then they must not overlap. These constraints are defined as*

$$ip^{k,l}_{i,j} + ip^{i,j}_{k,l} + id^{k,l}_{i,j} + id^{i,j}_{k,l} + nov^{k,l}_{i,j} = 1. \tag{4.75}$$

For each unique pair of computation job $J_{i,j}$ and communication block cc^u_r with overlapping active time intervals, if cc^u_r preempts or delays job $J_{i,j}$, then they cannot overlap. This constraint is defined as

$$ip^r_{i,j} + id^r_{i,j} + nov^r_{i,j} = 1. \tag{4.76}$$

The delay, preemption, and non-overlapping variables and constraints do not only define the execution order of jobs, but are as well used to calculate the start and preemption delays. Hence, the start delay $std_{i,j}$ of every job $J_{i,j}$ of each computation task $T_i \in \tau^u_{cp}$ is defined by the maximal time distance between the release time of $J_{i,j}$ and the end time of all computation jobs that delay job $J_{i,j}$ and the end time of all communication blocks that delay job $J_{i,j}$. The value of $std_{i,j}$ is defined in Constraint 4.13.

Constraint 4.13 (Start Delay): *The start delay $std_{i,j}$ of each job $J_{i,j}$ of each computation task T_i is defined as*

$$std_{i,j} = \max(\max(A_{i,j}), \max(B_{i,j})), \tag{4.77}$$

where $A_{i,j}$ and $B_{i,j}$ define the set of time distances as

$$A_{i,j} = \{id^{k,l}_{i,j} * (et_{k,l} - r_{i,j}) | \forall J_{k,l}, \forall T_k \in \tau^u_{cp}\} \tag{4.78}$$

$$B_{i,j} = \{id^r_{i,j} * (cc^u_r.end - r_{i,j}) | \forall cc^u_r \in C_u\}. \tag{4.79}$$

Only computation jobs and communication blocks that have an overlap of their active time intervals with $J_{i,j}$ are considered in the calculation of $std_{i,j}$.

The preemption delay $pt_{i,j}$ of each job $J_{i,j}$ is defined by the sum of WCETs of all computation jobs that preempt $J_{i,j}$ and the total duration of all communication blocks that preempt job $J_{i,j}$. The value of $pt_{i,j}$ is defined in Constraint 4.14.

Constraint 4.14 (Preemption Delay): *The preemption delay $pt_{i,j}$ of each job $J_{i,j}$ of each computation task $T_i \in \tau^u_{cp}$ is defined as*

$$pt_{i,j} = \sum_{\forall cc^u_r \in C_u} (d^u_r * ip^r_{i,j}) + \sum_{\forall J_{k,l}, T_k \in \tau^u_{cp}} (wcet_k + ov_t) * ip^{k,l}_{i,j}, \tag{4.80}$$

where d^u_r indicates the duration of the communication block cc^u_r. Only computation jobs and communication blocks that have an overlap of their active time intervals with $J_{i,j}$ are considered in the calculation of $pt_{i,j}$.

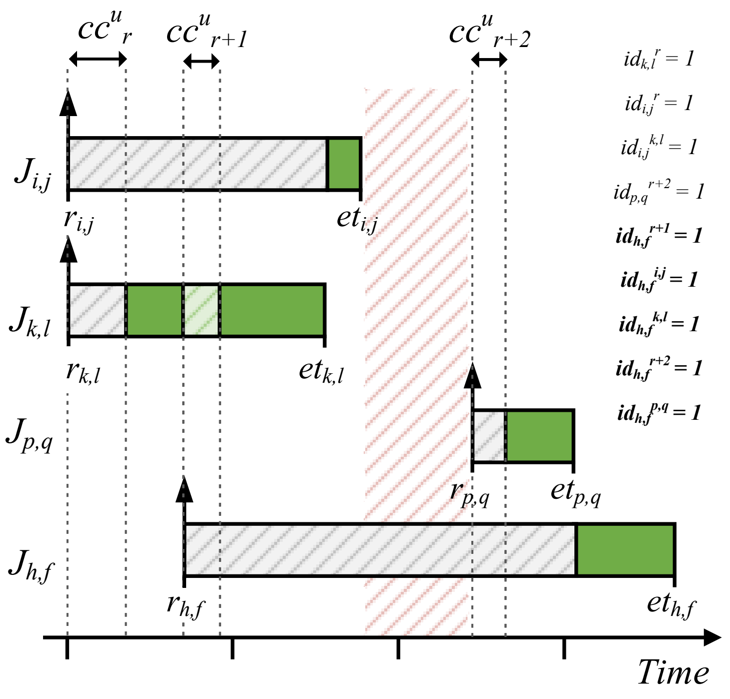
The *preemption* and *start delay* variables and constraints are insufficient to guarantee FPS semantics during delay situations. Figure 4.17a shows an example schedule that does not fulfill FPS semantics, although the variables described above are assigned correctly and constraints are fulfilled. In this example, the solver assigns value "1" to delay variables $id^r_{h,f}$, $id^{r+1}_{h,f}$, $id^{r+2}_{h,f}$, $id^{i,j}_{h,f}$, $id^{k,l}_{h,f}$, and $id^{p,q}_{h,f}$. In terms of constraints, the start delay of $J_{h,f}$ is assigned to the maximum end time of its delaying entities cc^u_r, cc^u_{r+1}, cc^u_{r+2}, $J_{i,j}$, $J_{k,l}$, and $J_{p,q}$, which corresponds to the end time of $J_{p,q}$. The schedule is invalid because the time interval between $et_{i,j}$ and $cc^u_{r+2}.start$, marked by the red pattern-filled box, is not utilized by any job although an active job exists, which in this case is job $J_{h,f}$. A correct schedule of this example is when job $J_{p,q}$ does not delay job $J_{h,f}$ and $J_{h,f}$ runs between $et_{i,j}$ and $cc^u_{r+2}.start$. If $J_{h,f}$ does not finish the execution before the start of cc^u_{r+2}, then it is preempted by cc^u_{r+2} and potentially by $J_{p,q}$ if the preemption occurs during the computation time of $J_{h,f}$. Therefore, the semantics of FPS are fulfilled only if the start delay time interval of each job is fully utilized by the execution of its delaying computation jobs and communication blocks. The correct schedule of Figure 4.17a is shown in Figure 4.17b. To avoid such invalid schedules, the *neighbor* variables and constraints are defined as follows.

Definition 4.14 (Neighbor Variables): *For each unique pair of jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, let $idrn^{k,l}_{i,j}$ indicate the Boolean variable that defines if job $J_{k,l}$ is the direct right neighbor of $J_{i,j}$ and $idrn^{i,j}_{k,l}$ the Boolean variable that defines if job $J_{i,j}$ is the direct right neighbor of $J_{k,l}$. The values of these variables are defined as*

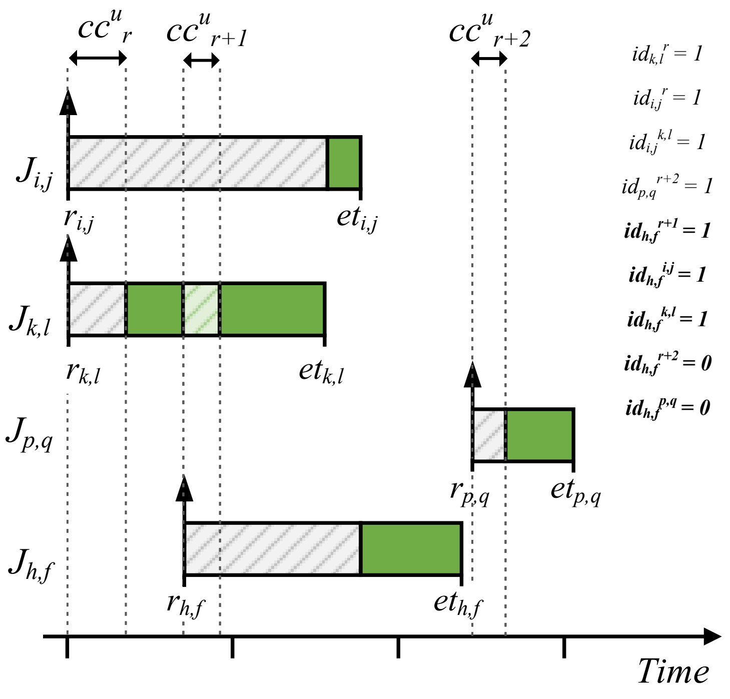
$$idrn^{k,l}_{i,j} = \begin{cases} 1, & \textit{if } J_{k,l} \textit{ is direct right neighbor of } J_{i,j} \\ 0, & \textit{otherwise} \end{cases}, \tag{4.81}$$

$$idrn^{i,j}_{k,l} = \begin{cases} 1, & \textit{if } J_{i,j} \textit{ is direct right neighbor of } J_{k,l} \\ 0, & \textit{otherwise} \end{cases}. \tag{4.82}$$

For each unique pair of job $J_{i,j}$ and communication block cc^u_r with overlapping active time intervals, let $idrn^r_{i,j}$ indicate the Boolean variable that defines if communication block cc^u_r is the direct right neighbor of job $J_{i,j}$ and $idrn^{i,j}_r$ the Boolean variable that defines if job $J_{i,j}$



(a) Start delay of job $J_{h,f}$ greater than the execution times of delaying jobs.



(b) Neighbor variables and correct execution order and distance among jobs.

Figure 4.17: An FPS schedule considering start and preemption delays. Priorities are $\pi_k > \pi_i > \pi_p > \pi_h$. The gray boxes show the start delay intervals and the dark green boxes the execution of jobs. The light green boxes show the preemption time of jobs. The red highlighted boxes indicate the underutilized time interval.

is the direct right neighbor of cc_r^u. The values of these variables are defined as

$$idrn_{i,j}^{r} = \begin{cases} 1, & \textit{if } cc_r^u \textit{ is direct right neighbor of } J_{i,j} \\ 0, & \textit{otherwise} \end{cases}, \tag{4.83}$$

$$idrn_{r}^{i,j} = \begin{cases} 1, & \textit{if } J_{i,j} \textit{ is direct right neighbor of } cc_r^u \\ 0, & \textit{otherwise} \end{cases}. \tag{4.84}$$

Although the communication block cc_r^u cannot be rescheduled, the variable $idrn_r^{i,j}$ is required to track if, e.g., a job $J_{i,j}$ starts directly after the end time of cc_r^u.

Constraint 4.15 (Neighbor Constraints 1): *For each unique pair of computation jobs $J_{i,j}$ and $J_{k,l}$ with overlapping active time intervals and $J_{i,j} \neq J_{k,l}$, if $J_{i,j}$ is direct right neighbor of $J_{k,l}$, then the opposite is not allowed. If a preemption relation exists between $J_{i,j}$ and $J_{k,l}$, then they cannot have direct right neighborhood relation. These constraints are defined as*

$$idrn_{i,j}^{k,l} + idrn_{k,l}^{i,j} \leq 1, \tag{4.85}$$

$$ip_{i,j}^{k,l} + ip_{k,l}^{i,j} = 1 \rightarrow idrn_{i,j}^{k,l} + idrn_{k,l}^{i,j} = 0. \tag{4.86}$$

If job $J_{k,l}$ is a direct right neighbor of $J_{i,j}$ then the end time $et_{i,j}$ of $J_{i,j}$ equals the start time $et_{k,l}$ of $J_{k,l}$, otherwise these times must be different. These constraints are defined as

$$idrn_{i,j}^{k,l} = \begin{cases} 1, & et_{i,j} = st_{k,l} \\ 0, & et_{i,j} \neq st_{k,l} \end{cases}. \tag{4.87}$$

For each unique pair of computation job $J_{i,j}$ and communication block cc_r^u with overlapping active time intervals, if communication block cc_r^u is a direct right neighbor of $J_{i,j}$, then the opposite is not allowed. If cc_r^u preempts $J_{i,j}$, then cc_r^u and $J_{i,j}$ cannot be direct right neighbors of each other. These constraints are defined as

$$idrn_{r}^{i,j} + idrn_{i,j}^{r} \leq 1, \tag{4.88}$$

$$ip_{i,j}^{r} = 1 \rightarrow idrn_{i,j}^{r} + idrn_{r}^{i,j} = 0. \tag{4.89}$$

If communication block cc_r^u is a direct right neighbor of $J_{i,j}$, then the end time $et_{i,j}$ of $J_{i,j}$ equals the start time $cc_r^u.start$ of cc_r^u, otherwise these times must be different. This constraint is defined as

$$idrn_{i,j}^{r} = \begin{cases} 1, & cc_r^u.start = et_{i,j} \\ 0, & cc_r^u.start \neq et_{i,j} \end{cases}. \tag{4.90}$$

If $J_{i,j}$ is a direct right neighbor of communication block cc^u_r, then the end time $cc^u_r.end$ of cc^u_r equals the start time $st_{i,j}$ of $J_{i,j}$, otherwise these times must be different. This constraint is defined as

$$idrn^{i,j}_r = \begin{cases} 1, & cc^u_r.end = st_{i,j} \\ 0, & cc^u_r.end \neq st_{i,j} \end{cases}. \tag{4.91}$$

Constraint 4.16 (Neighbor Constraints 2): *For each computation job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, let $hdrn_{i,j}$ define the Boolean variable that defines if $J_{i,j}$ has a direct right neighbor. The value of $hdrn_{i,j}$ is defined as*

$$hdrn_{i,j} = \sum_{\forall J_{k,l}, \forall T_k \in \tau^u_{cp}} idrn^{k,l}_{i,j} + \sum_{\forall cc^u_r \in C_u} idrn^r_{i,j}. \tag{4.92}$$

The Boolean type of $hdrn_{i,j}$ enforces that at most one direct right neighbor can exist for $J_{i,j}$.

For each communication block cc^u_r, let $hdrn^u_r$ define the Boolean variable that denotes if cc^u_r has a direct right neighbor. The $hdrn^u_r$ variable implies that either one or zero direct right neighbors exists for cc^u_r. The value of $hdrn^u_r$ is defined as

$$hdrn^u_r = \sum_{\forall J_{k,l}, \forall T_k \in \tau^u_{cp}} idrn^{k,l}_r. \tag{4.93}$$

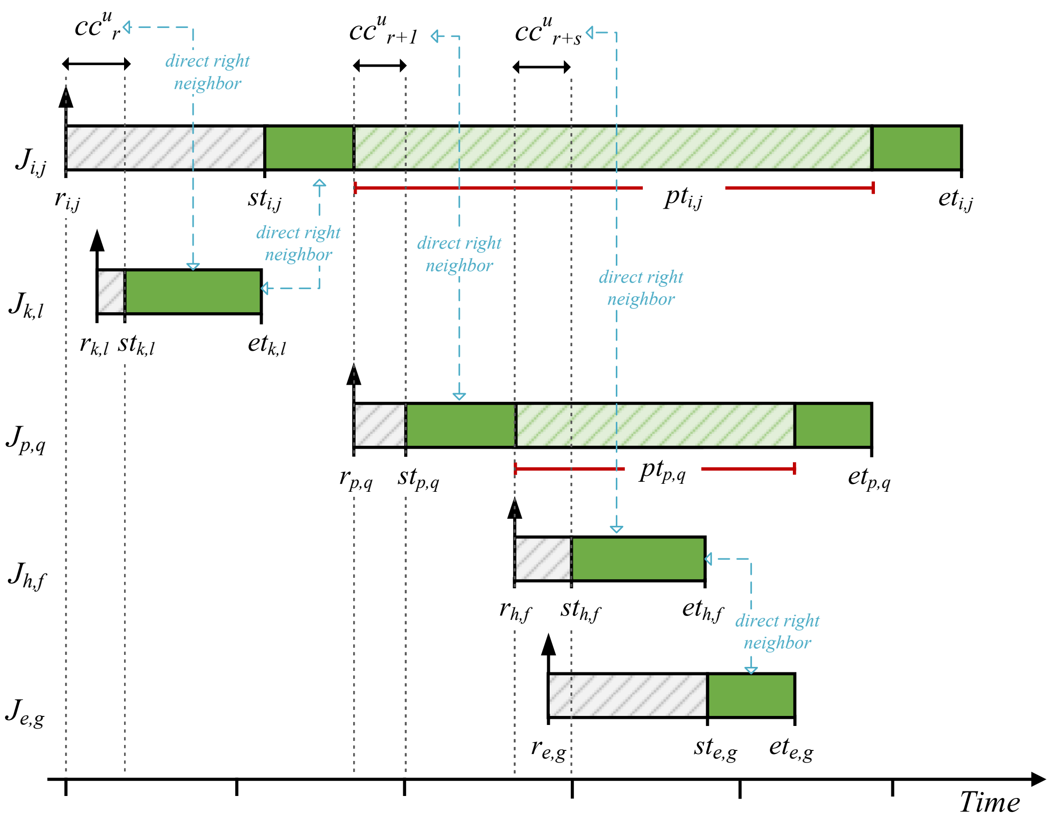
The Boolean type of $hdrn^u_r$ enforces that at most one direct right neighbor can exist for cc^u_r.

Constraint 4.17 (Delay Constraints for FPS Semantics): *For each computation job $J_{i,j}$ of each task $T_i \in \tau^u_{cp}$, the delaying computation jobs and communication blocks of $J_{i,j}$ must fulfill the following constraints*

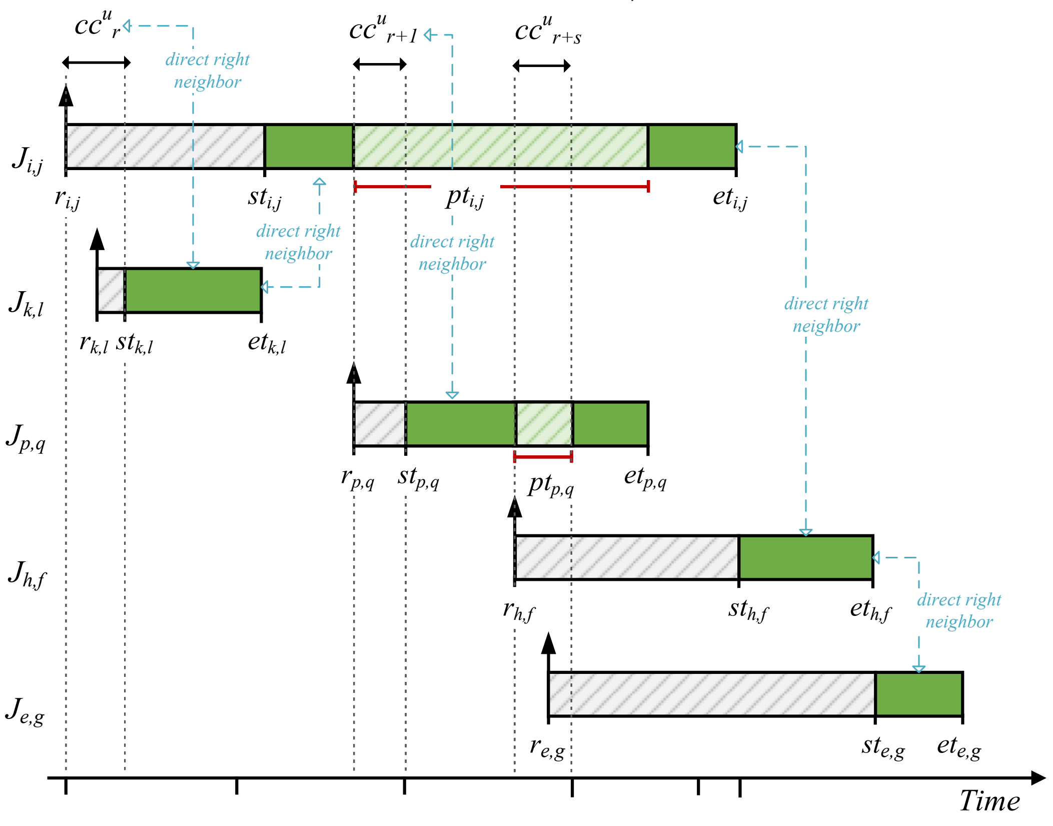
$$\forall J_{k,l}, \forall T_k \in \tau^u_{cp}, id^{k,l}_{i,j} = 1 \Longleftrightarrow (hdrn_{k,l} + \sum_{\forall J_{p,q}, \forall T_p \in \tau^u_{cp}} (ip^{k,l}_{p,q} * id^{p,q}_{i,j} * id^{k,l}_{i,j})) \geq 1, \tag{4.94}$$

$$\forall cc^u_r \in C_u, id^r_{i,j} = 1 \Longleftrightarrow (hdrn^u_r + \sum_{\forall J_{p,q}, \forall T_p \in \tau^u_{cp}} (ip^r_{p,q} * id^{p,q}_{i,j} * id^r_{i,j})) \geq 1. \tag{4.95}$$

In Constraint 4.17, if a job $J_{i,j}$ is delayed by other computation jobs and communication blocks, then the time interval between $r_{i,j}$ and $st_{i,j}$ is fully utilized by the delaying entities, i.e., all the delaying entities are running during this time frame. An entity is either a computation job or a communication block. The constraint in Equation (4.94) implies that each job $J_{k,l}$ that delays job $J_{i,j}$ must either have a direct right neighbor or must preempt at least one of the jobs that delays $J_{i,j}$. The equation is greater or equal to "1" because if job $J_{k,l}$ has no direct right neighbor then it can preempt one or more jobs that delay job $J_{i,j}$. Additionally, $J_{k,l}$ can have a direct right neighbor and preempt one or more jobs that delay job $J_{i,j}$. The constraint in Equation (4.94) implies that each



(a) Priorities are $\pi_k > \pi_h > \pi_p > \pi_i > \pi_e$.



(b) Priorities are $\pi_k > \pi_p > \pi_i > \pi_h > \pi_e$.

Figure 4.18: Example of start and preemption delays caused by higher-priority tasks and by communication blocks of the PTP protocol. The gray boxes show the start delay intervals and the dark green boxes the execution of jobs. The light green boxes show the preemption time of jobs.

communication block cc^u_r that delays job $J_{i,j}$ must either have a direct right neighbor or must preempt at least one of the jobs that delay $J_{i,j}$.

An example schedule of five jobs synthesized by the proposed approach is shown in Figure 4.18a. In this example, the priority ordering is defined as $\pi_k > \pi_h > \pi_p > \pi_i > \pi_e$ and job $J_{i,j}$ has the longest preemption delay because T_i has the lowest priority among tasks. Figure 4.18b shows the same example depicted in Figure 4.18a but with priority ordering defined as $\pi_k > \pi_p > \pi_i > \pi_h > \pi_e$. In this case, the new priority ordering changes the execution order of jobs and their respective start and preemption delays. The preemption delay of $J_{i,j}$ is reduced but the start delays of jobs $J_{e,g}$ and $J_{h,f}$ are increased. Job $J_{e,g}$ gets the longest start delay because T_e has the lowest priority among tasks. The start delay $std_{e,g}$ of job $J_{e,g}$ is defined by the maximal end time of its delaying entities, which in this case corresponds to the end time of $J_{h,f}$. Similarly, the $std_{h,f}$ is defined by the maximal end time of the delaying entities of $J_{h,f}$, which corresponds to the end time of $J_{i,j}$. The direct right neighbor relations are shown by the dotted bidirectional blue lines only for the entities that have a direct right neighbor. In Figure 4.18a, the communication block cc^u_{r+s} has job $J_{h,f}$ as direct right neighbor. After the change of the priorities in Figure 4.18b, the communication block cc^u_{r+s} does not have a direct right neighbor because at the end time of cc^u_{r+s} job $J_{p,q}$ resumes the execution. However, the constraints are still valid because cc^u_{r+s} preempts at least one job, which in this case is job $J_{p,q}$.

4.6.2.2 Optimizations

The preemption costs are optimized by minimizing the function $f^u_r(3)$, defined in Equation (4.96), which instructs the solver to produce solutions with minimal number of preemptions.

$$f^u_r(3) = \sum_{\forall J_{i,j}, \forall T_i \in \tau^u_{cp}} \Big(\sum_{\forall J_{k,j}, \forall T_k \in \tau^u_{cp}} ip^{k,l}_{i,j} + \sum_{\forall cc^u_r \in C_u} ip^r_{i,j} \Big) \tag{4.96}$$

This optimization applies only to preemptions of computation tasks. The number of preemptions for communication tasks is minimized by scheduling them non-preemptively when possible and with higher priority than computation tasks. If the solver is not configured to optimize the schedule by means of function $f^u_r(3)$, then it provides all feasible solutions that *satisfy* the constraints described in this section.

4.7 Evaluation

This evaluation focuses on the following aspects.

1) The scheduling success rate of TTS and FPS approaches is observed for applications that apply PTP and SBP protocols. The scheduling success rate defines the

ability of TTS and FPS synthesis algorithms to find a feasible schedule. Because PTP and SBP protocols have different demands in terms of processor resources, the success rate of TTS and FPS differs for each protocol and depends also on how each scheduling algorithm schedules tasks. Hence, this evaluation observes at which degree are TTS and FPS applicable for each buffering protocol.

2) The impact of application extensions with new functionalities on scheduling feasibility considering buffering demands of SBP and PTP is observed. The extensibility is expressed by the increase of the computation load of tasks. Hence, the scheduling success rate of both scheduling mechanisms considering the buffering run-time impact of PTP and SBP is evaluated.
3) Preemptions caused by timer's execution and scheduling semantics are evaluated and reduced for each scheduling strategy and each buffering protocol. PTP and SBP lead to different activation and context-switching overheads.
4) The run-time performance of each schedule synthesis algorithm is evaluated.

This evaluation uses a selection of metrics calculated in TA.Inspection option of TA Tool Suite [85] of Vector Informatik GmbH. The simulation of the application considering the produced schedule is not required because the proposed scheduling synthesis algorithms generate the execution trace of the application and exports it in the BTF format [161], which is imported in TA Tool Suite to evaluate timing requirements and calculate several metrics. The schedule synthesis algorithms are implemented in the CP-SAT solver of Google OR-Tools [162]. This evaluation is performed on a machine equipped with 64 GB RAM and Intel(R) Xeon(R) W-2133 CPU 3.60 GHz with 6 cores.

4.7.1 Configurations

This case study is based on synthetically generated application models with characteristics of the synthetic *Engine Management System (EMS)*s described in Section 3.5. The feasibility and performance of the proposed scheduling synthesis algorithms are influenced by task attributes such as load, periods, and the number of tasks. Chassis applications are not used in this evaluation because their task attributes are nearly a subset of the EMS applications. The buffering evaluation results of Section 3.5.1.2 are used in this case study to benchmark the buffering load of PTP and SBP.

The schedule synthesis is performed for tasks of each core separately. The EMS tasks, given in Table 3.2, can be mapped to different cores during software integration phase and it is assumed that any valid task-to-core mapping exists for these tasks. Therefore, the synthetically generated models of this evaluation are created to contain a unique subset or the complete set of EMS periods shown in Table 3.2. Each model contains unique periods with the assumption that tasks with equal periods are allocated on different cores as a mechanism to execute parts of the application in parallel. If two or

Configuration (∀ model)	**Load Ranges (%)**
Communication	$0.5 - 14$
ISR^{init}	$0.0002 - 0.006$
Initialization of indexes	$0.001 - 1$

Table 4.3: Configuration of the buffering load.

more tasks of equal periods are mapped to the same core, e.g., with period 5 ms, then they are merged into one to reduce the activation and context-switching overheads. In this way, the effect of unique task periods on scheduling is observed.

Models are generated considering the following characteristics. The impact of the computation load on scheduling capabilities of each scheduling strategy is observed by generating model sets with computation load between 30 % − 100 %. Each model set contains 50 models. In total, 1,200 models are generated. Models of different model sets have equal amount of tasks, equal total buffering load, and the same task characteristic such as the period, offset, and LET duration. For example, *Model-1* of the *ModelSet-30* and *Model-1* of the *ModelSet-40* have equal total buffering load, equal number of tasks, periods, offsets, and LET duration. But their total computation load is different. Hence, each model of *ModelSet-30* has the total computation load 30 % and each model of *ModelSet-40* has computation load 40 %. The buffering load varies among models of the same model set. For each model of each model set, the buffering load and the load of the timer come in addition to the computation load. The load of the timer depends on the amount of jobs released in the HP interval.

The ranges of the buffering load for both SBP and PTP are based on the experimental results of Section 3.5.1. Note that the buffering load results of the Section 3.5.1 show the buffering load of six cores. In this evaluation, an approximation of the total buffering load for tasks mapped to one core only is considered. In PTP, the buffering load represents the execution load of communication tasks. In SBP, the buffering load represents the execution load of ISR^{init}, executed on each core at the beginning of the HP, and the execution load of initialization operations of buffer indexes at the start of computation tasks. In SBP, the maximal buffer load includes as well the load when the *local programming style* is used. In this evaluation, the load of ISR^{init} represents the one of all cores because during the execution of ISR^{init} on one core the synchronization between other cores is assumed to take place.

In this evaluation, the buffering load is configured as in Table 4.3. The load ranges represent the buffer load for models with data accesses between 500 − 15,000. The buffer load for models with data accesses higher than 15,000 are not considered for the following reasons. Firstly, a buffering load of PTP greater than 15 % per core is considered an inefficient use of processor's capacity. Therefore, applying PTP for this complexity of applications is unrealistic and impractical. For this reason, the

Overheads	**Values** (µs)
Activation ov_a	40
Termination ov_t	30
Context-Switch ov_{cs}	20
Resource ov_{res}	5

Table 4.4: Run-times for overheads.

evaluation of scheduling performance is focused on applications with PTP buffering load up to 15 %. Secondly, the in-vehicle applications found in practice can apply LET semantics only for a limited amount of data. Therefore, the range of data accesses taken in this study assumes the typical use case of LET.

This evaluation focuses on two benchmarks. *Benchmark 1* evaluates models with *synchronous* tasks that have LET duration equal to their period. In this case, task offsets are equal to 0. *Benchmark 2* evaluates models with *asynchronous* tasks that have LET duration smaller than their period. This benchmark is designed to observe the effect that the offset and LET duration have in the overall schedulability of TTS and FPS when SBP and PTP are used. Models of *Benchmark 2* are generated considering the constraint in Equation (3.8), which ensures that the generated models have the termination of LET intervals of all jobs within the HP interval. This constraint is essential for the correct buffering behavior of SBP. In *Benchmark 2*, the offsets and LET duration of tasks are generated such that a dataflow exists between their LET intervals. Hence, the offset of a task corresponds to the LET end of the first LET interval occurrence of another task. The first task in the dataflow chain has an offset 0 ms and the rest of tasks have an offset greater than 0 ms but less than or equal to 50 % of their period. The LET duration is any random value between 50 % – 90 % of the period that fulfills the condition in Equation (3.8).

The WCETs of computation and communication tasks of both benchmarks are generated such that the following condition is fulfilled

$$\forall (T_i^S, T_i^E) \in \tau_{cc}^u, T_i \in \tau_{cp}^u \Rightarrow wcet_i^S + wcet_i^E + wcet_i + 3 * ov_a + 3 * ov_t < let_i, \quad (4.97)$$

where $wcet_i^S$, $wcet_i^E$, and $wcet_i$ are the respective WCETs of tasks T_i^S, T_i^E, and T_i. The value $3 * ov_a$ represents the execution time of the timer interrupt released to activate tasks T_i^S, T_i^E, and T_i, and value $3 * ov_t$ represents the termination overhead for these tasks. The condition in Equation (4.97) does not guarantee that a feasible schedule exists for these models. However, if it is not fulfilled, then the schedule is likely infeasible and the effects of scheduling and buffering cannot be properly observed.

The schedule is generated for all models considering the activation, termination, context-switching, and resource usage overheads shown in Table 4.4. These overheads are an approximation of the ones observed in an industrial real-time OS. Note that they are generally implementation and platform dependent and can vary among processors and different OSs. Although in TTS less context-switching overheads is expected than in FPS, i.e., due to the execution of the scheduler, this evaluation considers equal context-switching time in both TTS and FPS. This assumption is taken because the considered automotive AUTOSAR OSs does not have an implementation of TTS and the overheads for TTS could not be estimated for this evaluation.

The schedule duration synthesized by the proposed approaches is defined by the HP duration of tasks allocated on the core, referred as *local* HP. However, this is only applicable for schedule synthesis when PTP is used as buffering protocol. In SBP, the schedule duration is typically defined equal to the *global* HP, defined by periods of tasks running on all cores, because the buffer schedule is constructed for all cores and the ISR^{init} executes at the start of the global HP. The global HP of EMS tasks, given in Table 3.2, is 1000 ms. To reduce the memory resources for storing the schedule table, the schedule duration is defined as follows. In PTP, the schedule duration equals the local HP. In SBP, if the local HP of the model equals 1000 ms, then the schedule duration is 1000 ms. Otherwise, the schedule is generated for the duration of two times the local HP because the second occurrence of the local HP repeats until the end of one occurrence of the global HP. The first occurrence of the local HP repeats only once at the beginning of the global HP.

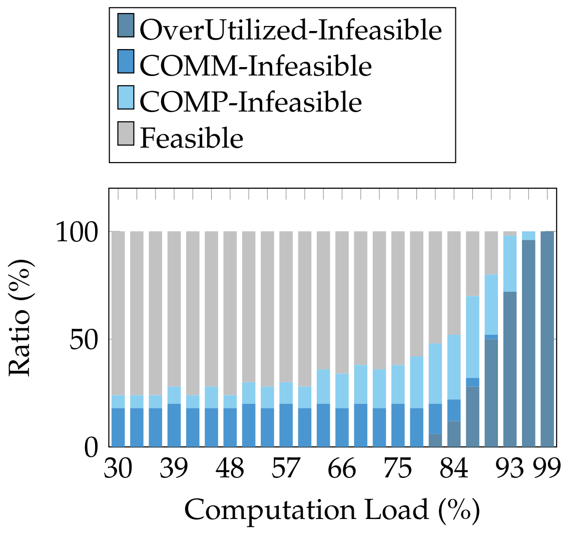
The load coming from the timer's execution, i.e., taking place for activation of tasks, is calculated statically based on the configured overheads and the amount of jobs as

$$U^{u}_{tr} = \frac{ov_a * jobs}{hp}, \tag{4.98}$$

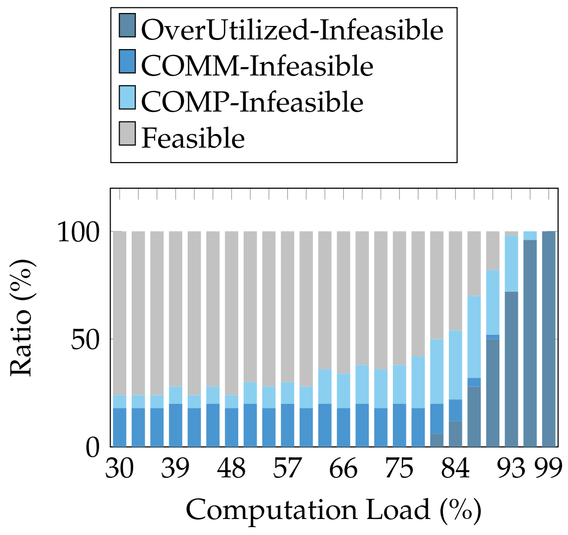
where U^{u}_{tr} defines the timer's load and *jobs* indicates the number of jobs in the HP interval with duration hp. In the evaluated models, the amount of jobs for SBP and PTP are in the range 27 – 577 and 78 – 1728, respectively. The amount of jobs increases linearly among models. Hence, the load coming from the timer execution is for SBP in the range 0.11 % – 2.31 % and for PTP in the range 0.21 % – 4.61 %. These ranges are equal in both benchmarks.

4.7.2 Feasibility

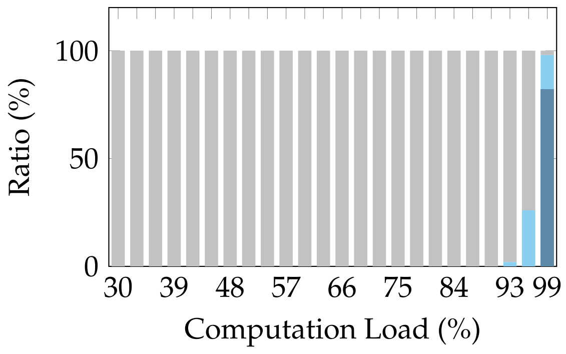
In this case study, the schedule synthesis algorithms are configured to terminate after finding the first *feasible* schedule, if one exists, or after finding an infeasible schedule. The feasibility results for the synchronous and asynchronous benchmarks are shown in Figure 4.19 and Figure 4.20, respectively. In all depicted graphs, the x-axis shows the *computation load* and the y-axis shows the *Ratio (%)* of models (out of the studied set) that are *Feasible*, *COMP-Infeasible*, *COMM-Infeasible*, and *OverUtilized-Infeasible* for each scheduling algorithm and each buffering protocol. Note that the



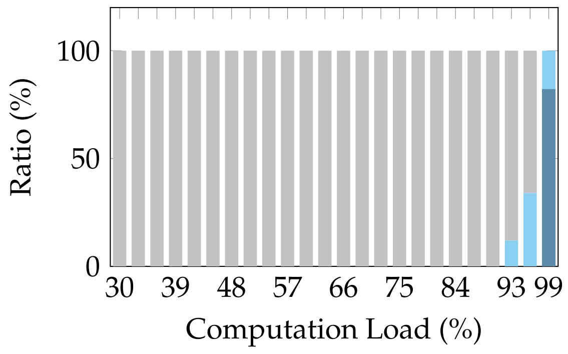
(a) TTS PTP



(b) FPS PTP



(c) TTS SBP



(d) FPS SBP

Figure 4.19: *Benchmark 1* - Synchronous (LET = Period). Schedule feasibility of TTS and FPS for applications that apply PTP and SBP protocols to integrate LET semantics.

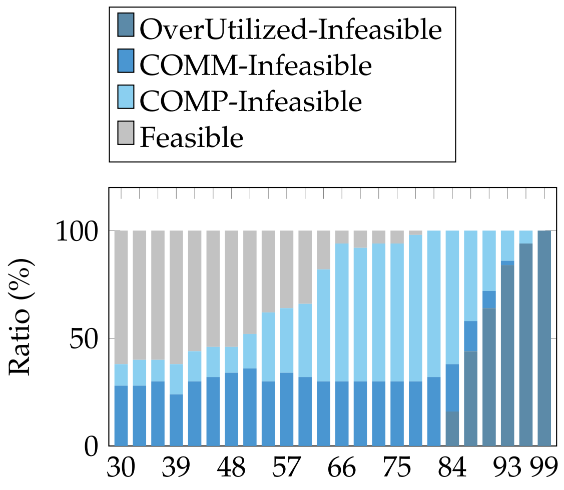
x-axis does not show the load coming from buffering and the timer interrupt. Hence, considering this additional load and the increasing computation load, the total load of certain models exceeds the value 100 %. Therefore, the schedule of these models is explicitly infeasible and is shown in the graphs by the *OverUtilized-Infeasible* plot. The *Feasible* plot indicates the ratio of models for which a feasible schedule is found. The *COMM-Infeasible* indicates the ratio of models that have an infeasible schedule for communication tasks. In this case, communication tasks cannot be scheduled to execute within their LET intervals. The *COMP-Infeasible* indicates the ratio of models that have an infeasible schedule for computation tasks. In this case, communication tasks have a feasible schedule, but the computation tasks not. In SBP, communication tasks do not exist, and, hence, the *COMM-Infeasible* plot is not shown for SBP results.

Figure 4.19a and Figure 4.19b show the feasibility results of PTP models synthesized by TTS and FPS for *Benchmark 1*, respectively. The results show that the *Feasible* ratio of PTP in both scheduling approaches decreases with the increase of the computation load. This occurs because of the stringent communication requirement of PTP and

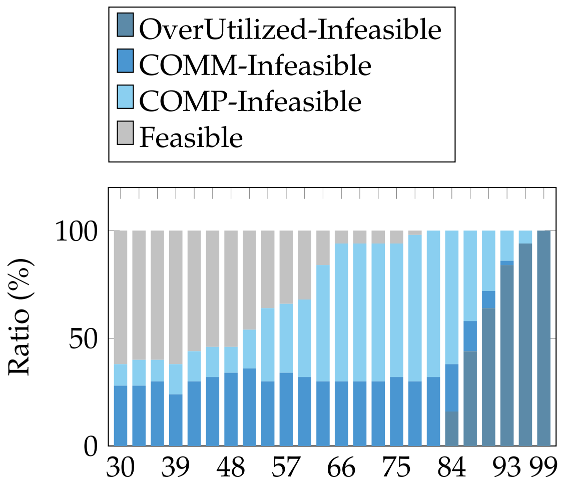
the extra load coming from the buffering and task activation. These aspects affect the schedule feasibility of some PTP models also at low computational load. The ratio of models with an infeasible communication schedule, depicted by plot *COMM-Infeasible*, remains unchanged for model sets with computation load up to 75 %, apart from small deviations, which occur due to the distribution of the buffering load to tasks of different LET intervals and periods. An infeasible schedule for communication tasks occurs in models that have the highest buffering load. In this case, the execution of all coinciding LET Start and End jobs does not fit the smallest LET interval duration among active tasks. If a schedule cannot be found for communication tasks, then the software integrator must allocate tasks to different cores or increase the processing resources until a feasible schedule is guaranteed.
The increase of the computation load increases the number of models exceeding the 100 % of total load. In both graphs, the ratio of *OverUtilized-Infeasible* models increases with the increase of the *computation load*, which is expected because within a model set of each computation load the buffering load is in the range of 0.5 % – 14 % and the timer interrupt load is between 0.21 % – 4.61 %. The *COMM-Infeasible* ratio drops when the *OverUtilized-Infeasible* ratio increases because models that have such infeasible communication schedule exceed the 100 % of total load and fall in the category of the *OverUtilized-Infeasible* models. The trend of the *Feasible* ratio coincides with the trend of the *COMM-Infeasible* and *COMP-Infeasible* ratios. Hence, at most 76 % of models with computation load up to 60 % have a feasible communication and computation schedule. This ratio decreases further when the computation load exceeds 60 %. The *COMP-Infeasible* ratio declines drastically starting with computation load 90 %, which occurs due to the significant increase of the *OverUtilized-Infeasible* ratio. Model sets with computation load $\geq$ 90 %, have 50 % – 100 % of models with a total utilization greater than 100 %. Therefore, the *Feasible* ratio declines after computation load 90 %. The described behavior is occurring in both TTS and FPS results.

Figure 4.19c and Figure 4.19d show the feasibility results of SBP models for TTS and FPS for *Benchmark 1*, respectively. The results show that TTS and FPS provide a feasible schedule for all SBP models with a total load of up to 93 %. This is due to the fact that SBP does not have stringent communication requirements and high buffer and timer load. The impact of the computation load on the ratio of *Feasible* models occurs in model sets with computation load between 93 % – 96 %. Because the buffering and the timer's load of SBP does not exceed the 2.31 % in total, the *OverUtilized-Infeasible* ratio increases only for model sets with computation load 99 %. Model sets of computation load 99 % have 82 % of models with total utilization greater than 100 %. Therefore, the strongest decline of the *Feasible* ratio occurs when the computation load is 99 %.

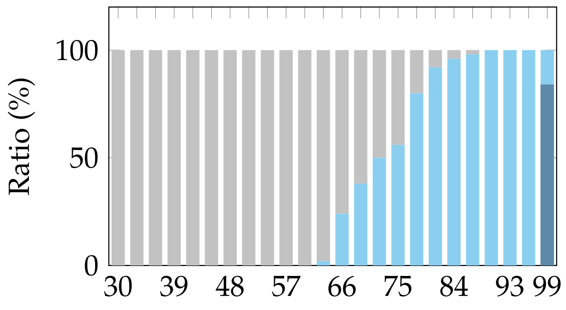
Figure 4.20a and Figure 4.20b show the feasibility results of PTP models synthesized by TTS and FPS for *Benchmark 2*, respectively. As in *Benchmark 1*, the *OverUtilized-Infeasible* ratio increases with the increase of the computation load. Similar tendency as in *Benchmark 1* occurs for *COMM-Infeasible*, *COMP-Infeasible*, and *Feasible* ratios. Therefore, this tendency is not further described here. In *Benchmark 2*, compared to *Benchmark 1*, the impact of the buffering and computation load on schedule feasibility



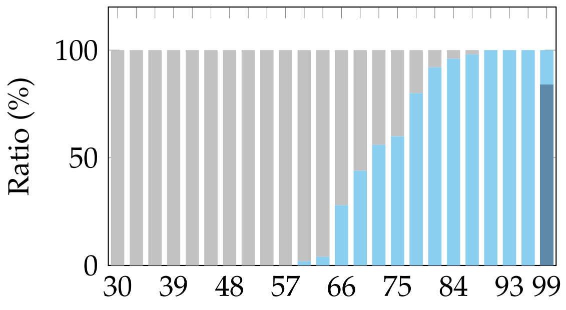
(a) TTS PTP



(b) FPS PTP

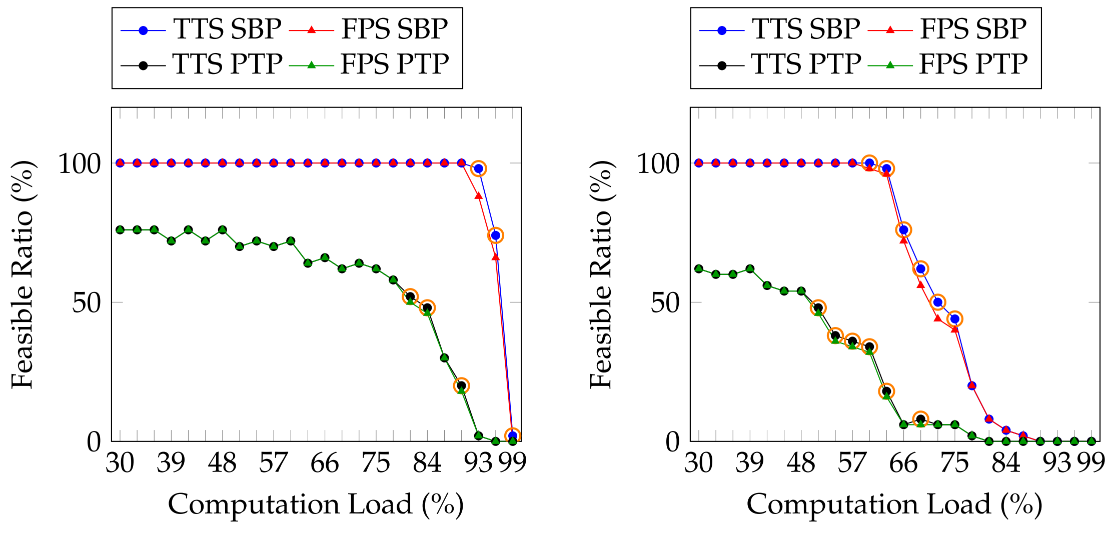


(c) TTS SBP



(d) FPS SBP

Figure 4.20: *Benchmark 2* - Asynchronous (LET < Period). Schedule feasibility of TTS and FPS for applications that apply PTP and SBP protocols to integrate LET semantics.



(a) Synchronous (LET = Period) (b) Asynchronous (LET < Period)

Figure 4.21: Feasibility ratio of TTS and FPS for applications that apply PTP and SBP protocols to integrate LET semantics. The results with higher *Feasible* ratio in TTS than in FPS are marked by the circles colored in orange.

is higher because of the reduced duration of LET intervals. This impact is evident in both scheduling mechanisms and buffering protocols.

Figure 4.20c and Figure 4.20d show the feasibility results of SBP models for TTS and FPS for *Benchmark 2*, respectively. In SBP, despite of the low buffering load, the *Feasible* ratio decreases at a faster rate than in *Benchmark 1*, which occurs due to the reduced duration of LET intervals. Hence, in SBP, considering the feasibility results of both TTS and FPS, the synchronous task set design (*Benchmark 1*) provides a 100 % *Feasible* ratio for computation load up to 90 % and the asynchronous task set design (*Benchmark 2*) provides the 100 % *Feasible* ratio for computation load up to 60 % for TTS and up to 57 % for FPS. These results indicate that, despite of the benefits, e.g., reduction of end-to-end delays, an asynchronous task set design limits functionality extensions of next generations of an embedded application, also when SBP is used, unless further design steps such as reallocation of new functions to other cores or the redesign of LET intervals is applied. In this case, more hardware resources would be needed.

To compare TTS and FPS in terms of schedulability, the *Feasible* ratios of *Benchmark 1* and *Benchmark 2* are plotted in Figure 4.21a and Figure 4.21b, respectively. In *Benchmark 1*, as shown in Figure 4.21a, TTS finds for PTP models with computation load 81 %, 84 %, and 90 % exactly 2 % more feasible schedules than FPS. Additionally, TTS finds for SBP models with computation load between 93 % – 94 % in average 6.6 % more feasible schedules than FPS. In *Benchmark 2*, as shown in Figure 4.21b, TTS finds for PTP models with computation load 51 % – 63 % and 69 % exactly 2 % more feasible schedules than FPS. Additionally, TTS finds for SBP models with computation load between 60 % – 75 % in average 4 % more feasible schedules than FPS. Better feasibility results come from TTS due to its ability to enforce an arbitrary execution

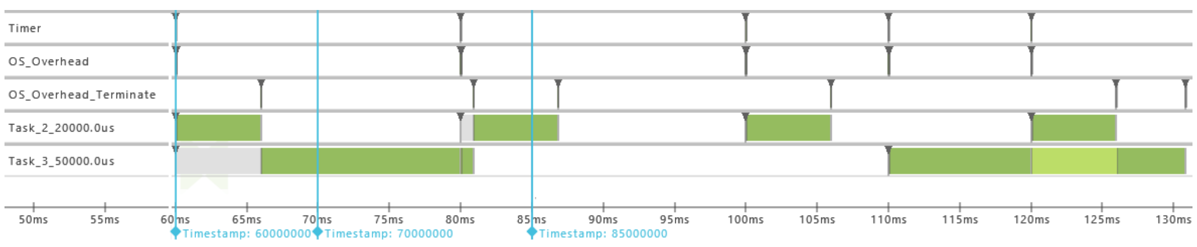


Figure 4.22: A schedule snapshot of a SBP model with a feasible schedule in TTS but infeasible schedule in FPS. The snapshot is taken in TA Tool Suite [163].

order between jobs, which is valid as long as they are completed within their LET intervals. In FPS, the execution of active jobs is closer to their activation times and is defined by the priority order of their tasks. A definite conclusion as to when TTS provides a better schedulability than FPS could not be derived from the obtained results. Nevertheless, it can be observed that in the asynchronous task sets, more models have a feasible schedule in TTS, but not in FPS. This is reasonable since the schedulability of a scheduling algorithm is determined by the sampled timing information of tasks such as periods, offsets, LET duration and WCETs.

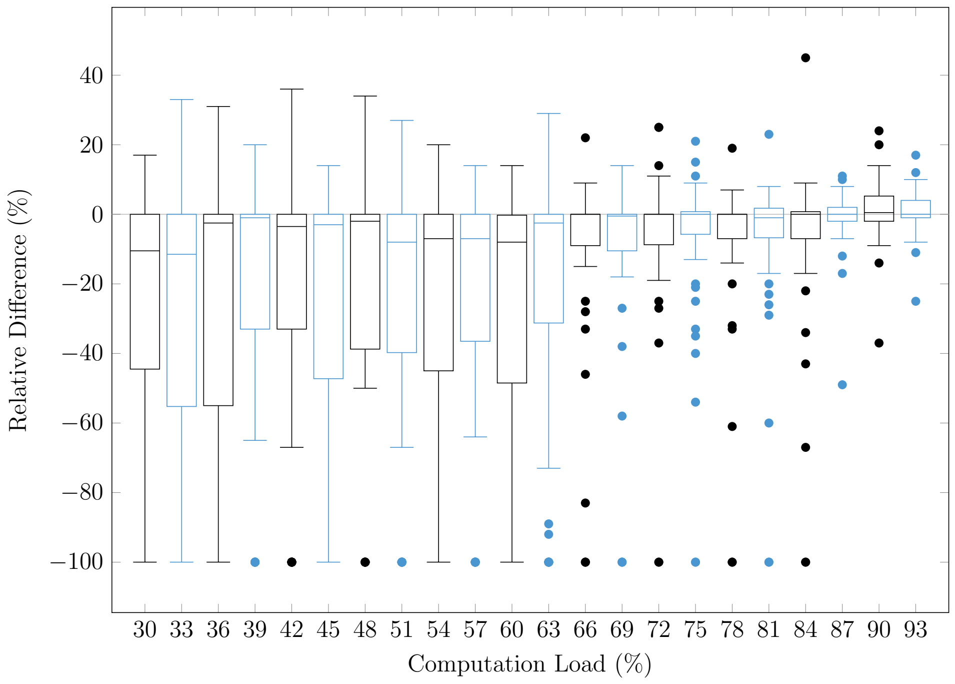
An example of a TTS schedule is given in Figure 4.22. The schedule snapshot belongs to one of the SBP models from the *Benchmark 2* for which a feasible schedule is found with TTS but not with FPS. Only the *Timer* interrupt, the jobs for overhead handling and tasks that directly impact the schedulability are shown, i.e., tasks *Task_2_20000.0us* and *Task_3_50000.0us*. The deadlines of *Task_2_20000.0us* and *Task_3_50000.0us* are 10 ms and 25 ms, respectively. At time 60 ms, highlighted by the first blue bookmark, a job is released for each task *Task_2_20000.0us* and *Task_3_50000.0us*.
In the FPS scenario, *Task_2_20000.0us* would execute first if it has a higher priority than *Task_3_50000.0us*. Otherwise, *Task_2_20000.0us* misses its deadline at time 60 ms, highlighted by the second blue bookmark. At time 80 ms another job of *Task_2_20000.0us* releases, which in the FPS scenario, different to what is shown in Figure 4.22, would be scheduled to execute after *Task_3_50000.0us* is preempted by the *Timer* interrupt because *Task_2_20000.0us* would have the highest priority. In this case, the preempted job of *Task_3_50000.0us* would miss its deadline at time 85 ms, highlighted by the third blue bookmark. Reversing priorities between these two jobs again results in an infeasible schedule in FPS.
In TTS, as shown in Figure 4.22, a feasible schedule exists because at time 80 ms, job *Task_3_50000.0us* resumes its execution after the preemption of the *Timer* interrupt. In this case, the execution of *Task_2_20000.0us* is shifted and deadlines of both tasks are fulfilled. This behavior is not possible in FPS, because the ordering of jobs is defined by priorities of their tasks. Therefore, a feasible schedule cannot be found for this model by FPS. Although these results show that TTS can provide higher schedulability than FPS, the FPS algorithm performs comparably well to TTS in terms of schedulability considering the overall results of this evaluation.

