Analysis of Real-Time Capabilities of Dynamic Scheduled System Applications

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Matthias Freier

aus Gera

Tag der mündlichen Prüfung: 08.02.2016

Erster Gutachter: Prof. Dr. Jian-Jia Chen

Zweiter Gutachter: Prof. Dr. Wolfgang Karl

Contents

Abstract

Future embedded real-time systems are expected to exploit increasingly more computing capabilities to accommodate the increasing application requirements. One of the most complex and computing-intensive real-time systems is the engine control unit of a vehicle, which is implemented by software. An engine control device regulates several sub-systems like fuel injection, exhaust control and diagnostic management. New features are expected to improve