4.7.3 Resource Optimization

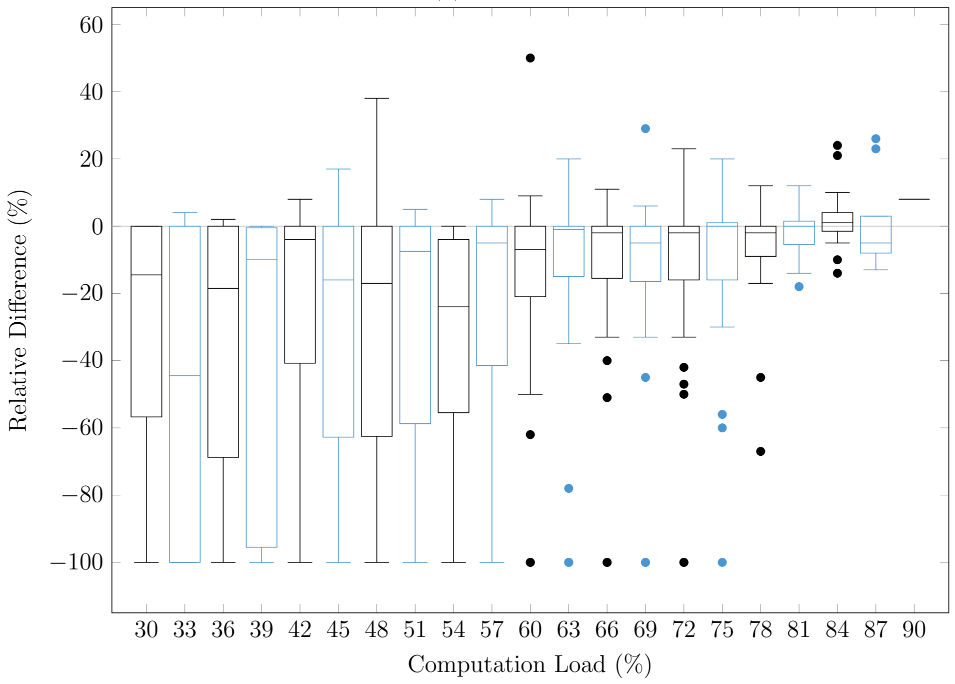
In this case study, the schedule synthesis algorithms are configured to provide an optimal solution. The optimization goal is to minimize the number of preemptions of computation tasks. To obtain an optimal solution within a reasonable amount of time, a time limit of 6 h is enforced to the CP program. This time limit was set to give the solver enough time to find the optimal solution for each model. Therefore, the solver terminates when the optimal solution is found or when this time limit is reached. Note that the optimal solution provided by the CP solver in this case is either the optimal or a near sub-optimal solution. The time limit of 6 h is reached by TTS for 6.39 % of the SBP models and for 5.56 % of the PTP models. This means that for these percentages of models, the solver certainly provides a near sub-optimal solution.

The results of this evaluation are shown in Figure 4.23 and Figure 4.24 for *Benchmark 1* and *Benchmark 2*, respectively. The box-plot graphs show the *Relative Difference (%)* of the total preemptions between TTS and FPS versus the computation load. Negative values of the *Relative Difference (%)* indicate that the optimal solution in FPS has a higher number of preemptions than the one in TTS, and positive values indicate the opposite. In these plots, only the results of models that have feasible schedules in both TTS and FPS are shown. As expected, the results show that the TTS algorithm provides overall optimal schedules with a smaller number of preemptions than FPS, and, hence, less preemption overheads. This is due to the ability of TTS to shift the execution of jobs to non-overlapping time intervals. It should be noted that the quality of optimal schedules in TTS also depends on the ability of the constraint solver to find the actual optimal solution. Based on our observations, in TTS, the feasible solution space is higher than in FPS. Therefore, a large feasible solution space reduces the solver's ability to find the actual optimal solution, resulting in a sub-optimal solution instead. In FPS, the solution space is smaller than in TTS due to symmetry constraints enforced by the priority ordering between tasks. Therefore, due to the internal search algorithm of the solver, only a statement about the general trend can be obtained for the results of Figure 4.23 and Figure 4.24.

To show the extent to which TTS delivers optimal solutions of lower quality than FPS, the *Relative Difference (%)* of Figure 4.23 and Figure 4.24 is summarized in Table 4.5. The attribute *TTS (%)* shows the percentage of models for which TTS provides an optimal schedule of lower quality than the one in FPS. The *FPS (%)* indicates the opposite. The *Equal (%)* attribute shows that both scheduling mechanisms provide optimal solutions of the same quality. For instance, in Benchmark 1, TTS provides for 17.5 % of PTP models an optimal solution of lower quality than the one provided by FPS, and FPS provides a lower quality solution than TTS for 53.6 % of models. The impact of the feasible solution space on the solution quality is evident also when the results of PTP and SBP are compared. The rate of models that have a lower quality in TTS than in FPS is smaller in PTP than in SBP. In PTP, the feasible solution space is

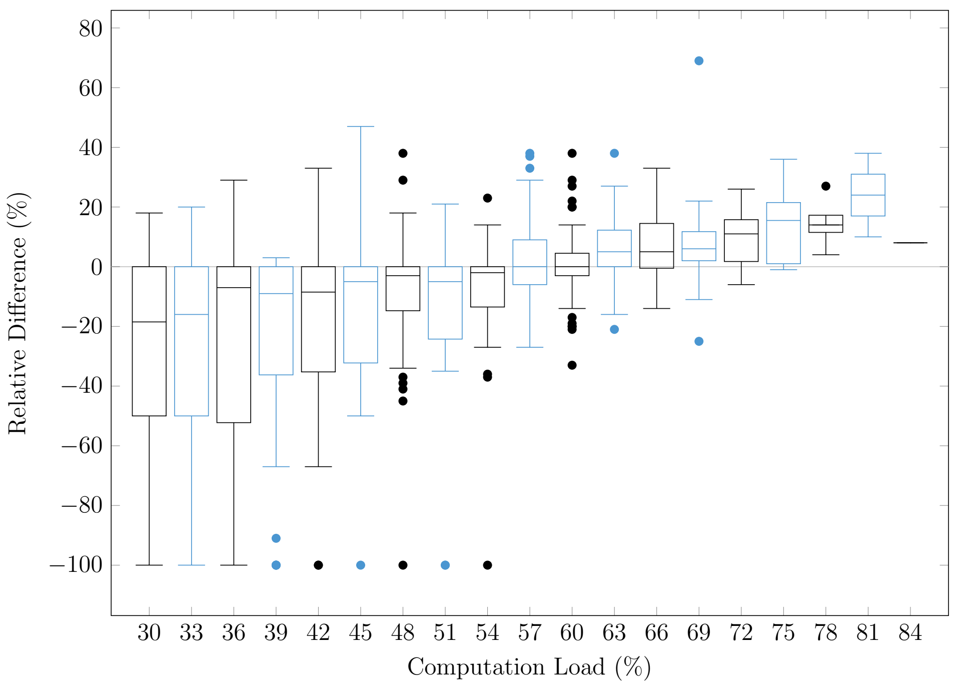


(a) SBP

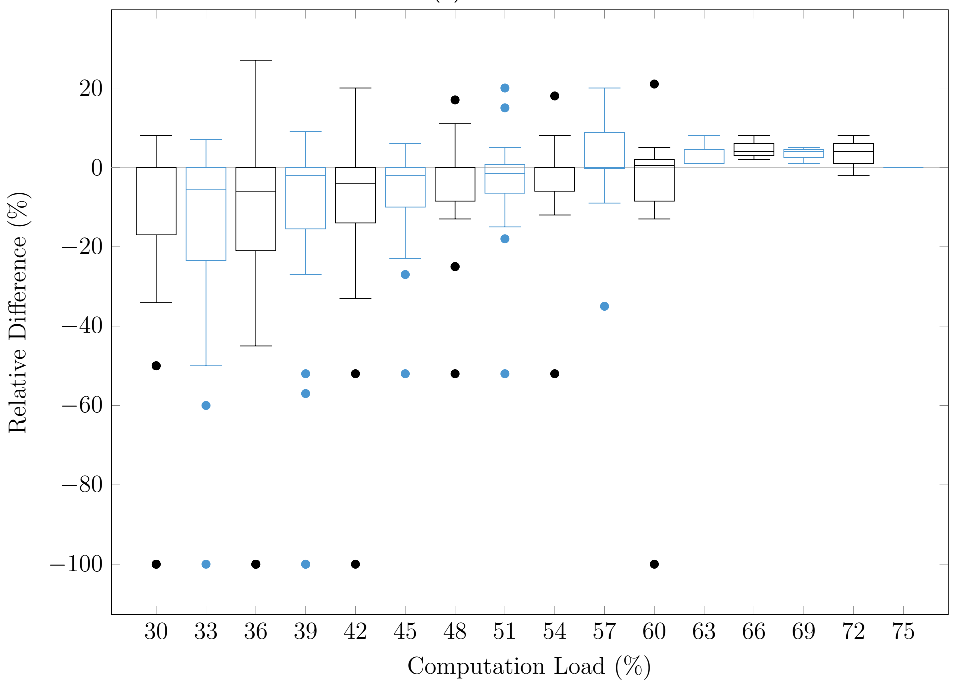


(b) PTP

Figure 4.23: *Benchmark 1* - Synchronous (LET = Period). The Relative Difference (%) of the preemption number of TTS and FPS schedule synthesis algorithms for the *optimal* found schedule.



(a) SBP



(b) PTP

Figure 4.24: *Benchmark 2* - Asynchronous (LET < Period). The Relative Difference (%) of the preemption number of TTS and FPS schedule synthesis algorithms for the *optimal* found schedule.

	Benchmark 1			Benchmark 2		
	TTS (%)	**FPS (%)**	**Equal (%)**	**TTS (%)**	**FPS (%)**	**Equal (%)**
SBP	17.5	53.6	28.9	27.1	50.1	22.8
PTP	10.8	60.2	29.0	20.7	49.5	29.8

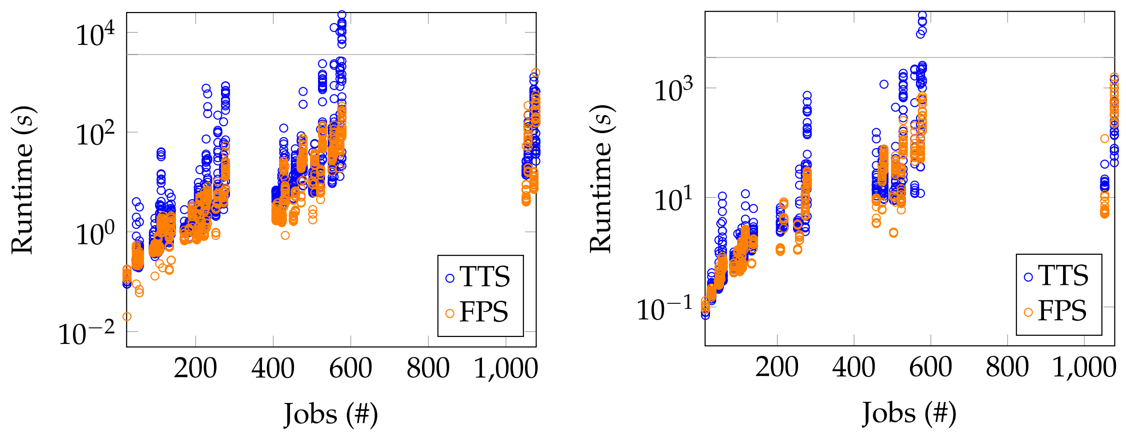
Table 4.5: Summary of Relative Difference (%) of the number of preemptions.

smaller than in SBP due to the strict communication requirements and high buffering and timer load.

These results show that TTS synthesis performs for the majority of models better than FPS synthesis in terms of optimal schedule quality. Although not fully shown by these results, it is expected that TTS synthesis provides equal or better optimization outcome than FPS in all cases because of the way TTS schedules tasks. To improve the search ability of the CP solver and the optimal solution quality of the schedule synthesis, several search heuristics and decision strategies can be enforced to the solver in OR-Tools. Based on our observations with several models from the model set, the search heuristics and strategies have a significant impact on the optimization results of TTS. Knowing which of the search and decision strategies lead to better results for TTS and FPS, an extensive experimentation for this specific problem is required. Identifying such strategies for both scheduling mechanisms is part of our future work. In this evaluation, the "automatic" search strategy is used to achieve a reasonable run-time of the solver, and no specific decision strategies are used.

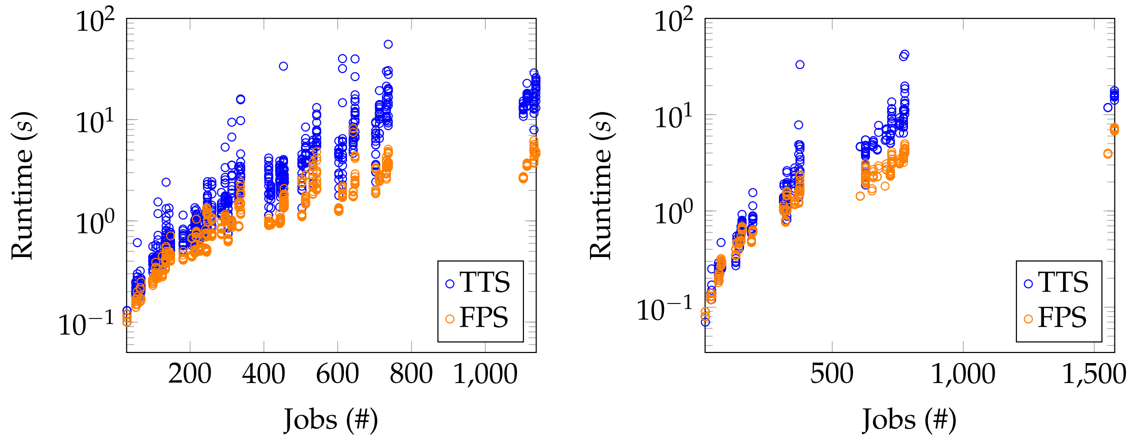
4.7.4 Performance

In this section, the run-time performance of TTS and FPS scheduling synthesis algorithms is discussed. The amount of time that each scheduling synthesis algorithm requires to find the *first feasible* and the *optimal* schedule is given. Figure 4.25 and Figure 4.26 shows the run-time results of the first *feasible* schedule of *Benchmark 1* and *Benchmark 2*, respectively. Only the results of models that provide a feasible schedule are shown. In graphs of Figure 4.26 less data are plotted than in graphs of Figure 4.25 because fewer models have a feasible schedule when PTP is used. The x-axis shows the *Jobs (#)*, which defines the sum of the number of computation blocks and of computation jobs that are solved by the synthesis algorithm. The *Jobs (#)* is chosen to show these results because they directly impact the number of variables and constraints in the CP program, and hence, the run-time performance. The y-axis shows the *Runtime (s)* in logarithmic scale. The *Runtime (s)* shows the amount of time that the CP solver takes to generate the computation schedule. Only this time is shown because the solver has the highest impact in the performance of both schedule synthesis algorithms.



(a) SBP (b) PTP

Figure 4.25: *Benchmark1* - Synchronous (LET = Period). The run-time performance of TTS and FPS schedule synthesis algorithms to provide the first *feasible* schedule. The gray horizontal line shows the bound of the reasonable amount of time of 3.600 s.



(a) SBP (b) PTP

Figure 4.26: *Benchmark 2* - Asynchronous (LET < Period). The run-time performance of TTS and FPS schedule synthesis algorithms to provide the first *feasible* schedule.

Although, the run-time behavior of the CP solver cannot be precisely evaluated due to its black-box implementation, the run-time of the proposed schedule synthesis algorithms is observed based on external influencing factors of the search algorithm's performance of the CP solver such as the application model's characteristics and complexity, which on the other hand define the size of the exploration and feasible solutions space. As expected, the increase of *Jobs (#)* increases the run-time of both schedule synthesis algorithms. However, the highest increase of the run-time is observed for models of the same amount of jobs. One reason this occurs is because models with different total load and equal number of jobs exist. Hence, in our understanding, when the load increases, the feasible solution space is lower, which means that the solver requires more time to search for a feasible solution, i.e., explores longer the search space and performs, e.g., more backtracking. However, although the run-time is generally increased with the increase of the load, one can notice that outliers exist and the increase is not continuously linear. Which leads us to the conclusion that the total load is one of the main influencing factors, but other aspects also play an important role on the run-time performance, such as, e.g., at which position on the search tree a feasible solution is found. This tendency is not explicitly drawn in Figure 4.25 and Figure 4.26, but can be observed based on the data given in Appendix A.2.

Did the scheduling synthesis algorithms provide a solution in a reasonable amount of time? Let 3.600 s be a reasonable amount of time to provide a solution by a CP program. In *Benchmark 1*, the TTS synthesis provides a first feasible schedule within this time for approximately 99 % of SBP and PTP models. The FPS synthesis provides a first feasible solution for all SBP and PTP models within this time. In *Benchmark 2*, both TTS and FPS provides the results in less than 60 s of time. In *Benchmark 2* overall less time is required to find a feasible schedule solution than in *Benchmark 1* for both TTS and FPS. This happens because in the asynchronous task sets, the constrained deadlines (LET duration < period) decrease not only the possibility of finding a feasible schedule but also the exploration space that is explored by the CP solver.

Which of TTS and FPS synthesis algorithms has a better run-time performance? Although it is challenging to draw a definite conclusion *if* and *why* FPS synthesis has a better run-time performance than TTS synthesis, or vice-versa, a rough comparison of their *absolute run-time difference* is given in Figure 4.27 and Figure 4.28. In all graphs, the *Ratio of models (%)* versus the *Absolute difference* of the run-times between TTS and FPS is shown. The absolute differences are grouped in ranges >0 s - 60 s, >60 s - 1 h, and >1 h. Since the majority of the absolute run-time differences of TTS and FPS are in the range >0 s - 60 s, i.e., differences are considered relatively insignificant, it can be inferred that both algorithms perform close to each-other in terms of run-time. In such case, it is challenging to identify why the small differences arise, which can be due to common operations in CP solver or due to exploration of the exploration space.
Benchmark 1 shows differences greater than zero between TTS and FPS for ranges >60 s - 1 h and >1 h. The results show that in absolute differences >60 s - 1 h, TTS takes more time to find the first feasible solution than FPS for 9.28 % of the SBP models and

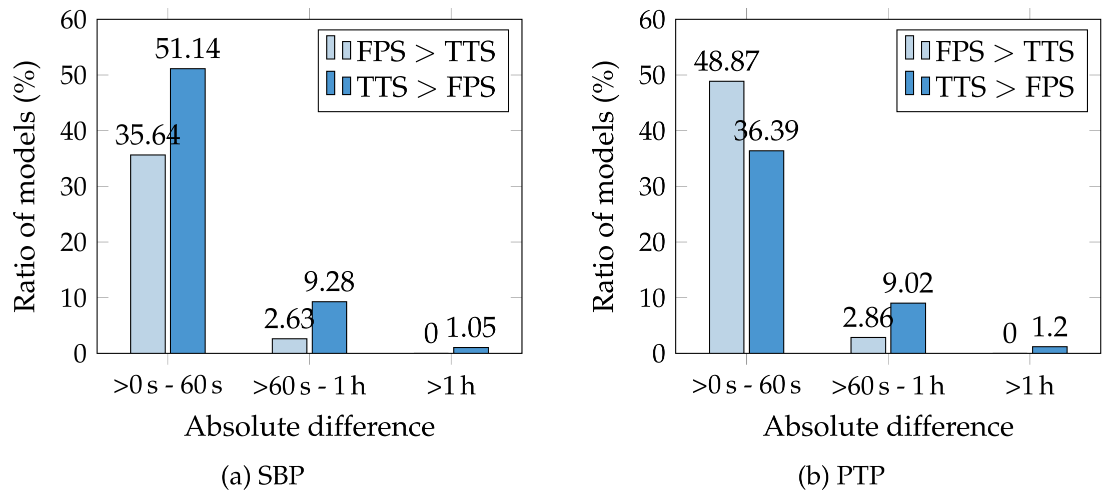


Figure 4.27: *Benchmark1* - Synchronous (LET = Period). The run-time comparison of TTS and FPS schedule synthesis algorithms for the first *feasible* schedule. Bars notated as *FPS* > *TTS* indicate that FPS requires more time to find a first feasible schedule than TTS and bars notated as *TTS* > *FPS* indicate the opposite.

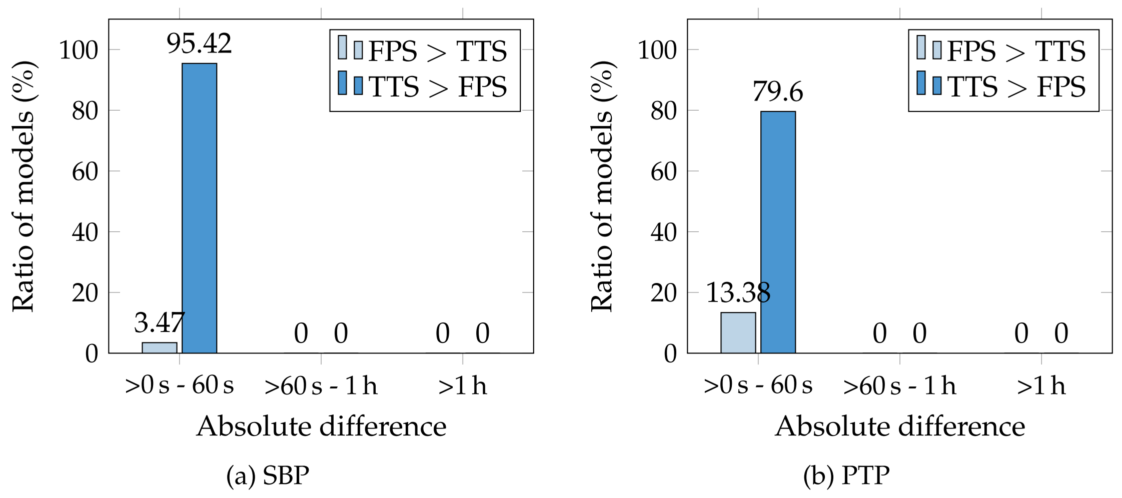


Figure 4.28: *Benchmark2* - Asynchronous (LET < Period). The run-time comparison of TTS and FPS schedule synthesis algorithms for the first *feasible* schedule. Bars notated as *FPS* > *TTS* indicate that FPS requires more time to find a first feasible schedule than TTS and bars notated as *TTS* > *FPS* indicate the opposite.

Runtime	**TTS-SBP (%)**	**FPS-SBP (%)**	**TTS-PTP (%)**	**FPS-PTP (%)**
0 s - 60 s	66.55	97.07	70.68	95.34
>60 s - 30 min	23.25	2.93	17.74	4.66
>30 min - 1 h	1.60	0.00	3.01	0.00
>1 h	8.61	0.00	8.57	0.00

Table 4.6: *Benchmark1* - Synchronous (LET = Period). The run-time of TTS and FPS schedule synthesis algorithms to provide the *optimal* found schedule.

Runtime	**TTS-SBP (%)**	**FPS-SBP (%)**	**TTS-PTP (%)**	**FPS-PTP (%)**
0 s - 60 s	97.08	100.00	96.32	100.00
>60 s - 30 min	2.78	0.00	3.68	0.00
>30 min - 1 h	0.00	0.00	0.00	0.00
>1 h	0.14	0.00	0.00	0.00

Table 4.7: *Benchmark2* - Asynchronous (LET < Period). The run-time of TTS and FPS schedule synthesis algorithms to provide the *optimal* found schedule.

9.02 % of the PTP models. In absolute differences >1 h, TTS takes more time for 1.05 % and 1.2 % of SBP and PTP models, respectively. For approximately 0.26 % of SBP models and 1.65 % of PTP models, TTS and FPS take the same amount of time. In our understanding, TTS has a bigger exploration space than FPS and if only few feasible solutions exists, it takes more time for the CP solver to find a first feasible solution. In TTS, compared to FPS, a higher preemption relation exists between overlapping jobs because they can preempt each-other at any position, which is not constrained by any priority ordering among them. In FPS, the exploration space is decreased by the symmetry constraints enforced by the priority ordering among jobs. However, as already emphasized, the run-time performance to find the first feasible solution is impacted, among several reasons, by the complexity of the application model defined by, e.g., the number of jobs, total load, and overlapping activation time intervals between jobs. A relationship of the run-time and application's characteristics could not be identified when TTS and FPS are compared. Finally, these times do not depend exclusively on the formalization of the described scheduling problems and on the studied application models, but as well on the CP-SAT solver of Google OR-Tools [162]. Therefore, it is expected that these times may vary when the proposed schedule synthesis approaches are implemented with a different solver.

What is the run-time of TTS and FPS synthesis algorithms to obtain an optimal schedule? The run-time results are shown in Table 4.6 and Table 4.7 for *Benchmark 1* and *Benchmark 2*, respectively. The run-time values are grouped in ranges >0 s - 60 s, >60 s - 30 min, >30 min - 1 h, and >1 h. The results of *Benchmark 1* show that FPS provides the optimal solution within the 30 min of time for all SBP and PTP models. The TTS provides the optimal solution within 1 h for up to 91.39 % and 91.43 % of the SBP and PTP models, respectively. For 8.57 % to 8.61 % of the models, TTS needs more than 1 h to provide the optimal solution. These results show that in TTS a larger number of models than in FPS requires more than 30 min to find the optimal solution. In TTS, the time limit of 6 h is reached for 6.39 % of the SBP models and for 5.56 % of the PTP models. In our understanding, this is due to the larger feasible solution and exploration space of TTS compared to FPS. Although the differences are less significant, the same observation holds also for the *Benchmark 2* results, except that none of the models reaches the 6 h time limit. Comparing the run-time performance of *Benchmark 1* and *Benchmark 2*, one notices that in the second case both TTS and FPS provide the optimal solution within the 1 h time for 99.86 % to 100.00 % of models, which also results due to a smaller feasible solution space.

4.7.5 Conclusions

The results of this evaluation demonstrate that the LET buffering overheads have a significant impact on the schedulability of the application. Hence, SBP is a convenient buffering approach to integrate LET also in terms of scheduling. PTP provides poorer schedulability than SBP because it has a higher buffering and timer load than SBP and stringent communication requirements for scheduling. Therefore, adding new functionalities to the application requires more resources in PTP compared to SBP in order to ensure a feasible schedule. The low scheduling performance of PTP is also observed when timing information such as the duration of LET intervals is reduced, which is usually needed to reduce the data ages and end-to-end delays.

Although both scheduling mechanisms demonstrate comparable feasibility capabilities, TTS shows that it finds a feasible schedule also when one does not exist for FPS and provides overall a schedule with lower number of preemptions. In general, TTS offers other advantages that surpass the capabilities of FPS, such as deterministic task execution and better resource utilization by avoiding peak loads, which is crucial for LET applications. Because FPS is already integrated into existing automotive OSs, it is typically the default approach chosen to schedule LET applications, which is acceptable as long as hardware resources are not limited. The integration of TTS into automotive systems is not only suitable for LET applications, but also for non-LET applications to ensure deterministic end-to-end delays across different ECUs.

As described in this chapter, the automatic schedule generation of in-vehicle applications is highly necessary to speed up their development time. The performance evaluation results have shown that the proposed schedule synthesis algorithms provide a feasible schedule in a reasonable amount of time for 99 % – 100 % of the studied models. Therefore, integrating the proposed algorithms into an automated development process is a solid practice. Although the reasonable time limit may be exceeded for model applications of certain complexity, executing constraint programming algorithms in future quantum computers [164] is a promising way to improve the run-time performance of CP solvers. Furthermore, as shown in the optimization results, especially of TTS, the search algorithm of the CP solver and the size of solution space have a significant impact on the quality of the optimal solution and the run-time performance of the synthesis algorithms. Therefore, to improve this, a combination of *reinforcement learning* and *constraint programming* [165] is an attractive approach to achieve the desired solution quality. Alternatively, the most suitable search heuristics and decision strategies for this specific problem can be applied.

5 | Realization in AUTOSAR Systems

This chapter describes the integration of the *Logical Execution Time (LET)* into the software architecture of the *classic platform* of *AUTomotive Open System ARchitecture (AUTOSAR)*. The successful integration of LET is demonstrated by a case study using a real world *Antilock Braking System (ABS)*.

5.1 Integration in Software Architecture

The following sections describe the methodology of integrating *Point-to-Point Protocol (PTP)* and *Static Buffering Protocol (SBP)* into AUTOSAR software architecture and aspects that impact the determinism of LET in highly event-driven AUTOSAR systems.

5.1.1 Methodology

In AUTOSAR, the LET paradigm is specified for the *implicit* Sender-Receiver communication between periodic runnables of different *Software Component (SWC)*s. Thus, when runnables are mapped to LET intervals, their implicit Sender-Receiver accesses use the semantics of LET. The *Runtime Environment (RTE)* enables the communication between SWCs by providing the actual implementation of the different AUTOSAR communication mechanisms (as shown in Section 2.2). Therefore, the PTP and SBP mechanisms are integrated into the RTE layer of AUTOSAR to enable the use of LET semantics between SWCs (as depicted in Figure 5.1). Whenever the embedded code of the RTE is generated, e.g., by means of tools, the necessary data structures and the respective *Application Programming Interface (API)*s are generated to accomplish the LET buffering behavior. Although PTP and SBP can be integrated into an AUTOSAR system in different ways, following their definition in this work, they are integrated as follows. In PTP, the RTE generates for each LET interval and for each implicit Sender-Receive communication between SWCs the global variables, buffers, and the respective copy-in and copy-out operations as dedicated LET buffering runnables. Because activation of runnables is enabled by RTE events, RTE also creates the events to periodically activate the LET runnables based on the timing information of their respective LET intervals, such as offsets and periods. Whenever necessary, the RTE also generates the spin-locks and interrupt locks to ensure consistent and stable data accesses within the LET buffering runnables. These runnables need to be mapped to dedicated *Operating System (OS)* tasks, which in AUTOSAR are scheduled via

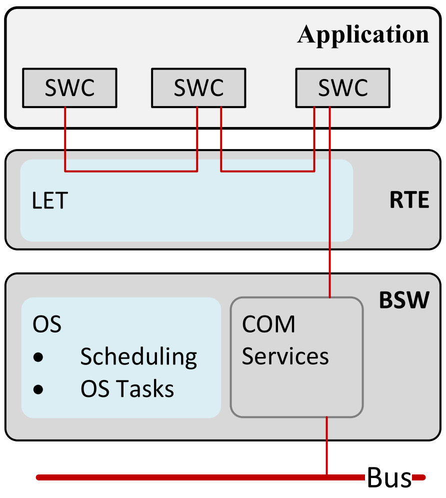


Figure 5.1: Integration of LET into the AUTOSAR Software Architecture

the *Fixed-Priority Scheduling (FPS)*. As for PTP, the RTE generates for SBP the global buffers, the buffer schedule, and the corresponding APIs for each runnable to access the assigned buffer elements. If runnables of a LET interval send data to runnables of another *Electronic Control Unit (ECU)* via LET semantics, the RTE calls inside the code of the LET runnables the respective APIs of the communication services of AUTOSAR. In case of SBP, the communication APIs are called by the computation runnables at the end of their execution. The explicit and implicit Sender-Receiver accesses for runnables that are not mapped to LET intervals are not affected by LET buffering.

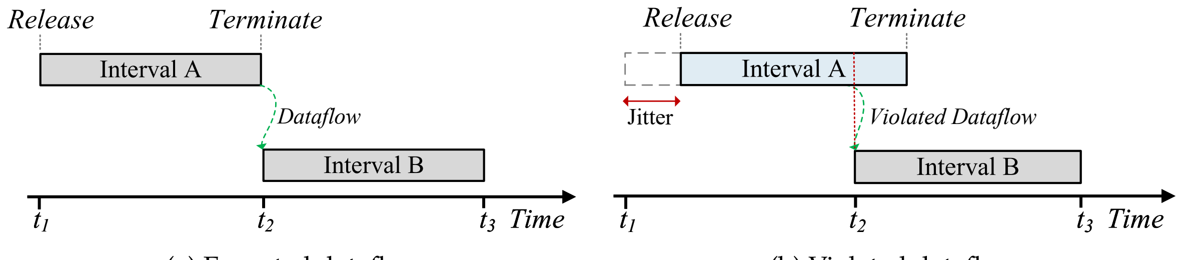
The LET paradigm can be applied to AUTOSAR systems only for a subset of global data elements for the following reasons. Typical in-vehicle applications have sporadic or a-periodic runnables that require a faster input than other runnables such as *crank angle* runnables of an *Engine Management System (EMS)* system. LET increases the duration of the control flow and shifts the time of delivering outputs, and, hence, the end-to-end delay requirements cannot be always satisfied in a highly event-based and loaded system, even if the duration of LET intervals is reduced to be less than their periods. Furthermore, due to the high data dependency between runnables, it is challenging to derive a LET design that simultaneously provides faster outputs and satisfies the data ages and end-to-end delay requirements for all Sender-Receiver communications between runnables [73]. Moreover, LET buffering has additional challenges and constraints in AUTOSAR systems. As described throughout this work, SBP can be applied only for Sender-Receiver communication between periodic runnables that access the data implicitly. A mixed LET and non-LET Sender-Receiver communication of the same data elements, be it implicit or explicit, is not possible in SBP, because runnables that read data via non-LET communication cannot know which buffer element to read from. This limitation of SBP is due to its global buffering methodology. Therefore, to reduce the complexity of handling such scenario in SBP, the PTP can be used in an AUTOSAR system, in addition to SBP, if there is at least one runnable that consumes a data element explicitly or via non-LET communication.

5.1.2 Determinism of LET

AUTOSAR systems are highly event-driven. A key challenge of integrating LET buffering in event-based AUTOSAR applications, especially in multi-core platforms, is guaranteeing time, value, and dataflow determinism. In SBP, determinism and buffering correctness is violated when tasks are not activated and executed according to their defined periods, LET intervals, and bounded activation jitters. In this case, scheduled accesses to buffers by tasks may differ from the buffer schedule planned at design time. This not only violates determinism but also can lead to data stability and consistency issues when accessing shared buffers. Determinism of LET is also affected when PTP is applied. The unpredictable activation jitters also affect the scheduling design of such systems. This means that the schedule planned at design time may deviate from the one on real target if tasks are not activated at expected times.
To ensure a correct execution behavior of LET buffering, the determinism of LET, and predictable scheduling on real target, dedicated mechanisms must be employed, such as reduction and bounding of activation jitters, monitoring of LET overruns at run-time, and safe system restart in case of timing violations. In the following sections, two mechanisms are described to ensure the above aspects of LET to an extent: the synchronization of LET intervals and the controlling of LET overruns.

Synchronization of LET intervals

Synchronization of LET intervals is necessary to reduce activation jitters such that these intervals occur at predicted times and the dataflow between them is guaranteed when the system runs on target. Regardless of the buffering protocol, the physical occurrence of the release and termination events of a LET interval correspond to the respective activation time and absolute deadline of the computation task. It should be noted that a LET interval is a logical time frame associated with a computation task and does not physically occur at execution time unless intentionally tracked by special interrupts. Hence, synchronization of LET intervals refers to the synchronization of activation times of their respective computation and communication tasks.
Figure 5.2 shows an example of the effects that activation jitters have on dataflow determinism between LET intervals. Figure 5.2a shows the expected dataflow between LET *Interval A* and *Interval B*. At time t_2, the outputs of *Interval A* are expected to be available such that *Interval B* can consume them immediately after *Interval A* terminates. In target execution, shown in Figure 5.2b, a long activation jitter occurs for *Interval A*, which shifts the terminate event of *Interval A* beyond time t_2 and violates the dataflow between intervals. Although this example is only an abstraction of the effects that activation jitters have on dataflow determinism between LET intervals, in the case of SBP and PTP, it is crucial to ensure that tasks do not execute during the unpredicted overlapping time frame between LET intervals. As mentioned earlier in this section, for SBP, concurrent execution of computation tasks during the overlapping interval is critical because it can affect the correctness of the buffering behavior. Therefore, the



(a) Expected dataflow. (b) Violated dataflow.

Figure 5.2: Synchronization of LET intervals in AUTOSAR systems. The LET interval is associated with a computation task. The release time of a LET interval corresponds to the release time of the computation task. In **(a)**, the dataflow is defined assuming zero activation jitters of the computation task, i.e., LET interval. In **(b)**, activation jitters occur during execution of tasks on target, which violates the expected dataflow.

activation of these tasks must be synchronized and the jitters must be reduced to a degree where the impact is insignificant and predictable during the design of the LET buffering and schedule synthesis.

To increase the activation precision of tasks running on the same core, the schedule tables of AUTOSAR OS can be used to activate tasks. In the previous chapter, such table is referred to as activation table to distinguish it from the schedule table of the *Time-Triggered Scheduling (TTS)*. In multi-core systems, to synchronize the physical occurrence of LET intervals of different cores, i.e., the corresponding computation and communication tasks, the synchronization of schedule tables of different cores is necessary. In this case, activation precision of tasks is affected by the synchronization precision of schedule tables. AUTOSAR does not specify explicit mechanisms for synchronizing schedule tables of different cores. The following two approaches are described to enable the synchronization of tasks of different cores.

(i) *Approach 1* – synchronizes only the start time of the schedule tables of different cores. Their synchronization is enabled via OS barriers at the initialization, i.e, startup phase of each core. In this case, the schedule tables are only started after both cores have finished their initialization phase and at the same point of time. This synchronization is only applicable if the hardware supports the synchronization of hardware timers. This implies that the timers that drive the schedule tables run on the same clock tick. The drawback of this approach is the active waiting between cores during the synchronization of barriers, especially in case of long start up phase or preemption delays of the startup routines. Additionally, because only the start time of the schedule tables is synchronized, delays on activation times at different expiry points can occur in case the timer interrupt is delayed or preempted by interrupts of higher priority. These delays can be reduced if the timer interrupt is defined to run on the highest interrupt level inside the OS, e.g, in the level of the timing protection interrupt [19].

(ii) *Approach 2* – synchronizes the activation of tasks of different cores by activating them using a global schedule table defined for all cores [19]. In this case, the activation of tasks of other cores is triggered by the timer interrupt of the core in which the schedule table is defined. The timer interrupt initiates a cross-core communication and the task is activated by the cross-core interrupt of the remote core. The drawback of this approach is the overhead for the cross-core communication and the violation of the "free from interference" requirement in case of safety systems. Therefore, it can be used only for applications of the same safety level. In this approach, activation delays may also occur if the timer and cross-core interrupts are delayed or preempted by higher priority interrupts.

A schedule table is driven by a timer. Hence, another way to increase further the task activation precision, such that it does not have major deviations to its initial configuration, is to use a periodic timer to activate tasks. Such timers are more accurate at determining the activation time of tasks. But their number is often limited and cannot be applied when the application contains hundreds of LET intervals with non-harmonic periods and duration. In case of non-harmonic periods, their use comes at the cost of a higher execution load because a minimal period for the timer has to be specified such that activation of all tasks is ensured. Therefore, the high-resolution timer is typically used in such systems. High-resolution timers minimize the effects of the high execution load of periodic timers in systems with non-harmonic periodic tasks, but in practice do not enable high task activation accuracy and have a higher execution load per instance of the timer interrupt. This is typically due to their internal algorithm of defining the next activation point. The synchronization precision of schedule tables across different ECUs also affects the activation precision of tasks. However, this work focuses only in one ECU.
It is crucial to bound the activation jitters during development time and to incorporate them into LET buffering design and scheduling synthesis, such that unpredictable dataflow does not occur during system's run-time. Such upper-bounding is usually possible at the development time through debugging and measurements of the AUTOSAR systems.

Controlling of LET Overruns

Regardless of the buffering protocol, overruns of LET intervals must be prevented to avoid violating the data flow determinism of LET. As with LET interval synchronization issues, preventing LET overruns is crucial for the correct functioning of the LET buffering behavior. Therefore, if a LET overrun is detected, then the task exceeding the LET interval must be forcibly terminated. In the development phase of AUTOSAR systems, LET overruns are prevented by constructing the schedule of tasks such that they execute within the boundaries of their LET intervals. However, LET overruns may happen when the system is running on target if unpredictable execution delays occur at run-time. If the TTS is integrated in an AUTOSAR OS, specific mechanisms must be implemented to prohibit the execution of tasks outside of the allocated time intervals in the schedule table. In FPS, existing AUTOSAR mechanisms can be used.

Two aspects are important for preventing LET overruns in AUTOSAR systems. Firstly, identifying that a LET overrun occurs and, secondly, taking an action to the detected overrun. LET overruns may be treated as deadline violations. AUTOSAR does not provide explicit deadline monitoring of tasks, but instead it defines timing protection mechanisms to prevent timing faults [19]. These mechanisms can be applied for LET applications to avoid LET overruns for the cases when these overruns occur due to (1) unpredictable long execution times of LET and non-LET tasks and *Interrupt Service Routine (ISR)*s, (2) early and unplanned arrival of certain tasks, and (3) unpredictable long critical sections and long disabling of interrupts. Hence, an *execution budget*, a *lock budget*, and an *inter-arrival time frame* can be specified for tasks and ISRs. If a violation is detected, then the faulty task or ISR is forcibly terminated. The main drawbacks of using timing protection mechanisms is the increased processing load and the forced termination of tasks. The later case is often not desired for several reasons. If the forcibly terminated task is a LET task, then the expected outputs may not all be written to the buffers, which affects the value determinism of LET. In this case, the next occurring reader LET intervals would read an old version of data. However, this is a trade-off, especially in SBP, as long as the correctness of the buffer accesses is guaranteed. Measurement and testing tools are also a practical approach to monitor LET overruns during the development phase.

5.2 Case Study: Antilock Braking System (ABS)

The realization of LET in AUTOSAR architecture is shown using a real ABS running on an AUTOSAR ECU. The abstraction of the ABS used in this case study is shown in Figure 5.3. The purpose of an ABS is to ensure safe steering of the vehicle by reducing the slip and preventing skidding and wheel lock in the event of high brake force, regardless of road conditions. An ABS adjusts the brake pressure to keep the slip in a certain ratio and thus shorten the braking distance. An important component of an ABS is the controller, which reacts to vehicle sensor data, such as the wheel speed, vehicle speed, accelerations, and pedal and steering angle, and calculates the slip ratio and the brake force pressure. The slip ratio is calculated based on two control variables: the vehicle and wheel speed and is later used to calculate the brake force pressure. The system responds by applying this force to the wheels, turning on the brake lights, and sending a throttle request to the engine to prevent further acceleration. When a certain vehicle acceleration is reached, then the full stop of the vehicle is enforced.

The application shown in Figure 5.3 consists of various SWCs and software compositions that encapsulate the functionalities described above. The dataflow between SWCs is indicated by arrows connecting different SWCs. The software composition *INPUT* contains SWCs that receive sensor data from the bus and check their plausibil-

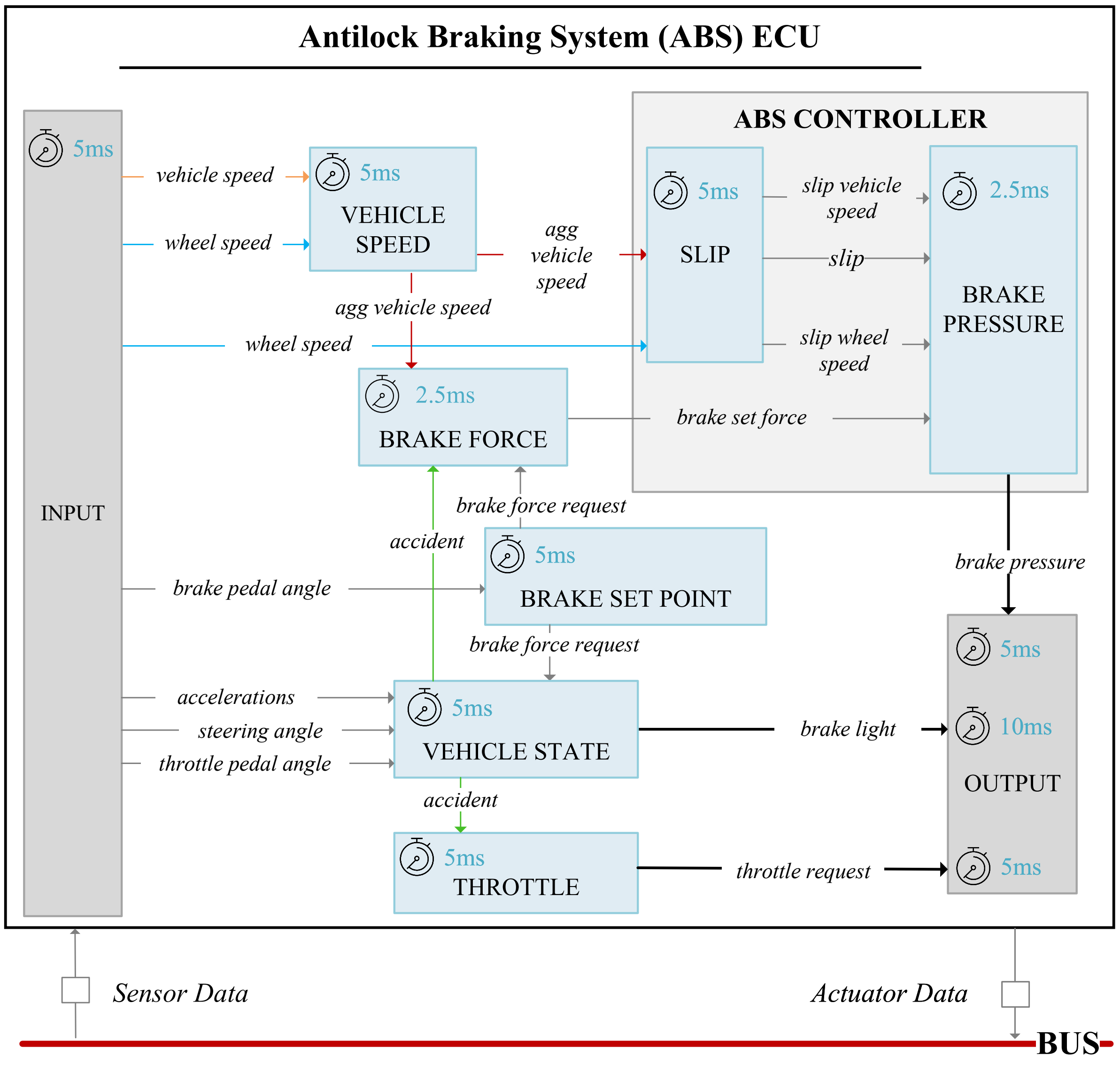


Figure 5.3: Abstraction of the Antilock Braking System (ABS) components. The INPUT and OUTPUT software compositions contain the SWCs that process sensor and actuator data, respectively. The light blue boxes represent the main application SWCs that implement the functionalities of the ABS. Three main outputs are provided to the actuator: the brake pressure, brake light activation, and throttle request. The arrows marked with red, yellow, green, and light blue indicate that the variables consumed by the SWCs must have the same data age, i.e., received and computed based on the same sensor data and of the same sampling time. The stop watch icon with the time in ms indicates the periodic activation of runnables inside the SWCs and compositions. In the OUTPUT composition, sending brake pressure and throttle request outputs is enabled by runnables activated every 5 ms, and sending of brake light by runnables activated each 10 ms.

ity. The processed sensor data is then stored in special global variables such as *vehicle speed*, *wheel speed*, *brake pedal angle*, *accelerations*, *steering angle*, and *throttle pedal angle*. This sensor data is later used by various components to calculate slip, brake pressure, a request for brake force, detection of a possible accident, rear light activation, and a throttle request to prevent acceleration. The software composition *OUTPUT* contains SWCs that convert into correct physical values the generated outputs such as brake pressure, brake rear light activation, and throttle request and send them to the actuator.

An important requirement of the ABS is to provide correct functionality, e.g., correct brake force pressure in a short amount of time. This means that the delays between the received sensor values and the provided actuator values must be as short as possible. This delay is referred to as end-to-end delay. The functional correctness of the ABS controller is affected by the occurrence of, e.g., poor slip response due to different data ages of the wheel and vehicle speed inputs. Several design configurations can increase the end-to-end delay and impact the functional correctness of the ABS. In multi-core systems, functional correctness is hard to achieve because the execution order of runnables involved in a dataflow must be handled explicitly if they are allocated to run on different cores. Approaches such as synchronization of runnable's execution among cores by means of, e.g., OS barriers are inefficient in terms of overheads and difficult to maintain in different software increments. Therefore, the LET paradigm is applied to the ABS of this case study to ensure correct dataflow and functional behavior without enforcing a specific execution order between runnables, regardless of which core they run on. In this case, the dataflow and data ages between runnables must be ensured by the design of LET intervals [73]. If LET intervals are not created to guarantee timing requirements, dataflow and data ages between SWC, then functional correctness of the system cannot be ensured.

The real world ABS used in this case study is of limited complexity and enables a straightforward demonstration of the integration and practicality of the LET paradigm and the synchronization aspects of task activation using the approaches described in the previous section. The PTP is used for LET buffering since it is integrated in MICROSAR RTE from Vector Informatik GmbH [1]. The practicality of LET and the effects of synchronization on buffering and functional correctness shown using ABS are feasible for both SBP and PTP.

5.2.1 Configurations

The ABS application is run on a real platform composed by an Infineon Aurix TC39x processor [166] and the MICROSAR OS/Stack of Vector Informatik GmbH [1]. The system is traced using the iC5700 debugger of iSystem [167] and the real measurement data are evaluated in TA Tool Suite of Vector Informatik GmbH. Tasks of the ABS are allocated to two different cores and the LET paradigm is used to exchange data for

[1]www.vector.com

16 data elements exchanged between 16 SWCs. The configuration of LET intervals is performed such that the correct dataflow, data ages, and the timing between SWCs of the same and different cores is ensured. The application after configuration of LET is composed of 11 LET intervals, 11 computation tasks, and 16 communication tasks. Each computation tasks is associated to one LET interval. Each LET interval contains runnables of equal periods belonging to the same SWC and the duration of the LET intervals is defined equal to the period. For each application SWC, one interval is created to simplify the LET design and apply the semantics of LET only for the Sender-Receiver communication between different SWCs as specified by AUTOSAR. Activation offsets of LET intervals are set to zero to observe the effects of activation jitters when multiple tasks are activated simultaneously. Runnables of the same SWC are mapped to one task to ensure the dataflow of the Inter-Runnable communication. AUTOSAR does not specify LET for the Inter-Runnable communication [46].

This case study applies LET for the dataflow between SWC that provide the following functionalities of the ABS:

- ▷ *Brake light activation* – the dataflow is defined between LET intervals: Input, Brake Set Point, Vehicle State, and Output Light
- ▷ *Throttle request to the engine* – the dataflow is defined between LET intervals: Input, Vehicle State, Throttle, and Output Throttle
- ▷ *Brake pressure calculation* – the dataflow is defined between LET intervals: Input, Vehicle Speed, Brake Set Point, Brake Force, Slip, Brake Pressure, and Output Pressure

Figure 5.4 depicts the dataflow between LET intervals for each of the aforementioned functionalities of the ABS. As indicated by the red arrows, the dataflow and data ages for the *brake pressure calculation* are ensured because computation of intermediate results and final output is performed based on sensor input data of the same sampling time. The same applies for the *throttle request* and *brake light activation* functionalities. This design of LET intervals guarantees functional correctness and control quality of the ABS. In this case, the end-to-end delays for *brake light activation*, *throttle request*, and *brake pressure calculation* functionalities are constant and consist of 25 ms, 20 ms, and 25 ms, respectively. The described design of LET intervals guarantees functional correctness at design time by itself, but this end-to-end delay and dataflow is expected when the system is running on the real target. This is ensured by synchronizing activation times of LET intervals and preventing long activation jitters.

The ABS is run on the target and traced for three configurations. In the *Unsynchronized* configuration, the synchronization of the activation of LET tasks on different cores is not enabled. However, tasks of the same core are synchronized by activating them via a schedule table created for each core. In the second configuration, the *Approach 1* is used to synchronize the start of the schedule tables of each core, and, hence, synchronize the activation of LET tasks of different cores. In the third configuration,

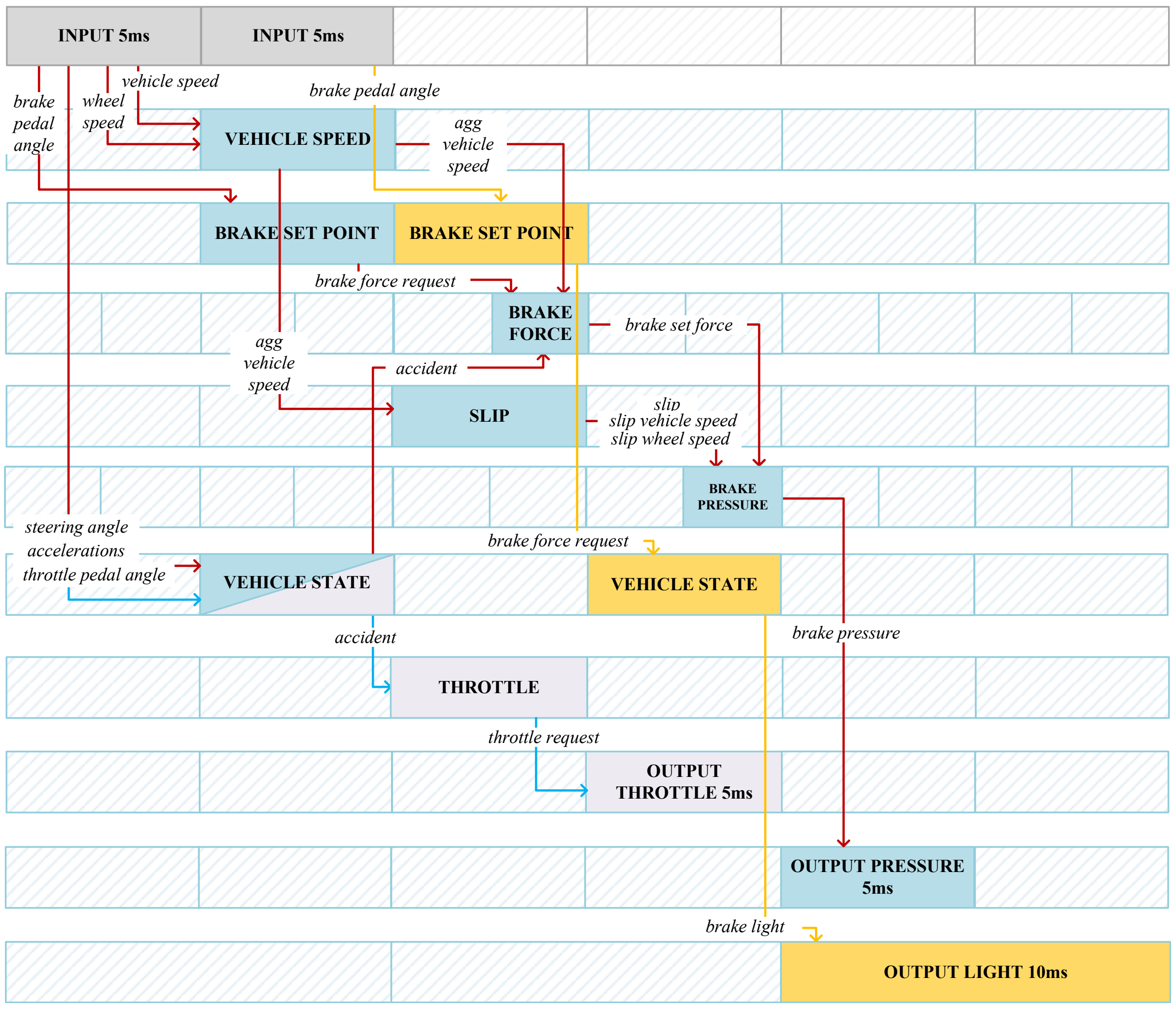


Figure 5.4: The dataflow between LET intervals of the ABS. The highlighted gray, blue, yellow, and purple boxes indicate the occurrences of LET intervals involved in the dataflow for the *brake light activation*, *throttle request*, and *brake pressure calculation* functionalities. The pattern filled gray and blue boxes indicate occurrences of LET intervals not involved in the highlighted dataflow. The dataflow of the *brake pressure calculation* functionality is marked by the red arrows connecting the respective LET intervals. The dataflow of the *brake light activation* and *throttle request* are marked by yellow and blue arrows, respectively.

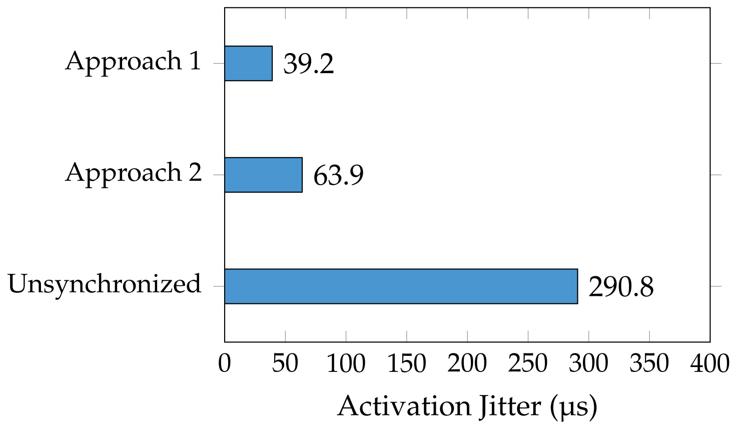


Figure 5.5: The maximal activation jitter between cores.

the *Approach 2* is used to synchronize the activation of LET tasks of all cores. In this case, one global schedule table is created to activate tasks. Because the timer interrupt of the master core, in which the global schedule table is defined, initiates the activation of other tasks to the remote core by triggering a call of the cross-core interrupt, this request is performed first and then tasks of the master core are activated. In this way, the activation of tasks occurs in parallel between the cores and the synchronization precision is reduced. In all configurations, the schedule tables are driven by a high-resolution timer.

5.2.2 Results

This section describes the evaluation results for the synchronization jitters of LET intervals, functional correctness, and timing of the ABS in LET.

5.2.2.1 Synchronization of LET Intervals

Synchronization accuracy between cores is measured as the time difference between the activation time of the first activated task on one core and the activation time of the last activated task on the other core. Synchronization accuracy is also referred to as activation jitter. The accuracy is measured only for LET Start and computation tasks because their activation represents the release time of their respective LET intervals. Figure 5.5 shows for each configuration the maximal activation jitter between the two cores. The maximal jitter occurs at activation points in which all LET Start and computation tasks are activated. Because the application has LET intervals of different periods, a different number of tasks are activated at different activation times. At most 12 tasks on Core 0 and 10 tasks on Core 1 are activated simultaneously. The results show a significant activation jitter between cores for the *Unsynchronized* configuration. This time exceeds the maximum net execution time of each core's timer

	Approach 1	**Approach 2**
Activation overhead (%)	7.62	8.82
OS barriers overhead (%)	0.005	–

Table 5.1: Task activation and synchronization run-time overheads.

interrupts, which are 54.5 µs and 59.2 µs for Core 0 and Core 1, respectively. This activation jitter causes the LET intervals of different cores to overlap incorrectly, thus affecting the dataflow and time determinism. This jitter is significantly improved when synchronization is applied for *Approach 1* and *Approach 2* configurations. In both cases, the maximum activation jitter does not exceed the time between the start of the timer interrupt of one core and the end time of the timer or cross-core interrupt of the other core. Because of the synchronization, these interrupts execute simultaneously on their respective cores. An absolute zero activation jitter cannot be achieved because the activation time of a task corresponds to the time that the timer interrupt sets the task in active state. If multiple tasks are activated in one expiry point of the schedule table, then the timer interrupt iterates over the list of tasks that must be activated and sets them to the active state. This also implies that the lower the number of tasks to be activated in an expiry point, the lower is the activation jitter of the tasks and the execution run-time of the timer interrupt. As long as these jitters and activation of tasks are bounded within the execution times of the timer and cross-core interrupts, the incorrect overlapping of LET intervals is avoided and buffering correctness is ensured in both PTP and SBP.

The activation jitter is higher in *Approach 2* than in *Approach 1* because when a global schedule table is used to activate tasks of all cores the cross-core communication delays are involved. In this evaluation, the high-resolution timer is not delayed or preempted by other interrupts. In systems with a large number of hardware interrupts that cause considerable delays of the timer interrupt, these jitters can be significantly higher in both approaches. The applicability of LET in such systems is then questioned by the extent to which these delays can affect the determinism requirement of LET and the synchronization of LET intervals.

Table 5.1 shows the run-time overheads associated with task activation and synchronization. The run-time overheads for task activation refers to the execution utilization of the timer interrupts and the cross-core interrupt (in the *Approach 2* only). As already mentioned, these overheads are not only dependent on the internal algorithm of these interrupts, but also on the number of tasks to be activated. They also dependent on the implementation of the OS and may vary between different versions or vendors of the OS. In this evaluation, the run-time activation overheads for *Approach 1* and *Approach 2* are 7.62 % and 8.82 %, respectively. A slightly higher activation utilization occurs for the *Approach 2* because of the cross-core communication. In *Approach 1*, an insignificant amount of 0.005 % execution load occurs in addition during the initialization phase

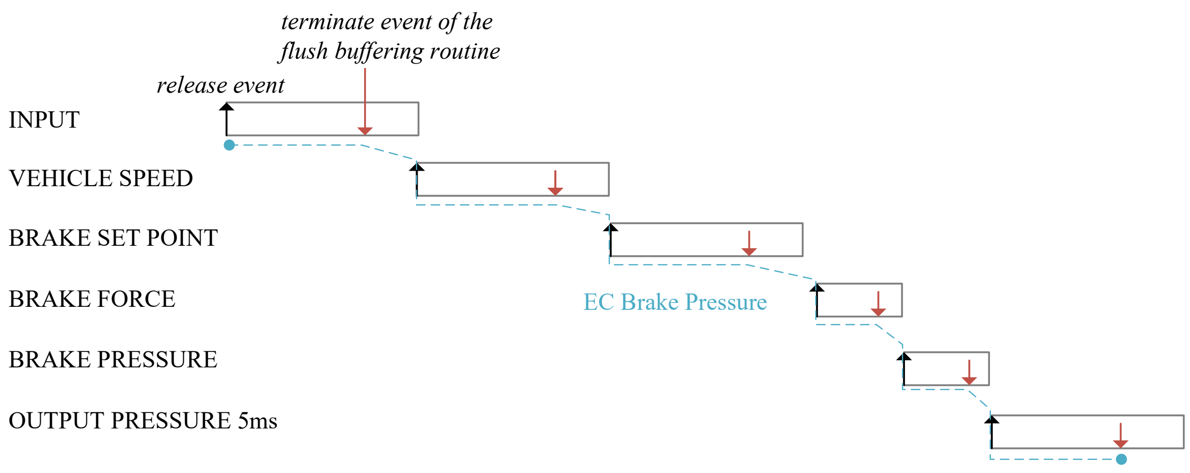


Figure 5.6: Event chain definition for the *brake pressure calculation* functionality. The blue dashed line represents the event chain of the dataflow between LET intervals illustrated by the white boxes.

due to the OS barrier synchronization. This corresponds to the active waiting time of the initialization task on Core 1 until the initialization task of Core 0 has completed all operations until barrier synchronization.

5.2.2.2 Functional Correctness and Timing

Considering the achieved synchronization precision between LET intervals and the configured LET design, the end-to-end delays, i.e, timing and the dataflow between application SWCs of the ABS are fulfilled. To evaluate the functional correctness and timing, the dataflow between LET intervals involved in each ABS functionality is evaluated by means of event chains [59]. An event chain is created for each functionality and their maximal duration is calculated based on the generated measurement traces of the ABS. The stimulus event of each event chain corresponds to the release event of the *Input* LET interval and the response event to the terminate event of the copy-out buffering routines of the respective *Output* LET intervals. To show the definition of event chains, the event chain for the *brake pressure calculation* is shown in Figure 5.6 as an example. Because the PTP is used in this evaluation, the actual flushing of the data to the buffers occurs “close” to the terminate event of the *Output* LET intervals. Hence, to capture the actual dataflow, the event chains are defined between release events of LET intervals involved in the chain and the terminate event of their respective copy-out buffering routines. An event chain is a composition of segments defined between each LET interval involved in the dataflow.

Table 5.2 provides the duration of the event chains of each functionality for each synchronization approach. The results show that the event chain duration of each functionality varies between synchronization approaches. This happens because, as shown in the previous section, the activation jitters between LET intervals are higher

Event Chain	**Duration Approach 1** (ms)	**Duration Approach 2** (ms)
EC Brake Pressure	24.047	24.125
EC Brake Light	24.065	24.129
EC Throttle	19.051	19.133

Table 5.2: The event chain duration of the dataflow between LET intervals.

in *Approach 2* than in *Approach 1*. In PTP, the end-to-end delay of a dataflow is not absolutely constant but is defined by the activation offsets of the copy-out buffering routines mapped to LET End tasks. This is shown by these results, in which the event chain duration of *brake light activation*, *throttle request*, and *brake pressure calculation* functionalities does not exceed their respective 25 ms, 20 ms, and 25 ms delays defined by the design of LET intervals. Overall the results show that the control quality is ensured because the dataflow and execution order between LET intervals is fulfilled and the slip value, the vehicle speed, and the wheel speed have the same data age when they are used to calculate the brake pressure. This is achieved by synchronizing LET intervals and by the LET buffering mechanism.

As described throughout this work, in PTP the actual flushing of the data is defined by the activation offsets of LET End tasks. Therefore, the dataflow and event chain duration between LET intervals can be ensured also in the *Unsynchronized* configuration if these offsets are assigned to define a correct activation order between LET End task of the producer LET interval and the LET Start task of the consumer LET interval. However, this approach is not recommended as it increases the effort and the complexity of verifying the dataflow and end-to-end delays between LET intervals. Furthermore, in systems of high complexity, relative to the ABS used in this case study, the synchronization jitter in the unsynchronized configuration is expected to be greater than 290.8 µs, which may result in an offset assignment that cannot satisfy the dataflow between LET intervals. Thus, looking to the future of highly complex systems using the LET paradigm, interval synchronization is an essential part of their LET integration. In SBP, any incorrect overlap of LET intervals endangers the correctness of buffering. Therefore, synchronizing LET intervals is mandatory in systems of any complexity. For this reason, the duration of event chains and the dataflow for the *Unsynchronized* configuration is not compared in Table 5.2.

5.2.3 Conclusions

The conducted case study showed that LET is a practical approach to ensure functional correctness between SWCs executing on different cores. Besides ensuring LET semantics through the buffering and scheduling design, the synchronization of LET intervals on the same and different cores is absolutely necessary to ensure the determinism of

LET in a highly event-driven system. Without this synchronization, the functional correctness of the LET system is affected. Hence, the described synchronization approaches are a practical way to reduce activation jitters of LET intervals and achieve the desired LET determinism in an AUTOSAR system. However, the case study was conducted on conditions where timer and cross-core interrupts are not delayed or preempted by other urgent interrupts. To ensure this synchronization accuracy, the delays or preemptions of these interrupts must be avoided.

6 | Conclusions and Future Work

This chapter presents the conclusions of this thesis and the future work. A summary of the conclusions and experiences regarding the proposed buffering and scheduling approaches are given, followed by the future work described at the end of this chapter.

6.1 Conclusions

In this work, buffering and scheduling techniques were presented for the resource efficient integration of the *Logical Execution Time (LET)* into automotive systems. The *Static Buffering Protocol (SBP)* was proposed as a resource efficient buffering mechanism to ensure data exchange semantics of LET (*Contribution C1*). SBP was designed to reduce the run-time of buffering operations and the memory capacity required to preserve LET semantics. This was done by incorporating a static and global buffering strategy and by describing a buffer synthesis algorithm that suppresses unnecessary writes. Since automotive applications are event-based, this work provided a comprehensive definition of the lock-based *Point-to-Point Protocol (PTP)* for LET and described its use for specific application configurations.

In addition to buffering, two schedule synthesis approaches have been proposed to guarantee LET semantics and fulfill timing and performance requirements under the influence of the *Operating System (OS)* overheads (*Contribution C2*). The approaches consider the buffer semantics of the two aforementioned LET buffering protocols. Since minimizing scheduling overheads was one of the goals of this work, the *Constraint Programming (CP)* technique was used to solve the schedule synthesis problem. Therefore, a constraint-based formulation was proposed for two different scheduling algorithms, *Time-Triggered Scheduling (TTS)* and *Fixed-Priority Scheduling (FPS)*, which have different scheduling semantics compared to each other.
Two different scheduling mechanisms were considered to observe the advantages and practicality of each approach for LET applications. Unlike related work, the proposed formulations take into account different types of overheads and delays caused by the execution of high-priority tasks, of context-switch operations, and of OS interrupts such as the *timer*. In particular, unlike related work, a unique method for validating the feasibility of scheduling as part of the constraint problem was proposed for FPS, which, as expected, showed to be a time-efficient approach in the conducted experiments.

Case studies considering characteristics of industrial applications and future functionality extensions (*Contribution C3*) were performed to evaluate the buffering protocols and scheduling mechanisms in terms of performance, memory requirements, and run-time. In this way, their capabilities and resource demands of LET could be estimated for the future industrial applications. To show the practicality of LET in automotive systems, the integration of LET buffering for the classic *AUTomotive Open System ARchitecture (AUTOSAR)* platform was described. Challenges and solutions to ensure determinism of LET and correctness of buffering in event-driven AUTOSAR systems were proposed. A case study based on a real world automotive system was conducted to show that LET is a feasible approach to ensure end-to-end delays and deterministic dataflow between *Software Component (SWC)*s of AUTOSAR applications.

Based on the experiences gained during this work, it can be concluded that LET is a feasible way to ensure time and dataflow determinism between periodic functions of automotive applications. Although it requires additional resources, these can be reduced by efficient integration of its semantics and by optimizing the schedule of tasks. Based on the conducted studies, it can be stated that SBP is an efficient approach to integrate LET semantics for automotive applications. The results showed that SBP performs better than PTP in several aspects. SBP incurs negligible processing load, provides zero communication time at the boundaries of LET intervals, and reduces the memory resources required to maintain LET semantics. Furthermore, it performs better in terms of schedulability on both TTS and FPS due to its low buffer and timer load. Therefore, it can be concluded that SBP provides more scope for future extensions of the application, as it allows to ensure a feasible schedule also during high load peaks, and more data elements can be exchanged via the LET semantics. However, due to its static buffer schedule configuration, it cannot ensure hybrid data exchange between periodic and a-periodic or sporadic tasks for write-write and read-write communication interactions and is more sensitive to activation jitters regarding buffering correctness. For this reason, PTP is used in these cases.

In terms of scheduling, it can be stated that any of the scheduling mechanisms can be used to schedule LET applications. However, as the conducted research shows, TTS has less complexity in solving the scheduling problem and it provides greater opportunities to reduce the run-time overheads caused by context switching. In addition, when implemented correctly in an OS, it ensures deterministic execution of tasks and can avoid LET buffering violations in the event of unpredictable long preemption and start delays without the need to use timing protection mechanisms of AUTOSAR [19]. In a dynamic operating environment, such as in an event-driven AUTOSAR OS, semantic violations of LET buffering must be prevented and monitored by specific mechanisms during system operation, regardless of the used buffering and scheduling mechanism. This is because automotive systems are highly event-based and it is difficult to completely avoid task activation jitters. Nevertheless, in both buffering mechanisms the effects of activation jitters on dataflow determinism between LET intervals must be reduced in all situations independent of the scheduling mechanism. This means that the dataflow and execution order between LET intervals must be identical at design time as in target execution. However, for SBP, such monitoring is more difficult to handle when FPS is used. Hence, TTS is the most convenient scheduling approach to ensure correct SBP buffering operation by means of deterministic task execution. It is important to note that integrating TTS into existing AUTOSAR OS is particularly challenging because this OS is purely event-based and is constructed to handle a variety of event types. Therefore, it is necessary to extend its foundations to be time-driven and to integrate the semantics of TTS and FPS into one system to obtain the characteristics of a wide range of applications.

Finally, it can further be concluded that it is essential to integrate automatic buffering and scheduling configuration of LET into commercial tools to automate the design process of highly complex industrial applications. The problem of scheduling in particular is highly complex and doing it manually is inefficient and increases development time and effort. As shown in this work, constraint programming is a necessary way to solve the scheduling problem because it is able to provide solutions of a satisfying quality and fulfill multiple requirements simultaneously. However, maintaining constraints with new functionalities in commercial products requires experienced engineers in the CP technology and considerable amount of effort and resources due to the abstraction of the CP logic. It should be noted that CP allows the description of the scheduling problem only for application attributes that are deterministic in nature, e.g., only for periodic tasks. Events with unknown arrival times are hard to describe by CP constraints without pessimistic assumptions as a trade-off.

6.2 Future Work

During this work, several attractive research directions were identified that can extend the work presented in this thesis. These research directions are outlined below.

Design of LET Intervals

The integration of the LET paradigm into automotive systems involves several design aspects. In this work, two of the most important designs aspects have been presented, as well as approaches to enable the practicality of LET in AUTOSAR systems. To complete the development flow of LET applications, the design of LET intervals is necessary. Although a basic LET design can be defined, for applications with highly interconnected runnables, an automated approach to derive LET intervals is required to handle the complexity of the problem. This is especially important because the timing information of LET intervals must ensure not only the dataflow and execution order between runnables, but also their timing requirements. This design step is usually performed before buffering and schedule synthesis. However, in order to design LET intervals, the impact of the buffer must be known in advance,

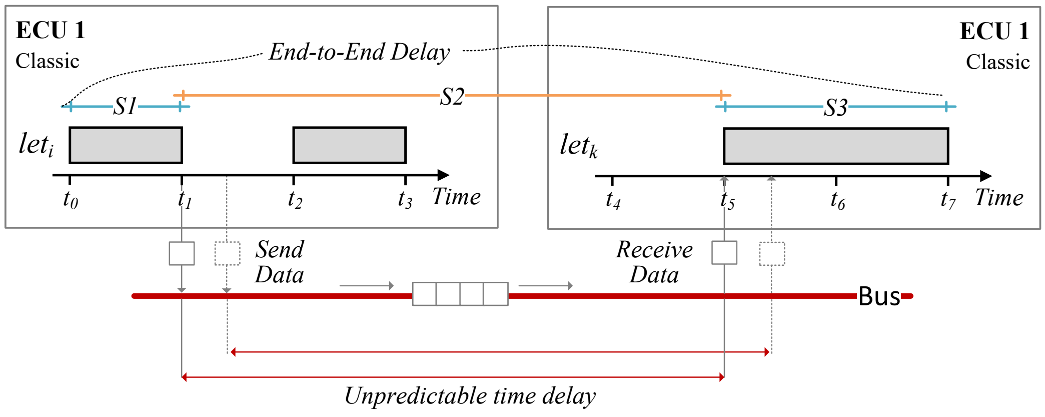


Figure 6.1: System-Level Determinism.

and a feasible schedule for these LET intervals must be guaranteed. Therefore, the design of LET intervals is coupled with the buffer and scheduling design in a closed development loop. The design of LET intervals for an AUTOSAR application includes the following tasks. Firstly, defining LET intervals and their timing information such as the activation offset, period, and LET interval duration. Secondly, the mapping of runnable entities to LET intervals, which enforces that they use LET semantics to exchange data with runnable entities of other LET intervals. Finally, it must be specified which data accesses within these runnables use the LET semantics. It should be noted that this design aspect is often coupled with the assignment of runnables to tasks and processor cores, which introduces more complexity to solving this design problem.

System-Level Determinism

The LET semantics, buffering, and scheduling design presented in this work are applicable to AUTOSAR classic applications within one *Electronic Control Unit (ECU)*. However, in-vehicle applications are distributed across multiple ECUs and have end-to-end timing requirements beyond one ECU. Moreover, these ECUs can contain not only AUTOSAR classic applications, but also AUTOSAR adaptive applications that normally run on Linux-based OSs. The LET paradigm, as currently defined, cannot guarantee timing and dataflow determinism for applications deployed on different ECUs and for communication between different platforms, i.e., classic and adaptive. The system-level LET is introduced in [78, 168] as an extension of LET to provide timing and dataflow determinism between applications deployed in different ECUs or hardware platforms.

To understand the need for LET at the system level, an example of data exchanges between two LET intervals from two different ECUs is given in Figure 6.1. In this example, the first occurrence of LET interval let_i of *ECU 1* provides outputs to the LET interval let_k of *ECU 2* at time t_1. These outputs are sent over the physical bus at time t_1 or later, depending on the scheduling, and are received in *ECU 2* just before the start of LET interval let_k or some time later. In this case, whether the interval let_k receives the data at its start time or not depends on the bus scheduling and traffic. In this example, the end-to-end delay requirement is the time interval defined by time intervals $S1$, $S2$, and $S3$. The duration of time interval $S2$ is unpredictable due to the unpredictable time delay caused by the bus traffic. Therefore, in certain situations, e.g., for different occurrences of LET intervals let_i and let_k, the data is received just before, some time after, or exactly at the start of the LET interval let_k. This enforces non-deterministic data flow and end-to-end delays between these LET intervals. Therefore, describing the time interval $S2$ as a system-level LET interval, the dataflow and time determinism between two local LET intervals is possible. The system-level LET guarantees determinism by enforcing that data sent by the sender LET interval is sent at the release time of the system-level LET interval and is made available to the receiver LET interval at the end of the system-level LET interval, even if the data is received well before that time. In this case, the system-level LET interval is a time interval with the release time in one ECU and the terminate time in the other ECU. Hence, LET and system-level LET can only together guarantee deterministic dataflow and end-to-end delays of distributed in-vehicle applications.

As with LET, the following research aspects are of interest for the system-level LET:

▷ *Design of system-level LET intervals*: Designing system-level LET intervals presents a similar problem to designing LET intervals. The assignment of timing information for system-level LET must take into account not only end-to-end delay requirements, but also realistic bus delays resulting from scheduling and timing synchronization delays. It should be noted that the clocks of the various ECUs are synchronized with some accuracy delay that can impact the timing of system-level LET. In practice, the design of system-level LET intervals is most effectively carried out together with the design of LET in a top-down development approach. These two problems are therefore best solved in one step.

▷ *Buffering mechanism for system-level LET*: The semantics of system-level LET must be ensured by a buffering mechanism in the receiver ECU. A buffering protocol shall store all incoming input from sending LET intervals and provide the receiving LET interval with the appropriate data corresponding to the system-level LET timing information. Without a specific buffering mechanism, system-level LET behavior cannot be guaranteed. It should be noted that buffering for system-level LET incurs additional memory and run-time overheads that must be kept to a minimum. Therefore, research on buffering protocols that reduce these overheads is highly necessary.

▷ *System-level schedule synthesis*: Bus and OS scheduling must guarantee that the sending and receiving of data between LET intervals of the various ECUs occurs on time. Within an ECU, the schedule synthesis algorithms presented in this work can be applied. However, since the data in the receiving ECU is usually processed using sporadic or a-periodic events, the proposed approaches need to be extended to account the execution of sporadic and a-periodic tasks. In addition, bus schedule synthesis that takes into account system-level LET semantics and the schedule mechanisms of different communication protocols is required.

Schedule Synthesis Extensions

The proposed algorithms for schedule synthesis can be enhanced to include the following aspects:

▷ *Start delays caused by critical sections*: In this work, it was assumed that low-priority computation tasks do not use interrupt locks during their execution. Nevertheless, these tasks can disable preemption for a certain time during critical non-LET data operations. This is especially important when the execution of sporadic and a-periodic schedules is considered in the schedule synthesis. Therefore, an extension of the proposed formalization to handle these delays is essential for tasks that exchange data via non-LET mechanisms.

▷ *Optimizations of the search strategy to improve the quality of the optimal solution*: Although the presented schedule synthesis algorithms were designed to be effective in terms of run-time by, for example, reducing the complexity of the problem and providing an efficient formalization, further optimizations in the search algorithm of the CP solver are needed. This can be realized by, e.g., implementing the proposed formulations in other CP solvers, combining the *reinforcement learning* and *constraint programming* [165], or configuring the most suitable search heuristic and decision strategy for this specific problem. This is especially necessary in TTS schedule synthesis because of the large solution and exploration space.

▷ *Synthesis of cooperative FPS scheduling*: Reduction of context-switching overheads and memory demands caused by preemption by employing and synthesizing cooperative FPS scheduling.

A | Appendix

A.1 Evaluation of Inter-Task Communication Design

Synthetic Benchmarks

This section provides the evaluation results of the synthetic benchmarks of *Static Buffering Protocol (SBP)* and *Point-to-Point Protocol (PTP)* evaluation.

A.1.0.1 Memory Evaluation

The attributes of the tables listed in this section are described as: M – the model identification number, D – the amount of data elements, A – the amount of data accesses, B_{ptp} – the amount of buffers for PTP, B_{sbpg} – the amount of buffers for SBP-G, B_{sbpl} – the amount of buffers for SBP-L, G_{ub} – the required global memory size for storing D data elements in the unbuffered model, G_{ptp}(kB) – the required global memory size for storing B_{ptp} amount of data in PTP, G_{sbpg}(kB) – the required global memory size for storing B_{sbpg} amount of data in SBP-G, G_{sbpl}(kB) – the required global memory size for storing B_{sbpl} amount of data in SBP-L, L_{ptp} – the amount of local variables for PTP, L_{sbpg} – the amount of local variables for SBP-G, L_{sbpl} – the amount of local variables for SBP-L, S_{ptp}(kB) – the maximal stack memory size for storing L_{ptp} amount of local data in PTP, S_{sbpg}(kB) – the maximal stack memory size for storing L_{sbpg} amount of local data in SBP-G, and S_{sbpl}(kB) – the maximal stack memory size for storing L_{sbpl} amount of local data in SBP-L.

Table A.1: Results of the *global buffer size* for EMS models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
1	100	500	586	426	358	1.1	6.5	4.7	4.0
2	200	1,000	1,157	863	718	2.2	12.8	9.6	7.8
3	300	1,500	1,754	1,281	1,057	3.1	18.2	13.5	11.0
4	400	2,000	2,332	1,718	1,437	4.2	24.2	17.9	15.0
5	500	2,500	2,903	2,109	1,760	4.7	27.5	20.0	16.5
6	600	3,000	3,477	2,575	2,168	6.0	34.7	25.6	21.5

Table A.1: Results of the *global buffer size* for EMS models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
7	700	3,500	4,060	3,001	2,539	7.3	42.3	31.6	27.0
8	800	4,000	4,639	3,420	2,838	8.8	51.2	37.6	31.2
9	900	4,500	5,223	3,829	3,185	9.6	55.6	40.8	34.1
10	1,000	5,000	5,836	4,277	3,585	9.9	57.7	42.3	35.4
11	1,100	5,500	6,407	4,674	3,934	10.9	63.7	46.6	39.2
12	1,200	6,000	6,947	5,104	4,271	12.2	71.0	52.4	44.0
13	1,300	6,500	7,549	5,551	4,627	13.3	76.9	56.7	47.5
14	1,400	7,000	8,151	5,994	4,979	14.5	84.7	62.6	51.8
15	1,500	7,500	8,713	6,433	5,354	16.2	94.2	69.4	57.1
16	1,600	8,000	9,308	6,817	5,689	16.5	95.8	70.1	58.5
17	1,700	8,500	9,912	7,234	6,050	17.6	102.4	75.6	63.0
18	1,800	9,000	10,488	7,696	6,445	18.1	106.1	78.1	65.2
19	1,900	9,500	11,059	8,146	6,787	19.9	116.0	84.6	70.5
20	2,000	10,000	11,632	8,531	7,104	21.8	126.9	93.5	78.1
21	2,100	10,500	12,188	8,905	7,445	22.2	128.8	94.4	79.0
22	2,200	11,000	12,834	9,363	7,801	23.3	136.3	99.3	82.5
23	2,300	11,500	13,382	9,832	8,208	24.4	141.8	104.4	87.1
24	2,400	12,000	13,944	10,218	8,508	26.2	152.4	110.9	92.3
25	2,500	12,500	14,532	10,664	8,816	26.7	155.2	114.0	94.1
26	2,600	13,000	15,090	11,174	9,357	27.7	160.5	118.8	99.3
27	2,700	13,500	15,734	11,550	9,689	29.7	173.2	127.3	106.7
28	2,800	14,000	16,266	12,017	9,991	29.0	168.6	124.0	103.2
29	2,900	14,500	16,827	12,358	10,340	31.2	181.4	133.2	110.7
30	3,000	15,000	17,449	12,750	10,556	31.2	182.0	132.6	110.2
31	3,100	15,500	17,881	13,253	10,959	32.5	187.4	138.9	115.1
32	3,200	16,000	18,643	13,666	11,438	32.8	190.9	139.9	116.8
33	3,300	16,500	19,232	14,205	11,853	35.0	203.8	150.3	125.5
34	3,400	17,000	19,823	14,647	12,284	36.5	213.1	157.6	132.1
35	3,500	17,500	20,286	14,879	12,370	36.7	212.2	155.8	129.4

Table A.1: Results of the *global buffer size* for EMS models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
36	3,600	18,000	20,911	15,306	12,769	37.8	219.7	160.9	134.4
37	3,700	18,500	21,419	15,752	13,094	38.5	223.4	163.9	135.9
38	3,800	19,000	22,006	16,257	13,478	39.9	230.7	170.1	141.6
39	3,900	19,500	22,707	16,667	13,930	41.1	239.0	175.9	146.9
40	4,000	20,000	23,301	17,161	14,290	42.1	244.9	180.1	150.0
41	4,100	20,500	23,875	17,438	14,523	43.1	250.5	183.2	152.1
42	4,200	21,000	24,427	17,879	14,951	44.0	255.7	187.1	156.2
43	4,300	21,500	25,017	18,353	15,306	44.8	260.6	190.5	158.5
44	4,400	22,000	25,552	18,827	15,563	45.4	263.1	194.1	160.2
45	4,500	22,500	26,174	19,240	16,026	46.9	273.3	201.7	167.9
46	4,600	23,000	26,748	19,612	16,347	48.1	279.6	205.1	170.9
47	4,700	23,500	27,324	20,024	16,717	49.1	285.5	209.7	175.1
48	4,800	24,000	27,937	20,409	17,069	49.9	290.5	212.1	178.0
49	4,900	24,500	28,493	20,890	17,406	51.5	299.3	218.4	182.1
50	5,000	25,000	29,061	21,328	17,764	51.7	300.2	220.4	183.8

Table A.2: Results of the *global buffer size* for Chassis models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
1	100	500	583	389	313	1.1	6.5	4.2	3.4
2	200	1,000	1,162	757	599	2.0	11.9	7.7	6.2
3	300	1,500	1,738	1,140	892	3.4	19.4	12.8	10.0
4	400	2,000	2,311	1,492	1,176	4.0	23.4	15.2	11.8
5	500	2,500	2,910	1,918	1,501	5.7	33.0	21.8	17.0
6	600	3,000	3,485	2,288	1,811	6.3	36.6	24.1	19.0
7	700	3,500	4,067	2,673	2,097	7.0	40.3	26.7	20.9
8	800	4,000	4,656	3,040	2,367	8.1	47.3	31.2	24.5
9	900	4,500	5,239	3,448	2,720	9.6	55.9	36.8	28.9
10	1,000	5,000	5,842	3,829	3,017	10.6	62.0	40.5	31.8
11	1,100	5,500	6,368	4,177	3,291	11.3	65.5	43.3	34.3

Table A.2: Results of the *global buffer size* for Chassis models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
12	1,200	6,000	6,998	4,617	3,643	12.8	74.8	49.5	39.2
13	1,300	6,500	7,553	4,981	3,932	14.0	81.4	54.2	43.0
14	1,400	7,000	8,156	5,360	4,212	14.7	85.5	56.0	44.1
15	1,500	7,500	8,708	5,736	4,530	15.9	92.1	60.4	48.0
16	1,600	8,000	9,294	6,139	4,824	16.9	98.5	65.4	51.5
17	1,700	8,500	9,922	6,567	5,209	17.5	102.4	67.8	53.8
18	1,800	9,000	10,494	6,904	5,460	19.4	112.7	73.8	58.4
19	1,900	9,500	11,060	7,255	5,688	19.7	114.9	75.7	59.5
20	2,000	10,000	11,628	7,681	6,103	20.4	118.4	78.2	62.0
21	2,100	10,500	12,213	8,006	6,270	22.9	133.4	87.6	68.8
22	2,200	11,000	12,785	8,481	6,725	22.7	131.6	87.3	69.2
23	2,300	11,500	13,366	8,882	7,077	24.6	142.6	94.9	75.8
24	2,400	12,000	13,957	9,143	7,220	24.3	140.6	92.4	73.1
25	2,500	12,500	14,555	9,537	7,522	25.4	147.9	96.5	76.0
26	2,600	13,000	15,119	9,928	7,846	26.8	156.5	103.3	81.5
27	2,700	13,500	15,678	10,369	8,208	28.6	165.7	110.0	87.2
28	2,800	14,000	16,240	10,725	8,484	29.7	172.2	113.4	89.5
29	2,900	14,500	16,897	11,154	8,849	30.3	176.7	116.3	92.1
30	3,000	15,000	17,482	11,481	9,055	30.5	177.8	116.3	91.6
31	3,100	15,500	18,059	11,891	9,405	32.9	191.7	126.8	100.5
32	3,200	16,000	18,637	12,238	9,674	33.7	196.3	129.3	102.4
33	3,300	16,500	19,228	12,669	10,031	34.7	201.9	132.4	104.8
34	3,400	17,000	19,725	12,959	10,239	34.7	202.1	132.8	104.7
35	3,500	17,500	20,363	13,378	10,578	37.1	215.5	141.2	111.5
36	3,600	18,000	20,970	13,849	10,962	38.0	220.9	145.7	115.0
37	3,700	18,500	21,501	14,205	11,257	39.1	227.3	149.8	118.8
38	3,800	19,000	22,100	14,603	11,559	40.2	233.2	153.9	121.8
39	3,900	19,500	22,632	14,853	11,679	41.5	240.8	158.3	124.2
40	4,000	20,000	23,284	15,327	12,075	40.5	236.5	155.4	122.1

Table A.2: Results of the *global buffer size* for Chassis models.

M	D	A	B_{ptp}	B_{sbpg}	B_{sbpl}	G_{ub}(kB)	G_{ptp}(kB)	G_{sbpg}(kB)	G_{sbpl}(kB)
41	4,100	20,500	23,890	15,789	12,494	43.8	255.4	169.7	135.1
42	4,200	21,000	24,467	15,998	12,618	43.9	256.0	167.5	132.0
43	4,300	21,500	25,008	16,406	12,937	44.2	257.7	169.1	133.4
44	4,400	22,000	25,605	16,862	13,344	45.3	264.0	174.4	138.8
45	4,500	22,500	26,189	17,240	13,650	47.0	273.5	179.9	142.6
46	4,600	23,000	26,785	17,613	13,963	48.1	280.2	185.0	147.4
47	4,700	23,500	27,307	17,954	14,184	48.7	282.7	186.6	148.3
48	4,800	24,000	27,956	18,440	14,558	49.3	287.5	189.7	149.8
49	4,900	24,500	28,533	18,695	14,731	51.9	301.9	197.2	155.0
50	5,000	25,000	29,106	19,191	15,143	52.0	302.3	198.7	156.5

Table A.3: Results of the *local buffer size* for EMS models.

M	D	A	L_{ptp}	L_{sbpg}	L_{sbpl}	S_{ptp}(kB)	S_{sbpg}(kB)	S_{sbpl}(kB)
1	100	500	0	459	656	0	0.5	2.7
2	200	1,000	0	918	1,308	0	0.9	5.3
3	300	1,500	0	1,366	1,958	0	1.4	7.5
4	400	2,000	0	1,841	2,621	0	1.8	9.9
5	500	2,500	0	2,259	3,238	0	2.3	11.5
6	600	3,000	0	2,714	3,888	0	2.7	14.3
7	700	3,500	0	3,155	4,515	0	3.2	17.4
8	800	4,000	0	3,619	5,175	0	3.6	20.8
9	900	4,500	0	4,106	5,853	0	4.1	22.8
10	1,000	5,000	0	4,570	6,528	0	4.6	24.0
11	1,100	5,500	0	5,030	7,179	0	5.0	26.4
12	1,200	6,000	0	5,449	7,779	0	5.4	29.3
13	1,300	6,500	0	5,935	8,477	0	5.9	31.9
14	1,400	7,000	0	6,394	9,147	0	6.4	35.1
15	1,500	7,500	0	6,811	9,737	0	6.8	38.4
16	1,600	8,000	0	7,306	10,442	0	7.3	39.6

Table A.3: Results of the *local buffer size* for EMS models.

M	D	A	L_{ptp}	L_{sbpg}	L_{sbpl}	S_{ptp}(kB)	S_{sbpg}(kB)	S_{sbpl}(kB)
17	1,700	8,500	0	7,803	11,138	0	7.8	42.3
18	1,800	9,000	0	8,244	11,770	0	8.2	43.8
19	1,900	9,500	0	8,625	12,341	0	8.6	47.5
20	2,000	10,000	0	9,108	13,014	0	9.1	51.8
21	2,100	10,500	0	9,542	13,634	0	9.5	52.7
22	2,200	11,000	0	10,052	14,349	0	10.1	55.7
23	2,300	11,500	0	10,477	14,964	0	10.5	58.1
24	2,400	12,000	0	10,878	15,567	0	10.9	62.1
25	2,500	12,500	0	11,343	16,217	0	11.3	63.3
26	2,600	13,000	0	11,792	16,866	0	11.8	65.7
27	2,700	13,500	0	12,336	17,606	0	12.3	70.1
28	2,800	14,000	0	12,728	18,207	0	12.7	69.4
29	2,900	14,500	0	13,194	18,848	0	13.2	74.2
30	3,000	15,000	0	13,671	19,535	0	13.7	74.8
31	3,100	15,500	0	13,791	19,802	0	13.8	76.7
32	3,200	16,000	0	14,674	20,925	0	14.7	79.0
33	3,300	16,500	0	15,003	21,444	0	15.0	83.2
34	3,400	17,000	0	15,533	22,190	0	15.5	87.1
35	3,500	17,500	0	15,897	22,713	0	15.9	87.1
36	3,600	18,000	0	16,383	23,427	0	16.4	90.3
37	3,700	18,500	0	16,674	23,883	0	16.7	91.9
38	3,800	19,000	0	17,129	24,524	0	17.1	94.5
39	3,900	19,500	0	17,784	25,406	0	17.8	98.0
40	4,000	20,000	0	18,238	26,039	0	18.2	100.1
41	4,100	20,500	0	18,733	26,760	0	18.7	103.0
42	4,200	21,000	0	19,130	27,342	0	19.1	105.0
43	4,300	21,500	0	19,554	27,965	0	19.6	107.0
44	4,400	22,000	0	19,915	28,504	0	19.9	108.5
45	4,500	22,500	0	20,562	29,350	0	20.6	112.4

Table A.3: Results of the *local buffer size* for EMS models.

M	D	A	L_{ptp}	L_{sbpg}	L_{sbpl}	S_{ptp}(kB)	S_{sbpg}(kB)	S_{sbpl}(kB)
46	4,600	23,000	0	20,925	29,897	0	20.9	114.8
47	4,700	23,500	0	21,387	30,586	0	21.4	117.5
48	4,800	24,000	0	21,981	31,349	0	22.0	119.2
49	4,900	24,500	0	22,306	31,888	0	22.3	122.8
50	5,000	25,000	0	22,749	32,510	0	22.7	123.9

Table A.4: Results of the *local buffer size* for Chassis models.

M	D	A	L_{ptp}	L_{sbpg}	L_{sbpl}	S_{ptp}(kB)	S_{sbpg}(kB)	S_{sbpl}(kB)
1	100	500	0	454	648	0	0.5	2.6
2	200	1,000	0	911	1,299	0	0.9	4.9
3	300	1,500	0	1,365	1,955	0	1.4	8.0
4	400	2,000	0	1,815	2,591	0	1.8	9.8
5	500	2,500	0	2,283	3,264	0	2.3	13.5
6	600	3,000	0	2,708	3,873	0	2.7	15.0
7	700	3,500	0	3,184	4,558	0	3.2	16.8
8	800	4,000	0	3,667	5,230	0	3.7	19.5
9	900	4,500	0	4,129	5,896	0	4.1	23.0
10	1,000	5,000	0	4,595	6,551	0	4.6	25.4
11	1,100	5,500	0	4,976	7,123	0	5.0	27.0
12	1,200	6,000	0	5,486	7,843	0	5.5	30.7
13	1,300	6,500	0	5,943	8,478	0	5.9	33.2
14	1,400	7,000	0	6,373	9,115	0	6.4	35.0
15	1,500	7,500	0	6,827	9,745	0	6.8	37.7
16	1,600	8,000	0	7,276	10,403	0	7.3	40.3
17	1,700	8,500	0	7,765	11,091	0	7.8	41.9
18	1,800	9,000	0	8,240	11,753	0	8.2	46.0
19	1,900	9,500	0	8,694	12,408	0	8.7	47.3
20	2,000	10,000	0	9,127	13,035	0	9.1	49.0
21	2,100	10,500	0	9,613	13,729	0	9.6	54.5

Table A.4: Results of the *local buffer size* for Chassis models.

M	D	A	L_{ptp}	L_{sbpg}	L_{sbpl}	S_{ptp}(kB)	S_{sbpg}(kB)	S_{sbpl}(kB)
22	2,200	11,000	0	10,037	14,324	0	10.0	54.1
23	2,300	11,500	0	10,517	15,003	0	10.5	58.2
24	2,400	12,000	0	10,989	15,682	0	11.0	58.4
25	2,500	12,500	0	11,432	16,307	0	11.4	60.9
26	2,600	13,000	0	11,853	16,933	0	11.9	64.5
27	2,700	13,500	0	12,301	17,573	0	12.3	68.2
28	2,800	14,000	0	12,721	18,172	0	12.7	70.6
29	2,900	14,500	0	13,240	18,912	0	13.2	72.4
30	3,000	15,000	0	13,715	19,587	0	13.7	73.3
31	3,100	15,500	0	14,233	20,290	0	14.2	78.5
32	3,200	16,000	0	14,599	20,852	0	14.6	80.3
33	3,300	16,500	0	15,111	21,565	0	15.1	82.8
34	3,400	17,000	0	15,468	22,091	0	15.5	83.4
35	3,500	17,500	0	16,035	22,857	0	16.0	88.3
36	3,600	18,000	0	16,441	23,486	0	16.4	90.8
37	3,700	18,500	0	16,877	24,090	0	16.9	93.0
38	3,800	19,000	0	17,308	24,748	0	17.3	95.9
39	3,900	19,500	0	17,694	25,313	0	17.7	98.8
40	4,000	20,000	0	18,286	26,112	0	18.3	97.9
41	4,100	20,500	0	18,789	26,818	0	18.8	104.7
42	4,200	21,000	0	19,168	27,371	0	19.2	105.1
43	4,300	21,500	0	19,654	28,041	0	19.7	106.1
44	4,400	22,000	0	20,083	28,669	0	20.1	108.6
45	4,500	22,500	0	20,526	29,329	0	20.5	112.7
46	4,600	23,000	0	21,040	30,030	0	21.0	114.8
47	4,700	23,500	0	21,458	30,661	0	21.5	116.6
48	4,800	24,000	0	21,891	31,290	0	21.9	118.5
49	4,900	24,500	0	22,420	32,007	0	22.4	124.2
50	5,000	25,000	0	22,862	32,621	0	22.9	124.0

A.1.0.2 Overhead Evaluation

The attributes of the tables listed in this section are described as: M – the model identification number, D – the amount of data elements, A – the amount of data accesses, SA – the amount of spin-lock accesses, O_{sbpg}(%) – buffering utilization of SBP-G, O_{sbpl}(%) – buffering utilization of SBP-L, $O_{ptp-wsp}$(%) – buffering utilization of PTP considering spin-locks, $O_{ptp-wosp}$(%) – buffering utilization of PTP without spin-locks, MAT(ms) – the total memory accessing time, MAT_{init}(ms) – the total memory accessing time to initialize buffers at the start of the hyper-period, MAT_{idx}(ms) – the total memory accessing time to initialize buffer indexes at the start of tasks, MAT_{local}(ms) – the total memory accessing time to fill and flush local buffers in SBP-L, NET_{sp}^{usage}(ms) – the total *Net Execution Time* for using spin-locks, and $NET_{sp}^{waiting}$(ms) – the total *Net Execution Time* for waiting for blocked spin-locks.

Table A.5: Results of the *buffering overhead* of PTP-WSP in EMS models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
1	100	500	936	2.67	0.01	0.09	0.000000
2	200	1,000	1,896	5.11	0.03	0.19	0.000035
3	300	1,500	2,808	7.94	0.04	0.28	0.000025
4	400	2,000	3,824	10.31	0.05	0.38	0.000130
5	500	2,500	4,734	13.69	0.06	0.47	0.000000
6	600	3,000	5,648	15.22	0.07	0.56	0.000065
7	700	3,500	6,642	19.48	0.09	0.66	0.000050
8	800	4,000	7,478	20.70	0.11	0.75	0.000110
9	900	4,500	8,504	24.45	0.12	0.85	0.000125
10	1,000	5,000	9,540	28.38	0.13	0.95	0.000045
11	1,100	5,500	10,404	31.00	0.14	1.04	0.000145
12	1,200	6,000	11,304	32.78	0.15	1.13	0.000015
13	1,300	6,500	12,314	35.91	0.16	1.23	0.000040
14	1,400	7,000	13,338	37.01	0.18	1.33	0.000040
15	1,500	7,500	14,180	39.10	0.20	1.42	0.000040
16	1,600	8,000	15,162	45.41	0.21	1.52	0.000045
17	1,700	8,500	16,192	47.74	0.22	1.62	0.000015
18	1,800	9,000	17,090	50.75	0.23	1.71	0.000105
19	1,900	9,500	17,994	51.32	0.25	1.80	0.000105

Table A.5: Results of the *buffering overhead* of PTP-WSP in EMS models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
20	2,000	10,000	18,868	57.16	0.27	1.89	0.000175
21	2,100	10,500	19,842	59.36	0.28	1.98	0.000060
22	2,200	11,000	20,984	62.37	0.29	2.10	0.000125
23	2,300	11,500	21,788	62.37	0.31	2.18	0.000195
24	2,400	12,000	22,692	62.87	0.32	2.27	0.000080
25	2,500	12,500	23,568	65.12	0.33	2.36	0.000175
26	2,600	13,000	24,456	66.32	0.34	2.45	0.000200
27	2,700	13,500	25,730	78.28	0.37	2.57	0.000090
28	2,800	14,000	26,420	76.26	0.36	2.64	0.000165
29	2,900	14,500	27,454	76.89	0.38	2.75	0.000105
30	3,000	15,000	28,342	82.10	0.39	2.83	0.000045
31	3,100	15,500	28,692	77.65	0.40	2.87	0.000055
32	3,200	16,000	30,482	90.18	0.41	3.05	0.000020
33	3,300	16,500	31,316	86.92	0.44	3.13	0.000045
34	3,400	17,000	32,332	97.05	0.45	3.23	0.000205
35	3,500	17,500	33,090	94.40	0.45	3.31	0.000335
36	3,600	18,000	33,976	102.73	0.47	3.40	0.000250
37	3,700	18,500	34,740	96.61	0.48	3.47	0.000075
38	3,800	19,000	35,676	95.44	0.49	3.57	0.000060
39	3,900	19,500	36,984	111.41	0.52	3.70	0.000085
40	4,000	20,000	37,860	105.98	0.53	3.79	0.000135
41	4,100	20,500	38,950	116.71	0.54	3.90	0.000170
42	4,200	21,000	39,786	119.92	0.56	3.98	0.000175
43	4,300	21,500	40,714	118.17	0.56	4.07	0.000185
44	4,400	22,000	41,478	110.56	0.56	4.15	0.000125
45	4,500	22,500	42,602	123.13	0.59	4.26	0.000150
46	4,600	23,000	43,398	124.82	0.60	4.34	0.000095
47	4,700	23,500	44,516	133.97	0.62	4.45	0.000120
48	4,800	24,000	45,564	138.63	0.63	4.56	0.000130

Table A.5: Results of the *buffering overhead* of PTP-WSP in EMS models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
49	4,900	24,500	46,252	134.55	0.65	4.63	0.000085
50	5,000	25,000	47,244	137.22	0.65	4.72	0.000125

Table A.6: Results of the *buffering overhead* of PTP-WOSP in EMS models.

M	D	A	$O_{ptp-wosp}$(%)	MAT(ms)
1	100	500	0.51	0.02
2	200	1,000	0.91	0.04
3	300	1,500	1.47	0.07
4	400	2,000	1.97	0.09
5	500	2,500	2.69	0.11
6	600	3,000	2.90	0.13
7	700	3,500	4.40	0.18
8	800	4,000	4.45	0.20
9	900	4,500	5.12	0.22
10	1,000	5,000	6.16	0.26
11	1,100	5,500	6.68	0.26
12	1,200	6,000	6.42	0.27
13	1,300	6,500	7.62	0.30
14	1,400	7,000	7.18	0.31
15	1,500	7,500	8.85	0.36
16	1,600	8,000	9.52	0.42
17	1,700	8,500	9.53	0.41
18	1,800	9,000	11.08	0.46
19	1,900	9,500	11.28	0.49
20	2,000	10,000	12.48	0.55
21	2,100	10,500	12.89	0.60
22	2,200	11,000	14.82	0.65
23	2,300	11,500	13.96	0.61
24	2,400	12,000	14.71	0.63

Table A.6: Results of the *buffering overhead* of PTP-WOSP in EMS models.

M	D	A	$O_{ptp-wosp}$(%)	MAT(ms)
25	2,500	12,500	16.25	0.71
26	2,600	13,000	15.33	0.67
27	2,700	13,500	19.46	0.84
28	2,800	14,000	17.28	0.70
29	2,900	14,500	19.32	0.88
30	3,000	15,000	19.59	0.77
31	3,100	15,500	19.26	0.81
32	3,200	16,000	20.66	0.86
33	3,300	16,500	20.62	0.95
34	3,400	17,000	22.70	0.94
35	3,500	17,500	22.79	1.00
36	3,600	18,000	25.49	1.08
37	3,700	18,500	23.35	1.05
38	3,800	19,000	25.15	1.06
39	3,900	19,500	27.48	1.22
40	4,000	20,000	28.16	1.22
41	4,100	20,500	27.20	1.16
42	4,200	21,000	30.83	1.29
43	4,300	21,500	28.05	1.18
44	4,400	22,000	25.63	1.19
45	4,500	22,500	29.71	1.29
46	4,600	23,000	30.79	1.29
47	4,700	23,500	33.15	1.42
48	4,800	24,000	35.94	1.51
49	4,900	24,500	31.83	1.41
50	5,000	25,000	32.01	1.52

Table A.7: Results of the *buffering overhead* of SBP-G in EMS models.

M	D	A	O_{sbpg}(%)	MAT_{idx}(ms)	MAT_{init}(ms)
1	100	500	0.08	0.003	0.003
2	200	1,000	0.15	0.006	0.006
3	300	1,500	0.23	0.009	0.009
4	400	2,000	0.30	0.012	0.013
5	500	2,500	0.39	0.015	0.015
6	600	3,000	0.45	0.018	0.018
7	700	3,500	0.55	0.021	0.022
8	800	4,000	0.60	0.024	0.025
9	900	4,500	0.71	0.027	0.027
10	1,000	5,000	0.81	0.030	0.029
11	1,100	5,500	0.90	0.034	0.033
12	1,200	6,000	0.94	0.036	0.037
13	1,300	6,500	1.04	0.040	0.039
14	1,400	7,000	1.06	0.043	0.043
15	1,500	7,500	1.12	0.045	0.047
16	1,600	8,000	1.31	0.049	0.047
17	1,700	8,500	1.38	0.052	0.051
18	1,800	9,000	1.46	0.055	0.053
19	1,900	9,500	1.48	0.058	0.058
20	2,000	10,000	1.64	0.061	0.062
21	2,100	10,500	1.71	0.064	0.064
22	2,200	11,000	1.78	0.067	0.067
23	2,300	11,500	1.79	0.070	0.071
24	2,400	12,000	1.81	0.073	0.076
25	2,500	12,500	1.88	0.076	0.078
26	2,600	13,000	1.92	0.079	0.081
27	2,700	13,500	2.23	0.082	0.084
28	2,800	14,000	2.19	0.085	0.084
29	2,900	14,500	2.21	0.088	0.091

Table A.7: Results of the *buffering overhead* of SBP-G in EMS models.

M	D	A	O_{sbpg}(%)	MAT_{idx}(ms)	MAT_{init}(ms)
30	3,000	15,000	2.36	0.091	0.090
31	3,100	15,500	2.25	0.092	0.097
32	3,200	16,000	2.60	0.098	0.096
33	3,300	16,500	2.50	0.100	0.104
34	3,400	17,000	2.77	0.104	0.104
35	3,500	17,500	2.72	0.106	0.107
36	3,600	18,000	2.95	0.109	0.109
37	3,700	18,500	2.78	0.111	0.111
38	3,800	19,000	2.76	0.114	0.118
39	3,900	19,500	3.18	0.119	0.119
40	4,000	20,000	3.06	0.122	0.123
41	4,100	20,500	3.34	0.125	0.124
42	4,200	21,000	3.42	0.128	0.128
43	4,300	21,500	3.38	0.130	0.131
44	4,400	22,000	3.20	0.133	0.135
45	4,500	22,500	3.54	0.137	0.137
46	4,600	23,000	3.60	0.140	0.140
47	4,700	23,500	3.83	0.143	0.141
48	4,800	24,000	3.97	0.147	0.143
49	4,900	24,500	3.86	0.149	0.149
50	5,000	25,000	3.93	0.152	0.151

Table A.8: Results of the *buffering overhead* of SBP-L in EMS models.

M	D	A	O_{sbpl}(%)	MAT_{idx}(ms)	MAT_{init}(ms)	MAT_{local}(ms)
1	100	500	0.19	0.003	0.003	0.005
2	200	1,000	0.37	0.006	0.006	0.010
3	300	1,500	0.58	0.009	0.009	0.016
4	400	2,000	0.77	0.012	0.012	0.021
5	500	2,500	1.03	0.015	0.013	0.026

Table A.8: Results of the *buffering overhead* of SBP-L in EMS models.

M	D	A	O_{sbpl}(%)	MAT_{idx}(ms)	MAT_{init}(ms)	MAT_{local}(ms)
6	600	3,000	1.08	0.018	0.018	0.029
7	700	3,500	1.50	0.021	0.020	0.039
8	800	4,000	1.58	0.024	0.024	0.043
9	900	4,500	1.96	0.027	0.026	0.053
10	1,000	5,000	2.26	0.030	0.027	0.056
11	1,100	5,500	2.26	0.034	0.030	0.060
12	1,200	6,000	2.49	0.036	0.035	0.063
13	1,300	6,500	2.73	0.040	0.037	0.072
14	1,400	7,000	2.80	0.043	0.041	0.076
15	1,500	7,500	3.11	0.045	0.045	0.086
16	1,600	8,000	3.66	0.049	0.043	0.095
17	1,700	8,500	3.83	0.052	0.047	0.098
18	1,800	9,000	4.12	0.055	0.050	0.109
19	1,900	9,500	4.16	0.058	0.054	0.110
20	2,000	10,000	4.78	0.061	0.057	0.128
21	2,100	10,500	4.72	0.064	0.059	0.120
22	2,200	11,000	4.80	0.067	0.062	0.124
23	2,300	11,500	4.93	0.070	0.067	0.132
24	2,400	12,000	4.93	0.073	0.072	0.137
25	2,500	12,500	5.42	0.076	0.074	0.152
26	2,600	13,000	5.25	0.079	0.077	0.145
27	2,700	13,500	6.70	0.082	0.077	0.188
28	2,800	14,000	6.38	0.085	0.078	0.166
29	2,900	14,500	6.23	0.088	0.086	0.166
30	3,000	15,000	6.51	0.091	0.084	0.174
31	3,100	15,500	6.33	0.092	0.092	0.174
32	3,200	16,000	7.11	0.098	0.090	0.189
33	3,300	16,500	7.06	0.100	0.099	0.197
34	3,400	17,000	8.07	0.104	0.096	0.205

Table A.8: Results of the *buffering overhead* of SBP-L in EMS models.

M	D	A	O_{sbpl}(%)	MAT_{idx}(ms)	MAT_{init}(ms)	MAT_{local}(ms)
35	3,500	17,500	7.59	0.106	0.100	0.202
36	3,600	18,000	8.70	0.109	0.100	0.228
37	3,700	18,500	7.78	0.111	0.105	0.214
38	3,800	19,000	7.86	0.114	0.114	0.218
39	3,900	19,500	9.53	0.119	0.111	0.247
40	4,000	20,000	9.06	0.122	0.117	0.242
41	4,100	20,500	9.80	0.125	0.114	0.261
42	4,200	21,000	10.72	0.128	0.118	0.301
43	4,300	21,500	10.15	0.130	0.122	0.275
44	4,400	22,000	9.16	0.133	0.129	0.259
45	4,500	22,500	10.55	0.137	0.128	0.278
46	4,600	23,000	10.79	0.140	0.131	0.295
47	4,700	23,500	11.67	0.143	0.130	0.314
48	4,800	24,000	12.53	0.147	0.132	0.336
49	4,900	24,500	11.46	0.149	0.139	0.309
50	5,000	25,000	11.74	0.152	0.141	0.321

Table A.9: Results of the *buffering overhead* of PTP-WSP in Chassis models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
1	100	500	936	3.57	0.01	0.09	0.000000
2	200	1,000	1,896	6.95	0.03	0.19	0.000035
3	300	1,500	2,808	10.64	0.04	0.28	0.000025
4	400	2,000	3,824	14.25	0.05	0.38	0.000130
5	500	2,500	4,734	18.15	0.06	0.47	0.000000
6	600	3,000	5,648	20.88	0.07	0.56	0.000065
7	700	3,500	6,642	24.89	0.09	0.66	0.000050
8	800	4,000	7,478	28.32	0.11	0.75	0.000110
9	900	4,500	8,504	31.85	0.12	0.85	0.000125
10	1,000	5,000	9,540	35.09	0.13	0.95	0.000045

Table A.9: Results of the *buffering overhead* of PTP-WSP in Chassis models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
11	1,100	5,500	10,404	38.95	0.14	1.04	0.000145
12	1,200	6,000	11,304	43.47	0.15	1.13	0.000015
13	1,300	6,500	12,314	46.99	0.16	1.23	0.000040
14	1,400	7,000	13,338	48.92	0.18	1.33	0.000040
15	1,500	7,500	14,180	53.80	0.20	1.42	0.000040
16	1,600	8,000	15,162	56.98	0.21	1.52	0.000045
17	1,700	8,500	16,192	60.39	0.22	1.62	0.000015
18	1,800	9,000	17,090	65.16	0.23	1.71	0.000105
19	1,900	9,500	17,994	66.80	0.25	1.80	0.000105
20	2,000	10,000	18,868	71.36	0.27	1.89	0.000175
21	2,100	10,500	19,842	74.70	0.28	1.98	0.000060
22	2,200	11,000	20,984	77.60	0.29	2.10	0.000125
23	2,300	11,500	21,788	83.65	0.31	2.18	0.000195
24	2,400	12,000	22,692	83.94	0.32	2.27	0.000080
25	2,500	12,500	23,568	88.70	0.33	2.36	0.000175
26	2,600	13,000	24,456	92.75	0.34	2.45	0.000200
27	2,700	13,500	25,730	95.03	0.37	2.57	0.000090
28	2,800	14,000	26,420	99.01	0.36	2.64	0.000165
29	2,900	14,500	27,454	104.95	0.38	2.75	0.000105
30	3,000	15,000	28,342	105.64	0.39	2.83	0.000045
31	3,100	15,500	28,692	111.80	0.40	2.87	0.000055
32	3,200	16,000	30,482	114.00	0.41	3.05	0.000020
33	3,300	16,500	31,316	118.35	0.44	3.13	0.000045
34	3,400	17,000	32,332	120.47	0.45	3.23	0.000205
35	3,500	17,500	33,090	124.06	0.45	3.31	0.000335
36	3,600	18,000	33,976	129.43	0.47	3.40	0.000250
37	3,700	18,500	34,740	133.04	0.48	3.47	0.000075
38	3,800	19,000	35,676	135.91	0.49	3.57	0.000060
39	3,900	19,500	36,984	135.39	0.52	3.70	0.000085

Table A.9: Results of the *buffering overhead* of PTP-WSP in Chassis models.

M	D	A	SA	$O_{ptp-wsp}$(%)	MAT(ms)	NET_{sp}^{usage}(ms)	$NET_{sp}^{waiting}$(ms)
40	4,000	20,000	37,860	141.91	0.53	3.79	0.000135
41	4,100	20,500	38,950	148.24	0.54	3.90	0.000170
42	4,200	21,000	39,786	148.97	0.56	3.98	0.000175
43	4,300	21,500	40,714	152.01	0.56	4.07	0.000185
44	4,400	22,000	41,478	158.59	0.56	4.15	0.000125
45	4,500	22,500	42,602	160.50	0.59	4.26	0.000150
46	4,600	23,000	43,398	165.44	0.60	4.34	0.000095
47	4,700	23,500	44,516	169.48	0.62	4.45	0.000120
48	4,800	24,000	45,564	170.09	0.63	4.56	0.000130
49	4,900	24,500	46,252	176.18	0.65	4.63	0.000085
50	5,000	25,000	47,244	179.11	0.65	4.72	0.000125

Table A.10: Results of the *buffering overhead* of PTP-WOSP in Chassis models.

M	D	A	$O_{ptp-wosp}$ (%)	MAT (ms)
1	100	500	0.57	0.02
2	200	1,000	1.12	0.03
3	300	1,500	2.19	0.07
4	400	2,000	2.38	0.08
5	500	2,500	3.96	0.12
6	600	3,000	3.74	0.12
7	700	3,500	4.46	0.14
8	800	4,000	5.74	0.20
9	900	4,500	5.87	0.18
10	1,000	5,000	7.22	0.24
11	1,100	5,500	7.86	0.25
12	1,200	6,000	8.74	0.28
13	1,300	6,500	9.64	0.31
14	1,400	7,000	8.71	0.30
15	1,500	7,500	10.71	0.33

Table A.10: Results of the *buffering overhead* of PTP-WOSP in Chassis models.

M	D	A	$O_{ptp-wosp}$(%)	MAT(ms)
16	1,600	8,000	11.88	0.37
17	1,700	8,500	12.16	0.39
18	1,800	9,000	14.83	0.48
19	1,900	9,500	13.20	0.42
20	2,000	10,000	14.87	0.50
21	2,100	10,500	16.53	0.55
22	2,200	11,000	16.11	0.52
23	2,300	11,500	18.08	0.55
24	2,400	12,000	17.87	0.59
25	2,500	12,500	17.29	0.62
26	2,600	13,000	21.38	0.71
27	2,700	13,500	19.05	0.63
28	2,800	14,000	22.53	0.73
29	2,900	14,500	23.89	0.79
30	3,000	15,000	23.51	0.72
31	3,100	15,500	26.20	0.83
32	3,200	16,000	26.69	0.83
33	3,300	16,500	27.24	0.85
34	3,400	17,000	28.04	0.94
35	3,500	17,500	27.98	0.90
36	3,600	18,000	30.05	0.99
37	3,700	18,500	30.66	1.02
38	3,800	19,000	32.86	1.06
39	3,900	19,500	30.77	1.08
40	4,000	20,000	33.62	1.16
41	4,100	20,500	35.79	1.13
42	4,200	21,000	34.15	1.15
43	4,300	21,500	31.81	1.07
44	4,400	22,000	35.30	1.25

Table A.10: Results of the *buffering overhead* of PTP-WOSP in Chassis models.

M	D	A	$O_{ptp-wosp}$(%)	MAT(ms)
45	4,500	22,500	34.82	1.22
46	4,600	23,000	38.73	1.19
47	4,700	23,500	37.76	1.28
48	4,800	24,000	39.76	1.19
49	4,900	24,500	42.15	1.38
50	5,000	25,000	42.50	1.39

Table A.11: Results of the *buffering overhead* of SBP-G in Chassis models.

M	D	A	O_{sbpg} (%)	MAT_{idx} (ms)	MAT_{init} (ms)
1	100	500	0.12	0.003	0.003
2	200	1,000	0.24	0.006	0.006
3	300	1,500	0.37	0.009	0.009
4	400	2,000	0.49	0.012	0.012
5	500	2,500	0.63	0.015	0.016
6	600	3,000	0.72	0.018	0.018
7	700	3,500	0.85	0.021	0.021
8	800	4,000	0.97	0.024	0.023
9	900	4,500	1.10	0.028	0.027
10	1,000	5,000	1.22	0.031	0.030
11	1,100	5,500	1.34	0.033	0.033
12	1,200	6,000	1.48	0.037	0.037
13	1,300	6,500	1.61	0.040	0.040
14	1,400	7,000	1.68	0.043	0.042
15	1,500	7,500	1.85	0.046	0.046
16	1,600	8,000	1.96	0.049	0.048
17	1,700	8,500	2.07	0.052	0.051
18	1,800	9,000	2.23	0.055	0.056
19	1,900	9,500	2.30	0.058	0.056
20	2,000	10,000	2.44	0.061	0.059

Table A.11: Results of the *buffering overhead* of SBP-G in Chassis models.

M	D	A	O_{sbpg}(%)	MAT_{idx}(ms)	MAT_{init}(ms)
21	2,100	10,500	2.58	0.064	0.065
22	2,200	11,000	2.68	0.067	0.066
23	2,300	11,500	2.87	0.070	0.071
24	2,400	12,000	2.89	0.073	0.071
25	2,500	12,500	3.03	0.076	0.073
26	2,600	13,000	3.17	0.079	0.079
27	2,700	13,500	3.28	0.082	0.083
28	2,800	14,000	3.42	0.085	0.085
29	2,900	14,500	3.59	0.088	0.088
30	3,000	15,000	3.63	0.091	0.090
31	3,100	15,500	3.83	0.095	0.094
32	3,200	16,000	3.92	0.097	0.099
33	3,300	16,500	4.07	0.101	0.099
34	3,400	17,000	4.13	0.103	0.102
35	3,500	17,500	4.28	0.107	0.106
36	3,600	18,000	4.43	0.110	0.112
37	3,700	18,500	4.56	0.113	0.112
38	3,800	19,000	4.66	0.115	0.116
39	3,900	19,500	4.69	0.118	0.120
40	4,000	20,000	4.89	0.122	0.120
41	4,100	20,500	5.09	0.125	0.128
42	4,200	21,000	5.10	0.128	0.126
43	4,300	21,500	5.21	0.131	0.128
44	4,400	22,000	5.43	0.134	0.132
45	4,500	22,500	5.49	0.137	0.134
46	4,600	23,000	5.65	0.140	0.139
47	4,700	23,500	5.81	0.143	0.142
48	4,800	24,000	5.81	0.146	0.142
49	4,900	24,500	6.04	0.150	0.148

Table A.11: Results of the *buffering overhead* of SBP-G in Chassis models.

M	D	A	O_{sbpg}(%)	MAT_{idx}(ms)	MAT_{init}(ms)
50	5,000	25,000	6.13	0.152	0.150

Table A.12: Results of the *buffering overhead* of SBP-L in Chassis models.

M	D	A	O_{sbpl} (%)	MAT_{idx} (ms)	MAT_{init} (ms)	MAT_{local} (ms)
1	100	500	0.27	0.003	0.003	0.004
2	200	1,000	0.50	0.006	0.006	0.008
3	300	1,500	0.85	0.009	0.009	0.014
4	400	2,000	1.04	0.012	0.011	0.017
5	500	2,500	1.49	0.015	0.015	0.023
6	600	3,000	1.66	0.018	0.017	0.026
7	700	3,500	1.94	0.021	0.019	0.029
8	800	4,000	2.17	0.024	0.022	0.035
9	900	4,500	2.61	0.028	0.025	0.040
10	1,000	5,000	2.99	0.031	0.028	0.046
11	1,100	5,500	3.22	0.033	0.030	0.050
12	1,200	6,000	3.61	0.037	0.034	0.054
13	1,300	6,500	3.72	0.040	0.037	0.057
14	1,400	7,000	3.89	0.043	0.038	0.060
15	1,500	7,500	4.68	0.046	0.041	0.082
16	1,600	8,000	4.44	0.049	0.045	0.068
17	1,700	8,500	5.24	0.052	0.047	0.090
18	1,800	9,000	5.68	0.055	0.051	0.096
19	1,900	9,500	5.49	0.058	0.052	0.086
20	2,000	10,000	5.83	0.061	0.055	0.087
21	2,100	10,500	6.64	0.064	0.059	0.112
22	2,200	11,000	7.13	0.067	0.061	0.121
23	2,300	11,500	7.59	0.070	0.067	0.125
24	2,400	12,000	7.47	0.073	0.065	0.130
25	2,500	12,500	7.63	0.076	0.068	0.123

Table A.12: Results of the *buffering overhead* of SBP-L in Chassis models.

M	D	A	O_{sbpl}(%)	MAT_{idx}(ms)	MAT_{init}(ms)	MAT_{local}(ms)
26	2,600	13,000	8.91	0.079	0.072	0.174
27	2,700	13,500	8.60	0.082	0.076	0.139
28	2,800	14,000	8.38	0.085	0.078	0.135
29	2,900	14,500	9.53	0.088	0.080	0.180
30	3,000	15,000	9.89	0.091	0.082	0.183
31	3,100	15,500	10.96	0.095	0.087	0.215
32	3,200	16,000	9.96	0.097	0.091	0.161
33	3,300	16,500	11.10	0.101	0.091	0.196
34	3,400	17,000	11.31	0.103	0.093	0.212
35	3,500	17,500	11.88	0.107	0.097	0.216
36	3,600	18,000	12.44	0.110	0.103	0.244
37	3,700	18,500	12.51	0.113	0.104	0.233
38	3,800	19,000	12.59	0.115	0.106	0.228
39	3,900	19,500	12.81	0.118	0.113	0.245
40	4,000	20,000	12.85	0.122	0.110	0.234
41	4,100	20,500	14.27	0.125	0.118	0.267
42	4,200	21,000	13.87	0.128	0.117	0.253
43	4,300	21,500	13.76	0.131	0.119	0.247
44	4,400	22,000	16.01	0.134	0.121	0.323
45	4,500	22,500	14.64	0.137	0.123	0.265
46	4,600	23,000	16.55	0.140	0.128	0.332
47	4,700	23,500	16.80	0.143	0.129	0.353
48	4,800	24,000	15.32	0.146	0.132	0.285
49	4,900	24,500	17.18	0.150	0.134	0.339
50	5,000	25,000	16.70	0.152	0.138	0.321

A.2 Evaluation of Scheduling Design

This section provides the detailed results of the evaluation benchmarks of *Time-Triggered Scheduling (TTS)* and *Fixed-Priority Scheduling (FPS)* synthesis.

The attributes of the tables listed below are described as: M – the model identification number, U_{cp}(%) – the computation load, U_t(%) – the total load, J – the number of jobs and communication blocks, F_{tts}(s) – the run-time of TTS synthesis to provide the first feasible solution, F_{fps}(s) – the run-time of FPS synthesis to provide the first feasible solution, O_{tts}(s) – the run-time of TTS synthesis to provide the optimal solution, O_{fps}(s) – the run-time of FPS synthesis to provide the optimal solution, P_{tts}(s) – the number of preemptions in the optimal solution of TTS, and P_{fps}(s) – the number of preemptions in the optimal solution of FPS. Values equal to "NA" for P_{fps} and P_{tts} indicate that the model has an infeasible schedule. In this case, F_{tts}(s) and F_{fps}(s) indicate the time it takes to find that the schedule is infeasible.

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-0	30	32.14	23	0.16	0.15	0.16	0.17	0	0
30-1	30	32.11	115	1.71	1.13	0.90	1.15	0	0
30-10	30	31.60	111	1.72	1.69	1.05	1.68	2	4
30-11	30	31.93	231	3.52	5.05	3.74	5.76	0	2
30-12	30	31.71	223	2.67	2.66	2.42	2.58	0	0
30-13	30	31.00	103	0.70	0.71	0.53	0.63	0	0
30-14	30	31.64	183	1.67	1.06	1.22	0.99	0	0
30-15	30	31.39	203	1.87	2.10	1.35	1.68	0	0
30-16	30	31.87	507	21.01	26.32	91.30	17.86	34	35
30-17	30	30.70	207	7.08	0.81	21.04	0.86	16	16
30-18	30	31.47	407	11.49	1.67	30.94	1.72	37	37
30-19	30	31.84	457	66.00	5.79	953.17	5.15	33	38
30-2	30	31.56	55	0.31	0.33	0.36	0.33	0	0
30-20	30	30.69	107	4.89	0.47	0.81	0.50	0	1
30-21	30	31.61	257	6.71	6.00	11.31	7.19	13	15
30-22	30	31.96	557	26.62	38.45	27.13	26.22	2	3
30-23	30	32.30	577	43.73	93.28	39.45	59.70	13	23
30-24	30	30.88	277	11.44	13.61	17.37	11.40	6	11

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-25	30	31.00	137	7.50	1.64	2.00	1.63	0	1
30-26	30	32.05	477	15.97	25.90	16.56	22.25	2	7
30-27	30	31.28	427	13.70	22.16	67.30	12.90	33	39
30-28	30	30.92	47	0.35	0.24	0.35	0.25	0	0
30-29	30	31.26	227	12.61	5.10	33.30	5.65	13	19
30-3	30	31.82	95	0.55	0.50	0.43	0.50	0	0
30-30	30	32.26	527	157.65	46.07	76.32	27.98	17	37
30-31	30	31.81	522	30.70	14.13	14.05	21.14	2	3
30-32	30	31.29	222	3.32	1.26	2.94	1.16	1	2
30-33	30	31.74	422	10.14	2.16	8.18	2.15	6	5
30-34	30	32.05	472	16.25	20.56	16.62	14.15	21	31
30-35	30	30.78	132	1.29	0.81	2.45	0.80	0	1
30-36	30	30.73	272	8.24	4.95	6.59	4.03	9	9
30-37	30	32.19	572	16.41	36.60	17.42	30.60	2	3
30-38	30	31.54	552	190.22	39.15	185.36	18.82	19	20
30-39	30	31.24	252	4.67	3.02	4.10	2.41	1	1
30-4	30	31.47	53	0.31	0.32	0.27	0.30	0	0
30-40	30	31.82	452	10.37	4.38	10.30	3.77	3	3
30-41	30	31.38	502	17.87	8.88	12.67	7.54	3	3
30-42	30	32.44	1,052	27.66	9.01	37.35	8.27	8	9
30-43	30	32.74	1,072	317.17	7.81	287.15	7.86	86	96
30-44	30	33.03	1,077	102.15	159.83	230.84	45.24	67	77
30-45	30	32.83	1,057	27.08	62.40	125.22	27.40	61	66
30-46	30	32.56	423	6.26	2.16	6.41	2.09	2	4
30-47	30	33.31	431	7.93	20.61	8.17	10.10	10	14
30-48	30	32.44	215	1.42	1.01	1.30	1.00	0	0
30-49	30	32.60	411	6.38	1.74	8.94	1.79	34	36
30-5	30	31.75	211	4.02	4.79	3.66	4.05	6	14
30-6	30	31.03	91	0.66	0.44	0.48	0.43	0	0

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-7	30	31.28	171	1.33	0.77	1.17	0.77	0	2
30-8	30	31.53	191	2.35	2.83	2.00	2.61	0	2
30-9	30	30.38	55	0.32	0.27	0.32	0.28	0	0
33-0	33	35.14	23	0.13	0.14	0.14	0.16	0	0
33-1	33	35.11	115	1.85	1.17	0.92	1.14	0	0
33-10	33	34.60	111	3.21	1.42	0.81	1.30	2	4
33-11	33	34.93	231	3.80	5.01	3.62	5.75	0	2
33-12	33	34.71	223	2.96	2.68	2.37	2.66	0	0
33-13	33	34.00	103	0.59	0.71	0.50	0.64	0	0
33-14	33	34.64	183	1.46	1.08	1.28	1.01	0	0
33-15	33	34.39	203	1.64	2.24	1.35	1.81	0	1
33-16	33	34.87	507	46.68	32.98	350.42	17.87	38	42
33-17	33	33.70	207	15.87	0.84	16.73	0.84	16	17
33-18	33	34.47	407	14.25	1.77	47.62	1.76	42	42
33-19	33	34.84	457	21.92	6.42	126.71	5.45	40	41
33-2	33	34.56	55	0.28	0.34	0.30	0.32	0	0
33-20	33	33.69	107	2.34	0.47	0.82	0.48	0	1
33-21	33	34.61	257	8.89	10.27	169.18	8.36	17	20
33-22	33	34.96	557	37.24	47.65	4,054.40	27.76	17	23
33-23	33	35.30	577	29.53	85.85	394.71	60.95	12	28
33-24	33	33.88	277	11.07	14.04	839.45	11.05	6	11
33-25	33	34.00	137	3.23	1.68	2.12	1.65	0	1
33-26	33	35.05	477	29.79	23.79	15.95	23.25	2	7
33-27	33	34.28	427	12.36	19.43	75.24	14.25	35	41
33-28	33	33.92	47	0.42	0.24	0.36	0.26	0	0
33-29	33	34.26	227	5.10	5.42	46.02	4.43	14	20
33-3	33	34.82	95	0.54	0.57	0.42	0.54	0	2
33-30	33	35.26	527	27.49	54.24	2,176.09	26.69	17	42
33-31	33	34.81	522	49.56	17.37	15.98	14.74	2	3

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-32	33	34.29	222	6.32	0.81	28.65	0.85	21	21
33-33	33	34.74	422	9.92	2.17	10.55	2.09	5	5
33-34	33	35.05	472	24.79	18.56	20.06	14.49	23	34
33-35	33	33.78	132	1.44	0.79	2.20	0.79	0	1
33-36	33	33.73	272	5.09	4.84	5.38	3.95	12	11
33-37	33	35.19	572	110.18	43.60	22.03	32.67	3	2
33-38	33	34.54	552	11.71	32.38	21.85	18.42	22	23
33-39	33	34.24	252	4.91	3.04	3.73	2.45	1	1
33-4	33	34.47	53	0.35	0.31	0.34	0.31	0	0
33-40	33	34.82	452	9.51	4.27	9.78	3.64	2	3
33-41	33	34.38	502	17.39	9.00	13.40	7.65	4	3
33-42	33	35.44	1,052	34.28	8.83	40.15	8.24	8	9
33-43	33	35.74	1,072	270.39	8.13	946.32	7.98	144	138
33-44	33	36.03	1,077	62.39	152.30	175.61	49.68	68	77
33-45	33	35.83	1,057	28.31	91.47	368.73	25.77	124	111
33-46	33	35.56	423	5.72	2.20	6.15	2.14	2	3
33-47	33	36.31	431	8.01	6.78	7.08	7.76	10	15
33-48	33	35.44	215	1.51	1.04	1.26	0.99	0	2
33-49	33	35.60	411	5.69	1.82	13.54	1.87	42	44
33-5	33	34.75	211	4.35	4.24	3.02	3.68	0	2
33-6	33	34.03	91	0.74	0.44	0.48	0.45	0	0
33-7	33	34.28	171	1.55	0.79	1.25	0.77	0	2
33-8	33	34.53	191	2.37	2.75	1.83	2.57	0	2
33-9	33	33.38	55	0.48	0.27	0.27	0.28	0	0
36-0	36	38.14	23	0.14	0.15	0.16	0.15	0	0
36-1	36	38.11	115	1.17	1.11	0.89	1.12	0	0
36-10	36	37.60	111	1.50	1.47	1.49	1.33	4	4
36-11	36	37.93	231	3.97	5.06	3.81	5.68	0	2
36-12	36	37.71	223	2.95	2.50	3.34	2.50	0	0

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
36-13	36	37.00	103	0.74	0.71	0.51	0.64	0	0
36-14	36	37.64	183	1.54	1.08	1.24	1.00	0	0
36-15	36	37.39	203	1.67	2.29	1.71	1.87	0	2
36-16	36	37.87	507	32.60	32.38	801.34	17.12	45	45
36-17	36	36.70	207	4.30	0.80	37.57	0.88	21	21
36-18	36	37.47	407	13.63	1.75	45.31	1.82	47	47
36-19	36	37.84	457	21.54	5.99	71.46	5.36	42	47
36-2	36	37.56	55	0.33	0.34	0.29	0.33	0	0
36-20	36	36.69	107	3.43	0.47	0.84	0.48	0	1
36-21	36	37.61	257	7.06	5.47	331.36	4.48	21	21
36-22	36	37.96	557	27.21	38.37	20.54	28.90	2	7
36-23	36	38.30	577	38.13	81.49	476.40	63.01	17	32
36-24	36	36.88	277	14.86	17.30	646.84	11.48	6	12
36-25	36	37.00	137	2.78	1.66	2.05	1.65	0	1
36-26	36	38.05	477	24.12	27.03	14.13	22.54	2	8
36-27	36	37.28	427	19.98	21.06	350.50	12.86	62	61
36-28	36	36.92	47	0.39	0.25	0.29	0.25	0	0
36-29	36	37.26	227	12.68	3.88	89.87	3.67	20	21
36-3	36	37.82	95	0.51	0.58	0.61	0.55	0	2
36-30	36	38.26	527	110.38	54.82	2,325.25	26.10	43	43
36-31	36	37.81	522	20.09	17.31	41.63	14.69	3	3
36-32	36	37.29	222	5.07	0.92	21.94	0.84	21	21
36-33	36	37.74	422	8.47	2.26	9.55	2.16	5	6
36-34	36	38.05	472	27.05	30.32	428.82	13.44	44	45
36-35	36	36.78	132	1.22	0.77	2.27	0.76	0	1
36-36	36	36.73	272	7.09	4.87	9.89	4.01	12	13
36-37	36	38.19	572	106.34	41.21	21.15	28.46	2	3
36-38	36	37.54	552	203.95	29.79	20.60	18.89	25	26
36-39	36	37.24	252	4.80	3.05	4.41	2.44	1	1

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
36-4	36	37.47	53	0.41	0.31	0.26	0.31	0	0
36-40	36	37.82	452	10.34	4.42	10.29	3.77	2	3
36-41	36	37.38	502	16.20	8.96	13.42	7.65	3	3
36-42	36	38.44	1,052	28.15	7.08	56.94	6.51	10	10
36-43	36	38.74	1,072	189.61	12.44	665.92	8.17	160	145
36-44	36	39.03	1,077	59.64	202.91	396.23	51.07	120	117
36-45	36	38.83	1,057	25.24	91.39	291.51	25.75	121	112
36-46	36	38.56	423	6.84	2.19	7.92	2.08	4	4
36-47	36	39.31	431	10.23	6.72	10.80	9.08	18	22
36-48	36	38.44	215	2.04	1.02	1.42	0.99	0	2
36-49	36	38.60	411	6.88	1.83	9.68	1.85	42	44
36-5	36	37.75	211	4.06	4.25	2.56	3.64	0	2
36-6	36	37.03	91	0.93	0.44	0.64	0.44	0	0
36-7	36	37.28	171	1.61	0.79	1.28	0.78	0	2
36-8	36	37.53	191	2.65	3.10	1.74	2.81	0	4
36-9	36	36.38	55	0.32	0.27	0.28	0.29	0	0
39-0	39	41.14	23	0.15	0.14	0.14	0.15	0	0
39-1	39	41.11	115	1.20	1.15	0.92	1.15	0	0
39-10	39	40.60	111	3.61	1.47	3.53	1.32	4	6
39-11	39	40.93	231	5.46	4.56	386.80	4.74	8	10
39-12	39	40.71	223	3.06	5.26	2.77	4.42	0	2
39-13	39	40.00	103	0.76	0.71	0.53	0.62	0	0
39-14	39	40.64	183	1.67	1.08	1.28	1.00	0	0
39-15	39	40.39	203	1.91	2.31	1.76	1.86	0	2
39-16	39	40.87	507	30.62	21.53	28.44	16.40	6	10
39-17	39	39.70	207	12.93	0.84	11.56	0.86	21	22
39-18	39	40.47	407	10.21	1.77	14.90	1.79	47	48
39-19	39	40.84	457	23.32	6.10	237.70	5.35	48	50
39-2	39	40.56	55	0.30	0.35	0.30	0.32	0	0

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-20	39	39.69	107	2.58	0.49	0.83	0.48	0	1
39-21	39	40.61	257	6.61	5.49	152.72	4.45	23	23
39-22	39	40.96	557	104.55	37.05	54.84	27.18	3	8
39-23	39	41.30	577	55.19	97.82	2,740.13	69.12	17	43
39-24	39	39.88	277	10.96	12.88	1,355.21	11.38	12	16
39-25	39	40.00	137	2.49	1.68	2.08	1.65	0	1
39-26	39	41.05	477	34.14	28.94	18.57	23.97	2	14
39-27	39	40.28	427	20.95	18.57	267.43	13.07	63	63
39-28	39	39.92	47	0.29	0.25	0.29	0.26	0	0
39-29	39	40.26	227	4.31	3.76	112.71	3.47	21	22
39-3	39	40.82	95	0.48	0.41	0.57	0.42	0	2
39-30	39	41.26	527	32.88	48.33	1,953.70	28.76	44	48
39-31	39	40.81	522	17.90	17.88	43.37	15.16	4	3
39-32	39	40.29	222	3.77	0.83	13.02	0.86	21	21
39-33	39	40.74	422	6.74	2.22	8.94	2.17	5	6
39-34	39	41.05	472	23.18	21.46	158.07	14.07	53	48
39-35	39	39.78	132	1.21	0.74	2.43	0.74	0	1
39-36	39	39.73	272	5.27	4.80	6.35	3.93	13	14
39-37	39	41.19	572	51.36	52.79	41.90	32.64	22	24
39-38	39	40.54	552	12.11	30.72	17.77	18.59	28	29
39-39	39	40.24	252	3.77	3.05	4.12	2.46	1	1
39-4	39	40.47	53	0.29	0.31	0.24	0.30	0	0
39-40	39	40.82	452	17.15	1.68	45.79	1.75	52	52
39-41	39	40.38	502	14.90	9.01	13.76	7.58	4	3
39-42	39	41.44	1,052	29.51	7.06	47.03	6.55	16	11
39-43	39	41.74	1,072	295.37	9.03	655.60	8.96	157	160
39-44	39	42.03	1,077	73.30	200.30	21,740.77	54.84	119	118
39-45	39	41.83	1,057	24.16	89.97	3,588.81	24.06	119	113
39-46	39	41.56	423	5.24	2.22	8.39	2.18	4	4

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-47	39	42.31	431	9.23	6.83	13.12	9.19	39	39
39-48	39	41.44	215	1.79	1.01	1.27	1.00	0	2
39-49	39	41.60	411	6.07	1.90	12.35	1.89	53	52
39-5	39	40.75	211	5.95	4.29	2.25	3.73	0	2
39-6	39	40.03	91	0.72	0.44	0.65	0.45	0	0
39-7	39	40.28	171	1.49	0.79	1.20	0.77	0	2
39-8	39	40.53	191	2.32	2.24	1.74	2.18	0	3
39-9	39	39.38	55	0.33	0.27	0.36	0.28	0	0
42-0	42	44.14	23	0.15	0.15	0.14	0.15	0	0
42-1	42	44.11	115	0.94	1.18	0.93	1.14	0	0
42-10	42	43.60	111	10.48	1.56	1.21	1.42	4	6
42-11	42	43.93	231	3.77	5.16	3.40	5.89	0	2
42-12	42	43.71	223	2.57	5.00	2.64	4.25	0	2
42-13	42	43.00	103	0.63	0.73	0.51	0.62	0	0
42-14	42	43.64	183	1.59	1.18	1.27	1.05	0	2
42-15	42	43.39	203	1.56	2.32	1.75	1.86	0	2
42-16	42	43.87	507	26.73	33.70	21,740.36	16.43	55	54
42-17	42	42.70	207	4.78	0.82	28.99	0.85	26	26
42-18	42	43.47	407	10.26	1.78	36.08	1.80	52	53
42-19	42	43.84	457	14.66	6.24	1,148.46	5.44	57	57
42-2	42	43.56	55	0.31	0.34	0.34	0.33	0	0
42-20	42	42.69	107	2.14	0.50	0.86	0.47	0	1
42-21	42	43.61	257	5.10	5.60	115.74	4.57	24	24
42-22	42	43.96	557	111.69	37.03	47.97	28.62	3	8
42-23	42	44.30	577	252.02	79.62	2,045.87	67.68	17	48
42-24	42	42.88	277	47.87	12.90	425.76	11.35	11	16
42-25	42	43.00	137	1.51	1.92	2.18	1.75	0	11
42-26	42	44.05	477	28.21	37.57	62.34	23.30	52	53
42-27	42	43.28	427	13.98	23.53	129.59	12.70	66	70

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-28	42	42.92	47	0.28	0.25	0.29	0.25	0	0
42-29	42	43.26	227	3.23	4.90	77.49	4.38	25	28
42-3	42	43.82	95	0.44	0.43	5.21	0.43	10	10
42-30	42	44.26	527	102.73	50.38	158.61	26.77	48	48
42-31	42	43.81	522	15.83	17.84	26.31	15.02	3	3
42-32	42	43.29	222	3.74	0.83	14.83	0.85	21	21
42-33	42	43.74	422	7.40	2.19	8.34	2.13	5	6
42-34	42	44.05	472	14.18	21.81	202.18	13.72	56	51
42-35	42	42.78	132	1.22	0.92	1.26	1.05	0	11
42-36	42	42.73	272	4.67	4.99	6.74	4.11	15	16
42-37	42	44.19	572	37.49	42.28	32.05	31.36	3	3
42-38	42	43.54	552	16.41	27.42	257.76	18.47	35	33
42-39	42	43.24	252	3.18	3.00	4.06	2.45	1	1
42-4	42	43.47	53	0.24	0.31	0.27	0.30	0	0
42-40	42	43.82	452	18.19	1.77	31.82	1.79	52	52
42-41	42	43.38	502	10.97	9.02	127.16	7.76	5	4
42-42	42	44.44	1,052	18.73	7.07	59.57	6.54	10	11
42-43	42	44.74	1,072	61.35	8.81	1,002.60	8.58	178	175
42-44	42	45.03	1,077	62.94	265.20	244.26	63.98	122	127
42-45	42	44.83	1,057	27.51	81.28	167.64	22.75	122	117
42-46	42	44.56	423	4.82	2.26	7.87	2.18	4	6
42-47	42	45.31	431	7.56	7.09	11.90	9.16	38	40
42-48	42	44.44	215	1.36	1.03	1.29	1.02	0	2
42-49	42	44.60	411	4.84	1.97	7.77	2.02	50	52
42-5	42	43.75	211	2.56	4.20	2.02	3.64	0	2
42-6	42	43.03	91	0.59	0.44	0.59	0.45	0	0
42-7	42	43.28	171	1.29	0.79	1.25	0.80	0	2
42-8	42	43.53	191	1.77	2.04	12.87	2.00	20	20
42-9	42	42.38	55	0.28	0.27	0.27	0.28	0	0

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-0	45	47.14	23	0.14	0.14	0.12	0.15	0	0
45-1	45	47.11	115	0.83	1.16	0.88	1.17	0	0
45-10	45	46.60	111	0.78	1.50	1.42	1.38	5	6
45-11	45	46.93	231	4.14	6.14	10.98	5.80	8	12
45-12	45	46.71	223	2.54	5.23	2.35	4.30	0	2
45-13	45	46.00	103	0.56	0.71	0.54	0.64	0	0
45-14	45	46.64	183	1.43	1.17	1.31	1.06	0	2
45-15	45	46.39	203	1.27	2.30	1.72	1.82	0	2
45-16	45	46.87	507	21.18	25.12	21,744.04	18.12	57	61
45-17	45	45.70	207	7.81	0.84	18.71	0.87	26	27
45-18	45	46.47	407	7.94	1.83	15.92	1.82	58	58
45-19	45	46.84	457	26.51	6.47	228.03	5.71	58	60
45-2	45	46.56	55	0.34	0.34	0.30	0.32	0	0
45-20	45	45.69	107	1.00	0.47	0.82	0.47	0	1
45-21	45	46.61	257	6.58	5.50	232.10	4.52	24	30
45-22	45	46.96	557	108.40	43.10	30.22	28.11	3	9
45-23	45	47.30	577	1,384.31	103.21	11,416.51	65.06	25	48
45-24	45	45.88	277	31.20	14.04	276.08	11.61	11	17
45-25	45	46.00	137	1.49	1.85	1.71	1.75	0	11
45-26	45	47.05	477	28.86	29.80	49.73	24.22	52	53
45-27	45	46.28	427	15.31	18.60	414.43	13.08	73	73
45-28	45	45.92	47	0.28	0.25	0.29	0.27	0	0
45-29	45	46.26	227	3.10	4.80	78.91	4.49	24	30
45-3	45	46.82	95	0.41	0.42	4.50	0.44	10	10
45-30	45	47.26	527	32.61	64.89	1,637.79	28.83	48	53
45-31	45	46.81	522	16.48	18.01	21.72	15.30	3	4
45-32	45	46.29	222	3.21	0.85	19.62	0.84	21	21
45-33	45	46.74	422	6.23	2.31	10.87	2.18	11	7
45-34	45	47.05	472	13.92	22.13	112.62	13.87	54	54

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-35	45	45.78	132	1.12	0.90	1.20	1.04	0	11
45-36	45	45.73	272	17.69	4.82	8.09	3.95	16	17
45-37	45	47.19	572	218.75	47.44	625.53	30.70	22	33
45-38	45	46.54	552	23.43	27.07	32.62	18.29	37	36
45-39	45	46.24	252	3.08	3.04	3.62	2.44	1	1
45-4	45	46.47	53	0.27	0.31	0.31	0.31	0	0
45-40	45	46.82	452	11.44	1.71	28.88	1.76	52	52
45-41	45	46.38	502	9.39	9.01	12.42	7.58	5	4
45-42	45	47.44	1,052	16.90	3.98	59.35	4.00	10	12
45-43	45	47.74	1,072	146.71	8.79	724.99	8.68	186	183
45-44	45	48.03	1,077	65.87	189.45	320.45	60.39	118	127
45-45	45	47.83	1,057	82.37	71.90	320.11	22.72	184	162
45-46	45	47.56	423	4.85	1.65	7.59	1.70	4	5
45-47	45	48.31	431	6.66	7.27	12.21	10.27	40	41
45-48	45	47.44	215	1.32	0.85	1.32	0.86	0	1
45-49	45	47.60	411	5.52	1.93	112.22	1.90	62	60
45-5	45	46.75	211	2.95	4.47	2.12	3.89	0	2
45-6	45	46.03	91	0.59	0.44	0.63	0.44	0	0
45-7	45	46.28	171	1.18	0.81	1.18	0.80	0	2
45-8	45	46.53	191	2.08	2.08	10.19	2.04	20	20
45-9	45	45.38	55	0.60	0.29	0.94	0.28	0	0
48-0	48	50.14	23	0.12	0.14	0.12	0.16	0	0
48-1	48	50.11	115	0.83	1.13	0.72	1.15	0	0
48-10	48	49.60	111	9.24	1.88	15.16	1.66	6	8
48-11	48	49.93	231	6.58	5.18	3.23	5.95	0	2
48-12	48	49.71	223	2.37	4.69	2.73	4.14	0	2
48-13	48	49.00	103	0.54	0.74	0.50	0.64	0	0
48-14	48	49.64	183	1.35	1.16	1.30	1.05	0	2
48-15	48	49.39	203	1.39	2.32	1.56	1.84	0	2

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
48-16	48	49.87	507	33.77	32.28	613.49	17.54	68	65
48-17	48	48.70	207	3.77	0.83	31.99	0.87	31	31
48-18	48	49.47	407	7.55	1.81	13.05	1.83	62	63
48-19	48	49.84	457	26.89	6.64	533.23	5.82	58	67
48-2	48	49.56	55	0.37	0.33	0.31	0.32	0	0
48-20	48	48.69	107	0.93	0.46	0.91	0.49	0	1
48-21	48	49.61	257	5.04	5.72	364.68	4.70	30	31
48-22	48	49.96	557	57.72	38.01	25.26	29.08	3	9
48-23	48	50.30	577	764.89	126.74	3,633.15	63.05	43	53
48-24	48	48.88	277	30.38	26.07	94.00	11.21	11	18
48-25	48	49.00	137	1.46	1.89	1.79	1.78	0	11
48-26	48	50.05	477	21.04	33.51	44.07	23.90	52	53
48-27	48	49.28	427	15.37	18.01	247.77	12.33	76	80
48-28	48	48.92	47	0.29	0.54	0.30	0.54	0	1
48-29	48	49.26	227	3.17	4.22	151.27	3.90	25	31
48-3	48	49.82	95	0.46	0.42	2.60	0.43	10	10
48-30	48	50.26	527	198.74	51.28	1,440.13	27.87	53	58
48-31	48	49.81	522	16.94	17.86	14.77	15.14	3	4
48-32	48	49.29	222	2.78	0.85	12.74	0.88	21	21
48-33	48	49.74	422	6.61	1.66	9.85	1.66	7	7
48-34	48	50.05	472	10.77	19.49	128.13	12.99	57	58
48-35	48	48.78	132	1.08	0.89	1.17	1.02	0	11
48-36	48	48.73	272	9.30	5.12	11.45	4.08	17	19
48-37	48	50.19	572	137.55	41.34	18.73	38.06	2	53
48-38	48	49.54	552	83.33	30.37	14.67	18.61	38	39
48-39	48	49.24	252	3.30	2.90	3.45	2.29	1	2
48-4	48	49.47	53	0.31	0.31	0.32	0.31	0	0
48-40	48	49.82	452	13.62	1.68	16.71	1.71	52	53
48-41	48	49.38	502	14.77	8.85	12.41	7.59	4	4

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
48-42	48	50.44	1,052	17.29	4.04	571.03	4.12	71	61
48-43	48	50.74	1,072	264.87	9.69	715.30	9.47	196	198
48-44	48	51.03	1,077	86.40	179.07	21,716.54	51.93	179	168
48-45	48	50.83	1,057	108.19	68.86	21,761.47	22.57	172	163
48-46	48	50.56	423	5.10	1.68	9.22	1.71	24	24
48-47	48	51.31	431	6.56	7.04	14.77	9.17	46	48
48-48	48	50.44	215	1.30	0.85	1.72	0.86	10	10
48-49	48	50.60	411	5.12	2.00	160.24	1.99	68	68
48-5	48	49.75	211	2.25	4.34	2.26	3.79	0	2
48-6	48	49.03	91	0.53	0.44	0.59	0.46	0	0
48-7	48	49.28	171	1.23	0.69	1.46	0.71	0	1
48-8	48	49.53	191	1.79	2.05	8.32	2.02	20	22
48-9	48	48.38	55	0.60	0.28	0.89	0.28	0	0
51-0	51	53.14	23	0.13	0.14	0.14	0.15	0	0
51-1	51	53.11	115	0.86	1.17	0.79	1.15	0	0
51-10	51	52.60	111	10.11	1.77	13.93	1.54	6	10
51-11	51	52.93	231	7.16	6.15	78.62	6.14	8	14
51-12	51	52.71	223	2.44	5.15	2.41	4.33	0	2
51-13	51	52.00	103	0.56	0.72	0.63	0.64	0	0
51-14	51	52.64	183	1.50	1.16	1.22	1.06	0	2
51-15	51	52.39	203	1.61	2.31	1.68	1.85	0	2
51-16	51	52.87	507	283.61	25.38	21,767.39	14.64	109	72
51-17	51	51.70	207	8.01	0.84	12.69	0.85	32	32
51-18	51	52.47	407	7.19	1.84	30.10	1.88	73	72
51-19	51	52.84	457	30.14	5.19	304.62	4.61	66	71
51-2	51	52.56	55	0.32	0.34	0.28	0.32	0	0
51-20	51	51.69	107	0.80	0.43	3.69	0.45	5	5
51-21	51	52.61	257	6.86	4.97	392.87	4.21	37	34
51-22	51	52.96	557	43.63	42.91	596.86	29.75	5	9

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-23	51	53.30	577	638.50	128.91	1,372.04	60.67	49	54
51-24	51	51.88	277	55.56	13.30	1,017.71	11.76	16	27
51-25	51	52.00	137	1.40	2.09	2.92	1.96	0	15
51-26	51	53.05	477	34.75	34.99	36.36	22.86	52	58
51-27	51	52.28	427	13.04	25.78	426.21	11.87	81	83
51-28	51	51.92	47	6.24	0.50	0.29	0.50	0	1
51-29	51	52.26	227	6.05	3.06	66.49	2.88	26	33
51-3	51	52.82	95	0.47	0.42	2.34	0.44	10	10
51-30	51	53.26	527	71.11	53.75	3,627.57	28.77	52	63
51-31	51	52.81	522	14.95	17.79	326.27	14.87	3	5
51-32	51	52.29	222	2.62	0.83	9.00	0.87	21	22
51-33	51	52.74	422	8.45	1.66	27.29	1.68	26	26
51-34	51	53.05	472	39.80	25.63	21,701.00	12.52	60	69
51-35	51	51.78	132	1.21	0.88	1.06	0.99	0	11
51-36	51	51.73	272	11.40	6.65	60.27	5.33	20	28
51-37	51	53.19	572	177.02	52.28	1,048.37	32.96	22	34
51-38	51	52.54	552	100.85	27.37	20.53	18.76	42	43
51-39	51	52.24	252	3.29	2.90	3.62	2.30	1	2
51-4	51	52.47	53	0.29	0.33	0.26	0.31	0	0
51-40	51	52.82	452	12.33	1.62	10.14	1.68	52	53
51-41	51	52.38	502	9.57	8.91	11.84	7.62	5	5
51-42	51	53.44	1,052	16.75	4.12	269.36	4.16	61	62
51-43	51	53.74	1,072	171.25	9.88	787.76	9.49	203	207
51-44	51	54.03	1,077	59.65	186.33	21,744.26	50.28	177	172
51-45	51	53.83	1,057	46.10	15.41	21,784.19	21.29	167	168
51-46	51	53.56	423	4.69	1.67	7.32	1.68	24	24
51-47	51	54.31	431	7.15	6.85	14.71	8.88	47	50
51-48	51	53.44	215	1.21	0.84	1.33	0.85	10	10
51-49	51	53.60	411	4.76	2.09	58.75	2.05	81	76

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-5	51	52.75	211	2.00	4.69	2.20	3.97	0	2
51-6	51	52.03	91	0.50	0.44	0.54	0.45	0	0
51-7	51	52.28	171	1.11	0.66	2.67	0.69	8	8
51-8	51	52.53	191	1.65	2.61	3.30	2.48	8	10
51-9	51	51.38	55	0.52	0.27	0.59	0.29	0	0
54-0	54	56.14	23	0.11	0.15	0.12	0.16	0	0
54-1	54	56.11	115	0.68	1.13	0.73	1.18	0	1
54-10	54	55.60	111	3.63	1.56	4.04	1.38	6	11
54-11	54	55.93	231	2.66	5.34	3.07	6.04	0	2
54-12	54	55.71	223	3.52	5.16	2.74	4.40	0	2
54-13	54	55.00	103	0.54	0.72	0.49	0.63	0	0
54-14	54	55.64	183	1.77	1.14	1.22	1.05	0	2
54-15	54	55.39	203	1.50	2.32	1.37	1.86	0	2
54-16	54	55.87	507	40.72	25.82	495.80	15.53	77	76
54-17	54	54.70	207	4.14	0.84	21.29	0.86	36	36
54-18	54	55.47	407	7.74	1.90	26.28	1.92	79	78
54-19	54	55.84	457	13.52	5.17	663.57	4.70	75	78
54-2	54	55.56	55	0.30	0.35	0.32	0.33	0	0
54-20	54	54.69	107	0.86	0.43	4.48	0.45	5	6
54-21	54	55.61	257	4.28	5.04	349.30	4.24	33	36
54-22	54	55.96	557	271.04	42.44	207.42	28.39	3	9
54-23	54	56.30	577	3,595.32	126.25	4,881.69	64.44	47	64
54-24	54	54.88	277	82.00	13.54	812.70	11.80	17	28
54-25	54	55.00	137	1.78	2.07	3.80	1.93	0	15
54-26	54	56.05	477	39.75	36.34	60.06	25.99	52	57
54-27	54	55.28	427	14.12	16.80	241.06	12.03	85	90
54-28	54	54.92	47	4.54	0.47	0.30	0.48	0	1
54-29	54	55.26	227	44.64	3.32	387.04	3.22	33	39
54-3	54	55.82	95	0.47	0.41	2.34	0.45	10	10

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
54-30	54	56.26	527	669.11	99.10	1,209.27	28.03	60	78
54-31	54	55.81	522	16.70	17.97	304.63	14.84	3	5
54-32	54	55.29	222	2.71	0.83	4.71	0.89	21	22
54-33	54	55.74	422	7.71	1.65	38.68	1.70	27	27
54-34	54	56.05	472	13.96	20.28	65.99	13.47	69	74
54-35	54	54.78	132	1.21	0.83	1.07	0.97	0	11
54-36	54	54.73	272	104.99	6.55	18.21	5.23	23	30
54-37	54	56.19	572	26.48	45.58	234.72	31.97	5	5
54-38	54	55.54	552	20.75	28.35	814.54	18.53	67	49
54-39	54	55.24	252	3.93	2.84	4.01	2.27	1	2
54-4	54	55.47	53	0.29	0.33	0.27	0.31	0	0
54-40	54	55.82	452	10.70	1.68	16.03	1.69	52	53
54-41	54	55.38	502	13.70	8.99	22.97	7.65	5	5
54-42	54	56.44	1,052	23.29	4.12	219.53	4.19	62	63
54-43	54	56.74	1,072	525.27	9.86	3,027.78	9.58	222	224
54-44	54	57.03	1,077	64.12	196.83	21,743.23	66.48	169	177
54-45	54	56.83	1,057	39.63	62.05	21,774.33	23.17	173	168
54-46	54	56.56	423	4.96	1.71	7.02	1.73	24	26
54-47	54	57.31	431	8.05	4.15	11.30	4.03	47	52
54-48	54	56.44	215	1.23	0.88	1.46	0.88	10	11
54-49	54	56.60	411	5.06	2.09	564.97	2.10	76	77
54-5	54	55.75	211	2.00	4.64	93.62	4.00	22	23
54-6	54	55.03	91	0.90	0.47	0.64	0.46	0	2
54-7	54	55.28	171	1.20	0.71	2.10	0.72	8	10
54-8	54	55.53	191	1.74	2.09	4.32	2.02	20	22
54-9	54	54.38	55	0.58	0.27	0.63	0.29	0	0
57-0	57	59.14	23	0.12	0.14	0.13	0.14	0	0
57-1	57	59.11	115	0.74	1.18	0.73	1.14	0	0
57-10	57	58.60	111	2.96	1.58	20.05	1.39	6	14

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-11	57	58.93	231	293.20	6.64	277.85	6.00	8	36
57-12	57	58.71	223	2.06	5.54	2.03	4.65	0	2
57-13	57	58.00	103	0.53	0.72	0.51	0.64	0	0
57-14	57	58.64	183	1.52	1.16	1.31	1.06	0	2
57-15	57	58.39	203	1.47	2.26	1.38	1.84	0	2
57-16	57	58.87	507	28.29	22.46	28.46	16.50	7	13
57-17	57	57.70	207	4.80	0.89	10.72	0.92	36	36
57-18	57	58.47	407	7.34	1.90	17.72	1.91	83	83
57-19	57	58.84	457	16.66	5.29	509.05	4.70	81	86
57-2	57	58.56	55	0.27	0.36	0.28	0.33	0	0
57-20	57	57.69	107	0.70	0.45	7.67	0.44	6	6
57-21	57	58.61	257	72.03	5.00	623.12	4.16	38	41
57-22	57	58.96	557	867.68	61.95	3,708.12	32.30	51	88
57-23	57	59.30	577	6,904.96	128.19	1,419.10	65.56	49	69
57-24	57	57.88	277	50.38	14.81	509.46	12.18	16	31
57-25	57	58.00	137	1.47	2.06	1.75	1.90	0	15
57-26	57	59.05	477	26.56	34.21	158.10	22.18	52	58
57-27	57	58.28	427	9.85	21.64	338.69	11.53	90	93
57-28	57	57.92	47	3.35	0.44	0.36	0.45	0	1
57-29	57	58.26	227	653.41	3.55	385.68	3.35	35	40
57-3	57	58.82	95	0.49	0.41	1.77	0.45	10	10
57-30	57	59.26	527	1,244.61	89.73	2,412.80	25.63	62	83
57-31	57	58.81	522	13.29	15.88	21.42	14.04	5	4
57-32	57	58.29	222	4.35	0.86	113.83	0.86	41	41
57-33	57	58.74	422	7.49	1.71	25.25	1.72	27	28
57-34	57	59.05	472	14.86	23.01	73.16	13.20	75	78
57-35	57	57.78	132	1.23	0.50	1.11	0.59	0	11
57-36	57	57.73	272	26.55	6.45	16.99	5.25	24	33
57-37	57	59.19	572	1,328.10	95.06	6,206.40	39.01	22	92

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-38	57	58.54	552	142.87	26.90	769.64	19.37	68	89
57-39	57	58.24	252	3.57	2.87	3.66	2.29	2	2
57-4	57	58.47	53	0.27	0.32	0.26	0.32	0	0
57-40	57	58.82	452	9.97	1.71	12.36	1.75	52	54
57-41	57	58.38	502	11.01	9.09	15.51	7.58	5	5
57-42	57	59.44	1,052	19.20	4.09	230.67	4.22	63	64
57-43	57	59.74	1,072	273.51	10.42	21,785.66	9.89	240	241
57-44	57	60.03	1,077	241.89	259.68	21,699.71	50.93	233	218
57-45	57	59.83	1,057	76.79	19.06	21,748.59	22.06	251	213
57-46	57	59.56	423	4.85	1.69	7.92	1.73	24	26
57-47	57	60.31	431	7.41	11.52	15.75	8.92	54	58
57-48	57	59.44	215	1.32	0.86	2.90	0.87	10	12
57-49	57	59.60	411	5.47	2.13	48.56	2.13	84	84
57-5	57	58.75	211	1.95	4.77	2.28	4.08	0	2
57-6	57	58.03	91	0.78	0.46	0.52	0.46	0	2
57-7	57	58.28	171	1.14	0.71	1.85	0.71	8	10
57-8	57	58.53	191	1.77	2.10	4.75	2.05	20	22
57-9	57	57.38	55	0.51	0.27	0.54	0.30	0	0
60-0	60	62.14	23	0.12	0.18	0.12	0.18	0	1
60-1	60	62.11	115	0.74	1.87	0.87	1.64	0	4
60-10	60	61.60	111	7.01	1.64	8.26	1.48	6	14
60-11	60	61.93	231	3.04	5.60	2.88	6.26	0	8
60-12	60	61.71	223	3.77	4.93	2.30	4.33	0	2
60-13	60	61.00	103	0.60	0.71	0.49	0.64	0	0
60-14	60	61.64	183	1.68	1.16	1.37	1.06	0	2
60-15	60	61.39	203	1.39	2.30	1.27	1.84	0	2
60-16	60	61.87	507	14.40	35.86	847.22	14.19	87	91
60-17	60	60.70	207	3.65	0.88	14.04	0.89	45	42
60-18	60	61.47	407	7.76	1.95	13.06	1.97	88	88

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
60-19	60	61.84	457	22.93	5.28	21,684.06	4.76	85	90
60-2	60	61.56	55	0.29	0.34	0.27	0.35	0	0
60-20	60	60.69	107	1.21	0.42	7.95	0.48	6	6
60-21	60	61.61	257	30.04	4.05	278.01	3.37	37	43
60-22	60	61.96	557	44.61	36.50	28.14	26.81	4	11
60-23	60	62.30	577	3,162.97	95.31	3,753.40	57.87	54	74
60-24	60	60.88	277	271.60	15.45	157.74	12.06	17	37
60-25	60	61.00	137	1.49	1.55	88.59	1.51	20	20
60-26	60	62.05	477	22.11	35.06	33.97	23.76	55	59
60-27	60	61.28	427	23.96	26.22	355.19	11.48	94	101
60-28	60	60.92	47	2.72	0.41	0.31	0.40	0	1
60-29	60	61.26	227	8.85	3.38	140.01	3.12	35	43
60-3	60	61.82	95	0.46	0.43	0.84	0.45	10	12
60-30	60	62.26	527	1,351.20	49.82	1,888.85	25.67	67	88
60-31	60	61.81	522	13.70	18.13	23.81	15.19	4	5
60-32	60	61.29	222	5.02	0.83	71.63	0.86	41	41
60-33	60	61.74	422	6.86	1.72	29.21	1.71	27	28
60-34	60	62.05	472	24.97	20.58	165.65	13.11	76	82
60-35	60	60.78	132	1.12	0.51	45.61	0.54	20	20
60-36	60	60.73	272	13.45	6.62	16.97	5.27	25	35
60-37	60	62.19	572	55.78	56.53	1,350.19	35.38	53	54
60-38	60	61.54	552	73.11	31.92	497.57	17.87	57	93
60-39	60	61.24	252	3.13	2.86	3.66	2.30	2	2
60-4	60	61.47	53	0.29	0.32	0.26	0.32	0	0
60-40	60	61.82	452	9.07	1.75	13.22	1.77	52	54
60-41	60	61.38	502	9.92	9.05	151.35	7.79	7	6
60-42	60	62.44	1,052	17.77	4.14	294.05	4.18	63	65
60-43	60	62.74	1,072	436.27	11.20	5,775.17	10.39	248	241
60-44	60	63.03	1,077	105.56	267.43	4,820.27	63.35	227	223

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
60-45	60	62.83	1,057	39.28	16.70	21,744.28	22.90	218	218
60-46	60	62.56	423	4.67	1.72	10.31	1.72	26	26
60-47	60	63.31	431	7.60	7.23	11.30	9.93	54	75
60-48	60	62.44	215	1.27	0.88	1.59	0.88	10	12
60-49	60	62.60	411	4.99	2.19	56.47	2.15	92	92
60-5	60	61.75	211	4.30	4.85	3.15	4.07	0	8
60-6	60	61.03	91	0.67	0.46	0.48	0.47	0	2
60-7	60	61.28	171	1.23	0.69	2.01	0.71	8	10
60-8	60	61.53	191	1.90	2.07	11.16	2.01	20	22
60-9	60	60.38	55	0.47	0.30	0.52	0.30	4	4
63-0	63	65.14	23	0.11	0.17	0.11	0.17	0	2
63-1	63	65.11	115	1.38	1.16	0.69	1.27	0	2
63-10	63	64.60	111	12.04	1.60	9.97	1.41	8	16
63-11	63	64.93	231	14.33	6.82	64.33	6.14	8	37
63-12	63	64.71	223	5.46	6.34	1.94	5.22	0	20
63-13	63	64.00	103	0.50	0.79	0.51	0.66	0	0
63-14	63	64.64	183	1.37	1.11	1.38	1.03	0	2
63-15	63	64.39	203	1.34	2.32	1.70	1.84	0	2
63-16	63	64.87	507	25.71	25.18	21.63	16.74	14	15
63-17	63	63.70	207	2.29	0.86	24.08	0.88	46	46
63-18	63	64.47	407	7.49	2.10	672.26	2.09	94	93
63-19	63	64.84	457	14.40	5.13	289.09	4.52	89	97
63-2	63	64.56	55	0.29	0.35	0.28	0.33	0	0
63-20	63	63.69	107	0.96	0.42	8.74	0.46	6	6
63-21	63	64.61	257	19.47	4.18	384.74	3.59	45	48
63-22	63	64.96	557	20.35	40.53	29.91	28.57	4	11
63-23	63	65.30	577	4,711.55	111.25	4,660.44	65.15	60	84
63-24	63	63.88	277	365.76	15.87	646.42	12.05	23	41
63-25	63	64.00	137	1.53	1.50	59.32	1.51	20	21

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-26	63	65.05	477	17.65	37.39	30.55	25.11	53	59
63-27	63	64.28	427	8.07	17.67	265.94	11.35	99	105
63-28	63	63.92	47	1.30	0.37	0.29	0.41	0	1
63-29	63	64.26	227	4.46	1.71	73.42	1.68	37	40
63-3	63	64.82	95	0.66	0.43	1.95	0.46	10	12
63-30	63	65.26	527	1,013.84	58.57	1,292.07	28.97	92	93
63-31	63	64.81	522	22.40	16.43	32.16	15.20	3	25
63-32	63	64.29	222	6.37	0.83	36.55	0.86	41	41
63-33	63	64.74	422	5.77	1.55	21.05	1.56	28	30
63-34	63	65.05	472	13.10	24.50	71.82	13.02	78	86
63-35	63	63.78	132	0.97	0.52	32.71	0.55	20	20
63-36	63	63.73	272	14.22	6.56	33.94	5.20	30	37
63-37	63	65.19	572	1,613.77	77.10	3,520.66	41.84	22	93
63-38	63	64.54	552	150.97	34.43	1,498.10	18.33	77	98
63-39	63	64.24	252	2.89	2.90	3.82	2.30	2	3
63-4	63	64.47	53	0.23	0.31	0.26	0.31	0	0
63-40	63	64.82	452	7.05	1.73	16.86	1.75	53	54
63-41	63	64.38	502	8.29	8.97	16.22	7.68	7	6
63-42	63	65.44	1,052	14.26	4.14	242.45	4.21	64	66
63-43	63	65.74	1,072	533.93	12.98	21,723.92	12.11	310	301
63-44	63	66.03	1,077	72.27	266.61	21,751.67	73.46	220	230
63-45	63	65.83	1,057	39.76	50.48	21,754.98	24.85	248	219
63-46	63	65.56	423	4.28	1.76	10.70	1.81	26	28
63-47	63	66.31	431	6.94	7.02	17.76	9.45	75	79
63-48	63	65.44	215	1.12	0.87	1.55	0.88	10	12
63-49	63	65.60	411	4.68	2.25	38.82	2.23	103	100
63-5	63	64.75	211	1.72	5.06	2.38	4.25	0	8
63-6	63	64.03	91	0.56	0.47	0.49	0.47	0	2
63-7	63	64.28	171	1.02	0.69	1.27	0.70	8	10

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-8	63	64.53	191	2.03	2.08	1.91	2.03	20	24
63-9	63	63.38	55	0.38	0.30	0.51	0.39	4	4
66-0	66	68.14	23	0.10	0.13	0.12	0.14	4	4
66-1	66	68.11	115	8.21	1.41	0.81	1.18	0	12
66-10	66	67.60	111	1.88	1.31	49.01	1.39	16	18
66-11	66	67.93	231	9.72	5.52	7.99	6.18	0	22
66-12	66	67.71	223	2.75	5.46	2.42	4.63	0	2
66-13	66	67.00	103	0.70	0.82	0.48	0.72	0	2
66-14	66	67.64	183	1.02	1.11	1.15	1.02	2	2
66-15	66	67.39	203	1.22	0.77	1.20	0.79	40	40
66-16	66	67.87	507	74.86	28.19	3,927.45	14.38	98	104
66-17	66	66.70	207	4.14	0.89	11.02	0.90	66	47
66-18	66	67.47	407	5.91	2.04	2,108.03	2.24	103	103
66-19	66	67.84	457	21.57	5.03	3,890.04	4.50	98	106
66-2	66	67.56	55	0.25	0.34	0.26	0.32	0	0
66-20	66	66.69	107	0.67	0.45	2.14	0.44	6	7
66-21	66	67.61	257	4.88	3.67	1,446.60	3.05	46	50
66-22	66	67.96	557	88.17	44.47	66.35	31.73	6	55
66-23	66	68.30	577	3,245.32	113.42	1,546.83	55.81	58	88
66-24	66	66.88	277	1,461.70	15.24	1,230.86	10.68	26	38
66-25	66	67.00	137	2.26	1.47	51.42	1.44	20	21
66-26	66	68.05	477	28.82	35.98	132.72	24.07	53	64
66-27	66	67.28	427	20.61	13.27	430.07	10.56	122	122
66-28	66	66.92	47	1.49	0.33	0.28	0.34	0	1
66-29	66	67.26	227	68.12	1.19	893.07	1.23	61	60
66-3	66	67.82	95	1.47	0.46	2.06	0.46	20	20
66-30	66	68.26	527	1,693.13	56.61	1,293.23	26.38	89	97
66-31	66	67.81	522	36.22	16.43	219.96	15.64	7	26
66-32	66	67.29	222	4.54	0.85	13.36	0.89	41	42

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
66-33	66	67.74	422	12.45	1.57	31.95	1.60	29	30
66-34	66	68.05	472	33.18	22.51	117.09	14.27	99	99
66-35	66	66.78	132	1.67	0.52	24.34	0.58	20	20
66-36	66	66.73	272	13.38	6.16	860.85	5.27	46	47
66-37	66	68.19	572	223.00	44.80	297.02	35.20	5	65
66-38	66	67.54	552	35.44	32.12	28.51	19.80	73	103
66-39	66	67.24	252	3.95	2.86	4.07	2.36	2	3
66-4	66	67.47	53	0.45	0.31	0.25	0.32	0	0
66-40	66	67.82	452	23.00	1.81	21,689.94	1.79	104	103
66-41	66	67.38	502	14.43	8.96	14.28	7.82	8	7
66-42	66	68.44	1,052	31.86	4.18	21,699.20	4.23	117	116
66-43	66	68.74	1,072	1,155.50	12.78	21,765.40	12.20	312	312
66-44	66	69.03	1,077	487.06	255.71	21,751.84	61.97	276	273
66-45	66	68.83	1,057	91.72	15.88	9,432.78	26.04	283	264
66-46	66	68.56	423	8.42	1.76	11.09	1.85	47	46
66-47	66	69.31	431	11.04	7.48	13.60	9.01	84	84
66-48	66	68.44	215	1.94	0.85	1.81	0.87	20	20
66-49	66	68.60	411	7.82	2.32	22.25	2.29	116	108
66-5	66	67.75	211	9.90	2.78	1.86	2.60	0	6
66-6	66	67.03	91	0.89	0.49	0.54	0.45	0	2
66-7	66	67.28	171	1.59	0.72	1.35	0.75	10	11
66-8	66	67.53	191	2.62	2.04	2.08	2.01	20	24
66-9	66	66.38	55	0.63	0.31	0.55	0.30	4	4
69-0	69	71.14	23	0.15	0.13	0.10	0.13	4	4
69-1	69	71.11	115	1.51	2.11	0.75	1.90	0	4
69-10	69	70.60	111	35.77	1.26	52.18	1.52	18	20
69-11	69	70.93	231	3.90	4.18	2.43	4.18	40	40
69-12	69	70.71	223	2.94	3.50	1.97	2.83	40	40
69-13	69	70.00	103	1.04	0.82	0.47	0.71	0	2

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
69-14	69	70.64	183	1.75	0.73	1.26	0.74	2	3
69-15	69	70.39	203	1.95	0.77	1.17	0.79	40	40
69-16	69	70.87	507	46.25	25.46	17,779.24	15.20	110	112
69-17	69	69.70	207	5.39	0.92	5.80	0.98	52	52
69-18	69	70.47	407	13.09	2.05	872.15	2.08	117	108
69-19	69	70.84	457	28.66	5.14	103.88	4.55	101	111
69-2	69	70.56	55	0.40	0.34	0.26	0.33	0	0
69-20	69	69.69	107	5.97	0.44	11.45	0.45	11	11
69-21	69	70.61	257	51.80	3.60	387.02	3.00	54	53
69-22	69	70.96	557	295.83	54.55	362.84	32.77	53	60
69-23	69	71.30	577	22,601.65	159.18	4,768.05	69.78	67	123
69-24	69	69.88	277	828.13	18.51	801.56	11.75	47	47
69-25	69	70.00	137	4.74	1.53	60.04	1.50	20	22
69-26	69	71.05	477	71.40	39.39	21,640.14	26.05	103	104
69-27	69	70.28	427	37.02	4.28	265.39	4.36	124	126
69-28	69	69.92	47	0.76	0.30	0.29	0.29	0	1
69-29	69	70.26	227	19.94	1.20	395.11	1.24	62	62
69-3	69	70.82	95	1.21	0.45	3.00	0.45	21	20
69-30	69	71.26	527	4,042.83	60.78	7,712.42	25.70	96	113
69-31	69	70.81	522	19.42	6.25	19.78	5.61	103	103
69-32	69	70.29	222	4.28	0.88	7.92	0.89	41	42
69-33	69	70.74	422	15.30	1.53	38.43	1.57	49	50
69-34	69	71.05	472	55.51	25.00	139.07	12.77	105	115
69-35	69	69.78	132	2.86	0.52	27.93	0.54	20	21
69-36	69	69.73	272	43.30	5.28	2,712.15	4.33	48	49
69-37	69	71.19	572	112.85	56.69	992.29	35.62	53	55
69-38	69	70.54	552	151.40	29.21	38.24	19.26	80	107
69-39	69	70.24	252	4.99	2.90	10.51	2.27	3	3
69-4	69	70.47	53	0.38	0.32	0.23	0.31	0	0

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
69-40	69	70.82	452	15.56	1.74	21,683.14	1.76	104	103
69-41	69	70.38	502	16.02	8.98	16.99	7.63	9	7
69-42	69	71.44	1,052	33.48	4.18	21,695.43	4.19	130	118
69-43	69	71.74	1,072	1,045.97	12.92	21,781.54	12.34	325	324
69-44	69	72.03	1,077	238.95	398.96	21,740.62	78.66	277	275
69-45	69	71.83	1,057	153.97	19.53	21,730.26	25.98	297	270
69-46	69	71.56	423	8.66	1.82	12.81	1.83	47	47
69-47	69	72.31	431	11.66	8.49	35.60	11.54	84	93
69-48	69	71.44	215	1.88	0.83	2.79	0.89	20	22
69-49	69	71.60	411	7.76	2.45	28.77	2.41	121	116
69-5	69	70.75	211	4.01	4.88	3.94	4.00	2	12
69-6	69	70.03	91	0.75	0.38	3.87	0.40	8	8
69-7	69	70.28	171	2.02	0.71	2.35	0.69	18	18
69-8	69	70.53	191	4.18	3.37	4.90	3.45	18	25
69-9	69	69.38	55	0.37	0.30	0.52	0.31	4	4
72-0	72	74.14	23	0.14	0.13	0.13	0.14	4	4
72-1	72	74.11	115	1.65	1.21	0.98	1.07	20	20
72-10	72	73.60	111	1.82	1.26	40.36	1.37	18	22
72-11	72	73.93	231	5.00	4.74	2.59	4.37	40	42
72-12	72	73.71	223	4.96	4.31	2.49	3.73	0	20
72-13	72	73.00	103	1.03	0.81	0.49	0.70	0	2
72-14	72	73.64	183	2.21	0.73	2.22	0.73	22	22
72-15	72	73.39	203	2.03	0.78	1.20	0.79	40	40
72-16	72	73.87	507	33.54	29.18	3,056.13	14.51	120	118
72-17	72	72.70	207	4.93	0.91	11.59	0.92	56	56
72-18	72	73.47	407	14.86	2.15	121.54	2.11	122	118
72-19	72	73.84	457	34.50	5.14	4,022.52	4.58	115	121
72-2	72	73.56	55	0.37	0.34	0.24	0.34	0	0
72-20	72	72.69	107	3.80	0.43	4.24	0.48	11	11

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
72-21	72	73.61	257	25.30	3.67	227.97	3.06	58	57
72-22	72	73.96	557	30.85	52.46	38.89	26.52	103	104
72-23	72	74.30	577	35,934.73	202.71	21,936.82	74.15	82	132
72-24	72	72.88	277	499.53	16.88	866.00	11.95	46	52
72-25	72	73.00	137	3.71	1.50	38.51	1.47	20	22
72-26	72	74.05	477	1,839.23	45.90	1,538.50	19.17	103	125
72-27	72	73.28	427	336.63	4.19	855.59	4.37	133	133
72-28	72	72.92	47	0.48	0.27	0.31	0.27	0	1
72-29	72	73.26	227	4.35	1.20	294.13	1.23	64	65
72-3	72	73.82	95	0.78	0.45	1.23	0.46	20	20
72-30	72	74.26	527	1,903.26	61.77	4,069.27	27.86	100	119
72-31	72	73.81	522	25.25	9.83	82.60	8.97	11	26
72-32	72	73.29	222	4.02	0.89	4.55	0.94	41	42
72-33	72	73.74	422	12.44	1.61	40.35	1.56	50	51
72-34	72	74.05	472	78.20	24.47	149.49	12.83	109	122
72-35	72	72.78	132	1.75	0.52	14.54	0.55	20	21
72-36	72	72.73	272	49.46	5.46	352.06	4.39	53	52
72-37	72	74.19	572	142.47	52.44	2,852.89	32.29	55	66
72-38	72	73.54	552	84.00	33.27	39.03	19.41	82	113
72-39	72	73.24	252	4.78	2.87	5.50	2.27	3	3
72-4	72	73.47	53	0.37	0.32	0.23	0.30	0	0
72-40	72	73.82	452	14.99	1.84	21,692.97	1.81	110	103
72-41	72	73.38	502	16.99	9.02	16.43	7.57	9	9
72-42	72	74.44	1,052	32.21	4.20	21,677.50	4.34	140	120
72-43	72	74.74	1,072	684.40	13.42	21,818.36	12.52	338	347
72-44	72	75.03	1,077	225.28	249.85	21,744.49	75.17	279	293
72-45	72	74.83	1,057	168.00	81.76	21,707.70	26.73	264	314
72-46	72	74.56	423	6.94	1.91	12.56	1.96	49	49
72-47	72	75.31	431	11.33	7.96	88.53	11.41	92	101

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
72-48	72	74.44	215	2.05	0.88	2.42	0.88	21	21
72-49	72	74.60	411	6.59	2.51	15.92	2.44	125	124
72-5	72	73.75	211	2.64	1.85	1.71	1.79	42	42
72-6	72	73.03	91	0.72	0.38	2.39	0.39	8	8
72-7	72	73.28	171	1.85	0.73	2.11	0.72	18	18
72-8	72	73.53	191	4.60	2.27	6.05	2.62	41	42
72-9	72	72.38	55	0.39	0.31	0.47	0.30	4	4
75-0	75	77.14	23	0.14	0.12	0.11	0.13	4	4
75-1	75	77.11	115	0.89	1.47	0.63	1.27	20	20
75-10	75	76.60	111	1.49	1.28	25.19	1.46	20	22
75-11	75	76.93	231	4.21	5.07	2.78	4.75	40	42
75-12	75	76.71	223	2.63	3.83	1.76	3.13	40	41
75-13	75	76.00	103	0.86	0.81	0.51	0.69	0	2
75-14	75	76.64	183	1.93	0.72	1.83	0.75	22	22
75-15	75	76.39	203	1.78	0.69	1.60	0.72	42	42
75-16	75	76.87	507	36.40	32.39	3,236.87	13.39	130	127
75-17	75	75.70	207	2.95	0.92	25.79	0.95	67	61
75-18	75	76.47	407	10.62	2.17	761.38	2.15	145	124
75-19	75	76.84	457	25.14	2.50	177.09	2.52	126	128
75-2	75	76.56	55	0.34	0.36	0.27	0.32	0	0
75-20	75	75.69	107	1.13	0.46	3.02	0.46	11	12
75-21	75	76.61	257	27.30	3.62	379.81	3.01	62	62
75-22	75	76.96	557	6,096.11	56.27	21,855.43	35.75	90	123
75-23	75	77.30	577	30,704.44	196.03	21,885.07	81.45	84	134
75-24	75	75.88	277	408.03	16.95	815.47	11.41	50	57
75-25	75	76.00	137	4.97	1.00	28.79	0.99	20	22
75-26	75	77.05	477	17.00	47.65	21,677.48	25.30	104	109
75-27	75	76.28	427	37.55	4.27	1,220.29	4.34	141	141
75-28	75	75.92	47	6.38	0.21	15.84	0.23	6	6

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
75-29	75	76.26	227	10.83	1.20	156.75	1.24	69	69
75-3	75	76.82	95	1.53	0.45	1.68	0.45	20	20
75-30	75	77.26	527	1,428.09	88.10	1,633.55	24.70	104	128
75-31	75	76.81	522	16.52	6.38	16.83	5.65	104	104
75-32	75	76.29	222	2.79	0.88	30.04	0.91	61	61
75-33	75	76.74	422	6.45	1.55	250.90	1.60	53	52
75-34	75	77.05	472	77.76	26.58	198.26	11.42	124	127
75-35	75	75.78	132	3.83	0.49	9.84	0.51	20	22
75-36	75	75.73	272	86.93	5.40	124.33	4.32	56	56
75-37	75	77.19	572	201.68	44.73	3,071.85	34.97	54	58
75-38	75	76.54	552	113.37	16.35	65.00	16.03	104	120
75-39	75	76.24	252	4.78	2.92	63.26	2.29	3	4
75-4	75	76.47	53	0.39	0.33	0.24	0.32	0	1
75-40	75	76.82	452	16.23	1.72	21,699.25	1.75	104	104
75-41	75	76.38	502	16.90	9.04	22.61	7.58	12	9
75-42	75	77.44	1,052	38.77	4.34	21,691.28	4.41	190	170
75-43	75	77.74	1,072	1,081.50	13.37	21,785.64	12.57	360	361
75-44	75	78.03	1,077	328.67	336.44	21,746.86	73.80	326	325
75-45	75	77.83	1,057	99.49	110.39	21,729.47	27.33	422	318
75-46	75	77.56	423	7.80	1.92	2,374.17	1.92	68	68
75-47	75	78.31	431	12.70	6.07	70.77	6.82	101	120
75-48	75	77.44	215	1.96	0.86	2.09	0.89	31	32
75-49	75	77.60	411	6.56	2.63	12.97	2.62	140	132
75-5	75	76.75	211	2.67	2.29	1.74	2.21	42	42
75-6	75	76.03	91	0.75	0.39	1.47	0.40	8	9
75-7	75	76.28	171	1.58	0.74	1.87	0.75	19	20
75-8	75	76.53	191	5.90	2.42	11.32	2.74	40	42
75-9	75	75.38	55	0.45	0.32	0.37	0.32	4	4
78-0	78	80.14	23	0.15	0.13	0.11	0.13	4	4

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
78-1	78	80.11	115	0.90	1.34	0.62	1.19	20	20
78-10	78	79.60	111	1.31	1.27	16.33	1.39	18	24
78-11	78	79.93	231	6.87	4.88	3.72	4.49	40	42
78-12	78	79.71	223	2.75	3.46	2.02	2.85	40	42
78-13	78	79.00	103	0.51	0.81	0.50	0.69	0	2
78-14	78	79.64	183	1.03	0.71	1.51	0.74	22	22
78-15	78	79.39	203	1.00	0.71	1.11	0.75	42	42
78-16	78	79.87	507	23.15	34.66	848.96	11.84	170	135
78-17	78	78.70	207	3.72	0.94	4.74	0.97	62	62
78-18	78	79.47	407	8.14	2.33	152.20	2.29	136	133
78-19	78	79.84	457	39.27	2.54	269.60	2.56	155	151
78-2	78	79.56	55	0.35	0.34	0.25	0.33	0	0
78-20	78	78.69	107	4.95	0.44	12.34	0.45	16	16
78-21	78	79.61	257	113.96	3.67	120.57	3.02	67	67
78-22	78	79.96	557	50.15	49.93	34.69	24.67	105	110
78-23	78	80.30	577	36,130.01	235.34	21,828.82	76.06	93	143
78-24	78	78.88	277	621.56	24.43	958.80	13.98	46	58
78-25	78	79.00	137	4.02	1.10	34.39	1.02	20	23
78-26	78	80.05	477	35.83	68.58	21,673.43	26.39	105	109
78-27	78	79.28	427	54.52	4.53	6,370.50	4.38	146	144
78-28	78	78.92	47	3.27	0.22	18.44	0.22	6	6
78-29	78	79.26	227	173.29	1.24	162.31	1.21	76	72
78-3	78	79.82	95	1.10	0.46	1.00	0.46	20	20
78-30	78	80.26	527	2,538.43	101.96	21,849.86	28.54	134	134
78-31	78	79.81	522	12.57	6.69	14.08	5.31	104	105
78-32	78	79.29	222	3.25	0.92	9.59	0.91	61	61
78-33	78	79.74	422	9.76	1.73	21,629.85	1.63	74	72
78-34	78	80.05	472	29.99	10.24	208.47	12.40	167	148
78-35	78	78.78	132	4.15	0.51	6.35	0.52	20	23

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
78-36	78	78.73	272	113.08	6.60	52.49	4.98	57	59
78-37	78	80.19	572	1,838.12	70.52	637.88	34.82	57	123
78-38	78	79.54	552	118.30	19.14	163.47	13.37	128	127
78-39	78	79.24	252	3.61	2.94	11.82	2.29	3	5
78-4	78	79.47	53	0.37	0.36	0.27	0.32	0	2
78-40	78	79.82	452	12.26	1.86	19.12	1.80	105	105
78-41	78	79.38	502	12.15	10.47	19.76	7.66	10	11
78-42	78	80.44	1,052	24.53	4.77	21,709.41	4.46	180	174
78-43	78	80.74	1,072	1,294.96	16.12	21,799.97	13.58	374	384
78-44	78	81.03	1,077	986.14	460.32	21,756.65	65.98	324	345
78-45	78	80.83	1,057	95.94	15.92	21,706.76	16.94	354	321
78-46	78	80.56	423	5.50	2.04	94.51	2.00	70	70
78-47	78	81.31	431	9.31	5.83	320.12	5.88	120	124
78-48	78	80.44	215	1.68	0.89	2.00	0.89	40	34
78-49	78	80.60	411	5.44	2.90	8.90	2.75	143	140
78-5	78	79.75	211	1.87	1.67	1.61	1.56	42	42
78-6	78	79.03	91	0.70	0.41	0.92	0.41	8	10
78-7	78	79.28	171	1.86	0.75	2.78	0.75	26	26
78-8	78	79.53	191	3.97	2.01	3.65	2.25	44	41
78-9	78	78.38	55	0.57	0.36	0.42	0.31	4	6
81-0	81	83.14	23	0.14	0.13	0.11	0.14	4	4
81-1	81	83.11	115	0.89	1.32	0.60	1.20	20	20
81-10	81	82.60	111	5.51	1.31	13.23	1.18	23	26
81-11	81	82.93	231	3.30	5.84	2.21	4.73	42	42
81-12	81	82.71	223	2.20	3.67	1.74	2.95	42	42
81-13	81	82.00	103	0.71	0.78	0.50	0.67	0	2
81-14	81	82.64	183	1.59	0.76	1.99	0.79	22	24
81-15	81	82.39	203	1.75	0.78	1.18	0.72	42	42
81-16	81	82.87	507	47.40	28.15	1,362.30	12.65	143	145

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
81-17	81	81.70	207	3.85	1.01	4.59	1.00	70	67
81-18	81	82.47	407	8.08	2.62	106.05	2.41	145	143
81-19	81	82.84	457	21.99	2.68	215.93	2.65	160	158
81-2	81	82.56	55	0.25	0.36	0.26	0.34	0	0
81-20	81	81.69	107	0.89	0.46	2.96	0.47	18	17
81-21	81	82.61	257	27.38	3.89	95.52	3.04	69	71
81-22	81	82.96	557	2,095.11	101.08	833.07	30.62	129	138
81-23	81	83.30	577	22,632.67	252.83	21,888.24	86.10	100	148
81-24	81	81.88	277	983.25	22.76	723.93	14.03	54	63
81-25	81	82.00	137	7.58	1.11	41.32	0.99	20	23
81-26	81	83.05	477	17.73	45.35	367.64	24.26	118	111
81-27	81	82.28	427	14.80	4.49	240.13	4.43	151	154
81-28	81	81.92	47	1.33	0.22	3.69	0.21	6	6
81-29	81	82.26	227	7.15	1.21	272.35	1.21	73	75
81-3	81	82.82	95	0.61	0.52	0.63	0.50	21	24
81-30	81	83.26	527	1,658.63	84.73	21,787.85	30.97	138	144
81-31	81	82.81	522	10.26	6.85	15.21	5.73	106	106
81-32	81	82.29	222	3.67	0.85	10.14	0.91	62	62
81-33	81	82.74	422	8.21	1.71	744.20	1.61	73	75
81-34	81	83.05	472	31.27	28.94	101.92	12.39	154	154
81-35	81	81.78	132	2.43	0.49	2.09	0.50	20	23
81-36	81	81.73	272	17.19	6.64	131.98	4.96	68	63
81-37	81	83.19	572	2,175.76	101.81	1,088.14	37.64	59	153
81-38	81	82.54	552	166.16	16.28	7,330.88	13.92	134	135
81-39	81	82.24	252	2.74	3.03	7.22	2.27	5	5
81-4	81	82.47	53	0.27	0.35	0.32	0.32	0	2
81-40	81	82.82	452	9.56	2.13	17.47	1.91	106	106
81-41	81	82.38	502	16.08	10.29	17.42	7.66	16	13
81-42	81	83.44	1,052	30.00	4.58	21,675.04	4.49	243	225

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
81-43	81	83.74	1,072	826.35	15.35	21,851.36	13.24	392	401
81-44	81	84.03	1,077	431.89	392.89	21,729.26	71.15	386	375
81-45	81	83.83	1,057	155.34	120.36	21,726.84	25.45	384	366
81-46	81	83.56	423	17.89	2.23	74.38	2.13	90	90
81-47	81	84.31	431	10.65	4.84	164.79	4.34	128	132
81-48	81	83.44	215	2.54	0.92	2.22	0.91	44	44
81-49	81	83.60	411	4.10	3.26	6.74	2.88	150	150
81-5	81	82.75	211	1.74	2.22	1.57	2.03	42	42
81-6	81	82.03	91	0.49	0.41	0.68	0.41	8	10
81-7	81	82.28	171	1.13	0.77	1.79	0.78	26	28
81-8	81	82.53	191	4.23	1.43	9.04	1.40	44	45
81-9	81	81.38	55	0.38	0.33	0.42	0.31	4	6
84-0	84	86.14	23	0.15	0.13	0.11	0.13	4	4
84-1	84	86.11	115	0.63	1.31	0.55	1.14	20	20
84-10	84	85.60	111	49.00	1.28	26.88	1.38	26	28
84-11	84	85.93	231	2.70	5.99	2.27	4.86	42	42
84-12	84	85.71	223	1.78	3.54	1.65	2.80	42	42
84-13	84	85.00	103	0.56	0.78	0.49	0.67	2	2
84-14	84	85.64	183	1.20	0.76	1.82	0.79	22	24
84-15	84	85.39	203	1.23	0.73	1.13	0.77	42	42
84-16	84	85.87	507	66.06	37.94	1,961.73	15.87	200	154
84-17	84	84.70	207	2.96	1.06	4.53	1.02	75	72
84-18	84	85.47	407	6.00	2.56	19.87	2.42	148	149
84-19	84	85.84	457	27.50	2.60	106.02	2.62	170	165
84-2	84	85.56	55	0.47	0.40	0.35	0.34	0	2
84-20	84	84.69	107	0.91	0.45	3.25	0.47	16	17
84-21	84	85.61	257	52.59	3.75	266.67	3.09	74	75
84-22	84	85.96	557	174.07	73.26	62.48	22.93	111	112
84-23	84	86.30	577	19,844.31	276.69	21,867.73	84.89	122	159

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
84-24	84	84.88	277	374.12	20.34	469.40	13.12	61	73
84-25	84	85.00	137	5.14	1.29	2.30	1.17	20	27
84-26	84	86.05	477	44.75	55.59	21,662.88	26.76	106	123
84-27	84	85.28	427	22.98	4.19	47.04	3.90	156	162
84-28	84	84.92	47	0.87	0.23	2.10	0.26	6	6
84-29	84	85.26	227	43.79	1.24	118.95	1.25	87	80
84-3	84	85.82	95	0.75	0.54	0.57	0.52	20	24
84-30	84	86.26	527	922.16	95.69	21,816.25	29.69	146	154
84-31	84	85.81	522	9.65	6.04	12.59	5.28	106	107
84-32	84	85.29	222	2.23	0.86	477.83	0.85	64	63
84-33	84	85.74	422	23.93	1.67	45.01	1.64	95	95
84-34	84	86.05	472	29.06	7.31	61.29	6.88	167	160
84-35	84	84.78	132	1.46	0.49	1.90	0.52	20	23
84-36	84	84.73	272	31.70	6.12	39.60	5.05	73	67
84-37	84	86.19	572	2,213.53	68.54	1,787.54	34.76	61	154
84-38	84	85.54	552	23.07	42.14	440.61	12.49	146	144
84-39	84	85.24	252	3.98	3.04	25.73	2.31	5	7
84-4	84	85.47	53	0.39	0.36	0.24	0.32	0	2
84-40	84	85.82	452	6.91	2.10	5.49	2.02	156	153
84-41	84	85.38	502	8.97	10.07	335.88	7.73	17	17
84-42	84	86.44	1,052	26.77	4.75	21,670.62	4.68	246	232
84-43	84	86.74	1,072	669.67	7.18	21,794.73	7.00	422	428
84-44	84	87.03	1,077	1,016.56	550.07	21,763.85	72.17	383	396
84-45	84	86.83	1,057	280.15	224.97	21,742.73	28.48	369	374
84-46	84	86.56	423	14.99	1.77	157.82	1.69	100	95
84-47	84	87.31	431	8.73	5.42	430.19	4.85	138	142
84-48	84	86.44	215	2.43	1.01	2.18	1.03	45	46
84-49	84	86.60	411	3.66	3.38	4.95	3.16	160	155
84-5	84	85.75	211	8.92	2.81	27.86	2.34	57	60

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
84-6	84	85.03	91	0.54	0.41	1.65	0.42	17	16
84-7	84	85.28	171	1.26	0.78	2.46	0.78	36	34
84-8	84	85.53	191	2.56	2.36	2.56	2.52	47	48
84-9	84	84.38	55	0.28	0.27	0.38	0.29	4	5
87-0	87	89.14	23	0.10	0.13	0.12	0.14	4	4
87-1	87	89.11	115	0.67	1.44	0.59	1.24	20	20
87-10	87	88.60	111	11.97	1.00	57.94	0.87	30	32
87-11	87	88.93	231	216.57	8.03	233.13	5.91	53	58
87-12	87	88.71	223	2.31	3.79	1.68	3.10	42	43
87-13	87	88.00	103	2.35	0.71	1.05	0.61	2	6
87-14	87	88.64	183	1.47	0.75	1.48	0.74	22	25
87-15	87	88.39	203	1.53	0.75	1.53	0.73	44	44
87-16	87	88.87	507	12.66	6.11	147.27	5.26	120	119
87-17	87	87.70	207	2.64	1.09	5.21	1.04	76	77
87-18	87	88.47	407	6.58	2.90	7.88	2.65	159	159
87-19	87	88.84	457	15.69	2.58	187.86	2.58	184	172
87-2	87	88.56	55	0.47	0.27	0.28	0.27	0	2
87-20	87	87.69	107	1.20	0.49	8.86	0.49	22	22
87-21	87	88.61	257	32.79	3.84	333.49	3.04	79	80
87-22	87	88.96	557	1,572.25	114.51	6,240.13	28.01	155	155
87-23	87	89.30	577	12,747.09	207.09	21,941.38	80.78	135	163
87-24	87	87.88	277	613.91	52.35	338.80	10.25	68	73
87-25	87	88.00	137	53.30	0.95	1.98	0.95	20	35
87-26	87	89.05	477	12.45	44.57	11.90	23.62	154	159
87-27	87	88.28	427	88.61	3.65	99.49	3.54	165	170
87-28	87	87.92	47	0.26	0.24	1.94	0.24	6	7
87-29	87	88.26	227	25.54	1.21	138.08	1.25	83	83
87-3	87	88.82	95	0.45	0.49	0.40	0.50	30	30
87-30	87	89.26	527	1,031.14	116.15	21,783.63	28.91	158	164

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
87-31	87	88.81	522	10.92	7.53	25.81	6.33	110	109
87-32	87	88.29	222	2.34	0.86	8.47	0.92	65	63
87-33	87	88.74	422	22.62	1.69	722.88	1.71	104	99
87-34	87	89.05	472	72.24	5.38	58.55	5.52	190	173
87-35	87	87.78	132	1.18	0.52	2.04	0.57	21	32
87-36	87	87.73	272	8.11	4.20	144.11	3.51	73	73
87-37	87	89.19	572	3,069.60	77.60	3,831.48	34.13	134	146
87-38	87	88.54	552	142.18	43.27	759.07	11.97	154	154
87-39	87	88.24	252	2.78	0.85	13.06	0.88	7	9
87-4	87	88.47	53	0.31	0.24	0.23	0.25	0	2
87-40	87	88.82	452	5.81	2.06	5.36	1.93	154	154
87-41	87	88.38	502	13.36	1.74	22.49	1.77	20	11
87-42	87	89.44	1,052	30.80	5.31	21,668.95	5.13	295	286
87-43	87	89.74	1,072	1,087.19	7.12	21,803.58	6.94	460	451
87-44	87	90.03	1,077	782.96	371.93	466.52	68.21	430	427
87-45	87	89.83	1,057	198.20	146.30	266.22	18.30	445	422
87-46	87	89.56	423	9.54	1.78	1,182.33	1.79	120	116
87-47	87	90.31	431	50.89	3.71	49.39	3.65	156	166
87-48	87	89.44	215	5.72	1.25	1.86	1.20	54	58
87-49	87	89.60	411	3.63	3.74	4.99	3.51	178	172
87-5	87	88.75	211	1.83	2.20	2.77	1.97	44	44
87-6	87	88.03	91	0.80	0.42	1.32	0.46	16	18
87-7	87	88.28	171	2.25	0.94	1.82	0.88	34	35
87-8	87	88.53	191	1.94	2.51	1.71	2.18	60	62
87-9	87	87.38	55	0.31	0.26	0.28	0.25	12	12
90-0	90	92.14	23	0.13	0.13	0.10	0.13	4	4
90-1	90	92.11	115	0.67	1.60	0.56	1.35	20	21
90-10	90	91.60	111	134.54	0.97	22.81	0.87	30	34
90-11	90	91.93	231	5.09	6.75	3.60	5.54	44	46

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
90-12	90	91.71	223	14.75	5.07	7.47	3.96	64	66
90-13	90	91.00	103	0.57	0.41	0.40	0.39	22	22
90-14	90	91.64	183	1.01	0.72	0.76	0.73	62	62
90-15	90	91.39	203	1.27	0.87	1.50	0.75	45	45
90-16	90	91.87	507	21.96	13.21	206.00	10.94	178	175
90-17	90	90.70	207	2.29	1.11	4.11	1.07	84	82
90-18	90	91.47	407	5.52	3.36	7.35	3.02	190	169
90-19	90	91.84	457	35.14	2.63	143.38	2.63	192	180
90-2	90	91.56	55	0.23	0.28	0.26	0.36	10	10
90-20	90	90.69	107	1.88	0.50	5.38	0.50	27	27
90-21	90	91.61	257	57.82	3.77	62.50	3.10	82	86
90-22	90	91.96	557	78.82	93.83	185.47	23.42	121	123
90-23	90	92.30	577	155,020.99	268.63	21,994.42	82.92	149	179
90-24	90	90.88	277	815.80	40.07	762.04	6.45	80	83
90-25	90	91.00	137	1.41	1.04	2.25	1.01	22	43
90-26	90	92.05	477	10.83	75.80	18.33	22.97	162	161
90-27	90	91.28	427	10.56	3.56	44.40	3.37	170	164
90-28	90	90.92	47	1.66	0.23	0.78	0.23	11	11
90-29	90	91.26	227	26.04	1.49	70.59	1.26	93	87
90-3	90	91.82	95	0.47	0.53	0.33	0.51	30	30
90-30	90	92.26	527	1,536.60	131.44	1,500.38	27.94	180	178
90-31	90	91.81	522	9.65	7.87	23.20	6.24	112	113
90-32	90	91.29	222	1.60	0.86	1.84	0.86	82	82
90-33	90	91.74	422	18.13	1.68	141.10	1.65	126	121
90-34	90	92.05	472	91.84	4.99	237.94	4.79	190	180
90-35	90	90.78	132	3.99	0.53	1.53	0.53	41	42
90-36	90	90.73	272	43.95	3.81	281.38	3.11	77	83
90-37	90	92.19	572	18.99	47.29	40.11	26.94	112	114
90-38	90	91.54	552	104.66	41.35	32.18	11.77	173	166

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
90-39	90	91.24	252	2.57	0.95	2.72	0.96	58	55
90-4	90	91.47	53	0.23	0.25	0.22	0.25	10	10
90-40	90	91.82	452	5.51	2.02	4.85	1.93	162	156
90-41	90	91.38	502	8.01	2.28	10.98	2.17	127	114
90-42	90	92.44	1,052	28.91	6.74	21,678.36	6.89	349	346
90-43	90	92.74	1,072	1,222.04	7.28	1,011.30	7.16	478	474
90-44	90	93.03	1,077	1,469.17	479.80	5,681.90	58.82	435	454
90-45	90	92.83	1,057	130.24	147.47	636.63	42.12	443	434
90-46	90	92.56	423	6.58	2.15	68.10	2.13	144	139
90-47	90	93.31	431	12.21	4.24	30.25	4.09	175	178
90-48	90	92.44	215	1.19	1.59	2.16	1.46	67	66
90-49	90	92.60	411	2.86	3.83	3.83	3.57	180	180
90-5	90	91.75	211	21.11	2.89	96.89	2.29	66	70
90-6	90	91.03	91	0.57	0.42	0.72	0.42	18	18
90-7	90	91.28	171	1.19	0.91	2.19	0.89	48	44
90-8	90	91.53	191	1.49	2.33	2.03	2.17	62	62
90-9	90	90.38	55	0.28	0.26	0.30	0.29	12	12
93-0	93	95.14	23	0.12	0.11	0.13	0.12	4	4
93-1	93	95.11	115	0.63	1.33	0.82	1.14	20	22
93-10	93	94.60	111	17.75	0.99	16.39	0.86	36	36
93-11	93	94.93	231	584.37	1.48	322.69	1.46	61	NA
93-12	93	94.71	223	2.15	3.86	4.47	2.99	48	49
93-13	93	94.00	103	0.47	0.39	0.55	0.38	25	24
93-14	93	94.64	183	0.93	0.74	9.75	0.77	46	50
93-15	93	94.39	203	1.03	0.76	1.79	0.76	51	48
93-16	93	94.87	507	76.16	14.95	224.50	11.20	208	186
93-17	93	93.70	207	2.00	1.24	2.42	1.24	86	87
93-18	93	94.47	407	4.39	3.42	7.02	3.35	184	183
93-19	93	94.84	457	36.39	2.63	50.03	2.63	205	189

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
93-2	93	94.56	55	3.15	0.06	4.11	0.07	14	NA
93-20	93	93.69	107	0.75	0.51	3.27	0.51	41	33
93-21	93	94.61	257	20.94	3.76	108.01	3.02	97	91
93-22	93	94.96	557	16,376.34	85.22	14,042.61	24.12	104	165
93-23	93	95.30	577	15,993.86	239.54	21,891.17	63.75	174	189
93-24	93	93.88	277	933.86	7.03	1,264.10	6.27	80	87
93-25	93	94.00	137	0.99	0.27	0.92	0.29	51	NA
93-26	93	95.05	477	24.29	56.48	49.74	20.82	166	169
93-27	93	94.28	427	15.31	3.25	79.99	3.11	190	188
93-28	93	93.92	47	2.70	0.24	1.83	0.22	11	12
93-29	93	94.26	227	416.90	1.50	75.24	1.26	104	92
93-3	93	94.82	95	0.42	0.52	0.31	0.49	30	30
93-30	93	95.26	527	2,120.54	138.84	1,256.07	31.44	182	188
93-31	93	94.81	522	12.02	6.00	18.75	5.74	132	119
93-32	93	94.29	222	1.36	0.87	1.99	0.87	83	83
93-33	93	94.74	422	3.96	2.00	436.07	2.02	177	173
93-34	93	95.05	472	66.34	3.79	110.20	3.71	199	189
93-35	93	93.78	132	0.70	0.17	0.70	0.18	51	NA
93-36	93	93.73	272	51.31	3.98	83.22	3.05	103	89
93-37	93	95.19	572	2,667.91	18.66	11,288.74	15.99	167	NA
93-38	93	94.54	552	81.39	45.41	38.58	13.27	193	180
93-39	93	94.24	252	2.48	0.95	2.79	0.97	61	58
93-4	93	94.47	53	0.29	0.21	0.23	0.22	12	12
93-40	93	94.82	452	5.19	2.02	4.97	1.96	166	166
93-41	93	94.38	502	8.55	2.28	11.63	2.15	131	122
93-42	93	95.44	1,052	57.99	52.32	239.39	17.29	468	457
93-43	93	95.74	1,072	106.62	7.57	171.90	7.10	519	496
93-44	93	96.03	1,077	1,000.90	1,575.80	914.76	93.42	493	496
93-45	93	95.83	1,057	153.42	342.23	126.56	45.80	501	490

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
93-46	93	95.56	423	3.32	2.12	5.87	2.12	187	185
93-48	93	95.44	215	0.96	0.86	1.44	0.89	90	91
93-49	93	95.60	411	1.95	2.19	2.36	2.16	192	196
93-5	93	94.75	211	1.56	1.41	3.21	1.42	49	51
93-6	93	94.03	91	0.54	0.45	0.99	0.44	27	26
93-7	93	94.28	171	1.09	1.08	1.18	1.04	70	53
93-8	93	94.53	191	2.57	1.56	4.96	1.55	65	66
93-9	93	93.38	55	0.36	0.23	0.46	0.24	12	14
96-10	96	97.60	111	5.59	0.94	10.43	0.86	34	38
96-11	96	97.93	231	62.59	2.08	39.12	2.07	73	78
96-12	96	97.71	223	8.48	1.18	2.51	1.21	64	67
96-13	96	97.00	103	0.52	0.39	0.42	0.41	26	27
96-14	96	97.64	183	0.98	0.71	0.84	0.75	78	70
96-15	96	97.39	203	1.31	0.77	1.81	0.79	67	62
96-16	96	97.87	507	18.27	9.51	23.57	7.74	225	198
96-17	96	96.70	207	1.64	0.91	2.10	0.94	92	93
96-18	96	97.47	407	3.71	2.22	4.15	2.05	196	194
96-19	96	97.84	457	16.55	2.74	24.65	2.62	199	198
96-2	96	97.56	55	1.58	0.06	0.73	0.08	19	NA
96-20	96	96.69	107	0.67	0.52	2.18	0.50	39	39
96-21	96	97.61	257	53.99	3.77	27.28	3.12	104	97
96-22	96	97.96	557	168.58	137.93	249.98	18.70	189	191
96-24	96	96.88	277	1,185.68	7.54	1,342.30	6.15	88	96
96-25	96	97.00	137	1.20	0.27	1.11	0.30	51	NA
96-26	96	98.05	477	133.82	10.94	21,730.66	9.69	191	158
96-27	96	97.28	427	32.47	3.15	66.15	2.98	197	196
96-28	96	96.92	47	0.22	0.22	0.20	0.23	16	16
96-29	96	97.26	227	24.78	1.22	35.78	1.23	100	97
96-3	96	97.82	95	0.43	0.13	0.39	0.14	34	NA

Table A.13: Results for Benchmark 1 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
96-31	96	97.81	522	12.96	4.45	34.29	4.26	181	182
96-32	96	97.29	222	1.64	0.97	1.36	0.98	87	86
96-33	96	97.74	422	3.53	2.48	3.00	2.08	193	193
96-34	96	98.05	472	28.33	3.92	19.83	3.55	218	199
96-35	96	96.78	132	1.06	0.17	1.22	0.18	52	NA
96-36	96	96.73	272	65.57	3.81	54.64	3.14	103	95
96-38	96	97.54	552	20.39	78.40	7.50	14.08	196	196
96-39	96	97.24	252	2.33	0.98	3.34	0.95	70	67
96-4	96	97.47	53	0.21	0.21	0.27	0.22	14	15
96-40	96	97.82	452	4.67	2.34	4.21	2.24	188	187
96-41	96	97.38	502	9.22	2.80	12.70	2.67	162	155
96-5	96	97.75	211	1.47	1.38	4.10	1.35	68	71
96-6	96	97.03	91	0.43	0.38	0.32	0.39	34	34
96-7	96	97.28	171	0.85	0.74	1.01	0.75	70	70
96-8	96	97.53	191	1.77	1.23	1.55	1.22	77	74
96-9	96	96.38	55	0.26	0.24	0.34	0.24	12	16
99-9	99	99.38	55	0.25	0.07	0.32	0.08	22	NA

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-0	30	34.93	13	0.08	0.11	0.10	0.13	0	0
30-1	30	40.08	59	0.44	0.70	0.46	0.70	0	0
30-2	30	43.01	29	0.16	0.19	0.17	0.18	0	0
30-3	30	36.63	49	0.25	0.29	0.28	0.31	0	0
30-4	30	39.05	28	0.17	0.18	0.19	0.21	0	0
30-5	30	42.86	107	0.91	1.59	2.21	1.46	4	8
30-6	30	36.95	47	0.27	0.31	0.31	0.32	0	0
30-7	30	33.42	87	0.67	0.64	0.69	0.65	0	1
30-8	30	47.18	97	0.84	1.10	0.91	1.12	0	1

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-9	30	36.97	29	0.17	0.19	0.19	0.21	0	0
30-10	30	36.42	57	0.39	0.62	0.51	0.63	1	2
30-11	30	40.99	117	1.46	2.52	1.90	2.57	0	1
30-12	30	35.28	113	1.36	1.32	1.26	1.35	0	0
30-13	30	40.52	53	3.33	0.36	0.41	0.36	0	0
30-15	30	44.30	103	0.69	1.13	0.90	1.04	0	1
30-16	30	35.38	508	12.52	31.58	130.20	17.29	34	35
30-17	30	39.96	208	3.65	1.03	6.79	1.08	16	17
30-19	30	41.24	458	13.99	9.47	172.95	8.61	37	41
30-20	30	39.86	108	0.83	0.84	0.90	0.88	0	1
30-21	30	45.40	258	16.74	1.87	46.28	1.98	17	21
30-22	30	37.30	558	11.83	61.75	32.01	34.03	4	7
30-23	30	47.08	578	995.42	231.54	888.23	77.48	17	33
30-24	30	34.94	278	7.35	25.56	11.43	12.43	6	11
30-25	30	42.83	138	2.50	1.74	1.95	1.79	0	11
30-26	30	34.42	478	9.40	31.83	16.34	27.21	2	7
30-28	30	39.52	48	0.28	0.31	0.47	0.32	0	0
30-30	30	37.19	528	21.01	59.75	1,809.08	35.78	17	41
30-31	30	36.02	523	12.93	29.59	28.73	17.44	3	3
30-34	30	41.03	473	11.04	20.24	21.60	15.67	25	32
30-35	30	36.98	133	1.13	0.63	1.80	0.70	0	1
30-36	30	39.82	273	3.91	12.28	6.22	8.67	11	11
30-37	30	40.63	573	11.90	49.83	19.47	26.53	2	3
30-39	30	33.20	253	3.36	3.29	4.32	2.69	1	1
30-41	30	34.78	503	8.79	10.07	14.07	8.65	3	3
30-42	30	37.38	1,053	15.54	10.33	44.73	9.54	9	9
30-44	30	42.28	1,078	70.58	238.53	205.44	63.99	62	117
30-47	30	44.62	217	3.12	7.79	4.67	7.09	9	10
30-48	30	39.04	109	0.68	0.66	0.71	0.69	0	1

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-0	33	37.93	13	0.08	0.11	0.12	0.12	0	0
33-1	33	43.08	59	0.44	0.67	0.52	0.68	0	0
33-2	33	46.01	29	0.15	0.18	0.18	0.23	0	0
33-3	33	39.63	49	0.25	0.34	0.28	0.34	0	1
33-4	33	42.05	28	0.17	0.19	0.18	0.20	0	0
33-5	33	45.86	107	1.90	1.43	1.15	1.45	0	1
33-6	33	39.95	47	0.36	0.31	0.39	0.32	0	0
33-7	33	36.42	87	1.01	0.62	0.65	0.64	0	1
33-8	33	50.18	97	0.91	1.11	1.06	1.13	0	1
33-9	33	39.97	29	0.17	0.19	0.25	0.22	0	0
33-10	33	39.42	57	0.55	0.64	0.85	0.67	1	2
33-11	33	43.99	117	1.46	2.51	1.71	2.62	0	1
33-12	33	38.28	113	1.07	1.36	1.27	1.38	0	1
33-13	33	43.52	53	0.39	0.43	0.32	0.42	0	1
33-15	33	47.30	103	0.68	1.13	0.89	1.03	0	1
33-16	33	38.38	508	443.96	33.08	1,186.64	17.71	45	43
33-17	33	42.96	208	5.89	1.14	51.20	1.18	21	21
33-19	33	44.24	458	17.23	9.54	4,700.03	8.82	47	49
33-20	33	42.86	108	1.48	0.84	1.09	0.87	0	1
33-21	33	48.40	258	8.59	1.90	1,122.38	1.94	22	23
33-22	33	40.30	558	12.24	60.45	4,265.73	29.88	26	28
33-23	33	50.08	578	614.40	164.29	1,234.28	82.56	17	43
33-24	33	37.94	278	61.48	21.26	3,772.81	11.86	6	11
33-25	33	45.83	138	2.25	1.71	2.06	2.14	0	11
33-26	33	37.42	478	11.89	30.93	16.17	25.58	2	8
33-28	33	42.52	48	0.29	0.31	0.32	0.33	0	0
33-30	33	40.19	528	161.08	76.45	939.69	31.85	24	43
33-31	33	39.02	523	18.43	31.81	28.07	15.60	3	3
33-34	33	44.03	473	15.36	21.79	176.02	15.11	44	44

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-35	33	39.98	133	1.12	0.62	1.67	0.64	0	1
33-36	33	42.82	273	4.53	11.10	11.95	8.54	12	13
33-37	33	43.63	573	279.80	53.07	22.75	35.66	2	12
33-39	33	36.20	253	3.55	3.14	4.45	2.58	1	2
33-41	33	37.78	503	8.93	10.19	14.51	8.61	4	4
33-42	33	40.38	1,053	15.65	10.41	41.57	10.10	9	9
33-44	33	45.28	1,078	56.49	222.23	356.63	47.95	124	122
33-47	33	47.62	217	3.44	7.71	5.76	6.78	10	19
33-48	33	42.04	109	0.71	0.65	0.84	0.66	0	1
36-0	36	40.93	13	0.09	0.11	0.10	0.12	0	0
36-1	36	46.08	59	0.43	0.68	0.49	0.67	0	0
36-2	36	49.01	29	0.16	0.19	0.17	0.19	0	0
36-3	36	42.63	49	0.26	0.28	0.33	0.30	0	0
36-4	36	45.05	28	0.15	0.19	0.18	0.21	0	0
36-5	36	48.86	107	0.97	1.44	1.33	1.51	0	1
36-6	36	42.95	47	0.33	0.31	0.37	0.33	0	0
36-7	36	39.42	87	0.90	0.63	0.76	0.65	0	1
36-8	36	53.18	97	0.84	1.00	4.45	1.02	10	10
36-9	36	42.97	29	0.17	0.19	0.26	0.20	0	0
36-10	36	42.42	57	0.53	0.76	0.67	0.78	3	3
36-11	36	46.99	117	1.64	2.51	1.60	2.66	0	1
36-12	36	41.28	113	1.17	1.33	1.09	1.38	0	1
36-13	36	46.52	53	0.31	0.44	0.32	0.44	0	1
36-15	36	50.30	103	0.79	1.15	0.74	1.04	0	1
36-16	36	41.38	508	29.68	26.47	415.20	17.58	49	49
36-17	36	45.96	208	3.48	1.06	9.44	1.08	21	22
36-19	36	47.24	458	65.90	9.70	206.19	8.66	46	52
36-20	36	45.86	108	1.41	0.82	1.07	0.84	0	1
36-21	36	51.40	258	4.22	2.02	135.93	2.01	25	29

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
36-22	36	43.30	558	40.95	62.11	39.58	30.67	3	8
36-23	36	53.08	578	1,276.24	177.17	3,984.36	80.61	23	48
36-24	36	40.94	278	14.25	20.40	1,952.21	11.41	14	16
36-25	36	48.83	138	2.02	1.73	2.06	2.10	0	11
36-26	36	40.42	478	22.02	37.27	16.75	25.40	2	5
36-28	36	45.52	48	0.28	0.32	0.37	0.33	0	0
36-30	36	43.19	528	33.02	68.62	1,785.20	34.42	43	48
36-31	36	42.02	523	12.50	34.03	15.01	19.65	3	3
36-34	36	47.03	473	44.18	24.14	111.69	15.44	48	48
36-35	36	42.98	133	1.13	0.62	1.51	0.64	0	2
36-36	36	45.82	273	3.81	11.12	57.46	8.45	14	14
36-37	36	46.63	573	36.51	61.83	23.09	33.97	2	14
36-39	36	39.20	253	3.04	3.15	4.44	2.60	1	2
36-41	36	40.78	503	9.81	10.96	14.72	9.40	4	4
36-42	36	43.38	1,053	16.97	8.58	47.71	7.96	12	11
36-44	36	48.28	1,078	43.11	322.01	358.15	60.58	119	124
36-47	36	50.62	217	2.99	7.41	5.26	7.27	19	19
36-48	36	45.04	109	0.63	0.65	0.81	0.68	0	1
39-0	39	43.93	13	0.10	0.11	0.09	0.13	0	0
39-1	39	49.08	59	0.39	0.67	0.47	0.69	0	0
39-2	39	52.01	29	0.15	0.18	0.17	0.19	0	0
39-3	39	45.63	49	0.25	0.28	0.29	0.28	5	5
39-4	39	48.05	28	0.15	0.19	0.17	0.20	0	0
39-5	39	51.86	107	0.95	1.47	1.21	1.53	0	1
39-6	39	45.95	47	0.31	0.31	0.37	0.33	0	0
39-7	39	42.42	87	0.86	0.62	0.67	0.64	0	1
39-9	39	45.97	29	0.17	0.19	0.20	0.21	0	0
39-10	39	45.42	57	0.82	0.83	0.97	0.85	2	3
39-11	39	49.99	117	1.32	2.44	3.35	2.70	4	6

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-12	39	44.28	113	1.09	1.31	1.34	1.40	0	1
39-13	39	49.52	53	0.29	0.43	0.41	0.42	0	0
39-15	39	53.30	103	0.99	1.16	0.90	1.04	0	1
39-16	39	44.38	508	28.91	29.14	14.76	17.90	3	8
39-17	39	48.96	208	5.36	1.13	41.23	1.18	26	26
39-19	39	50.24	458	19.40	9.67	117.09	8.78	53	59
39-20	39	48.86	108	0.82	0.83	0.92	0.85	0	1
39-21	39	54.40	258	7.58	2.11	686.48	2.10	25	31
39-23	39	56.08	578	607.75	305.15	3,479.41	73.43	45	53
39-24	39	43.94	278	8.89	15.36	425.04	12.32	15	16
39-25	39	51.83	138	1.84	1.66	1.86	2.08	0	11
39-26	39	43.42	478	13.45	32.84	15.23	21.69	2	14
39-28	39	48.52	48	0.27	0.32	0.75	0.32	0	0
39-30	39	46.19	528	26.64	72.61	4,270.44	31.65	49	48
39-31	39	45.02	523	11.67	35.50	18.74	18.57	3	4
39-34	39	50.03	473	18.84	24.63	501.12	15.30	50	51
39-35	39	45.98	133	1.12	0.63	1.39	0.66	0	12
39-36	39	48.82	273	4.71	10.05	7.22	7.84	15	17
39-37	39	49.63	573	120.55	56.25	1,380.21	37.37	22	33
39-39	39	42.20	253	3.10	3.12	3.63	2.53	1	2
39-41	39	43.78	503	13.32	10.17	13.40	8.54	4	4
39-42	39	46.38	1,053	15.77	5.16	41.88	5.21	9	11
39-44	39	51.28	1,078	71.14	349.48	21,750.05	54.62	119	167
39-47	39	53.62	217	2.95	7.29	5.64	6.63	19	24
39-48	39	48.04	109	0.68	0.60	0.89	0.64	0	1
42-0	42	46.93	13	0.08	0.11	0.11	0.13	0	0
42-1	42	52.08	59	0.38	0.67	0.51	0.68	0	0
42-2	42	55.01	29	0.18	0.20	0.18	0.19	0	0
42-3	42	48.63	49	0.26	0.25	0.30	0.29	5	5

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-4	42	51.05	28	0.17	0.19	0.18	0.20	0	0
42-5	42	54.86	107	0.93	1.46	0.99	1.50	0	1
42-6	42	48.95	47	0.29	0.31	0.36	0.33	0	0
42-7	42	45.42	87	0.81	0.62	0.75	0.63	0	1
42-8	42	59.18	97	0.85	0.97	3.17	1.00	10	11
42-9	42	48.97	29	0.27	0.19	0.25	0.19	0	0
42-10	42	48.42	57	2.44	0.58	0.55	0.62	2	3
42-11	42	52.99	117	1.34	2.46	1.59	2.62	0	1
42-12	42	47.28	113	1.13	1.34	1.29	1.36	0	1
42-13	42	52.52	53	0.29	0.41	0.32	0.39	0	1
42-15	42	56.30	103	0.70	1.13	0.88	1.04	0	1
42-16	42	47.38	508	85.83	33.66	1,998.22	17.43	57	60
42-17	42	51.96	208	3.29	1.06	6.93	1.09	26	27
42-19	42	53.24	458	16.23	9.78	374.36	9.05	58	63
42-20	42	51.86	108	0.83	0.83	1.01	0.85	0	1
42-21	42	57.40	258	3.41	1.67	899.24	1.70	32	33
42-22	42	49.30	558	216.69	45.53	595.97	27.06	4	9
42-23	42	59.08	578	1,409.54	260.17	4,624.76	78.95	49	55
42-24	42	46.94	278	41.07	17.18	464.64	12.10	11	17
42-25	42	54.83	138	1.53	1.75	2.06	1.78	0	15
42-26	42	46.42	478	26.02	47.26	52.51	28.60	52	53
42-28	42	51.52	48	0.28	0.32	0.34	0.32	0	0
42-30	42	49.19	528	47.16	83.37	1,443.47	32.16	47	54
42-31	42	48.02	523	12.24	28.29	1,127.42	17.84	3	4
42-34	42	53.03	473	29.08	20.59	94.42	15.07	53	55
42-35	42	48.98	133	1.12	0.63	1.19	0.66	0	12
42-36	42	51.82	273	8.86	9.40	33.69	7.30	17	18
42-37	42	52.63	573	71.42	55.66	179.17	27.82	3	5
42-39	42	45.20	253	3.05	3.14	4.73	2.55	1	2

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-41	42	46.78	503	11.26	2.31	13.86	2.31	4	4
42-42	42	49.38	1,053	15.66	5.12	66.33	5.13	11	60
42-44	42	54.28	1,078	164.79	318.00	3,477.09	61.63	169	169
42-47	42	56.62	217	2.95	7.67	5.45	4.55	23	24
42-48	42	51.04	109	0.62	0.59	0.93	0.62	0	5
45-0	45	49.93	13	0.08	0.11	0.10	0.12	0	0
45-1	45	55.08	59	0.39	0.60	0.38	0.62	0	0
45-2	45	58.01	29	0.17	0.18	0.18	0.19	0	0
45-3	45	51.63	49	0.25	0.26	0.30	0.28	5	5
45-4	45	54.05	28	0.17	0.19	0.16	0.19	0	0
45-5	45	57.86	107	0.90	1.48	1.01	1.51	0	1
45-6	45	51.95	47	0.29	0.32	0.31	0.33	0	0
45-7	45	48.42	87	0.59	0.43	0.68	0.45	0	0
45-9	45	51.97	29	0.25	0.19	0.26	0.21	0	0
45-10	45	51.42	57	0.36	0.61	0.70	0.62	4	4
45-11	45	55.99	117	1.88	2.41	3.40	2.50	4	7
45-12	45	50.28	113	1.01	1.34	1.24	1.36	0	1
45-13	45	55.52	53	0.28	0.42	0.33	0.41	0	1
45-15	45	59.30	103	0.96	1.13	0.88	1.05	0	1
45-16	45	50.38	508	19.30	35.28	804.43	17.84	68	64
45-17	45	54.96	208	6.55	1.21	32.79	1.18	31	31
45-19	45	56.24	458	17.40	8.93	4,774.66	8.11	67	71
45-20	45	54.86	108	0.81	0.85	1.08	0.86	0	1
45-21	45	60.40	258	26.57	1.67	592.78	1.71	38	35
45-23	45	62.08	578	2,261.17	200.57	1,772.08	99.78	50	69
45-24	45	49.94	278	16.20	17.74	21.80	12.04	13	18
45-25	45	57.83	138	1.44	1.74	1.68	1.74	0	15
45-26	45	49.42	478	54.52	43.01	131.66	24.26	52	52
45-28	45	54.52	48	0.27	0.31	0.36	0.33	0	0

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-30	45	52.19	528	161.68	62.08	1,142.13	32.58	54	59
45-31	45	51.02	523	12.65	29.73	266.55	18.26	3	5
45-34	45	56.03	473	38.35	26.46	132.01	15.83	61	67
45-35	45	51.98	133	1.47	0.63	1.27	0.68	0	12
45-36	45	54.82	273	10.86	10.31	19.83	8.22	19	28
45-37	45	55.63	573	94.94	60.71	2,051.39	36.57	23	34
45-39	45	48.20	253	3.09	3.15	3.50	2.57	1	2
45-41	45	49.78	503	8.89	2.32	18.27	2.31	5	5
45-42	45	52.38	1,053	15.25	5.09	2,155.69	5.16	60	61
45-44	45	57.28	1,078	78.66	314.49	21,756.45	72.39	179	175
45-47	45	59.62	217	2.98	7.57	12.64	4.11	24	26
45-48	45	54.04	109	0.62	0.58	0.98	0.61	5	5
48-0	48	52.93	13	0.08	0.11	0.11	0.12	0	0
48-1	48	58.08	59	0.37	0.66	0.44	0.65	0	1
48-2	48	61.01	29	0.17	0.18	0.17	0.18	0	0
48-3	48	54.63	49	0.24	0.26	0.30	0.27	5	5
48-4	48	57.05	28	0.16	0.19	0.18	0.23	0	0
48-5	48	60.86	107	0.90	1.48	1.19	1.51	0	1
48-6	48	54.95	47	0.27	0.31	0.30	0.33	0	0
48-7	48	51.42	87	0.58	0.44	0.98	0.44	4	4
48-8	48	65.18	97	0.85	0.95	1.41	0.98	10	10
48-9	48	54.97	29	0.24	0.19	0.24	0.21	0	0
48-10	48	54.42	57	5.53	0.67	5.03	0.65	3	5
48-11	48	58.99	117	1.27	2.43	1.74	2.52	0	2
48-12	48	53.28	113	1.24	1.34	1.46	1.37	0	1
48-13	48	58.52	53	0.28	0.41	0.34	0.42	0	1
48-15	48	62.30	103	0.67	1.12	0.90	1.04	0	1
48-16	48	53.38	508	30.67	26.73	21,766.88	17.52	80	71
48-17	48	57.96	208	2.50	1.15	6.86	1.25	31	32

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
48-19	48	59.24	458	23.12	8.64	326.93	7.76	75	76
48-20	48	57.86	108	0.80	0.51	4.13	0.54	5	6
48-21	48	63.40	258	33.01	1.65	365.83	1.68	42	41
48-22	48	55.30	558	82.15	53.44	44.27	30.96	3	9
48-23	48	65.08	578	2,584.95	312.60	2,071.08	84.33	49	70
48-24	48	52.94	278	38.19	17.31	1,683.18	12.44	17	27
48-25	48	60.83	138	1.48	1.57	1.54	1.59	0	20
48-26	48	52.42	478	19.04	36.66	155.78	24.96	52	58
48-28	48	57.52	48	1.66	0.31	0.34	0.32	0	0
48-30	48	55.19	528	129.79	103.49	1,362.55	33.59	53	63
48-31	48	54.02	523	15.30	23.61	303.08	18.55	3	5
48-34	48	59.03	473	15.83	26.05	97.04	16.86	67	71
48-35	48	54.98	133	1.37	0.63	1.54	0.64	0	13
48-36	48	57.82	273	8.75	15.92	16.89	8.43	22	30
48-37	48	58.63	573	211.17	59.52	54.40	41.47	4	53
48-39	48	51.20	253	3.00	3.20	3.81	2.61	1	2
48-41	48	52.78	503	8.93	2.31	12.94	2.38	6	5
48-42	48	55.38	1,053	14.79	5.10	530.49	5.07	64	62
48-44	48	60.28	1,078	135.34	504.10	729.38	78.79	180	215
48-47	48	62.62	217	2.92	8.30	9.77	4.11	28	30
48-48	48	57.04	109	0.62	0.51	0.77	0.54	5	6
51-0	51	55.93	13	0.08	0.11	0.13	0.13	0	0
51-1	51	61.08	59	0.36	0.66	0.44	0.67	0	0
51-2	51	64.01	29	0.17	0.18	0.21	0.20	0	0
51-3	51	57.63	49	0.24	0.26	0.30	0.27	5	5
51-4	51	60.05	28	0.16	0.19	0.17	0.20	0	0
51-5	51	63.86	107	0.90	1.49	1.02	1.50	0	4
51-6	51	57.95	47	0.40	0.28	0.37	0.29	0	1
51-7	51	54.42	87	0.57	0.42	1.22	0.45	4	5

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-9	51	57.97	29	0.23	0.19	0.25	0.21	0	0
51-10	51	57.42	57	1.11	0.75	1.64	0.71	3	6
51-11	51	61.99	117	11.66	2.64	11.67	3.01	4	18
51-12	51	56.28	113	0.97	1.35	1.20	1.35	0	1
51-13	51	61.52	53	0.29	0.32	0.31	0.34	0	1
51-15	51	65.30	103	0.67	1.14	0.85	1.09	0	1
51-16	51	56.38	508	36.43	34.74	896.98	15.41	76	76
51-17	51	60.96	208	4.50	1.15	17.17	1.16	36	37
51-19	51	62.24	458	16.64	8.68	414.02	7.77	81	86
51-20	51	60.86	108	0.88	0.51	9.41	0.53	6	6
51-23	51	68.08	578	2,189.57	202.43	6,937.57	74.06	61	83
51-24	51	55.94	278	21.26	14.89	426.91	13.13	16	28
51-25	51	63.83	138	1.42	1.54	1.55	1.60	0	21
51-26	51	55.42	478	15.55	44.67	32.62	28.56	52	57
51-28	51	60.52	48	1.05	0.32	0.33	0.33	0	0
51-30	51	58.19	528	284.43	113.53	1,018.48	35.51	60	79
51-31	51	57.02	523	13.37	14.54	25.88	18.37	3	5
51-34	51	62.03	473	13.61	23.68	80.90	15.31	67	76
51-35	51	57.98	133	1.27	0.61	1.67	0.62	0	13
51-36	51	60.82	273	15.07	13.98	19.41	9.14	25	33
51-37	51	61.63	573	1,180.85	68.57	3,389.24	40.87	23	93
51-39	51	54.20	253	3.82	3.16	4.11	2.57	1	2
51-41	51	55.78	503	10.94	2.22	15.52	2.30	8	5
51-42	51	58.38	1,053	14.74	4.96	2,041.26	5.05	63	63
51-44	51	63.28	1,078	207.13	391.72	21,765.15	77.21	223	225
51-47	51	65.62	217	2.86	4.31	9.11	4.66	28	30
51-48	51	60.04	109	0.62	0.53	0.79	0.55	5	6
54-0	54	58.93	13	0.08	0.11	0.10	0.12	0	0
54-1	54	64.08	59	0.33	0.64	0.53	0.66	0	2

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
54-2	54	67.01	29	0.16	0.18	0.18	0.20	0	0
54-3	54	60.63	49	0.24	0.26	0.30	0.27	5	5
54-4	54	63.05	28	0.16	0.19	0.19	0.20	0	0
54-5	54	66.86	107	1.79	1.27	3.77	1.18	18	18
54-6	54	60.95	47	0.37	0.28	0.36	0.28	0	1
54-7	54	57.42	87	0.57	0.43	1.07	0.44	4	5
54-8	54	71.18	97	1.03	0.98	1.91	1.00	10	12
54-9	54	60.97	29	0.23	0.18	0.19	0.19	2	2
54-10	54	60.42	57	2.44	0.69	3.37	0.66	3	7
54-11	54	64.99	117	1.26	2.22	1.41	2.19	0	3
54-12	54	59.28	113	2.28	1.35	1.37	1.33	0	1
54-13	54	64.52	53	0.29	0.32	0.33	0.34	0	1
54-15	54	68.30	103	2.26	1.12	1.07	1.05	0	1
54-16	54	59.38	508	20.40	31.14	740.72	15.57	87	83
54-17	54	63.96	208	3.16	1.13	20.79	1.19	41	41
54-19	54	65.24	458	38.51	8.67	1,535.90	7.95	90	92
54-20	54	63.86	108	0.72	0.50	6.06	0.52	6	6
54-22	54	61.30	558	117.32	56.52	32.66	32.23	4	11
54-23	54	71.08	578	2,417.78	315.21	2,171.73	94.23	62	84
54-24	54	58.94	278	178.07	18.74	1,021.23	12.22	16	36
54-25	54	66.83	138	1.39	1.50	153.58	1.52	20	21
54-26	54	58.42	478	21.47	51.02	39.88	24.47	53	58
54-28	54	63.52	48	0.65	0.32	0.33	0.34	0	0
54-30	54	61.19	528	1,102.21	97.91	954.39	31.06	63	88
54-31	54	60.02	523	18.77	32.62	20.07	17.01	4	5
54-34	54	65.03	473	11.95	19.02	19,065.26	15.62	76	81
54-35	54	60.98	133	1.07	0.61	1.17	0.63	0	22
54-36	54	63.82	273	27.85	7.47	10.26	8.61	38	36
54-37	54	64.63	573	215.71	82.86	136.32	35.44	5	26

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
54-39	54	57.20	253	3.06	3.19	3.99	2.57	2	3
54-42	54	61.38	1,053	15.90	5.13	332.23	5.28	63	64
54-44	54	66.28	1,078	139.53	409.73	21,723.35	75.75	226	231
54-47	54	68.62	217	2.83	4.20	55.28	4.15	42	43
54-48	54	63.04	109	0.63	0.52	1.06	0.54	5	5
57-0	57	61.93	13	0.08	0.11	0.10	0.11	0	0
57-1	57	67.08	59	0.34	0.63	0.40	0.65	0	1
57-2	57	70.01	29	0.18	0.18	0.17	0.20	0	0
57-3	57	63.63	49	0.25	0.26	0.42	0.27	5	5
57-4	57	66.05	28	0.17	0.19	0.19	0.22	0	0
57-5	57	69.86	107	1.16	1.20	1.00	1.23	0	4
57-6	57	63.95	47	0.36	0.27	0.36	0.29	0	1
57-7	57	60.42	87	0.59	0.44	0.75	0.44	4	5
57-9	57	63.97	29	0.19	0.19	0.28	0.27	2	2
57-10	57	63.42	57	1.90	0.76	3.53	0.74	6	8
57-11	57	67.99	117	4.01	2.56	11.62	2.92	11	19
57-12	57	62.28	113	0.91	1.37	1.00	1.39	0	1
57-13	57	67.52	53	0.28	0.32	0.33	0.33	0	1
57-15	57	71.30	103	1.91	0.71	0.78	0.77	1	1
57-16	57	62.38	508	11.08	26.20	18.25	19.27	4	11
57-17	57	66.96	208	2.43	1.18	6.21	1.25	42	42
57-19	57	68.24	458	13.10	8.44	212.41	7.50	91	100
57-20	57	66.86	108	0.73	0.50	4.27	0.52	6	6
57-22	57	64.30	558	1,655.26	64.03	21,776.65	37.35	71	98
57-23	57	74.08	578	11,112.43	563.00	21,864.49	92.36	70	128
57-24	57	61.94	278	139.74	17.83	490.04	13.01	16	37
57-25	57	69.83	138	1.51	1.51	35.12	1.48	20	21
57-26	57	61.42	478	16.45	61.12	72.47	26.24	53	59
57-28	57	66.52	48	0.39	0.55	0.46	0.55	0	1

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-30	57	64.19	528	661.50	73.44	7,604.97	30.43	68	89
57-31	57	63.02	523	14.39	9.22	23.25	9.27	4	25
57-34	57	68.03	473	21.58	33.17	132.90	14.86	79	93
57-35	57	63.98	133	1.07	0.64	44.73	0.67	20	22
57-36	57	66.82	273	21.61	6.73	14.35	7.42	29	38
57-37	57	67.63	573	1,244.59	89.40	21,786.37	47.05	43	94
57-39	57	60.20	253	3.01	3.16	4.28	2.70	2	3
57-42	57	64.38	1,053	15.16	5.19	21,722.99	5.23	64	66
57-44	57	69.28	1,078	251.03	452.67	21,725.79	77.29	228	266
57-47	57	71.62	217	3.00	4.19	71.73	4.27	42	47
57-48	57	66.04	109	0.60	0.51	1.38	0.54	5	10
60-0	60	64.93	13	0.08	0.13	0.12	0.14	0	1
60-1	60	70.08	59	0.35	0.55	0.40	0.56	10	10
60-2	60	73.01	29	0.16	0.22	0.20	0.20	0	0
60-3	60	66.63	49	0.24	0.27	0.33	0.29	5	6
60-4	60	69.05	28	0.16	0.18	0.17	0.19	0	0
60-5	60	72.86	107	0.87	1.22	1.09	1.26	20	21
60-6	60	66.95	47	0.38	0.29	0.31	0.28	0	1
60-7	60	63.42	87	0.59	0.43	0.72	0.44	4	5
60-8	60	77.18	97	2.16	0.95	2.58	1.21	21	21
60-9	60	66.97	29	0.16	0.22	0.27	0.21	2	2
60-10	60	66.42	57	10.58	0.57	0.85	0.56	4	8
60-11	60	70.99	117	3.06	2.10	1.52	2.49	2	20
60-12	60	65.28	113	1.13	1.33	1.03	1.35	0	1
60-13	60	70.52	53	0.28	0.32	0.32	0.34	0	1
60-15	60	74.30	103	0.62	0.75	0.65	0.74	20	20
60-16	60	65.38	508	15.24	37.16	633.04	16.20	94	96
60-17	60	69.96	208	2.79	1.16	11.12	1.19	46	47
60-19	60	71.24	458	153.82	6.76	376.07	6.26	100	109

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
60-20	60	69.86	108	0.71	0.50	2.47	0.53	6	6
60-22	60	67.30	558	117.41	56.11	83.69	35.88	7	56
60-23	60	77.08	578	16,040.18	380.03	21,959.01	90.71	80	134
60-24	60	64.94	278	235.04	21.58	1,206.14	13.77	22	41
60-25	60	72.83	138	1.39	1.52	24.62	1.90	20	21
60-26	60	64.42	478	21.11	56.29	35.24	24.87	54	59
60-28	60	69.52	48	0.27	0.50	0.32	0.50	0	1
60-30	60	67.19	528	1,616.85	70.00	2,099.22	34.50	70	98
60-31	60	66.02	523	11.36	12.85	298.50	19.44	6	6
60-34	60	71.03	473	48.11	24.20	124.80	15.12	102	103
60-35	60	66.98	133	1.05	0.62	20.71	0.63	20	23
60-36	60	69.82	273	31.57	7.85	98.74	5.87	52	48
60-37	60	70.63	573	59.59	67.93	2,010.49	42.31	54	55
60-39	60	63.20	253	2.93	3.20	4.37	2.58	3	3
60-42	60	67.38	1,053	16.25	5.27	21,702.85	5.36	115	115
60-44	60	72.28	1,078	158.89	616.66	21,741.23	83.72	282	289
60-47	60	74.62	217	2.74	4.22	36.29	4.89	49	51
60-48	60	69.04	109	0.64	0.51	1.12	0.56	10	11
63-0	63	67.93	13	0.08	0.10	0.08	0.11	2	2
63-1	63	73.08	59	0.52	0.62	0.35	0.62	0	6
63-2	63	76.01	29	0.15	0.19	0.19	0.19	0	0
63-3	63	69.63	49	0.94	0.27	0.49	0.41	10	10
63-4	63	72.05	28	0.15	0.19	0.18	0.21	0	0
63-5	63	75.86	107	1.12	1.20	1.12	1.19	20	20
63-6	63	69.95	47	0.27	0.26	0.35	0.26	4	4
63-7	63	66.42	87	0.53	0.45	0.92	0.46	5	5
63-9	63	69.97	29	0.22	0.20	0.27	0.21	2	3
63-10	63	69.42	57	0.71	0.60	2.44	0.71	8	9
63-11	63	73.99	117	2.51	2.41	26.15	2.41	11	22

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-12	63	68.28	113	0.86	1.25	0.90	1.26	20	10
63-15	63	77.30	103	0.59	0.70	0.76	0.68	21	21
63-16	63	68.38	508	8.93	39.59	16.16	18.00	104	104
63-17	63	72.96	208	2.31	1.18	22.77	1.17	53	51
63-19	63	74.24	458	117.34	8.48	487.56	7.51	110	116
63-20	63	72.86	108	0.74	0.52	2.18	0.55	6	7
63-23	63	80.08	578	21,267.33	711.50	18,746.22	80.54	87	139
63-24	63	67.94	278	146.33	19.76	78.77	13.87	24	47
63-25	63	75.83	138	3.51	1.46	16.71	1.45	20	22
63-26	63	67.42	478	16.16	48.81	27.68	29.01	54	60
63-28	63	72.52	48	0.27	0.49	0.42	0.49	0	1
63-30	63	70.19	528	688.58	78.30	4,501.36	27.64	98	103
63-31	63	69.02	523	9.31	8.96	12.98	8.73	103	103
63-34	63	74.03	473	24.82	32.11	123.35	16.25	107	118
63-36	63	72.82	273	2.91	5.63	129.56	4.43	50	51
63-37	63	73.63	573	9,575.36	76.23	2,169.68	39.11	43	114
63-39	63	66.20	253	3.53	1.10	4.90	1.12	2	3
63-42	63	70.38	1,053	15.39	5.54	21,725.95	5.47	130	118
63-44	63	75.28	1,078	313.00	725.83	21,754.36	87.18	276	321
63-47	63	77.62	217	4.13	4.08	3,246.76	4.00	50	56
63-48	63	72.04	109	0.60	0.52	0.92	0.54	10	12
66-0	66	70.93	13	0.07	0.10	0.09	0.11	2	2
66-1	66	76.08	59	0.38	0.55	0.41	0.56	10	10
66-2	66	79.01	29	0.16	0.18	0.17	0.20	0	0
66-3	66	72.63	49	0.48	0.27	0.71	0.29	10	10
66-4	66	75.05	28	0.15	0.19	0.18	0.21	0	0
66-5	66	78.86	107	0.82	1.16	0.89	1.16	21	21
66-6	66	72.95	47	0.28	0.25	0.33	0.25	4	4
66-7	66	69.42	87	0.72	0.42	0.95	0.45	9	9

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
66-8	66	83.18	97	0.93	0.97	1.37	1.20	23	21
66-9	66	72.97	29	0.19	0.20	0.26	0.21	2	3
66-10	66	72.42	57	1.10	0.57	2.83	0.71	9	10
66-11	66	76.99	117	1.24	2.17	2.01	2.13	20	21
66-12	66	71.28	113	0.97	1.49	1.30	1.48	0	10
66-15	66	80.30	103	0.57	0.50	0.75	0.51	21	21
66-16	66	71.38	508	25.66	33.47	21,785.93	16.05	142	113
66-19	66	77.24	458	27.40	3.45	487.92	3.47	122	127
66-20	66	75.86	108	1.48	0.53	9.10	0.56	11	11
66-22	66	73.30	558	13.47	71.02	29.31	30.17	107	105
66-23	66	83.08	578	26,657.36	654.21	21,874.12	100.08	111	145
66-24	66	70.94	278	415.59	20.11	1,448.29	11.54	47	51
66-25	66	78.83	138	1.36	1.17	14.74	1.13	20	23
66-26	66	70.42	478	21.73	55.81	45.45	27.31	64	75
66-28	66	75.52	48	0.28	0.43	0.40	0.46	0	1
66-30	66	73.19	528	1,535.08	86.05	2,644.12	30.33	99	118
66-31	66	72.02	523	13.52	12.90	33.62	20.20	6	27
66-34	66	77.03	473	16.29	21.99	109.47	13.69	118	125
66-36	66	75.82	273	2.78	5.39	159.82	4.29	56	54
66-37	66	76.63	573	36.00	72.14	20.94	31.45	106	105
66-39	66	69.20	253	2.84	1.10	8.73	1.12	3	4
66-42	66	73.38	1,053	15.85	5.39	21,710.78	5.66	119	120
66-44	66	78.28	1,078	455.73	548.89	21,759.27	95.29	322	341
66-47	66	80.62	217	2.71	4.29	28.83	4.16	65	65
66-48	66	75.04	109	0.62	0.56	1.05	0.55	11	17
69-0	69	73.93	13	0.08	0.10	0.10	0.12	2	2
69-1	69	79.08	59	0.31	0.54	0.33	0.54	10	10
69-2	69	82.01	29	0.16	0.18	0.17	0.18	0	0
69-3	69	75.63	49	0.31	0.27	0.42	0.30	10	10

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
69-4	69	78.05	28	0.15	0.20	0.25	0.20	0	1
69-5	69	81.86	107	0.75	1.14	0.97	1.15	21	21
69-6	69	75.95	47	0.27	0.25	0.33	0.27	4	4
69-7	69	72.42	87	0.61	0.43	1.19	0.44	9	9
69-9	69	75.97	29	0.17	0.20	0.26	0.21	2	3
69-10	69	75.42	57	3.63	0.59	11.14	0.70	9	11
69-11	69	79.99	117	1.13	2.00	1.73	1.94	21	21
69-12	69	74.28	113	0.84	1.52	0.99	1.43	20	21
69-15	69	83.30	103	0.57	0.44	0.79	0.48	21	21
69-16	69	74.38	508	51.17	8.20	14,544.92	6.01	120	122
69-19	69	80.24	458	25.49	3.44	1,061.80	3.53	151	151
69-20	69	78.86	108	2.78	0.62	5.91	0.54	11	12
69-23	69	86.08	578	16,626.61	514.35	21,806.80	107.56	123	160
69-24	69	73.94	278	157.74	22.84	1,175.01	9.82	47	52
69-25	69	81.83	138	2.64	1.39	53.01	1.38	20	23
69-26	69	73.42	478	23.43	76.83	21,644.91	25.28	109	109
69-28	69	78.52	48	0.28	0.40	0.41	0.42	0	1
69-30	69	76.19	528	934.31	110.06	21,824.49	34.48	108	124
69-31	69	75.02	523	8.41	7.72	13.90	7.05	104	104
69-34	69	80.03	473	20.85	29.96	101.98	13.48	106	147
69-36	69	78.82	273	25.51	6.06	77.54	4.87	60	58
69-37	69	79.63	573	1,161.54	90.11	917.59	45.03	57	116
69-39	69	72.20	253	2.80	1.10	6.64	1.12	3	5
69-42	69	76.38	1,053	16.12	5.91	21,700.17	5.64	172	171
69-44	69	81.28	1,078	534.63	1,010.91	21,751.39	104.73	328	372
69-47	69	83.62	217	2.75	4.07	20.53	4.07	65	71
69-48	69	78.04	109	0.57	0.55	1.32	0.55	19	17
72-0	72	76.93	13	0.08	0.09	0.09	0.11	2	2
72-1	72	82.08	59	0.31	0.54	0.35	0.55	10	10

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
72-2	72	85.01	29	0.28	0.20	0.18	0.21	0	1
72-3	72	78.63	49	0.33	0.27	0.60	0.29	14	10
72-4	72	81.05	28	0.15	0.22	0.25	0.20	0	1
72-5	72	84.86	107	0.77	1.16	1.17	1.16	21	22
72-6	72	78.95	47	0.27	0.25	0.42	0.28	4	5
72-7	72	75.42	87	0.52	0.44	1.12	0.45	9	10
72-8	72	89.18	97	0.65	1.03	1.17	1.25	32	31
72-9	72	78.97	29	0.16	0.16	0.31	0.19	2	3
72-10	72	78.42	57	0.54	0.56	3.39	0.70	9	11
72-11	72	82.99	117	1.14	2.04	1.19	1.98	21	21
72-12	72	77.28	113	0.81	1.56	1.20	1.45	20	21
72-15	72	86.30	103	0.54	0.47	0.92	0.47	21	22
72-16	72	77.38	508	30.48	34.74	291.36	15.81	140	131
72-19	72	83.24	458	11.90	3.38	67.26	3.52	162	159
72-22	72	79.30	558	16.79	69.51	195.55	26.99	110	110
72-23	72	89.08	578	67,811.62	363.72	21,827.05	85.46	139	174
72-24	72	76.94	278	376.57	29.00	759.66	11.78	48	57
72-25	72	84.83	138	2.91	1.47	3.03	1.49	20	23
72-26	72	76.42	478	1,167.89	48.42	1,817.46	19.04	125	133
72-28	72	81.52	48	0.28	0.25	0.42	0.27	5	5
72-30	72	79.19	528	712.15	94.57	1,609.21	33.57	112	130
72-31	72	78.02	523	28.06	11.28	13.23	11.20	105	105
72-34	72	83.03	473	20.41	38.59	99.46	11.79	157	153
72-36	72	81.82	273	14.51	6.32	269.41	5.10	60	62
72-37	72	82.63	573	1,877.41	93.17	734.98	43.10	64	116
72-39	72	75.20	253	2.74	1.16	49.96	1.13	4	5
72-42	72	79.38	1,053	16.88	5.78	21,693.51	5.94	176	176
72-44	72	84.28	1,078	567.90	838.94	21,807.13	106.78	385	397
72-47	72	86.62	217	2.76	2.74	57.12	2.79	72	73

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
72-48	72	81.04	109	1.12	0.56	1.19	0.57	20	22
75-0	75	79.93	13	0.08	0.10	0.10	0.11	2	2
75-1	75	85.08	59	0.30	0.54	0.33	0.54	10	10
75-2	75	88.01	29	0.36	0.20	0.17	0.23	0	1
75-3	75	81.63	49	0.29	0.28	0.32	0.28	13	10
75-4	75	84.05	28	0.15	0.20	0.17	0.21	0	1
75-5	75	87.86	107	0.70	1.15	0.78	1.16	21	22
75-6	75	81.95	47	0.27	0.26	0.37	0.27	4	5
75-7	75	78.42	87	0.58	0.44	0.85	0.46	13	13
75-9	75	81.97	29	0.16	0.17	0.25	0.18	2	3
75-10	75	81.42	57	0.57	0.56	1.57	0.70	10	13
75-11	75	85.99	117	1.03	2.07	1.44	2.02	21	22
75-12	75	80.28	113	0.79	1.42	1.30	1.35	21	21
75-15	75	89.30	103	0.56	0.46	0.67	0.53	22	22
75-16	75	80.38	508	24.76	31.37	1,582.82	12.55	138	141
75-19	75	86.24	458	14.98	3.32	105.36	3.26	173	167
75-22	75	82.30	558	538.74	202.92	21,837.76	33.01	145	140
75-23	75	92.08	578	47,278.86	500.17	21,897.25	77.98	163	185
75-24	75	79.94	278	4.51	30.58	377.51	10.97	57	58
75-25	75	87.83	138	1.33	1.66	4.68	1.62	21	36
75-26	75	79.42	478	14.70	72.70	21,684.05	30.15	109	111
75-28	75	84.52	48	0.27	0.25	0.41	0.26	5	5
75-30	75	82.19	528	395.58	97.09	21,774.76	33.66	139	144
75-31	75	81.02	523	8.21	8.54	16.20	7.89	105	106
75-34	75	86.03	473	18.61	8.21	96.22	8.16	168	161
75-36	75	84.82	273	2.38	6.71	52.07	5.57	71	67
75-37	75	85.63	573	1,740.79	145.56	3,869.11	50.17	63	118
75-39	75	78.20	253	2.66	1.09	15.88	1.11	3	6
75-42	75	82.38	1,053	18.82	6.08	21,687.36	6.31	246	228

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
75-44	75	87.28	1,078	991.43	1,163.14	1,395.50	104.19	385	426
75-47	75	89.62	217	3.06	3.80	11.47	3.84	89	89
75-48	75	84.04	109	0.80	0.62	1.29	0.64	24	24
78-0	78	82.93	13	0.08	0.10	0.09	0.11	2	2
78-1	78	88.08	59	0.30	0.53	0.34	0.56	10	10
78-2	78	91.01	29	0.14	0.17	0.17	0.25	5	5
78-3	78	84.63	49	0.23	0.30	0.42	0.32	10	12
78-4	78	87.05	28	0.19	0.16	0.22	0.18	0	1
78-5	78	90.86	107	0.77	1.01	1.34	1.02	23	22
78-6	78	84.95	47	0.26	0.26	0.40	0.28	4	5
78-7	78	81.42	87	0.50	0.44	0.67	0.47	13	14
78-8	78	95.18	97	0.61	0.79	1.44	0.79	34	33
78-9	78	84.97	29	0.16	0.17	0.27	0.18	2	5
78-10	78	84.42	57	1.56	0.56	1.51	0.74	12	14
78-11	78	88.99	117	1.21	2.31	1.58	2.34	22	22
78-12	78	83.28	113	0.75	1.48	1.03	1.36	21	21
78-15	78	92.30	103	0.53	0.49	0.69	0.50	27	24
78-16	78	83.38	508	19.01	45.59	358.69	13.26	153	151
78-19	78	89.24	458	13.76	3.44	81.01	3.49	196	176
78-22	78	85.30	558	14.92	79.49	83.00	29.36	115	114
78-24	78	82.94	278	362.46	25.69	761.29	11.76	56	67
78-25	78	90.83	138	6.28	1.37	7.60	1.37	32	46
78-26	78	82.42	478	20.95	71.82	21,656.25	27.46	109	111
78-28	78	87.52	48	0.27	0.27	0.42	0.26	5	6
78-30	78	85.19	528	397.64	140.91	21,811.05	36.55	146	154
78-31	78	84.02	523	9.47	8.05	13.69	7.59	107	108
78-34	78	89.03	473	21.79	7.91	147.29	7.29	177	175
78-36	78	87.82	273	32.08	4.79	30.26	4.19	75	60
78-37	78	88.63	573	2,172.36	116.71	2,898.56	46.67	69	156

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
78-42	78	85.38	1,053	19.76	6.97	21,683.90	7.08	293	283
78-44	78	90.28	1,078	1,434.18	1,591.52	21,837.96	118.78	456	468
78-48	78	87.04	109	0.56	0.65	1.05	0.67	29	29
81-0	81	85.93	13	0.08	0.10	0.09	0.11	2	2
81-1	81	91.08	59	0.32	0.60	0.49	0.59	10	11
81-2	81	94.01	29	0.14	0.17	0.19	0.26	5	5
81-3	81	87.63	49	0.26	0.30	0.29	0.31	17	15
81-4	81	90.05	28	0.15	0.16	0.17	0.18	5	5
81-5	81	93.86	107	0.71	0.90	1.23	0.94	29	30
81-6	81	87.95	47	0.30	0.28	0.70	0.27	9	9
81-7	81	84.42	87	0.82	0.48	1.33	0.48	17	18
81-9	81	87.97	29	0.16	0.17	0.22	0.19	2	6
81-10	81	87.42	57	0.99	0.50	4.09	0.48	13	12
81-11	81	91.99	117	1.09	1.64	4.63	1.64	23	25
81-12	81	86.28	113	0.80	1.38	1.04	1.30	21	22
81-15	81	95.30	103	0.51	0.50	1.01	0.51	31	32
81-16	81	86.38	508	15.86	48.15	231.39	9.93	163	161
81-19	81	92.24	458	13.87	3.43	50.01	3.47	186	186
81-22	81	88.30	558	2,214.22	161.39	6,795.43	35.65	172	160
81-24	81	85.94	278	187.82	25.60	236.16	12.05	66	73
81-25	81	93.83	138	2.33	0.27	5.45	0.31	42	NA
81-26	81	85.42	478	40.77	75.87	99.82	30.31	108	127
81-28	81	90.52	48	0.27	0.27	0.88	0.27	5	6
81-30	81	88.19	528	1,183.19	141.60	4,245.26	37.23	147	165
81-31	81	87.02	523	8.25	9.67	85.51	8.99	109	111
81-34	81	92.03	473	74.69	6.13	63.35	6.22	185	184
81-36	81	90.82	273	37.40	4.91	231.80	4.27	85	85
81-37	81	91.63	573	172.39	114.95	2,519.68	49.97	89	161
81-42	81	88.38	1,053	21.66	11.34	9,912.05	12.16	356	344

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
84-0	84	88.93	13	0.08	0.10	0.08	0.11	2	2
84-1	84	94.08	59	0.30	0.54	0.50	0.53	12	14
84-2	84	97.01	29	0.13	0.16	0.19	0.18	7	7
84-3	84	90.63	49	0.24	0.29	0.28	0.30	17	15
84-4	84	93.05	28	0.14	0.16	0.17	0.16	6	6
84-6	84	90.95	47	0.28	0.26	0.44	0.28	10	9
84-7	84	87.42	87	0.76	0.51	0.91	0.53	17	18
84-9	84	90.97	29	0.16	0.16	0.24	0.17	6	7
84-10	84	90.42	57	0.89	0.48	5.13	0.48	14	17
84-11	84	94.99	117	7.02	1.21	5.32	1.24	37	39
84-12	84	89.28	113	0.69	1.44	1.31	1.33	22	23
84-16	84	89.38	508	15.20	41.44	261.20	17.36	179	172
84-19	84	95.24	458	10.70	4.19	29.33	4.19	206	196
84-22	84	91.30	558	59.14	97.00	9,770.64	26.97	133	132
84-24	84	88.94	278	560.42	27.83	701.79	9.41	78	82
84-25	84	96.83	138	2.67	0.28	1.83	0.29	55	NA
84-26	84	88.42	478	10.63	66.11	32.04	25.64	156	160
84-28	84	93.52	48	0.26	0.27	0.61	0.28	5	5
84-30	84	91.19	528	766.30	176.67	4,331.19	38.45	169	179
84-31	84	90.02	523	8.06	9.66	200.45	8.91	116	115
84-34	84	95.03	473	68.82	5.91	31.80	5.81	210	195
84-36	84	93.82	273	41.07	4.56	76.07	4.02	93	91
84-37	84	94.63	573	1,505.29	633.58	22,138.98	22.26	173	190
84-42	84	91.38	1,053	40.56	118.83	197.30	26.80	445	464
87-0	87	91.93	13	0.08	0.10	0.09	0.11	2	2
87-3	87	93.63	49	0.23	0.30	0.24	0.31	19	15
87-6	87	93.95	47	0.23	0.27	0.40	0.28	17	13
87-7	87	90.42	87	0.73	0.55	1.51	0.55	23	22
87-9	87	93.97	29	0.15	0.15	0.23	0.17	6	7

Table A.14: Results for Benchmark 1 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
87-10	87	93.42	57	7.77	0.48	2.66	0.48	20	18
87-12	87	92.28	113	0.72	1.45	1.43	1.38	26	25
87-16	87	92.38	508	12.27	58.11	2,369.99	14.58	135	134
87-22	87	94.30	558	2,136.64	116.19	885.84	27.17	196	195
87-24	87	91.94	278	381.25	26.09	1,119.80	9.10	83	87
87-26	87	91.42	478	9.19	57.62	27.15	23.51	164	163
87-28	87	96.52	48	0.23	0.25	1.25	0.27	12	12
87-30	87	94.19	528	1,852.05	255.50	20,638.99	34.02	186	191
87-31	87	93.02	523	9.20	5.93	28.64	5.98	128	142
87-36	87	96.82	273	14.23	4.60	182.49	4.01	97	97
90-0	90	94.93	13	0.08	0.09	0.09	0.10	3	3
90-6	90	96.95	47	0.21	0.26	0.24	0.26	23	17
90-7	90	93.42	87	0.56	0.57	1.02	0.56	26	30
90-9	90	96.97	29	0.42	0.04	0.33	0.04	7	NA
90-10	90	96.42	57	1.41	0.49	1.97	0.51	17	19
90-12	90	95.28	113	1.01	1.09	2.52	1.14	39	30
90-16	90	95.38	508	36.52	9.78	61.78	8.91	203	197
90-24	90	94.94	278	731.89	25.19	1,081.32	6.89	86	93
90-26	90	94.42	478	28.83	63.97	70.05	21.79	169	177
90-28	90	99.52	48	0.22	0.27	0.65	0.29	18	19
93-7	93	96.42	87	0.42	0.54	0.43	0.56	39	36

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-0	30	31.50	29	0.13	0.10	0.11	0.10	0	0
30-1	30	32.11	147	0.51	0.47	0.46	0.42	0	0
30-2	30	31.48	67	0.22	0.17	0.20	0.18	0	0
30-3	30	31.32	107	0.33	0.34	0.37	0.35	0	10
30-4	30	32.04	65	0.17	0.16	0.19	0.16	0	0

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-5	30	31.61	259	1.29	0.75	3.14	0.74	10	18
30-6	30	31.12	99	0.40	0.25	0.37	0.26	0	0
30-7	30	31.53	181	0.75	0.41	0.80	0.43	0	0
30-8	30	32.06	217	1.05	0.77	1.00	0.76	0	20
30-9	30	30.57	57	0.30	0.17	0.28	0.17	0	0
30-10	30	31.22	135	1.32	0.48	1.05	0.46	4	8
30-11	30	32.11	295	1.53	1.00	1.48	1.00	0	0
30-12	30	31.55	285	1.36	0.74	1.17	0.75	0	0
30-13	30	30.66	125	0.35	0.29	0.35	0.28	0	0
30-14	30	31.79	205	0.77	0.46	0.79	0.47	0	0
30-15	30	31.47	245	0.95	0.50	0.99	0.49	0	0
30-16	30	31.36	613	6.00	1.72	197.63	1.78	54	57
30-17	30	31.00	213	1.02	0.50	1.95	0.52	17	21
30-18	30	31.03	413	2.87	0.91	5.39	0.89	36	36
30-19	30	31.62	513	8.45	1.66	18.32	1.64	42	48
30-20	30	30.69	113	0.46	0.28	0.36	0.30	0	0
30-21	30	31.51	313	2.85	0.94	5.55	0.91	23	28
30-22	30	32.07	713	13.81	2.46	80.40	2.42	31	37
30-23	30	31.92	737	16.90	3.42	27.73	3.35	21	48
30-24	30	31.11	337	3.19	1.28	18.46	1.24	16	23
30-25	30	30.77	143	1.34	0.50	0.97	0.47	0	1
30-26	30	31.50	543	6.63	2.93	21.33	2.62	27	58
30-27	30	31.73	453	3.93	1.60	13.62	1.56	37	56
30-28	30	30.18	53	0.22	0.15	0.19	0.15	0	0
30-29	30	31.03	247	1.41	0.71	3.15	0.74	14	16
30-30	30	31.65	647	9.50	2.33	60.06	2.40	27	46
30-31	30	31.70	642	6.06	1.77	7.44	1.76	1	1
30-32	30	30.77	242	1.10	0.69	1.16	0.68	0	20
30-33	30	31.04	443	3.40	0.94	4.48	0.99	1	2

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-34	30	31.66	533	4.42	3.79	6.83	3.21	3	53
30-35	30	30.72	133	0.59	0.42	0.59	0.40	0	1
30-36	30	30.73	332	1.78	1.42	2.71	1.27	14	24
30-37	30	32.22	732	10.78	3.38	25.32	4.24	1	33
30-38	30	31.50	703	6.11	1.86	13.14	1.81	35	35
30-39	30	31.45	303	1.53	0.65	1.62	0.66	0	1
30-40	30	31.24	503	3.77	2.35	5.02	2.05	1	52
30-41	30	31.76	603	4.52	1.25	5.69	1.28	3	4
30-42	30	33.00	1,103	11.47	2.60	15.48	2.55	3	4
30-43	30	32.76	1,132	16.06	4.33	24.95	4.15	60	104
30-44	30	32.74	1,137	15.21	4.68	45.61	4.74	70	108
30-45	30	33.14	1,113	15.47	3.65	26.29	3.49	71	58
30-46	30	32.35	445	2.26	1.03	4.07	0.99	24	42
30-47	30	32.60	455	2.56	1.39	2.45	1.39	20	20
30-48	30	32.82	227	0.78	0.54	0.63	0.56	0	2
30-49	30	32.54	419	1.90	0.92	3.06	0.95	33	40
33-0	33	34.50	29	0.12	0.10	0.13	0.11	0	0
33-1	33	35.11	147	0.55	0.41	0.43	0.41	0	0
33-2	33	34.48	67	0.23	0.17	0.21	0.18	0	0
33-3	33	34.32	107	0.39	0.39	0.37	0.35	0	10
33-4	33	35.04	65	0.18	0.16	0.26	0.16	0	0
33-5	33	34.61	259	1.01	0.76	1.00	0.73	0	2
33-6	33	34.12	99	0.31	0.27	0.35	0.26	0	0
33-7	33	34.53	181	0.72	0.43	1.74	0.42	8	16
33-8	33	35.06	217	0.91	0.78	1.13	0.72	0	4
33-9	33	33.57	57	0.28	0.18	0.28	0.18	0	0
33-10	33	34.22	135	0.69	0.46	1.25	0.46	4	10
33-11	33	35.11	295	2.54	1.11	1.55	1.08	0	12
33-12	33	34.55	285	1.18	0.74	1.13	0.75	0	0

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-13	33	33.66	125	0.43	0.28	0.35	0.31	0	0
33-14	33	34.79	205	0.70	0.45	0.77	0.47	0	0
33-15	33	34.47	245	0.89	0.52	0.74	0.52	0	0
33-16	33	34.36	613	5.25	1.75	31.81	1.80	56	62
33-17	33	34.00	213	1.35	0.49	1.82	0.52	16	21
33-18	33	34.03	413	3.21	0.90	3.86	0.90	42	41
33-19	33	34.62	513	6.52	1.55	16.32	1.56	45	52
33-20	33	33.69	113	0.39	0.27	0.51	0.29	0	0
33-21	33	34.51	313	6.73	0.91	40.89	0.88	25	31
33-22	33	35.07	713	12.77	2.47	39.32	2.51	37	37
33-23	33	34.92	737	30.54	3.65	84.81	3.72	26	58
33-24	33	34.11	337	3.46	1.21	8.43	1.27	18	25
33-25	33	33.77	143	0.78	0.44	0.84	0.46	0	1
33-26	33	34.50	543	5.91	2.56	7.90	3.27	1	52
33-27	33	34.73	453	3.85	1.67	11.43	1.59	39	64
33-28	33	33.18	53	0.20	0.14	0.19	0.17	0	0
33-29	33	34.03	247	1.36	0.73	7.45	0.72	16	17
33-30	33	34.65	647	11.03	2.45	31.09	2.31	27	51
33-31	33	34.70	642	5.61	1.73	8.78	1.70	1	2
33-32	33	33.77	242	1.16	0.51	1.01	0.53	20	20
33-33	33	34.04	443	3.35	0.96	8.20	0.99	21	42
33-34	33	34.66	533	5.06	2.46	12.21	2.87	32	49
33-35	33	33.72	133	0.47	0.40	0.58	0.39	0	1
33-36	33	33.73	332	2.32	1.35	3.92	1.22	16	27
33-37	33	35.22	732	10.48	3.35	13.05	3.14	1	33
33-38	33	34.50	703	7.39	1.86	11.92	1.87	39	41
33-39	33	34.45	303	1.63	0.64	1.76	0.64	3	3
33-40	33	34.24	503	3.54	1.12	4.67	1.12	2	3
33-41	33	34.76	603	4.69	1.30	6.33	1.27	1	3

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-42	33	36.00	1,103	10.78	2.62	14.60	2.58	3	4
33-43	33	35.76	1,132	12.68	4.88	47.32	4.69	104	148
33-44	33	35.74	1,137	16.72	4.65	30.08	5.39	137	109
33-45	33	36.14	1,113	13.62	3.46	26.01	3.43	64	66
33-46	33	35.35	445	1.95	0.98	5.00	0.99	40	42
33-47	33	35.60	455	2.59	1.45	3.50	1.43	22	24
33-48	33	35.82	227	0.66	0.54	0.63	0.52	0	2
33-49	33	35.54	419	1.94	0.91	3.31	0.93	40	40
36-0	36	37.50	29	0.12	0.10	0.11	0.13	0	0
36-1	36	38.11	147	0.51	0.41	0.64	0.42	0	0
36-2	36	37.48	67	0.22	0.17	0.21	0.18	0	0
36-3	36	37.32	107	0.58	0.35	0.33	0.33	0	10
36-4	36	38.04	65	0.20	0.16	0.27	0.17	0	0
36-5	36	37.61	259	1.10	0.75	1.17	0.73	0	2
36-6	36	37.12	99	0.37	0.25	0.32	0.26	0	0
36-7	36	37.54	181	0.83	0.42	1.05	0.43	16	16
36-8	36	38.06	217	0.95	1.04	0.89	0.94	0	20
36-9	36	36.57	57	0.19	0.16	0.28	0.17	0	0
36-10	36	37.22	135	1.01	0.48	1.15	0.47	6	10
36-11	36	38.11	295	1.71	1.01	1.91	1.02	0	2
36-12	36	37.55	285	1.29	0.74	1.11	0.73	0	0
36-13	36	36.66	125	0.38	0.29	0.42	0.29	0	0
36-14	36	37.79	205	0.70	0.46	1.31	0.46	0	0
36-15	36	37.47	245	0.83	0.51	1.06	0.51	0	0
36-16	36	37.36	613	31.96	1.85	31.91	1.82	64	67
36-17	36	37.00	213	0.97	0.49	1.75	0.50	5	6
36-18	36	37.03	413	4.18	0.92	4.33	0.91	42	46
36-19	36	37.62	513	3.85	1.61	15.10	1.56	52	60
36-20	36	36.69	113	0.56	0.28	0.87	0.28	10	10

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
36-21	36	37.51	313	3.55	0.91	9.38	0.87	31	34
36-22	36	38.07	713	10.07	2.46	11.45	2.43	1	2
36-23	36	37.92	737	13.83	3.74	51.14	4.85	27	63
36-24	36	37.11	337	4.59	1.46	11.96	1.37	24	34
36-25	36	36.77	143	0.70	0.44	0.81	0.47	0	1
36-26	36	37.50	543	7.91	2.30	17.88	2.25	46	48
36-27	36	37.73	453	33.74	1.43	15.96	1.39	64	66
36-28	36	36.18	53	0.20	0.16	0.26	0.17	0	5
36-29	36	37.03	247	2.44	0.75	10.98	0.73	23	24
36-30	36	37.65	647	11.83	2.53	63.27	2.50	53	76
36-31	36	37.70	642	6.60	1.75	7.63	1.71	1	2
36-32	36	36.77	242	1.04	0.51	1.02	0.52	20	20
36-33	36	37.04	443	3.58	0.95	4.28	0.93	1	2
36-34	36	37.66	533	4.27	1.95	13.45	1.81	55	53
36-35	36	36.72	133	0.47	0.42	0.45	0.40	0	1
36-36	36	36.73	332	1.99	1.51	3.20	1.24	21	28
36-37	36	38.22	732	13.99	3.57	16.20	3.22	21	53
36-38	36	37.50	703	7.82	2.04	12.39	1.86	44	45
36-39	36	37.45	303	1.48	0.67	1.80	0.62	6	6
36-40	36	37.24	503	3.77	2.40	4.98	1.94	1	51
36-41	36	37.76	603	4.54	1.27	5.99	1.26	12	11
36-42	36	39.00	1,103	13.38	2.66	21.48	2.56	107	103
36-43	36	38.76	1,132	15.14	5.43	73.02	5.00	161	172
36-44	36	38.74	1,137	24.60	4.58	35.30	4.59	113	109
36-45	36	39.14	1,113	17.39	3.53	29.86	3.45	63	65
36-46	36	38.35	445	2.24	0.98	3.27	1.02	40	42
36-47	36	38.60	455	2.84	1.60	5.84	1.57	60	66
36-48	36	38.82	227	0.67	0.54	0.90	0.54	0	0
36-49	36	38.54	419	1.77	0.92	3.73	0.93	48	48

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-0	39	40.50	29	0.10	0.10	0.11	0.11	0	0
39-1	39	41.11	147	0.46	0.41	0.61	0.41	0	0
39-2	39	40.48	67	0.20	0.17	0.20	0.19	0	0
39-3	39	40.32	107	0.54	0.29	0.42	0.29	0	10
39-4	39	41.04	65	0.20	0.16	0.19	0.17	0	0
39-5	39	40.61	259	1.36	0.75	1.19	0.75	0	2
39-6	39	40.12	99	0.44	0.25	0.32	0.27	0	0
39-7	39	40.54	181	0.80	0.42	0.78	0.44	18	16
39-8	39	41.06	217	1.17	0.69	2.53	0.69	0	20
39-9	39	39.57	57	0.24	0.18	0.38	0.26	2	4
39-10	39	40.22	135	0.58	0.40	0.90	0.40	0	8
39-11	39	41.11	295	1.68	1.16	2.45	1.09	8	20
39-12	39	40.55	285	1.14	0.76	1.28	0.75	0	0
39-13	39	39.66	125	0.35	0.33	0.51	0.31	0	2
39-14	39	40.79	205	0.77	0.44	0.80	0.46	0	0
39-15	39	40.47	245	0.93	0.49	1.03	0.52	0	0
39-16	39	40.36	613	5.41	1.89	17.89	1.91	68	74
39-17	39	40.00	213	1.26	0.49	1.36	0.51	5	6
39-18	39	40.03	413	2.72	0.92	4.11	0.92	49	51
39-19	39	40.62	513	3.71	1.66	18.86	1.61	61	64
39-20	39	39.69	113	0.38	0.28	0.56	0.29	14	10
39-21	39	40.51	313	2.84	0.90	15.58	0.89	33	35
39-22	39	41.07	713	9.17	2.48	23.75	2.43	44	52
39-23	39	40.92	737	16.50	3.73	112.86	4.85	32	68
39-24	39	40.11	337	2.72	1.80	8.41	1.56	26	37
39-25	39	39.77	143	0.65	0.44	0.63	0.43	0	1
39-26	39	40.50	543	6.04	2.39	14.65	3.13	1	52
39-27	39	40.73	453	3.08	1.50	14.50	1.48	71	69
39-28	39	39.18	53	0.19	0.17	0.25	0.16	0	5

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-29	39	40.03	247	1.80	0.97	12.66	0.86	25	42
39-30	39	40.65	647	15.09	2.73	17.90	3.61	51	81
39-31	39	40.70	642	7.27	1.78	8.81	1.70	1	2
39-32	39	39.77	242	1.14	0.52	1.14	0.52	20	20
39-33	39	40.04	443	3.90	0.96	4.58	0.97	1	2
39-34	39	40.66	533	4.01	2.16	22.82	2.00	54	64
39-35	39	39.72	133	0.56	0.41	0.62	0.39	0	1
39-36	39	39.73	332	2.13	1.48	3.98	1.28	22	20
39-37	39	41.22	732	21.41	3.87	13.76	4.49	21	53
39-38	39	40.50	703	7.28	1.96	8.52	1.83	50	50
39-39	39	40.45	303	1.66	0.62	1.68	0.62	6	6
39-40	39	40.24	503	4.00	1.15	4.18	1.12	1	51
39-41	39	40.76	603	5.09	1.27	6.05	1.24	13	12
39-42	39	42.00	1,103	14.47	2.62	19.49	2.59	105	103
39-43	39	41.76	1,132	28.96	5.40	4,644.48	4.92	170	180
39-44	39	41.74	1,137	25.75	4.65	27.19	4.65	113	119
39-45	39	42.14	1,113	22.84	3.49	27.25	3.49	68	116
39-46	39	41.35	445	2.77	0.98	3.24	0.99	56	42
39-47	39	41.60	455	3.20	1.83	8.40	2.32	61	72
39-48	39	41.82	227	0.82	0.53	1.31	0.53	4	4
39-49	39	41.54	419	2.23	0.95	2.43	0.92	56	48
42-0	42	43.50	29	0.13	0.10	0.11	0.11	0	0
42-1	42	44.11	147	0.64	0.42	0.53	0.42	0	0
42-2	42	43.48	67	0.27	0.17	0.21	0.18	0	0
42-3	42	43.32	107	0.40	0.28	0.38	0.27	10	10
42-4	42	44.04	65	0.21	0.16	0.20	0.15	0	0
42-5	42	43.61	259	1.85	0.83	2.39	0.82	25	34
42-6	42	43.12	99	0.40	0.24	0.32	0.26	0	0
42-7	42	43.54	181	0.72	0.42	0.89	0.42	17	24

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-8	42	44.06	217	1.15	0.68	0.95	0.68	20	20
42-9	42	42.57	57	0.26	0.16	0.36	0.17	4	4
42-10	42	43.22	135	1.09	0.40	1.00	0.41	0	0
42-11	42	44.11	295	2.35	1.03	1.44	1.00	0	2
42-12	42	43.55	285	1.38	0.75	1.07	0.74	0	0
42-13	42	42.66	125	0.46	0.30	0.42	0.31	0	2
42-14	42	43.79	205	0.85	0.46	0.77	0.47	20	20
42-15	42	43.47	245	1.14	0.53	0.75	0.52	0	0
42-16	42	43.36	613	6.49	2.06	21.76	2.00	79	88
42-17	42	43.00	213	1.01	0.57	1.46	0.50	5	11
42-18	42	43.03	413	3.40	0.96	3.64	0.90	51	56
42-19	42	43.62	513	5.21	1.72	16.04	1.65	64	70
42-20	42	42.69	113	0.45	0.30	0.62	0.28	10	15
42-21	42	43.51	313	2.77	0.92	10.30	0.90	39	41
42-22	42	44.07	713	10.61	2.46	20.96	2.39	1	3
42-23	42	43.92	737	20.30	3.80	105.06	4.97	38	73
42-24	42	43.11	337	3.17	1.82	13.51	1.59	30	42
42-25	42	42.77	143	0.64	0.45	0.76	0.45	0	11
42-26	42	43.50	543	9.35	3.03	29.29	2.80	49	74
42-27	42	43.73	453	3.16	1.52	12.36	1.49	71	75
42-28	42	42.18	53	0.19	0.15	0.19	0.16	5	5
42-29	42	43.03	247	1.74	0.99	11.27	0.93	26	44
42-30	42	43.65	647	13.01	2.91	17.22	3.70	65	86
42-31	42	43.70	642	5.94	1.76	7.78	1.74	1	2
42-32	42	42.77	242	0.86	0.82	0.99	0.67	23	39
42-33	42	43.04	443	3.16	1.18	4.29	1.13	2	22
42-34	42	43.66	533	3.60	2.14	13.86	2.02	60	68
42-35	42	42.72	133	0.63	0.34	0.53	0.37	0	1
42-36	42	42.73	332	1.92	1.47	4.02	1.31	26	41

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-37	42	44.22	732	12.72	3.95	65.31	4.57	21	53
42-38	42	43.50	703	5.64	1.88	8.74	1.83	56	56
42-39	42	43.45	303	1.59	0.64	1.87	0.65	6	6
42-40	42	43.24	503	3.58	1.15	4.27	1.10	51	51
42-41	42	43.76	603	4.78	1.35	5.67	1.25	14	15
42-42	42	45.00	1,103	13.06	2.67	20.56	2.59	106	104
42-43	42	44.76	1,132	12.92	5.45	65.10	4.97	190	188
42-44	42	44.74	1,137	19.84	4.74	27.31	4.66	115	159
42-45	42	45.14	1,113	16.96	3.53	23.91	3.53	120	116
42-46	42	44.35	445	2.14	0.96	3.59	0.96	8	10
42-47	42	44.60	455	3.26	1.84	8.34	2.34	75	74
42-48	42	44.82	227	0.77	0.54	1.46	0.55	4	6
42-49	42	44.54	419	2.11	0.92	3.41	0.91	56	56
45-0	45	46.50	29	0.12	0.10	0.10	0.11	0	0
45-1	45	47.11	147	0.59	0.42	0.51	0.40	0	0
45-2	45	46.48	67	0.26	0.17	0.28	0.17	0	0
45-3	45	46.32	107	0.42	0.29	0.30	0.29	10	10
45-4	45	47.04	65	0.25	0.16	0.19	0.16	0	0
45-5	45	46.61	259	1.14	1.02	2.73	0.96	29	42
45-6	45	46.12	99	0.42	0.23	0.33	0.24	0	0
45-7	45	46.53	181	1.14	0.42	1.40	0.43	18	24
45-8	45	47.06	217	1.29	0.69	1.01	0.68	20	20
45-9	45	45.57	57	0.26	0.16	0.45	0.17	6	4
45-10	45	46.22	135	0.79	0.54	0.95	0.67	8	16
45-11	45	47.11	295	2.37	1.00	2.62	0.99	8	24
45-12	45	46.55	285	1.91	0.75	1.42	0.73	0	0
45-13	45	45.66	125	0.46	0.32	0.50	0.30	2	4
45-14	45	46.79	205	0.84	0.46	1.35	0.47	0	2
45-15	45	46.47	245	1.02	0.50	0.88	0.53	0	0

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-16	45	46.36	613	40.08	2.14	37.24	2.05	85	92
45-17	45	46.00	213	0.98	0.49	1.38	0.50	10	11
45-18	45	46.03	413	2.22	0.93	4.70	0.91	72	72
45-19	45	46.62	513	4.42	1.78	15.01	1.71	74	78
45-20	45	45.69	113	1.54	0.34	1.09	0.30	10	15
45-21	45	46.51	313	2.20	1.02	9.77	0.93	37	44
45-22	45	47.07	713	11.77	2.52	21.88	2.45	56	67
45-23	45	46.92	737	19.19	4.61	110.87	4.28	61	98
45-24	45	46.11	337	3.29	1.84	19.73	1.63	37	44
45-25	45	45.77	143	0.70	0.44	0.82	0.45	10	11
45-26	45	46.50	543	7.48	2.05	8.52	1.98	51	52
45-27	45	46.73	453	3.10	1.76	13.45	1.72	80	83
45-28	45	45.18	53	0.22	0.15	0.19	0.17	5	5
45-29	45	46.03	247	2.15	1.14	15.71	1.03	28	50
45-30	45	46.65	647	11.59	4.14	28.27	3.69	63	106
45-31	45	46.70	642	7.26	1.74	7.35	1.71	2	2
45-32	45	45.77	242	1.00	0.76	1.24	0.69	25	41
45-33	45	46.04	443	3.10	0.98	4.91	0.95	11	12
45-34	45	46.66	533	3.82	2.20	11.08	2.05	63	74
45-35	45	45.72	133	0.58	0.34	0.45	0.34	0	1
45-36	45	45.73	332	1.82	1.46	4.13	1.30	28	44
45-37	45	47.22	732	14.00	2.59	15.10	2.60	21	64
45-38	45	46.50	703	6.49	1.89	8.54	1.85	61	63
45-39	45	46.45	303	1.57	0.64	1.88	0.64	7	8
45-40	45	46.24	503	2.66	2.58	4.13	2.08	56	95
45-41	45	46.76	603	4.64	1.26	5.68	1.28	16	17
45-42	45	48.00	1,103	13.57	2.74	41.66	2.57	112	111
45-43	45	47.76	1,132	14.09	6.23	48.87	5.52	194	205
45-44	45	47.74	1,137	23.42	4.74	29.33	4.69	180	159

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-45	45	48.14	1,113	16.09	3.47	23.70	3.43	114	117
45-46	45	47.35	445	1.96	1.05	4.29	1.03	14	18
45-47	45	47.60	455	2.38	2.09	7.22	1.94	76	80
45-48	45	47.82	227	0.73	0.56	1.05	0.56	4	8
45-49	45	47.54	419	2.18	0.94	2.88	0.92	56	64
48-0	48	49.50	29	0.11	0.10	0.15	0.11	0	0
48-1	48	50.11	147	0.63	0.41	0.62	0.40	0	0
48-2	48	49.48	67	0.24	0.17	0.19	0.18	0	0
48-3	48	49.32	107	0.37	0.30	0.36	0.28	10	10
48-4	48	50.04	65	0.25	0.16	0.24	0.17	0	0
48-5	48	49.61	259	1.01	1.08	3.02	1.02	28	46
48-6	48	49.12	99	0.37	0.26	0.57	0.26	11	8
48-7	48	49.53	181	0.71	0.42	1.06	0.43	26	24
48-8	48	50.06	217	1.01	0.69	1.21	0.69	20	20
48-9	48	48.57	57	0.61	0.16	0.62	0.17	4	6
48-10	48	49.22	135	0.62	0.56	1.37	0.52	10	18
48-11	48	50.11	295	2.69	1.06	3.79	1.01	28	40
48-12	48	49.55	285	1.51	0.76	1.45	0.74	0	0
48-13	48	48.66	125	0.40	0.30	0.68	0.31	2	4
48-14	48	49.79	205	0.64	0.67	1.71	0.61	26	40
48-15	48	49.47	245	0.95	0.51	0.76	0.50	0	0
48-16	48	49.36	613	14.72	2.21	1,536.15	2.21	95	97
48-17	48	49.00	213	1.02	0.49	1.40	0.50	10	11
48-18	48	49.03	413	3.14	0.93	4.76	0.92	27	29
48-19	48	49.62	513	5.30	2.79	9.83	2.41	66	101
48-20	48	48.69	113	0.65	0.28	0.91	0.29	16	15
48-21	48	49.51	313	3.55	0.99	10.51	0.95	45	51
48-22	48	50.07	713	11.53	2.70	12.74	2.54	51	53
48-23	48	49.92	737	20.14	4.65	79.18	5.60	66	98

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
48-24	48	49.11	337	3.81	1.93	15.21	1.71	36	47
48-25	48	48.77	143	0.67	0.45	0.82	0.45	10	11
48-26	48	49.50	543	13.17	3.14	31.75	2.83	79	84
48-27	48	49.73	453	3.14	1.48	13.65	1.47	84	90
48-28	48	48.18	53	0.21	0.15	0.25	0.16	0	0
48-29	48	49.03	247	1.53	1.23	4.14	1.08	31	51
48-30	48	49.65	647	17.59	4.29	49.49	3.70	94	111
48-31	48	49.70	642	7.62	1.77	8.99	1.70	5	4
48-32	48	48.77	242	1.05	0.74	1.20	0.68	27	44
48-33	48	49.04	443	2.87	0.97	4.09	0.96	16	17
48-34	48	49.66	533	5.51	1.86	6.65	1.79	53	54
48-35	48	48.72	133	0.65	0.35	0.43	0.35	0	1
48-36	48	48.73	332	1.79	1.53	4.19	1.33	31	47
48-37	48	50.22	732	13.42	2.59	48.74	2.57	31	54
48-38	48	49.50	703	6.96	1.96	12.01	1.94	113	60
48-39	48	49.45	303	1.54	0.64	1.72	0.62	10	11
48-40	48	49.24	503	4.54	1.10	4.55	1.13	51	51
48-41	48	49.76	603	6.16	1.26	6.49	1.26	22	23
48-42	48	51.00	1,103	12.79	2.63	22.80	2.58	104	104
48-43	48	50.76	1,132	16.88	4.73	19.21	4.33	181	157
48-44	48	50.74	1,137	24.12	4.66	22.36	4.64	158	159
48-45	48	51.14	1,113	16.71	3.52	31.25	3.47	124	109
48-46	48	50.35	445	2.18	0.99	3.19	0.96	48	42
48-47	48	50.60	455	2.42	1.52	7.05	1.48	79	86
48-48	48	50.82	227	0.79	0.56	1.39	0.55	9	18
48-49	48	50.54	419	1.80	0.93	2.83	0.94	80	64
51-0	51	52.50	29	0.13	0.11	0.14	0.11	2	2
51-1	51	53.11	147	0.56	0.42	0.61	0.42	0	0
51-2	51	52.48	67	0.24	0.17	0.29	0.17	0	0

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-3	51	52.32	107	0.38	0.29	0.41	0.28	14	10
51-4	51	53.04	65	0.21	0.21	0.19	0.20	10	10
51-5	51	52.61	259	2.25	1.06	4.04	1.00	39	46
51-6	51	52.12	99	0.41	0.25	0.73	0.25	11	16
51-7	51	52.53	181	0.72	0.42	0.76	0.43	28	24
51-8	51	53.06	217	1.06	0.70	1.09	0.70	20	22
51-9	51	51.57	57	0.24	0.16	0.36	0.17	6	6
51-10	51	52.22	135	2.41	0.42	2.89	0.40	11	18
51-11	51	53.11	295	1.75	1.06	3.39	1.03	39	40
51-12	51	52.55	285	1.18	0.86	1.17	0.82	40	40
51-13	51	51.66	125	0.41	0.30	0.69	0.29	6	6
51-14	51	52.79	205	0.72	0.67	1.83	0.61	38	42
51-15	51	52.47	245	0.96	1.35	0.88	1.09	40	40
51-16	51	52.36	613	5.27	1.81	13.67	1.71	102	105
51-17	51	52.00	213	0.89	0.49	1.38	0.50	15	16
51-18	51	52.03	413	2.61	0.92	5.58	0.91	34	39
51-19	51	52.62	513	5.45	2.84	10.26	2.38	63	106
51-20	51	51.69	113	0.43	0.29	3.01	0.29	25	22
51-21	51	52.51	313	9.43	0.87	5.65	0.87	49	57
51-22	51	53.07	713	19.38	2.61	30.72	2.50	118	112
51-23	51	52.92	737	19.81	5.10	90.66	4.64	77	103
51-24	51	52.11	337	15.71	2.38	14.55	1.98	40	63
51-25	51	51.77	143	0.71	0.45	0.90	0.44	10	11
51-26	51	52.50	543	7.68	2.05	8.27	2.00	51	53
51-27	51	52.73	453	3.30	1.50	11.88	1.46	89	93
51-28	51	51.18	53	0.24	0.15	0.29	0.16	8	5
51-29	51	52.03	247	2.29	1.03	5.36	0.98	36	52
51-30	51	52.65	647	16.06	4.31	39.15	3.73	93	116
51-31	51	52.70	642	7.82	7.50	7.27	5.20	102	102

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-32	51	51.77	242	1.22	0.53	1.82	0.52	22	40
51-33	51	52.04	443	3.22	0.96	4.22	0.96	51	42
51-34	51	52.66	533	6.01	1.83	7.25	1.76	57	54
51-35	51	51.72	133	0.64	0.34	0.43	0.36	0	1
51-36	51	51.73	332	2.16	0.89	8.63	0.87	33	50
51-37	51	53.22	732	15.66	2.90	41.14	3.96	94	103
51-38	51	52.50	703	9.49	3.23	16.04	2.83	115	116
51-39	51	52.45	303	1.33	0.67	1.95	0.64	15	15
51-40	51	52.24	503	4.11	1.14	4.88	1.14	53	51
51-41	51	52.76	603	5.13	1.27	5.74	1.25	31	32
51-42	51	54.00	1,103	15.31	2.64	20.24	2.60	106	154
51-43	51	53.76	1,132	18.44	5.33	17.83	4.80	172	168
51-44	51	53.74	1,137	20.66	4.81	29.07	4.76	161	170
51-45	51	54.14	1,113	17.94	3.49	39.78	3.44	230	233
51-46	51	53.35	445	2.54	0.98	3.28	0.98	44	62
51-47	51	53.60	455	2.65	1.48	4.14	1.48	61	60
51-48	51	53.82	227	0.81	0.57	1.49	0.57	23	22
51-49	51	53.54	419	2.03	0.92	2.90	0.93	85	72
54-0	54	55.50	29	0.12	0.11	0.11	0.12	4	4
54-1	54	56.11	147	0.59	0.40	0.43	0.42	0	0
54-2	54	55.48	67	0.23	0.17	0.32	0.19	0	0
54-3	54	55.32	107	0.36	0.28	0.40	0.30	10	10
54-4	54	56.04	65	0.23	0.16	0.37	0.16	9	8
54-5	54	55.61	259	2.30	1.08	3.36	1.02	42	48
54-6	54	55.12	99	0.33	0.25	0.50	0.25	12	16
54-7	54	55.54	181	0.67	0.46	1.41	0.45	21	24
54-8	54	56.06	217	0.89	0.75	1.42	0.69	24	22
54-9	54	54.57	57	0.28	0.15	0.51	0.16	6	6
54-10	54	55.22	135	0.81	0.45	1.73	0.42	13	20

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
54-11	54	56.11	295	1.77	1.11	1.75	1.06	40	42
54-12	54	55.55	285	1.35	0.73	1.26	0.74	0	0
54-13	54	54.66	125	0.43	0.29	0.55	0.31	0	0
54-14	54	55.79	205	0.69	0.67	1.84	0.60	31	44
54-15	54	55.47	245	0.77	1.19	0.98	0.96	40	40
54-16	54	55.36	613	5.27	1.70	14.44	1.75	114	119
54-17	54	55.00	213	1.00	0.51	1.74	0.51	22	23
54-18	54	55.03	413	2.74	1.00	5.93	0.98	44	50
54-19	54	55.62	513	5.92	2.84	9.35	2.38	78	111
54-20	54	54.69	113	0.42	0.30	1.28	0.28	28	22
54-21	54	55.51	313	1.67	0.87	6.63	0.89	56	61
54-22	54	56.07	713	11.27	2.75	10.75	2.62	102	103
54-23	54	55.92	737	27.96	3.55	99.14	3.29	78	113
54-24	54	55.11	337	9.78	2.18	36.67	1.85	47	64
54-25	54	54.77	143	0.74	0.45	0.82	0.46	10	11
54-26	54	55.50	543	11.37	4.83	17.80	4.80	94	127
54-27	54	55.73	453	4.04	1.54	13.32	1.52	104	112
54-28	54	54.18	53	0.25	0.17	0.25	0.17	0	5
54-29	54	55.03	247	1.45	1.20	3.59	1.09	40	58
54-30	54	55.65	647	26.61	4.41	86.43	5.00	98	121
54-31	54	55.70	642	8.49	1.75	11.44	1.73	3	4
54-32	54	54.77	242	1.12	0.53	1.21	0.51	15	16
54-33	54	55.04	443	3.53	0.96	4.19	0.96	46	42
54-34	54	55.66	533	4.99	3.10	11.82	2.69	87	125
54-35	54	54.72	133	0.56	0.33	0.43	0.34	0	1
54-36	54	54.73	332	3.10	0.87	6.68	0.87	37	52
54-37	54	56.22	732	16.69	2.91	12.05	2.81	51	54
54-38	54	55.50	703	5.64	3.23	10.72	2.76	122	122
54-39	54	55.45	303	1.32	0.65	2.29	0.65	25	21

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
54-40	54	55.24	503	3.69	1.10	4.95	1.11	56	51
54-41	54	55.76	603	4.53	1.24	5.47	1.26	43	42
54-42	54	57.00	1,103	13.49	2.64	25.95	2.63	187	154
54-43	54	56.76	1,132	18.50	5.08	28.81	4.72	170	207
54-44	54	56.74	1,137	26.00	4.83	24.26	4.69	164	209
54-45	54	57.14	1,113	17.71	3.50	27.63	3.55	177	159
54-46	54	56.35	445	2.06	1.00	3.38	1.00	72	62
54-47	54	56.60	455	2.60	1.59	3.11	1.53	61	64
54-48	54	56.82	227	0.76	0.52	1.46	0.51	27	28
54-49	54	56.54	419	1.96	0.94	2.33	0.91	100	80
57-0	57	58.50	29	0.11	0.11	0.11	0.11	4	4
57-1	57	59.11	147	0.60	0.46	0.41	0.46	20	20
57-2	57	58.48	67	0.24	0.17	0.28	0.19	0	0
57-3	57	58.32	107	0.38	0.30	0.40	0.30	14	12
57-4	57	59.04	65	0.32	0.17	0.36	0.17	9	8
57-5	57	58.61	259	2.44	0.72	3.72	0.74	46	52
57-6	57	58.12	99	0.56	0.25	0.28	0.24	16	16
57-7	57	58.54	181	0.57	0.50	1.40	0.47	24	26
57-8	57	59.06	217	0.82	0.74	1.07	0.70	24	24
57-9	57	57.57	57	0.21	0.15	0.35	0.17	6	6
57-10	57	58.22	135	0.73	0.38	1.33	0.38	14	22
57-11	57	59.11	295	5.35	1.07	4.14	1.05	43	50
57-12	57	58.55	285	1.13	0.83	1.38	0.81	40	40
57-13	57	57.66	125	0.42	0.29	0.57	0.29	0	0
57-14	57	58.79	205	0.57	0.68	1.71	0.62	34	46
57-15	57	58.47	245	0.79	1.32	0.96	1.05	40	40
57-16	57	58.36	613	6.13	1.77	27.50	1.74	121	124
57-17	57	58.00	213	0.89	0.52	1.85	0.52	29	30
57-18	57	58.03	413	3.90	1.06	5.06	1.05	60	60

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-19	57	58.62	513	4.52	2.81	8.62	2.40	94	116
57-20	57	57.69	113	0.36	0.28	1.14	0.29	25	23
57-21	57	58.51	313	1.87	0.97	1.87	0.91	50	51
57-22	57	59.07	713	14.12	2.61	29.45	2.56	128	132
57-23	57	58.92	737	55.55	3.51	84.37	3.43	85	134
57-24	57	58.11	337	16.02	2.19	19.08	2.84	48	65
57-25	57	57.77	143	0.68	0.45	0.76	0.44	10	11
57-26	57	58.50	543	10.96	4.24	14.38	4.30	99	132
57-27	57	58.73	453	3.76	1.59	10.02	1.56	103	115
57-28	57	57.18	53	0.24	0.15	0.22	0.16	5	5
57-29	57	58.03	247	1.14	1.16	5.58	1.06	44	60
57-30	57	58.65	647	39.80	2.30	41.77	2.24	116	131
57-31	57	58.70	642	9.38	1.78	11.37	1.73	81	82
57-32	57	57.77	242	1.16	0.53	0.95	0.52	40	40
57-33	57	58.04	443	3.01	0.96	3.85	0.97	81	62
57-34	57	58.66	533	6.27	3.41	9.31	2.82	99	129
57-35	57	57.72	133	0.62	0.34	0.67	0.33	0	1
57-36	57	57.73	332	2.44	0.88	6.12	0.87	46	55
57-37	57	59.22	732	30.05	2.70	26.38	2.62	112	127
57-38	57	58.50	703	9.21	3.27	16.72	2.82	128	130
57-39	57	58.45	303	1.30	0.65	2.81	0.64	25	26
57-40	57	58.24	503	2.96	1.13	5.61	1.14	54	50
57-41	57	58.76	603	4.28	1.27	5.74	1.26	57	53
57-42	57	60.00	1,103	12.62	2.73	20.61	2.63	179	154
57-43	57	59.76	1,132	16.91	5.08	18.26	4.74	208	208
57-44	57	59.74	1,137	21.45	4.76	25.10	4.69	217	210
57-45	57	60.14	1,113	18.20	3.59	312.08	3.43	157	199
57-46	57	59.35	445	1.93	1.00	3.60	1.01	65	82
57-47	57	59.60	455	2.37	1.55	3.34	1.53	84	80

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-48	57	59.82	227	0.67	0.52	1.41	0.52	35	36
57-49	57	59.54	419	1.63	0.93	1.91	0.93	101	88
60-0	60	61.50	29	0.12	0.11	0.10	0.11	4	4
60-1	60	62.11	147	0.52	0.46	0.41	0.44	20	20
60-2	60	61.48	67	0.25	0.17	0.31	0.18	0	0
60-3	60	61.32	107	0.31	0.31	0.41	0.31	28	20
60-4	60	62.04	65	0.21	0.17	0.30	0.17	10	12
60-5	60	61.61	259	1.43	0.71	2.56	0.70	48	52
60-6	60	61.12	99	0.44	0.26	0.27	0.27	16	16
60-7	60	61.53	181	0.52	0.42	1.45	0.44	34	34
60-8	60	62.06	217	1.11	0.69	0.90	0.69	24	20
60-9	60	60.57	57	0.20	0.15	0.48	0.16	8	10
60-10	60	61.22	135	0.54	0.50	0.61	0.48	20	20
60-11	60	62.11	295	1.59	1.09	1.54	1.05	40	42
60-12	60	61.55	285	1.45	0.98	1.87	0.91	60	40
60-13	60	60.66	125	0.44	0.29	0.42	0.31	0	0
60-14	60	61.79	205	0.60	0.46	0.89	0.47	52	48
60-15	60	61.47	245	0.81	1.18	0.97	0.96	40	40
60-16	60	61.36	613	4.51	1.73	13.55	1.73	135	134
60-17	60	61.00	213	0.94	0.52	1.37	0.49	33	33
60-18	60	61.03	413	3.52	0.94	4.30	0.94	70	70
60-19	60	61.62	513	4.00	1.47	7.46	1.48	135	116
60-20	60	60.69	113	0.38	0.29	0.85	0.30	28	26
60-21	60	61.51	313	1.88	0.97	2.16	0.91	50	51
60-22	60	62.07	713	11.64	2.67	27.32	2.58	135	143
60-23	60	61.92	737	19.05	3.37	61.53	4.64	101	139
60-24	60	61.11	337	2.53	1.51	2.95	1.41	50	51
60-25	60	60.77	143	1.18	0.13	1.48	0.14	21	NA
60-26	60	61.50	543	5.43	2.12	9.90	2.07	70	64

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
60-27	60	61.73	453	2.75	1.78	10.86	1.56	111	123
60-28	60	60.18	53	0.18	0.16	0.25	0.16	8	5
60-29	60	61.03	247	1.00	1.28	3.54	1.08	50	62
60-30	60	61.65	647	9.99	2.31	23.07	2.24	126	132
60-31	60	61.70	642	6.33	1.79	13.05	1.76	99	122
60-32	60	60.77	242	1.04	0.52	0.99	0.53	50	40
60-33	60	61.04	443	2.58	0.98	3.88	0.98	72	62
60-34	60	61.66	533	4.86	3.41	9.07	2.90	100	134
60-35	60	60.72	133	0.47	0.35	0.59	0.34	10	11
60-36	60	60.73	332	1.63	0.89	6.68	0.89	45	60
60-37	60	62.22	732	12.32	2.98	16.15	4.03	110	124
60-38	60	61.50	703	4.55	3.30	17.83	2.90	143	138
60-39	60	61.45	303	1.17	0.64	2.01	0.66	30	31
60-40	60	61.24	503	2.61	1.16	6.33	1.16	63	48
60-41	60	61.76	603	3.76	1.27	4.74	1.25	70	65
60-42	60	63.00	1,103	12.44	2.67	11.62	2.57	326	204
60-43	60	62.76	1,132	14.21	5.69	19.44	5.27	219	218
60-44	60	62.74	1,137	18.23	4.83	24.84	4.73	228	220
60-45	60	63.14	1,113	16.53	3.58	27.68	3.54	272	308
60-46	60	62.35	445	1.86	0.98	2.23	1.00	106	82
60-47	60	62.60	455	2.86	1.76	2.88	1.60	95	84
60-48	60	62.82	227	0.72	0.51	1.75	0.51	41	42
60-49	60	62.54	419	1.93	0.92	2.35	0.97	110	88
63-0	63	64.50	29	0.11	0.12	0.16	0.12	4	4
63-1	63	65.11	147	0.77	0.44	0.51	0.45	20	20
63-2	63	64.48	67	0.26	0.17	0.25	0.18	0	0
63-3	63	64.32	107	0.44	0.27	0.40	0.28	28	20
63-4	63	65.04	65	0.23	0.16	0.39	0.17	14	12
63-5	63	64.61	259	0.90	0.72	2.71	0.70	56	58

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-6	63	64.12	99	0.39	0.26	0.27	0.26	16	16
63-7	63	64.53	181	0.60	0.42	1.03	0.43	40	36
63-8	63	65.06	217	0.88	0.79	1.02	0.72	45	42
63-9	63	63.57	57	0.22	0.20	0.23	0.17	10	12
63-10	63	64.22	135	0.51	0.50	0.54	0.47	20	20
63-11	63	65.11	295	3.36	1.08	4.05	1.06	46	58
63-12	63	64.55	285	1.37	0.83	1.70	0.80	40	40
63-13	63	63.66	125	0.44	0.29	0.36	0.30	0	0
63-14	63	64.79	205	0.62	0.46	0.97	0.48	56	52
63-15	63	64.47	245	0.87	1.31	0.97	1.02	40	40
63-16	63	64.36	613	4.24	1.80	16.22	1.80	146	148
63-17	63	64.00	213	0.89	0.49	1.59	0.50	40	40
63-18	63	64.03	413	2.44	0.91	4.02	0.92	80	81
63-19	63	64.62	513	4.67	1.46	5.81	1.47	145	126
63-20	63	63.69	113	0.46	0.29	2.17	0.29	27	30
63-21	63	64.51	313	2.03	0.95	2.20	0.92	50	51
63-22	63	65.07	713	13.26	2.63	15.50	2.55	146	143
63-23	63	64.92	737	9.28	4.75	12.79	4.11	155	157
63-24	63	64.11	337	2.74	1.50	3.17	1.42	50	51
63-25	63	63.77	143	0.86	0.13	1.07	0.13	23	NA
63-26	63	64.50	543	6.57	2.09	7.15	2.04	109	103
63-27	63	64.73	453	3.33	1.79	15.20	1.69	113	131
63-28	63	63.18	53	0.25	0.14	0.22	0.16	8	5
63-29	63	64.03	247	1.17	1.17	3.68	1.04	51	64
63-30	63	64.65	647	12.03	2.47	12.28	2.26	129	142
63-31	63	64.70	642	6.33	1.97	10.02	1.75	97	122
63-32	63	63.77	242	1.02	0.54	0.79	0.54	50	40
63-33	63	64.04	443	2.37	1.00	3.73	1.07	82	82
63-34	63	64.66	533	4.92	2.28	5.20	2.19	104	105

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-35	63	63.72	133	0.52	0.34	0.66	0.34	10	11
63-36	63	63.73	332	2.32	0.89	3.76	0.90	51	63
63-37	63	65.22	732	10.76	2.96	17.66	3.94	140	144
63-38	63	64.50	703	3.61	3.25	12.23	2.87	148	146
63-39	63	64.45	303	1.13	0.66	2.41	0.65	37	36
63-40	63	64.24	503	2.41	2.60	4.70	2.15	71	106
63-41	63	64.76	603	2.96	1.28	3.86	1.29	105	77
63-43	63	65.76	1,132	14.84	3.80	40.55	3.61	255	258
63-44	63	65.74	1,137	13.06	4.58	21.47	4.56	270	259
63-45	63	66.14	1,113	13.75	3.66	25.23	3.40	298	318
63-46	63	65.35	445	1.71	1.08	1.87	0.99	130	102
63-47	63	65.60	455	2.72	1.40	4.79	1.33	91	92
63-48	63	65.82	227	0.70	0.52	1.18	0.51	46	46
63-49	63	65.54	419	1.34	0.96	1.63	0.94	122	98
66-1	66	68.11	147	0.46	0.71	0.55	0.61	32	30
66-2	66	67.48	67	0.21	0.25	0.26	0.24	10	10
66-4	66	68.04	65	0.19	0.16	0.22	0.16	12	12
66-5	66	67.61	259	0.85	0.71	2.21	0.70	67	58
66-6	66	67.12	99	0.33	0.24	0.30	0.24	20	16
66-7	66	67.54	181	0.52	0.42	1.00	0.42	40	38
66-9	66	66.57	57	0.19	0.15	0.29	0.16	14	12
66-11	66	68.11	295	1.58	1.12	2.05	1.05	40	42
66-12	66	67.55	285	1.21	0.85	1.16	0.82	40	40
66-13	66	66.66	125	0.46	0.45	0.47	0.41	20	20
66-14	66	67.79	205	0.65	0.46	0.89	0.46	30	22
66-16	66	67.36	613	4.02	1.74	123.04	1.76	157	156
66-17	66	67.00	213	0.86	0.48	1.40	0.49	44	43
66-18	66	67.03	413	1.87	0.94	4.86	0.92	96	91
66-19	66	67.62	513	3.74	1.49	7.57	1.49	149	131

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
66-20	66	66.69	113	1.18	0.29	1.39	0.32	32	31
66-22	66	68.07	713	10.18	2.66	18.23	2.59	172	158
66-23	66	67.92	737	8.72	4.85	14.90	4.19	157	158
66-25	66	66.77	143	0.61	0.13	0.66	0.15	24	NA
66-26	66	67.50	543	5.48	2.02	8.56	1.95	113	103
66-27	66	67.73	453	2.93	1.34	8.82	1.34	148	135
66-28	66	66.18	53	0.19	0.15	0.21	0.15	8	5
66-29	66	67.03	247	1.68	0.72	1.91	0.73	73	70
66-30	66	67.65	647	7.62	2.30	13.05	2.30	169	147
66-32	66	66.77	242	1.12	0.51	0.92	0.53	40	40
66-33	66	67.04	443	2.50	0.98	3.23	0.96	109	82
66-34	66	67.66	533	3.61	3.10	8.52	2.62	124	157
66-35	66	66.72	133	0.57	0.09	0.50	0.10	29	NA
66-36	66	66.73	332	3.43	0.89	8.67	0.87	82	74
66-37	66	68.22	732	10.37	3.12	19.64	3.94	146	164
66-38	66	67.50	703	3.05	3.28	10.21	2.82	167	157
66-39	66	67.45	303	1.03	0.64	1.63	0.64	52	43
66-40	66	67.24	503	2.04	1.13	5.43	1.10	96	114
66-41	66	67.76	603	2.86	1.29	4.22	1.29	109	90
66-43	66	68.76	1,132	13.03	3.71	41.42	3.67	265	259
66-45	66	69.14	1,113	13.52	3.56	44.30	3.47	280	254
66-47	66	68.60	455	2.42	1.35	4.47	1.32	100	100
66-48	66	68.82	227	0.54	0.50	1.34	0.52	50	50
69-4	69	71.04	65	0.23	0.16	0.17	0.18	14	12
69-5	69	70.61	259	1.06	0.73	2.04	0.74	89	60
69-6	69	70.12	99	0.33	0.24	0.37	0.25	24	16
69-7	69	70.54	181	0.58	0.41	0.89	0.42	40	42
69-9	69	69.57	57	0.18	0.16	0.41	0.16	12	14
69-11	69	71.11	295	1.47	1.09	3.69	1.05	69	66

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
69-13	69	69.66	125	0.37	0.45	0.40	0.39	20	20
69-16	69	70.36	613	3.52	1.78	9.48	1.75	179	167
69-17	69	70.00	213	0.72	0.50	1.18	0.49	55	50
69-18	69	70.03	413	1.83	0.97	3.45	0.93	105	102
69-19	69	70.62	513	3.38	1.51	5.54	1.46	168	136
69-20	69	69.69	113	0.72	0.29	2.42	0.29	39	32
69-22	69	71.07	713	7.32	2.79	12.30	2.65	191	173
69-23	69	70.92	737	9.67	3.82	20.82	4.94	102	118
69-27	69	70.73	453	2.46	1.37	7.45	1.36	148	143
69-28	69	69.18	53	0.20	0.15	0.21	0.15	15	10
69-29	69	70.03	247	0.98	0.71	2.12	0.69	76	72
69-30	69	70.65	647	6.82	2.35	11.39	2.27	199	167
69-31	69	70.70	642	5.08	1.80	8.51	1.79	127	122
69-32	69	69.77	242	0.86	0.52	0.91	0.53	60	40
69-34	69	70.66	533	2.71	3.44	10.39	2.78	148	163
69-36	69	69.73	332	3.44	0.90	6.20	0.87	73	78
69-37	69	71.22	732	9.91	4.28	21.44	3.75	174	170
69-38	69	70.50	703	3.51	3.26	8.70	2.82	177	166
69-39	69	70.45	303	1.00	0.64	2.48	0.65	49	50
69-40	69	70.24	503	2.20	1.16	2.45	1.18	129	123
69-41	69	70.76	603	2.56	1.30	5.62	1.27	114	104
69-43	69	71.76	1,132	16.66	1.19	33.28	1.21	285	NA
69-45	69	72.14	1,113	15.09	3.73	32.09	3.54	305	319
69-47	69	71.60	455	2.45	0.46	4.64	0.48	120	NA
69-48	69	71.82	227	0.73	0.15	0.83	0.16	68	NA
72-4	72	74.04	65	0.16	0.17	0.23	0.18	26	8
72-5	72	73.61	259	0.83	0.73	2.22	0.71	92	74
72-7	72	73.54	181	0.49	0.42	0.56	0.43	51	52
72-9	72	72.57	57	0.17	0.16	0.19	0.16	18	14

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
72-10	72	73.22	135	0.89	0.39	0.99	0.40	27	36
72-16	72	73.36	613	2.91	1.75	10.44	1.75	186	181
72-17	72	73.00	213	0.75	0.48	1.38	0.49	56	55
72-18	72	73.03	413	1.66	0.93	3.45	0.94	118	113
72-19	72	73.62	513	3.27	1.51	5.24	1.49	160	151
72-20	72	72.69	113	0.30	0.29	0.83	0.30	39	33
72-22	72	74.07	713	6.52	2.67	13.68	2.58	180	173
72-23	72	73.92	737	10.84	1.13	16.96	1.19	208	NA
72-27	72	73.73	453	1.95	1.36	4.64	1.32	160	147
72-29	72	73.03	247	2.30	0.73	6.62	0.73	85	75
72-30	72	73.65	647	6.64	2.38	10.50	2.29	185	182
72-31	72	73.70	642	4.65	1.93	8.63	1.86	126	124
72-34	72	73.66	533	2.35	3.52	6.18	2.86	150	168
72-36	72	72.73	332	2.07	0.89	4.54	0.90	78	83
72-38	72	73.50	703	2.43	3.27	5.34	2.84	193	177
72-39	72	73.45	303	0.74	0.64	2.37	0.66	64	57
72-40	72	73.24	503	2.00	1.17	2.55	1.15	142	133
72-41	72	73.76	603	2.11	1.32	4.59	1.27	126	119
72-43	72	74.76	1,132	11.55	1.20	21.37	1.23	324	NA
72-48	72	74.82	227	0.55	0.15	0.88	0.15	78	NA
72-49	72	74.54	419	0.96	0.96	1.35	0.98	155	134
75-4	75	77.04	65	0.17	0.22	0.17	0.21	26	20
75-5	75	76.61	259	1.25	0.70	1.79	0.71	78	76
75-7	75	76.54	181	0.61	0.43	0.51	0.44	66	54
75-10	75	76.22	135	0.58	0.39	1.21	0.39	46	40
75-11	75	77.11	295	1.37	0.35	2.95	0.34	72	NA
75-16	75	76.36	613	2.27	1.75	6.74	1.71	212	189
75-17	75	76.00	213	0.57	0.49	0.85	0.48	70	62
75-18	75	76.03	413	1.14	0.92	1.63	0.91	153	123

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
75-20	75	75.69	113	0.40	0.29	0.39	0.30	47	35
75-22	75	77.07	713	7.30	2.89	10.61	2.65	192	188
75-23	75	76.92	737	14.70	3.72	22.55	4.98	229	229
75-27	75	76.73	453	2.11	1.34	3.58	1.30	191	156
75-29	75	76.03	247	0.95	0.70	4.64	0.70	86	78
75-30	75	76.65	647	6.62	2.34	11.46	2.28	229	194
75-31	75	76.70	642	4.98	8.17	4.81	5.62	203	202
75-36	75	75.73	332	1.90	0.89	6.44	0.89	116	100
75-38	75	76.50	703	3.03	3.37	4.91	2.92	189	191
75-39	75	76.45	303	0.76	0.64	1.94	0.66	72	65
75-40	75	76.24	503	1.34	1.14	3.23	1.18	167	143
75-41	75	76.76	603	1.78	1.33	3.47	1.31	136	135
75-43	75	77.76	1,132	12.91	1.22	15.41	1.23	393	NA
75-44	75	77.74	1,137	19.87	4.69	31.70	4.74	393	419
78-16	78	79.36	613	2.61	1.76	9.34	1.76	209	207
78-17	78	79.00	213	0.62	0.48	1.00	0.49	84	65
78-18	78	79.03	413	1.37	0.94	1.29	0.96	207	133
78-26	78	79.50	543	5.78	3.94	11.11	3.36	196	197
78-27	78	79.73	453	1.75	1.35	3.48	1.34	201	160
78-29	78	79.03	247	0.85	0.71	4.26	0.70	103	83
78-33	78	79.04	443	1.37	1.01	1.48	0.99	178	139
78-36	78	78.73	332	1.61	0.89	2.97	0.90	105	104
78-39	78	79.45	303	0.85	0.66	1.34	0.65	83	73
78-44	78	80.74	1,137	22.43	4.73	37.24	4.70	447	442
81-17	81	82.00	213	0.55	0.49	0.92	0.52	98	72
81-18	81	82.03	413	1.13	0.93	1.73	0.92	166	143
81-21	81	82.51	313	0.99	0.86	1.66	0.87	125	107
81-43	81	83.76	1,132	7.93	4.02	16.99	3.90	431	413
84-24	84	85.11	337	2.88	1.19	6.87	1.16	135	121

Table A.15: Results for Benchmark 2 - SBP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
84-25	84	84.77	143	0.47	0.47	0.97	0.44	61	38
87-24	87	88.11	337	1.85	1.16	3.27	1.19	131	120

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-0	30	34.93	16	0.08	0.08	0.14	0.10	0	0
30-1	30	40.65	78	0.29	0.30	0.29	0.31	0	0
30-10	30	44.31	76	0.27	0.30	0.48	0.32	3	6
30-11	30	38.07	156	0.80	0.65	0.83	0.69	0	1
30-12	30	38.88	152	0.62	0.49	0.66	0.52	0	0
30-13	30	44.56	71	0.23	0.22	0.24	0.23	1	1
30-15	30	42.25	142	0.48	0.39	0.49	0.38	0	0
30-16	30	43.63	707	7.13	2.70	32.85	2.57	84	85
30-19	30	45.88	657	5.39	2.28	22.64	2.35	86	86
30-2	30	35.39	38	0.14	0.13	0.19	0.14	0	0
30-21	30	38.89	357	2.80	1.17	9.20	1.18	34	40
30-22	30	46.90	757	7.92	3.13	31.72	3.29	52	48
30-23	30	47.34	777	13.17	4.20	73.65	4.10	52	68
30-24	30	45.59	377	2.54	1.51	10.97	1.54	28	34
30-25	30	39.27	197	0.89	0.60	1.34	0.63	0	11
30-27	30	34.68	627	5.47	1.94	16.80	2.06	87	86
30-28	30	41.74	67	0.23	0.19	0.33	0.20	0	0
30-29	30	33.52	327	1.85	1.00	7.03	0.99	33	34
30-3	30	45.30	68	0.21	0.23	0.33	0.23	5	5
30-30	30	37.61	727	8.21	3.13	23.06	2.92	58	57
30-31	30	41.16	722	6.31	2.42	10.74	2.35	3	3
30-35	30	33.01	192	0.66	0.50	1.15	0.50	10	11
30-36	30	33.62	372	1.99	1.43	4.24	1.44	19	29
30-37	30	45.52	772	10.83	3.77	13.70	7.15	22	44

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
30-4	30	46.89	37	0.13	0.12	0.21	0.13	0	0
30-44	30	40.96	1,577	14.10	6.77	239.94	6.66	215	213
30-47	30	45.69	316	1.91	0.98	3.23	1.01	21	25
30-5	30	45.46	146	0.53	0.48	1.23	0.50	12	17
30-6	30	35.35	66	0.29	0.19	0.25	0.20	0	0
30-8	30	39.28	136	0.55	0.47	0.83	0.47	10	10
30-9	30	39.77	40	0.14	0.13	0.22	0.15	2	2
33-0	33	37.93	16	0.08	0.08	0.14	0.09	0	0
33-1	33	43.65	78	0.26	0.29	0.39	0.30	0	0
33-10	33	47.31	76	0.29	0.31	0.66	0.44	4	10
33-11	33	41.07	156	0.76	0.68	1.16	0.68	5	10
33-12	33	41.88	152	0.69	0.49	0.67	0.51	0	0
33-13	33	47.56	71	0.24	0.22	0.34	0.23	1	1
33-16	33	46.63	707	6.93	2.66	26.97	2.59	98	96
33-19	33	48.88	657	5.18	2.34	26.46	2.39	100	102
33-2	33	38.39	38	0.14	0.13	0.16	0.16	0	0
33-21	33	41.89	357	2.21	1.20	8.53	1.20	42	43
33-22	33	49.90	757	9.29	3.08	23.08	3.17	52	57
33-23	33	50.34	777	16.65	4.48	119.16	5.82	63	73
33-24	33	48.59	377	3.40	1.77	12.44	1.77	35	46
33-25	33	42.27	197	0.87	0.60	1.14	0.64	0	11
33-26	33	46.22	677	7.74	2.94	19.25	4.04	52	63
33-27	33	37.68	627	5.47	2.08	20.85	2.01	95	115
33-29	33	36.52	327	1.91	1.03	5.97	1.05	34	38
33-3	33	48.30	68	0.21	0.21	0.31	0.23	5	10
33-30	33	40.61	727	10.69	3.03	77.09	3.05	62	87
33-31	33	44.16	722	6.16	2.31	12.25	2.52	3	3
33-35	33	36.01	192	0.65	0.49	1.12	0.50	10	11
33-36	33	36.62	372	2.14	1.48	4.46	1.40	23	32

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
33-37	33	48.52	772	11.23	3.95	28.30	3.90	32	54
33-4	33	49.89	37	0.14	0.12	0.22	0.14	0	0
33-41	33	50.04	702	5.41	1.81	8.58	2.03	4	4
33-44	33	43.96	1,577	16.73	6.86	88.66	6.89	241	224
33-47	33	48.69	316	1.81	1.01	2.87	1.03	32	41
33-5	33	48.46	146	0.57	0.46	0.91	0.49	0	0
33-6	33	38.35	66	0.27	0.19	0.24	0.21	0	0
33-9	33	42.77	40	0.17	0.13	0.18	0.14	2	2
36-0	36	40.93	16	0.08	0.08	0.12	0.09	0	0
36-1	36	46.65	78	0.26	0.29	0.35	0.31	0	0
36-10	36	50.31	76	0.25	0.32	0.42	0.43	4	10
36-11	36	44.07	156	0.74	0.64	1.11	0.67	0	1
36-12	36	44.88	152	0.60	0.49	0.63	0.51	0	0
36-16	36	49.63	707	5.49	2.87	24.07	2.87	106	111
36-19	36	51.88	657	5.14	2.56	22.11	2.60	110	109
36-2	36	41.39	38	0.14	0.13	0.16	0.15	0	0
36-21	36	44.89	357	2.57	1.26	11.04	1.25	45	46
36-22	36	52.90	757	8.55	3.20	16.66	3.35	54	3
36-23	36	53.34	777	12.89	4.34	49.31	6.10	67	89
36-24	36	51.59	377	4.37	1.81	9.60	1.77	38	52
36-25	36	45.27	197	0.88	0.64	1.28	0.62	0	11
36-27	36	40.68	627	4.81	2.20	18.38	2.22	123	124
36-28	36	47.74	67	0.22	0.21	0.44	0.24	5	5
36-29	36	39.52	327	1.90	1.03	9.81	1.07	41	44
36-3	36	51.30	68	0.20	0.22	0.28	0.22	10	10
36-30	36	43.61	727	9.58	3.27	32.35	3.34	87	92
36-31	36	47.16	722	6.22	2.44	11.22	2.47	4	4
36-35	36	39.01	192	0.65	0.50	1.00	0.52	10	11
36-36	36	39.62	372	7.84	1.45	3.65	1.38	26	35

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
36-37	36	51.52	772	40.10	3.19	34.53	3.20	42	54
36-4	36	52.89	37	0.14	0.12	0.16	0.14	0	0
36-40	36	52.91	652	4.87	1.61	7.01	1.71	102	102
36-42	36	46.90	1,552	11.89	3.88	58.33	3.92	243	308
36-44	36	46.96	1,577	15.22	7.01	80.89	6.97	242	263
36-47	36	51.69	314	1.35	1.04	5.96	1.06	72	74
36-5	36	51.46	146	0.57	0.48	0.76	0.51	0	1
36-6	36	41.35	66	0.25	0.19	0.25	0.21	0	0
36-9	36	45.77	40	0.14	0.13	0.19	0.14	0	4
39-0	39	43.93	16	0.08	0.08	0.13	0.11	0	0
39-1	39	49.65	78	0.26	0.28	0.42	0.32	0	0
39-11	39	47.07	156	0.79	0.63	1.32	0.67	8	12
39-12	39	47.88	152	0.63	0.49	0.92	0.52	0	0
39-16	39	52.63	707	6.10	3.09	23.30	2.96	118	122
39-18	39	47.84	607	4.68	1.43	8.38	1.46	116	112
39-19	39	54.88	657	4.86	2.70	22.33	2.75	122	124
39-2	39	44.39	38	0.14	0.13	0.16	0.13	0	0
39-21	39	47.89	357	2.35	1.26	15.42	1.24	53	53
39-22	39	55.90	757	7.68	3.21	43.56	3.14	72	78
39-23	39	56.34	777	18.61	4.99	58.86	4.75	95	109
39-24	39	54.59	377	3.63	1.89	11.09	1.84	45	55
39-25	39	48.27	197	0.84	0.61	1.27	0.63	0	11
39-26	39	52.22	677	6.55	2.77	13.96	2.85	140	102
39-27	39	43.68	627	4.70	2.26	19.84	2.29	136	134
39-28	39	50.74	67	0.21	0.19	0.24	0.21	10	10
39-29	39	42.52	327	1.82	1.29	10.64	1.68	50	64
39-3	39	54.30	68	0.19	0.21	0.25	0.22	11	10
39-30	39	46.61	727	11.55	3.45	28.92	3.31	94	102
39-31	39	50.16	722	6.13	2.30	14.10	2.39	4	5

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
39-35	39	42.01	192	0.65	0.49	1.12	0.52	10	11
39-36	39	42.62	372	2.09	1.64	5.60	1.54	31	47
39-37	39	54.52	772	11.30	3.71	24.12	5.19	57	104
39-4	39	55.89	37	0.14	0.12	0.21	0.13	0	0
39-40	39	55.91	652	4.80	1.70	7.34	1.55	102	102
39-42	39	49.90	1,552	11.83	3.99	66.63	4.13	332	308
39-44	39	49.96	1,577	16.10	6.92	84.87	6.81	266	313
39-47	39	54.69	314	1.31	1.09	3.88	1.08	75	80
39-5	39	54.46	146	0.57	0.47	0.84	0.49	0	2
39-6	39	44.35	66	0.23	0.19	0.33	0.20	0	1
39-9	39	48.77	40	0.14	0.13	0.24	0.15	3	5
42-0	42	46.93	16	0.08	0.08	0.12	0.10	0	0
42-1	42	52.65	78	0.27	0.28	0.35	0.30	0	0
42-11	42	50.07	156	0.76	0.64	0.92	0.66	0	1
42-12	42	50.88	152	0.59	0.48	0.62	0.49	0	0
42-14	42	49.90	132	0.40	0.34	0.49	0.35	20	20
42-18	42	50.84	607	4.63	1.42	7.95	1.49	134	122
42-2	42	47.39	38	0.13	0.14	0.20	0.14	0	0
42-21	42	50.89	357	2.13	1.30	5.73	1.29	55	56
42-23	42	59.34	777	19.86	4.76	49.61	4.78	98	114
42-24	42	57.59	377	4.86	2.29	12.58	2.09	47	64
42-25	42	51.27	197	0.83	0.62	1.60	0.65	20	21
42-27	42	46.68	627	4.44	2.60	20.01	2.55	144	143
42-28	42	53.74	67	0.21	0.20	0.28	0.22	10	10
42-29	42	45.52	327	1.98	1.47	17.04	1.40	66	70
42-3	42	57.30	68	0.19	0.20	0.29	0.21	11	10
42-30	42	49.61	727	10.64	3.78	55.89	3.65	98	123
42-31	42	53.16	722	6.09	2.30	10.50	2.47	5	5
42-32	42	51.13	322	1.16	0.99	2.07	0.94	60	61

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
42-34	42	55.99	672	4.40	2.26	18.26	2.27	113	132
42-35	42	45.01	192	0.65	0.49	1.15	0.50	10	21
42-36	42	45.62	372	1.93	1.57	5.99	1.53	49	50
42-37	42	57.52	772	7.38	3.45	25.45	4.87	49	114
42-4	42	58.89	37	0.13	0.12	0.16	0.14	0	0
42-44	42	52.96	1,577	17.76	7.18	63.61	7.07	331	324
42-47	42	57.69	314	1.39	1.15	4.47	1.17	83	95
42-6	42	47.35	66	0.26	0.19	0.24	0.20	0	1
42-8	42	51.28	136	0.51	0.45	0.87	0.48	20	20
42-9	42	51.77	40	0.15	0.13	0.31	0.15	4	5
45-0	45	49.93	16	0.08	0.08	0.13	0.09	0	0
45-1	45	55.65	78	0.27	0.28	0.42	0.29	0	0
45-11	45	53.07	156	0.78	0.63	2.33	0.66	8	12
45-12	45	53.88	152	0.59	0.50	0.80	0.52	0	0
45-19	45	60.88	657	4.62	2.21	20.89	2.24	151	153
45-2	45	50.39	38	0.13	0.13	0.22	0.14	0	0
45-21	45	53.89	357	1.71	1.17	7.31	1.21	63	64
45-22	45	61.90	757	8.20	3.18	26.26	3.35	116	93
45-23	45	62.34	777	42.51	4.46	45.59	4.39	109	129
45-24	45	60.59	377	4.65	2.37	25.58	2.32	57	71
45-25	45	54.27	197	0.82	0.62	2.31	0.63	20	21
45-27	45	49.68	627	4.19	3.10	19.23	3.02	148	154
45-28	45	56.74	67	0.21	0.20	0.23	0.22	12	10
45-29	45	48.52	327	2.62	1.54	8.29	1.47	73	74
45-3	45	60.30	68	0.19	0.21	0.31	0.22	11	10
45-30	45	52.61	727	9.15	4.12	20.59	3.94	108	133
45-31	45	56.16	722	6.21	2.48	13.55	2.43	5	5
45-32	45	54.13	322	1.07	0.80	1.62	0.79	80	80
45-34	45	58.99	672	4.29	2.27	14.82	2.48	120	139

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
45-35	45	48.01	192	0.64	0.48	1.25	0.50	10	21
45-36	45	48.62	372	1.69	1.62	5.50	1.53	52	54
45-37	45	60.52	772	9.28	3.68	26.10	4.89	119	124
45-4	45	61.89	37	0.14	0.12	0.15	0.14	0	0
45-44	45	55.96	1,577	15.70	6.99	82.74	7.12	350	374
45-47	45	60.69	314	1.24	1.02	3.86	1.04	91	99
45-6	45	50.35	66	0.22	0.18	0.24	0.20	0	1
45-9	45	54.77	40	0.13	0.14	0.33	0.14	6	7
48-0	48	52.93	16	0.07	0.09	0.10	0.10	2	2
48-1	48	58.65	78	0.25	0.30	0.28	0.31	10	10
48-10	48	62.31	76	0.47	0.27	0.61	0.29	11	15
48-11	48	56.07	156	0.92	0.67	1.87	0.69	19	23
48-12	48	56.88	152	0.57	0.49	1.54	0.51	0	0
48-14	48	55.90	132	0.35	0.40	0.95	0.40	31	29
48-2	48	53.39	38	0.13	0.13	0.18	0.14	0	0
48-21	48	56.89	357	1.61	1.16	5.28	1.20	67	72
48-22	48	64.90	757	8.14	3.32	18.68	4.82	54	55
48-23	48	65.34	777	13.97	3.96	42.06	4.01	111	144
48-24	48	63.59	377	2.63	2.09	12.17	2.03	61	75
48-25	48	57.27	197	0.81	0.62	1.37	0.64	20	21
48-27	48	52.68	627	4.09	2.34	15.26	2.32	160	164
48-29	48	51.52	327	2.51	1.46	7.89	1.39	77	81
48-3	48	63.30	68	0.19	0.20	0.37	0.22	11	12
48-30	48	55.61	727	11.50	3.89	25.67	3.94	122	138
48-31	48	59.16	722	6.25	2.57	9.01	2.62	103	104
48-32	48	57.13	322	1.02	0.78	2.90	0.80	89	85
48-35	48	51.01	192	0.64	0.48	1.11	0.50	10	21
48-36	48	51.62	372	4.88	1.21	13.92	1.15	56	65
48-37	48	63.52	772	9.08	3.53	36.62	3.46	142	135

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
48-4	48	64.89	37	0.13	0.13	0.15	0.14	5	5
48-44	48	58.96	1,577	15.94	7.11	62.37	7.16	421	414
48-47	48	63.69	314	1.24	1.01	3.70	1.03	99	105
48-5	48	63.46	146	0.48	0.48	1.78	0.49	36	34
48-6	48	53.35	66	0.21	0.19	0.24	0.21	12	12
48-9	48	57.77	40	0.14	0.13	0.33	0.15	7	7
51-0	51	55.93	16	0.08	0.08	0.12	0.10	1	1
51-1	51	61.65	78	0.30	0.29	0.44	0.31	0	0
51-10	51	65.31	76	0.25	0.27	0.80	0.29	13	15
51-11	51	59.07	156	0.70	0.67	2.39	0.67	27	27
51-12	51	59.88	152	0.56	0.54	0.73	0.57	20	20
51-14	51	58.90	132	0.35	0.40	0.67	0.39	32	31
51-2	51	56.39	38	0.13	0.12	0.16	0.13	0	0
51-23	51	68.34	777	10.95	4.08	46.74	4.06	157	165
51-24	51	66.59	377	33.12	1.59	11.79	1.63	88	94
51-25	51	60.27	197	1.55	0.18	2.68	0.21	40	NA
51-27	51	55.68	627	3.80	2.37	13.21	2.45	173	190
51-28	51	62.74	67	0.21	0.19	0.27	0.21	12	10
51-29	51	54.52	327	1.39	1.43	6.61	1.33	84	83
51-30	51	58.61	727	13.58	3.16	19.91	3.18	122	163
51-31	51	62.16	722	6.32	2.59	8.82	2.59	104	104
51-32	51	60.13	322	1.39	0.80	1.80	0.79	61	81
51-35	51	54.01	192	0.63	0.47	1.25	0.50	10	21
51-36	51	54.62	372	3.88	1.20	20.43	1.20	60	68
51-37	51	66.52	772	8.50	3.49	28.17	3.52	134	145
51-4	51	67.89	37	0.12	0.13	0.17	0.14	5	5
51-44	51	61.96	1,577	15.52	7.36	52.63	7.03	474	424
51-6	51	56.35	66	0.20	0.20	0.28	0.21	12	12
51-8	51	60.28	136	0.48	0.47	0.99	0.47	21	20

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
51-9	51	60.77	40	0.14	0.13	0.24	0.13	8	8
54-0	54	58.93	16	0.08	0.09	0.09	0.11	2	2
54-11	54	62.07	156	0.68	0.69	0.90	0.70	20	21
54-14	54	61.90	132	0.32	0.35	0.61	0.36	34	40
54-2	54	59.39	38	0.14	0.13	0.16	0.15	0	0
54-21	54	62.89	357	2.58	1.17	7.64	1.20	83	84
54-23	54	71.34	777	11.07	4.43	30.14	4.06	162	170
54-24	54	69.59	377	3.17	1.67	12.88	1.69	90	101
54-25	54	63.27	197	1.13	0.17	1.69	0.21	50	NA
54-27	54	58.68	627	3.94	2.40	12.26	2.45	200	196
54-29	54	57.52	327	1.43	1.56	6.95	1.46	86	91
54-30	54	61.61	727	8.52	3.20	24.36	3.00	170	173
54-35	54	57.01	192	0.62	0.48	1.25	0.50	10	21
54-36	54	57.62	372	1.59	1.20	5.39	1.20	67	72
54-37	54	69.52	772	6.43	4.19	10.74	3.95	156	154
54-4	54	70.89	37	0.12	0.12	0.18	0.14	10	8
54-5	54	69.46	146	0.43	0.48	1.14	0.49	40	38
54-6	54	59.35	66	0.23	0.19	0.25	0.20	16	16
54-8	54	63.28	136	0.47	0.47	0.83	0.48	26	22
54-9	54	63.77	40	0.13	0.13	0.31	0.15	9	11
57-0	57	61.93	16	0.08	0.09	0.11	0.10	2	2
57-11	57	65.07	156	0.74	0.67	1.78	0.68	29	31
57-14	57	64.90	132	0.30	0.34	0.43	0.35	44	42
57-2	57	62.39	38	0.14	0.13	0.20	0.14	0	0
57-21	57	65.89	357	1.92	1.25	2.34	1.26	51	51
57-23	57	74.34	777	11.30	4.37	36.61	5.75	221	215
57-25	57	66.27	197	0.74	0.18	0.99	0.20	50	NA
57-27	57	61.68	627	4.00	2.76	16.37	2.66	211	211
57-29	57	60.52	327	1.47	1.60	5.56	1.49	92	98

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
57-30	57	64.61	727	7.23	3.11	21.57	3.07	194	178
57-31	57	68.16	722	5.40	2.33	16.27	2.38	145	164
57-32	57	66.13	322	1.25	0.76	1.55	0.80	99	81
57-35	57	60.01	192	0.62	0.47	0.83	0.50	10	21
57-36	57	60.62	372	1.77	1.21	4.97	1.17	70	76
57-37	57	72.52	772	8.04	3.60	29.13	3.99	173	175
57-4	57	73.89	37	0.13	0.12	0.13	0.15	8	8
57-6	57	62.35	66	0.21	0.19	0.22	0.21	16	16
57-9	57	66.77	40	0.13	0.13	0.26	0.14	11	12
60-12	60	68.88	152	0.54	0.54	0.82	0.58	31	30
60-14	60	67.90	132	0.29	0.35	0.44	0.38	48	44
60-2	60	65.39	38	0.14	0.13	0.16	0.14	0	0
60-22	60	76.90	757	6.44	3.47	19.65	4.96	229	199
60-23	60	77.34	777	10.12	4.41	38.95	4.26	252	240
60-27	60	64.68	627	3.68	2.93	12.86	2.90	221	222
60-29	60	63.52	327	1.47	1.60	19.38	1.52	100	101
60-30	60	67.61	727	6.85	3.13	19.69	3.00	229	193
60-31	60	71.16	722	5.09	2.38	11.89	2.59	165	165
60-32	60	69.13	322	1.16	0.78	1.50	0.79	81	81
60-35	60	63.01	192	0.61	0.49	0.94	0.50	20	31
60-36	60	63.62	372	1.38	1.21	6.16	1.22	81	82
60-37	60	75.52	772	7.58	1.03	24.39	1.06	197	NA
60-4	60	76.89	37	0.12	0.12	0.19	0.14	10	8
60-5	60	75.46	146	0.41	0.48	0.95	0.50	42	46
60-6	60	65.35	66	0.19	0.19	0.25	0.22	18	16
60-9	60	69.77	40	0.12	0.13	0.24	0.16	13	13
63-11	63	71.07	156	0.72	0.67	1.35	0.68	37	35
63-14	63	70.90	132	0.27	0.35	0.48	0.36	58	46
63-2	63	68.39	38	0.13	0.13	0.16	0.15	0	1

Table A.16: Results for Benchmark 2 - PTP.

M	U_{cp}(%)	U_t(%)	J	F_{tts}(s)	F_{fps}(s)	O_{tts}(s)	O_{fps}(s)	P_{tts}(s)	P_{fps}(s)
63-27	63	67.68	627	2.98	2.07	15.63	2.14	245	244
63-29	63	66.52	327	1.57	1.00	5.47	1.03	105	120
63-31	63	74.16	722	5.03	2.60	9.59	2.58	167	165
63-35	63	66.01	192	0.58	0.13	0.76	0.14	51	NA
63-36	63	66.62	372	2.75	1.22	6.96	1.21	95	94
63-9	63	72.77	40	0.25	0.13	0.33	0.14	13	14
66-27	66	70.68	627	2.88	1.98	12.89	2.10	258	256
66-29	66	69.52	327	1.77	1.03	10.11	1.02	124	123
66-36	66	69.62	372	1.96	1.22	7.36	1.21	108	99
69-11	69	77.07	156	0.62	0.20	1.01	0.21	50	NA
69-27	69	73.68	627	2.39	1.97	13.80	2.06	279	268
69-29	69	72.52	327	1.06	1.01	3.24	1.01	142	131
69-36	69	72.62	372	1.97	1.24	5.80	1.24	107	105
72-27	72	76.68	627	2.30	2.04	6.24	2.06	288	284
72-29	72	75.52	327	1.11	1.01	2.84	1.01	142	135
72-36	72	75.62	372	1.72	1.25	8.62	1.20	128	123
75-27	75	79.68	627	1.86	2.11	5.27	2.14	307	296
75-29	75	78.52	327	0.88	1.00	6.50	1.03	155	143
75-36	75	78.62	372	1.41	1.23	5.28	1.23	127	129
78-29	78	81.52	327	0.81	1.02	2.28	1.02	151	151

References

[1] N. Navet and F. Simonot-Lion. *Automotive embedded systems handbook*. CRC press, 2017.

[2] D. Claraz, S. Kuntz, U. Margull, M. Niemetz, and G. Wirrer. “Deterministic Execution Sequence in Component Based Multi-Contributor Powertrain Control Systems”. In: *Embedded Real Time Software and Systems (ERTS2012)*. Toulouse, France, 2012.

[3] G. Macher, A. Höller, E. Armengaud, and C. Kreiner. “Automotive embedded software: Migration challenges to multi-core computing platforms”. In: *IEEE 13th International Conference on Industrial Informatics (INDIN)*. 2015, pp. 1386–1393.

[4] S. Kehr, E. Quiñones, B. Böddeker, and G. Schäfer. “Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication”. In: *ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[5] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. “Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?” In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2008, pp. 342–353.

[6] C. M. Kirsch and A. Sokolova. “The Logical Execution Time Paradigm”. In: *Advances in Real-Time Systems (ARTS)*. Springer, 2012, pp. 103–120.

[7] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. “Giotto: A Time-Triggered Language for Embedded Programming”. In: *Embedded Software*. Vol. 2211. Lecture Notes in Computer Science. Springer, 2001, pp. 166–184.

[8] S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree. “Applying Real-Time Programming to Legacy Embedded Control Software”. In: *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2015, pp. 1–8.

[9] S. Resmerita, A. Naderlinger, and S. Lukesch. “Efficient Realization of Logical Execution Times in Legacy Embedded Software”. In: *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. ACM, 2017, pp. 36–45.

[10] C. Sofronis, S. Tripakis, and P. Caspi. “A Memory-Optimal Buffering Protocol for Preservation of Synchronous Semantics Under Preemptive Scheduling”. In: *ACM and IEEE International Conference on Embedded Software (EMSOFT)*. ACM, 2006, pp. 21–33.

[11] F. Scheler and W. Schröder-Preikschat. “The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems”. In: *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2010, pp. 34–41.

[12] E. Yip, E. Lalo, G. Lüttgen, and A. Sailer. “Lightweight Semantics-Preserving Communication for Real-Time Automotive Software”. In: *IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2019, pp. 372–379.

[13] E. Yip, E. Lalo, G. Lüttgen, M. Deubzer, and A. Sailer. *Optimized Buffering of Time-Triggered Automotive Software*. eng. Tech. rep. Bamberg, 2018, pp. 76, 8. DOI: 10.20378/irbo-52917. URL: https://fis.uni-bamberg.de/handle/uniba/44463.

[14] E. Lalo, R. Weber, A. Sailer, J. Mottok, and C. Siemers. “On Solving Task Allocation and Schedule Generation for Time-Triggered LET Systems using Constraint Programming”. In: *ARCS Workshop 2019; IEEE 32nd International Conference on Architecture of Computing Systems*. 2019, pp. 1–8.

[15] E. Lalo, A. Sailer, J. Mottok, and C. Siemers. “Overhead-Aware Schedule Synthesis for Logical Execution Time (LET) in Automotive Systems”. In: *IEEE 35th International System-on-Chip Conference (SOCC)*. 2022, pp. 1–6.

[16] Vector Informatik GmbH. *TA Simulation Module*. 2020-09. URL: http://www.vector.com.

[17] ISO. *Road vehicles – functional safety. ISO 26262.* Standard. International Organisation for Standardisation, 2011.

[18] ARINC Specification 653-1. *Avionics application standard interface. Aeronautical Radio Inc Software.* 2003-10.

[19] AUTOSAR. *Specification of Operating System. Release R21-11.* Available at http://www.autosar.org. 2021-11.

[20] D. Reinhardt and G. Morgan. “An embedded hypervisor for safety-relevant automotive E/E-systems”. In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. 2014, pp. 189–198.

[21] G. Han, H. Zeng, M. Di Natale, X. Liu, and W. Dou. “Experimental Evaluation and Selection of Data Consistency Mechanisms for Hard Real-Time Applications on Multicore Platforms”. In: *IEEE Transactions on Industrial Informatics* 10.2 (2014), pp. 903–918.

[22] M. Raynal. “Solving Mutual Exclusion”. In: *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013, pp. 15–60.

[23] J. B. Goodenough and L. Sha. “The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks”. In: *ACM SIGAda Ada Letters* 8.7 (1988), pp. 20–31.

[24] H. Kopetz and J. Reisinger. “The Non-Blocking Write Protocol NBW: A solution to a Real-Time Synchronization Problem”. In: *Real-Time Systems Symposium*. IEEE, 1993, pp. 131–137.

[25] M. Herlihy. “A Methodology for Implementing Highly Concurrent Data Structures”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 1990, pp. 197–206.

[26] D. Paret. *Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire*. John Wiley & Sons, 2007.

[27] R. I. Davis, A. Zabos, and A. Burns. “Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems”. In: *IEEE Transactions on Computers* 57.9 (2008), pp. 1261–1276.

[28] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. “Is Semi-Partitioned Scheduling Practical?” In: *23rd Euromicro Conference on Real-Time Systems*. 2011, pp. 125–135.

[29] B. Andersson and J. Jonsson. “Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition”. In: *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. 2000, pp. 337–346.

[30] M. Lowinski, D. Ziegenbein, and S. Glesner. “Splitting tasks for migrating real-time automotive applications to multi-core ECUs”. In: *11th IEEE Symposium on Industrial Embedded Systems (SIES)*. 2016, pp. 1–8.

[31] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. “Proportionate progress: A notion of fairness in resource allocation”. In: *Algorithmica* 15.6 (1996), pp. 600–625.

[32] H. Alhussian, N. Zakaria, and A. Patel. “An unfair semi-greedy real-time multiprocessor scheduling algorithm”. In: *Computers and Electrical Engineering* 50 (2014), pp. 143–165.

[33] M. Bertogna and S. Baruah. “Tests for Global EDF Schedulability Analysis”. In: *J. Syst. Archit.* 57.5 (2011), pp. 487–497.

[34] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. “Optimal Selection of Preemption Points to Minimize Preemption Overhead”. In: *23rd Euromicro Conference on Real-Time Systems*. 2011, pp. 217–227.

[35] J. M. Marinho, V. Nélis, S. M. Petters, and I. Puaut. “Preemption delay analysis for floating non-preemptive region scheduling”. In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 497–502.

[36] G. Yao, G. Buttazzo, and M. Bertogna. “Comparative evaluation of limited preemptive methods”. In: *IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*. 2010, pp. 1–8.

[37] G. C. Buttazzo, M. Bertogna, and G. Yao. “Limited Preemptive Scheduling for Real-Time Systems. A Survey”. In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 3–15.

[38] H. Kopetz. “Event-Triggered Versus Time-Triggered Real-Time Systems”. In: *Operating Systems of the 90s and Beyond*. Vol. 563. Lecture Notes in Computer Science. Springer, 1991, pp. 87–101.

[39] F. Scheler and W. Schroeder-Preikschat. “Time-Triggered vs. Event-Triggered: A matter of configuration?” In: *ITG FA 6.2 Workshop on Model-Based Testing, GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 13th GI/ITG Conference Measuring, Modelling, and Evaluation of Computer and Communications*. 2006, pp. 1–6.

[40] S. Schorr and G. Fohler. “Integrated time- and event-triggered scheduling - An overhead analysis on the ARM architecture”. In: *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. 2013, pp. 165–174.

[41] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. Springer Publishing Company, Incorporated, 2011.

[42] E. Massa, G. Lima, and P. Regnier. “Revealing the Secrets of RUN and QPS: New Trends for Optimal Real-Time Multiprocessor Scheduling”. In: *Brazilian Symposium on Computing Systems Engineering*. 2014, pp. 150–155.

[43] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. “RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor”. In: *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*. IEEE Computer Society, 2011, pp. 104–115.

[44] AUTOSAR. *Software Component Template. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[45] AUTOSAR. *Methodology for Classic Platform. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[46] AUTOSAR. *Specification of RTE Software. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[47] AUTOSAR. *Basic Software Module Description Template. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[48] A. Lofwenmark and S. Nadjm-Tehrani. “Challenges in future avionic systems on multi-core platforms”. In: *Proceedings - IEEE 25th International Symposium on Software Reliability Engineering Workshops, ISSREW 2014* (2014), pp. 115–119.

[49] R. Rajkumar. “Real-time synchronization protocols for shared memory multiprocessors”. In: *Proceedings.,10th International Conference on Distributed Computing Systems*. 1990, pp. 116–123.

[50] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers. “An efficient spin-lock based multi-core resource sharing protocol”. In: *IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*. 2014, pp. 1–7.

[51] L. Michel, T. Flaemig, D. Claraz, and R. Mader. “Shared SW development in multi-core automotive context”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. Toulouse, France, 2016.

[52] R. Rivett. “The Challenge of Technological Change in the Automotive Industry”. In: *Achieving Systems Safety*. Ed. by C. Dale and T. Anderson. Springer London, 2012, pp. 35–42.

[53] J. Martinez, I. Sañudo, and M. Bertogna. “Analytical Characterization of End-to-End Communication Delays With Logical Execution Time”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2244–2254.

[54] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree. “From control models to real- time code using Giotto.” In: *IEEE Control Systems Magazine 23(1)*. Springer, 2003, pp. 50–64.

[55] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido. “Event-driven programming with logical execution times”. In: *In Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC)* 2993 (2004), pp. 357–371.

[56] E. Farcas, C. Farcas, W. Pree, and J. Templ. “Transparent Distribution of Real-time Components Based on Logical Execution Time”. In: *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* 40 (2005), pp. 31–39.

[57] P. Derler and S. Resmerita. “Flexible Static Scheduling of Software with Logical Execution Time Constraints”. In: *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*. CIT ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1719–1726.

[58] A. Biondi, P. Pazzaglia, A. Balsini, and M. D. Natale. “Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores”. In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Available at `https://www.ecrts.org/forum/viewtopic.php?f=32&t=87`, last accessed August 2018. 2017.

[59] AUTOSAR. *Specification of Timing Extensions. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[60] N. Halbwachs. “Synchronous programming of reactive systems”. In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 1–16.

[61] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. “The Synchronous Languages 12 Years Later”. In: *IEEE* 91.1 (2003), pp. 64–83.

[62] P. Jungklass and M. Berekovic. “Effects of Concurrent Access to Embedded Multicore Microcontrollers with Hard Real-Time Demands”. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2018, pp. 1–9.

[63] 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 2017)*. Ed. by M. Bertogna. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:20. DOI: 10.4230/LIPIcs.ECRTS.2017.10. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7162.

[64] E. Ntaryamira, C. Maxim, and L. Cucu-Grosjean. "Data Consistency and Temporal Validity under the Circular Buffer Communication Paradigm". In: *Proceedings of the Conference on Research in Adaptive and Convergent Systems*. RACS '19. Association for Computing Machinery, 2019, pp. 51–56.

[65] E. Ntaryamira, C. Maxim, T. Niyonsaba, and L. Cucu-Grosjean. "An efficient FIFO buffer management to ensure task level and effect-chain level data properties". In: *IEEE International Conference on Embedded Software and Systems (ICESS)*. 2020, pp. 1–8.

[66] G. Wang, M. D. Natale, and A. Sangiovanni-Vincentelli. *An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protocols*. Tech. rep. UCB/EECS-2007-81. EECS Department, University of California, Berkeley, 2007. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-81.html.

[67] G. Wang, M. D. Natale, and A. Sangiovanni-Vincentelli. "Optimal Synthesis of Communication Procedures in Real-Time Synchronous Reactive Models". In: *IEEE Transactions on Industrial Informatics* 6.4 (2010), pp. 729–743.

[68] M. D. Natale, G. Wang, and A. S. Vincentelli. "Optimizing the Implementation of Communication in Synchronous Reactive Models". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2008, pp. 169–179.

[69] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. "Improving the Size of Communication Buffers in Synchronous Models With Time Constraints". In: *IEEE Transactions on Industrial Informatics* 5.3 (2009), pp. 229–240.

[70] H. Zeng and M. Di Natale. "Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms". In: *6th IEEE International Symposium on Industrial and Embedded Systems*. 2011, pp. 140–149.

[71] H. Zeng and M. D. Natale. "Efficient Implementation of AUTOSAR Components with Minimal Memory Usage". In: *IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2012, pp. 130–137.

[72] AUTOSAR. *Release R21-11*. Available at http://www.autosar.org. 2021-11.

[73] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger. "Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[74] C. Bradatsch, F. Kluge, and T. Ungerer. "Data Age Diminution in the Logical Execution Time Model". In: *International Conference on Architecture of Computing Systems (ARCS)*. Vol. 9637. Lecture Notes in Computer Science. Springer, 2016, pp. 173–184.

[75] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. "End-to-end timing analysis of cause-effect chains in automotive embedded systems". In: *Journal of Systems Architecture* 80 (2017), pp. 104–113.

[76] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein. "WATERS Industrial Challenge 2017". In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Available at `https://waters2017.inria.fr/challenge`, last accessed August 2018. Inria, 2017.

[77] C. Brandberg and M. Di Natale. "A SimEvents Model for the Analysis of Scheduling and Memory Access Delays in Multicores". In: *IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–10.

[78] R. Ernst, L. Ahrendts, K.-B. Gemlau, S. Quinton, H. Von Hasseln, and J. Hennig. *System Level LET with Application to Automotive Design*. Research Report. TU Braunschweig, 2018, pp. 1–11.

[79] F. Kluge, M. Schoeberl, and T. Ungerer. "Support for the Logical Execution Time Model on a Time-predictable Multicore Processor". In: *SIGBED Review–Special Issue on International Workshop on RealTime Networks (RTN)* 13.4 (2016), pp. 61–66.

[80] M. Beckert, M. Möstl, and R. Ernst. "Zero-time communication for automotive multi-core systems under SPP scheduling". In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–9.

[81] M. Beckert. "Scheduling Mechanisms for Efficient and Safe Automotive Systems Integration". PhD thesis. 2020-01.

[82] M. Ogawa, S. Honda, and H. Takada. "Efficient Approach to Ensure Temporal Determinism in Automotive Control Systems". In: *8th International Symposium on Embedded Computing and System Design (ISED)*. 2018, pp. 53–57.

[83] A. Biondi and M. Di Natale. "Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2018, pp. 240–250.

[84] P. Haefele, U. Hartmann, D. Ziegenbein, and S. Kramer. "Method and device for operating a control device". Patent US11115232B2 (United States). 2021-09.

[85] Vector Informatik GmbH. *Timing Architects Tool Suite*. `http://www.vector.com`. 2018-04.

[86] AUTOSAR. *Specification of Software Cluster Connection module. Release R21-11*. Available at `http://www.autosar.org`. 2021-11.

[87] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. "The Worst-Case Execution-time Problem–Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (2008), 36:1–36:53.

[88] E. Wozniak, M. Di Natale, H. Zeng, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. "Assigning Time Budgets to Component Functions in the Design of Time-Critical Automotive Systems". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 235–246.

[89] A. A. Paul and B. A. S. Pillai. "Reducing the Number of Context Switches in Real Time Systems". In: *International Conference on Process Automation, Control and Computing*. 2011, pp. 1–6.

[90] OSEK/VDX. *Operating Sytem*. Specification 2.3.3. Available at `http://www.osek-vdx.org`, last accessed on November 2017. 2005-02.

[91] S. M. Enosh and N. S. George. "An efficient implementation of protocol operation control unit of FlexRay communication controller". In: *First International Conference on Computational Systems and Communications (ICCSC)*. 2014, pp. 256–259.

[92] IEEE 802.1 Working Group. *Time-sensitive networking task group*. Last accessed September 2021. URL: `http://www.ieee802.org/1/pages/tsn.html`.

[93] C. Barrett and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing, 2018, pp. 305–343.

[94] S. Samii, A. Cervin, P. Eles, and Z. Peng. "Integrated Scheduling and Synthesis of Control Applications on Distributed Embedded Systems". In: *Design, Automation & Test in Europe Conf. & Exhib.* 2009, pp. 57–62.

[95] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty. "Constraint-Driven Synthesis and Tool-Support for FlexRay-Based Automotive Control Systems". In: *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. Association for Computing Machinery, 2011, pp. 139–148.

[96] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. "Time-triggered implementations of mixed-criticality automotive software". In: *Design, Automation & Test in Europe Conference Exhibition (DATE)*. 2012, pp. 1227–1232.

[97] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty. "Multi-Objective Co-Optimization of FlexRay-Based Distributed Control Systems". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, pp. 1–12.

[98] F. Eisenbrand, K. Kesavan, R. S. Mattikalli, M. Niemeier, A. W. Nordsieck, M. Skutella, J. Verschae, and A. Wiese. “Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods”. In: *Proceedings of the 18th Annual European Conference on Algorithms: Part I*. ESA’10. 2010, pp. 11–22.

[99] M. Blikstad, E. Karlsson, T. Lööw, and E. Rönnberg. “An optimisation approach for pre-runtime scheduling of tasks and communication in an integrated modular avionic system”. In: *Optimization and Engineering* 19 (2018), pp. 1–28.

[100] S. Voss and B. Schätz. “Deployment and Scheduling Synthesis for Mixed-Critical Shared-Memory Applications”. In: *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. 2013, pp. 100–109.

[101] S. Zverlov and S. Voss. “Synthesis of Pareto Efficient Technical Architectures for Multi-core Systems”. In: *IEEE 38th International Computer Software and Applications Conference Workshops*. 2014, pp. 366–371.

[102] S. Zverlov, M. Khalil, and M. Chaudhary. “Pareto-efficient deployment synthesis for safety-critical applications in seamless model-based development”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS)*. Toulouse, France, 2016.

[103] G. Igna, L. Dieudonne, S. Voss, and B. Schatz. “Model-based deployment generation for safety-critical avionics systems”. In: *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2017, pp. 1–8.

[104] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Stähle, S. Chakraborty, and A. C. Knoll. “Schedule Integration Framework for Time-Triggered Automotive Architectures”. In: *The 51st Annual Design Automation Conference 2014, DAC ’14*. ACM, 2014, 20:1–20:6.

[105] F. Sagstetter. “Schedule Synthesis for Time-Triggered Automotive Architectures”. PhD thesis. Technical University Munich, Germany, 2016.

[106] A. Darbandi, S. Yoon, and M. K. Kim. “Schedule construction under precedence constraints in FlexRay in-vehicle networks”. In: *International Journal of Automotive Technology* 18.4 (2017), pp. 671–683.

[107] A. Minaeva, B. Akesson, Z. Hanzálek, and D. Dasari. “Time-Triggered Co-Scheduling of Computation and Communication with Jitter Requirements”. In: *IEEE Transactions on Computers* 67.1 (2018), pp. 115–129.

[108] W. Wang, F. Camut, and B. Miramond. “Generation of schedule tables on multi-core systems for AUTOSAR applications”. In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2016, pp. 191–198.

[109] M. Hu, J. Luo, Y. Wang, M. Lukasiewycz, and Z. Zeng. “Holistic Scheduling of Real-Time Applications in Time-Triggered In-Vehicle Networks”. In: *IEEE Transactions on Industrial Informatics* 10.3 (2014), pp. 1817–1828.

[110] M. Hu, J. Luo, Y. Wang, and B. Veeravalli. “Scheduling periodic task graphs for safety-critical time-triggered avionic systems”. In: *IEEE Transactions on Aerospace and Electronic Systems* 51.3 (2015), pp. 2294–2304.

[111] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. “Extensible and scalable time triggered scheduling”. In: *Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*. 2005, pp. 132–141.

[112] H. Zeng, W. Zheng, M. Di Natale, A. Ghosal, P. Giusto, and A. Sangiovanni-Vincentelli. “Scheduling the FlexRay bus using optimization techniques”. In: *2009 46th ACM/IEEE Design Automation Conference*. 2009, pp. 874–877.

[113] H. Zeng, M. Di Natale, A. Ghosal, and A. Sangiovanni-Vincentelli. “Schedule Optimization of Time-Triggered Systems Communicating Over the FlexRay Static Segment”. In: *IEEE Transactions on Industrial Informatics* 7.1 (2011), pp. 1–17.

[114] R. Hilbrich and H.-J. Goltz. “Model-based Generation of Static Schedules for Safety Critical Multi-core Systems in the Avionics Domain”. In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. IWMSE '11. ACM, 2011, pp. 9–16.

[115] R. Hilbrich. “Platzierung von Softwarekomponenten auf Mehrkernprozessoren: automatisierte Konstruktion und Analyse für funktionssichere Systeme”. PhD thesis. Brandenburg University of Technology, Cottbus-Senftenberg, Germany, 2015.

[116] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. “Modular scheduling of distributed heterogeneous time-triggered automotive systems”. In: *17th Asia and South Pacific Design Automation Conference*. 2012, pp. 665–670.

[117] Y. Zhou, S. Samii, P. Eles, and Z. Peng. “Partitioned and Overhead-aware Scheduling of Mixed-criticality Real-time Systems”. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC '19. New York, NY, USA: ACM, 2019, pp. 39–44.

[118] P. Han, Z. Zhai, B. Nielsen, and U. Nyman. “Model-Based Optimization of ARINC-653 Partition Scheduling”. In: *Int. J. Softw. Tools Technol. Transf.* 23.5 (2021), pp. 721–740.

[119] S. D. McLean, S. S. Craciunas, E. Alexander Juul Hansen, and P. Pop. “Mapping and Scheduling Automotive Applications on ADAS Platforms using Metaheuristics”. In: *25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 329–336.

[120] eCos. *Embedded Configurable Operating System*. Available at `http://ecos.sourceware.org`. 2012.

[121] Y.-K. Kwok and I. Ahmad. “Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 7.5 (1996), pp. 506–521.

[122] H. Topcuoglu, S. Hariri, and M.-Y. Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274.

[123] J. Theis, G. Fohler, and S. Baruah. “Schedule table generation for time-triggered mixed criticality systems”. In: *Proc. WMC, RTSS* (2013), pp. 79–84.

[124] A. Burns and R. I. Davis. “A Survey of Research into Mixed Criticality Systems”. In: *ACM Comput. Surv.* 50.6 (2017), 82:1–82:37.

[125] S. Vestal. “Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance”. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 239–243.

[126] W. Steiner. “An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks”. In: *31st IEEE Real-Time Systems Symposium*. 2010, pp. 375–384.

[127] W. Steiner. “Synthesis of Static Communication Schedules for Mixed-Criticality Systems”. In: *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 2011, pp. 11–18.

[128] J. Lehoczky. “Fixed priority scheduling of periodic task sets with arbitrary deadlines”. In: *Proceedings 11th Real-Time Systems Symposium*. 1990, pp. 201–209.

[129] P. Pazzaglia, A. Biondi, and M. Di Natale. “Simple and General Methods for Fixed-Priority Schedulability in Optimization Problems”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019, pp. 1543–1548.

[130] Y. Wang and M. Saksena. “Scheduling Fixed-Priority Tasks with Preemption Threshold”. In: *International Conference on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1999, pp. 328–335.

[131] M. Bertogna, G. Buttazzo, and G. Yao. “Improving Feasibility of Fixed Priority Tasks Using Non-Preemptive Regions”. In: *IEEE 32nd Real-Time Systems Symposium*. 2011, pp. 251–260.

[132] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns. “A review of priority assignment in real-time systems”. In: *Journal of Systems Architecture - Embedded Systems Design* 65 (2016), pp. 64–82.

[133] N. C. Audsley. “On priority assignment in fixed priority scheduling”. In: *Information Processing Letters* 79.1 (2001), pp. 39–44.

[134] R. I. Davis and A. Burns. “Robust Priority Assignment for Fixed Priority Real-Time Systems”. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 3–14.

[135] M. Grenier, J. Goossens, and N. Navet. “Near-Optimal Fixed Priority Preemptive Scheduling of Offset Free Systems”. In: *14th International Conference on Real-Time and Networks Systems (RTNS’06)*. 2006, pp. 35–42.

[136] J. Goossens and R. Devillers. “The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems”. In: *Real-Time Systems*. 13. 1997, pp. 107–126.

[137] R. Garibay-Martínez, G. Nelissen, L. L. Ferreira, and L. M. Pinho. “Task partitioning and priority assignment for distributed hard real-time systems”. In: *Journal of Computer and System Sciences* 81.8 (2015), pp. 1542–1555.

[138] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. “PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling”. In: *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[139] J. J. G. García and M. G. Harbour. “Optimized priority assignment for tasks and messages in distributed hard real-time systems”. In: *Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems*. 1995, pp. 124–132.

[140] Q. Zhu, Y. Yang, E. Scholte, M. D. Natale, and A. Sangiovanni-Vincentelli. “Optimizing Extensibility in Hard Real-Time Distributed Systems”. In: *15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2009, pp. 275–284.

[141] Q. Zhu, Y. Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli. “Optimizing the Software Architecture for Extensibility in Hard Real-Time Distributed Systems”. In: *IEEE Transactions on Industrial Informatics* 6.4 (2010), pp. 621–636.

[142] E. Azketa, J. P. Uribe, M. Marcos, L. Almeida, and J. J. Gutierrez. “Permutational Genetic Algorithm for the Optimized Assignment of Priorities to Tasks and Messages in Distributed Real-Time Systems”. In: *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. 2011, pp. 958–965.

[143] A. Hamann, M. Jersak, K. Richter, and R. Ernst. “Design space exploration and system optimization with SymTA/S - symbolic timing analysis for systems”. In: *25th IEEE International Real-Time Systems Symposium*. 2004, pp. 469–478.

[144] M. N. S. M. Sayuti and L. S. Indrusiak. “Simultaneous Optimisation of Task Mapping and Priority Assignment for Real-Time Embedded NoCs”. In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015, pp. 692–695.

[145] R. Bouaziz, L. Lemarchand, F. Singhoff, B. Zalila, and M. Jmaiel. “Architecture Exploration of Real-Time Systems Based on Multi-objective Optimization”. In: *20th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2015, pp. 1–10.

[146] I. Bate and P. Emberson. “Incorporating Scenarios And Heuristics To Improve Flexibility In Real-Time Embedded Systems”. In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06)*. 2006, pp. 221–230.

[147] R. Racu, M. Jersak, and R. Ernst. “Applying sensitivity analysis in real-time distributed systems”. In: *11th IEEE Real Time and Embedded Technology and Applications Symposium*. 2005, pp. 160–169.

[148] A. Mehiaoui, E. Wozniak, S. Tucci-Piergiovanni, C. Mraidha, M. Di Natale, H. Zeng, J.-P. Babau, L. Lemarchand, and S. Gerard. “A Two-Step Optimization Technique for Functions Placement, Partitioning, and Priority Assignment in Distributed Systems”. In: *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. Association for Computing Machinery, 2013, pp. 121–132.

[149] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. “An Optimization Approach for the Synthesis of AUTOSAR Architectures”. In: *Conference on Emerging Technologies Factory Automation (ETFA)*. IEEE, 2013, pp. 1–10.

[150] A. Metzner and C. Herde. “RTSAT– An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures”. In: *27th IEEE International Real-Time Systems Symposium (RTSS’06)*. 2006, pp. 147–158.

[151] W. Zheng, Q. Zhu, M. D. Natale, and A. S. Vincentelli. “Definition of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems”. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 161–170.

[152] Q. Zhu, H. Zeng, W. Zheng, M. D. Natale, and A. Sangiovanni-Vincentelli. “Optimization of Task Allocation and Priority Assignment in Hard Real-time Distributed Systems”. In: *ACM Trans. Embed. Comput. Syst.* 11.4 (2013), 85:1–85:30.

[153] J. Schlatow, M. Mostl, S. Tobuschat, T. Ishigooka, and R. Ernst. “Data-Age Analysis and Optimisation for Cause-Effect Chains in Automotive Control Systems”. In: *IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–9.

[154] 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)*. 2013, pp. 49–58.

[155] H. Zeng and M. Di Natale. “An Efficient Formulation of the Real-Time Feasibility Region for Design Optimization”. In: *IEEE Transactions on Computers* 62.4 (2013), pp. 644–661.

[156] Y. Zhao and H. Zeng. “The concept of unschedulability core for optimizing priority assignment in real-time systems”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 232–237.

[157] S. Igarashi, T. Ishigooka, T. Horiguchi, R. Koike, and T. Azumi. “Heuristic Contention-Free Scheduling Algorithm for Multi-core Processor using LET Model”. In: *IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2020, pp. 1–10.

[158] A. Yano, S. Igarashi, and T. Azumi. “Contention-Free Scheduling Algorithm Using LET Paradigm for Clustered Many-core Processor”. In: *IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2021, pp. 1–4.

[159] H. Zeng, M. D. Natale, and Q. Zhu. “Minimizing Stack and Communication Memory Usage in Real-Time Embedded Applications”. In: *ACM Transactions on Embedded Computing Systems* 13.5s (2014), 149:1–149:25.

[160] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gérard, and F. Terrier. “Enabling Scheduling Analysis for AUTOSAR Systems”. In: *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2011, pp. 152–159.

[161] "Timing Architects Embedded Systems GmbH." *BTF Specification (Version 2.1.3)*. 2014-04. URL: `https://wiki.eclipse.org/images/e/e6/TA_BTF_Specification_2.1.3_Eclipse_Auto_IWG.pdf`.

[162] L. Perron and V. Furnon. *OR-Tools*. Version 7.4. Google, 2019-10. URL: `https://developers.google.com/optimization/`.

[163] Vector Informatik GmbH. *TA Inspection Module*. 2022-09. URL: `http://www.vector.com`.

[164] K. A. Valiev. “Quantum computers and quantum computations”. In: *Physics-Uspekhi* 48.1 (2005), pp. 1–36.

[165] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, and A. A. Cire. “Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.5 (2021), pp. 3677–3687.

[166] Infineon. *AURIX^{TM} TC39x - B: User's Manual*. 2.0. Infineon Technologies AG. Germany, 2021.

[167] iSYSTEM. *IC5700 Debugger & On-Chip Analyzer User Manual*. iSYSTEM AG. Germany, 2022.

[168] K.-B. Gemlau, L. Köhler, and R. Ernst. “A Platform Programming Paradigm for Heterogeneous Systems Integration”. In: *Proceedings of the IEEE* 109.4 (2021), pp. 582–603.

Acronyms

ABS	Antilock Braking System
API	Application Programming Interface
AUTOSAR	AUTomotive Open System ARchitecture
BCET	Best-Case Execution Time
BET	Bounded-Execution Time
BSW	Basic Software
BTF	Best Trace Format
CLP	Constraint Logic Programming
CP	Constraint Programming
CPU	Control Processing Unit
CRPD	Cache-Related Preemption Delay
CSP	Constraint Satisfaction Problem
DBP	Dynamic Buffering Protocol
DM	Deadline Monotonic
ECU	Electronic Control Unit
EDF	Earliest Deadline First
EMS	Engine Management System
EPS	Electric Power Steering
ET	Event-Triggered
FIFO	First In - First Out
FPS	Fixed-Priority Scheduling
HOPA	Heuristic Optimized Priority Assignment

HP	Hyper-Period
ILP	Integer Linear Programming
IP	Integer Programming
ISR	Interrupt Service Routine
LCM	Least Common Multiple
LDBP	LET Dynamic Buffering Protocol
LET	Logical Execution Time
LIFO	Last In - First Out
MCAL	Microcontroller Abstraction Layer
MILP	Mixed Integer Linear Programming
MIP	Mixed Integer Programming
NBW	Non-Blocking Write
OPA	Optimal Priority Assigment
OS	Operating System
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen
PCP	Priority Ceiling Protocol
PTP	Point-to-Point Protocol
RM	Rate Monotonic
RTE	Runtime Environment
SBP	Static Buffering Protocol
SMT	Satisfiability Modulo Theories
SR	Synchronous-Reactive
SWC	Software Component
TCCP	Temporal Concurrency Control Protocol
TICP	Timed Implicit Communication Protocol

TIMEX	Timing Extensions
TSN	Time-Sensitive Networking
TT	Time-Triggered
TTS	Time-Triggered Scheduling
VFB	Virtual Functional Bus
WCCT	Worst-Case Communication Time
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time
ZET	Zero-Execution Time

List of Figures

List of Tables

List of Algorithms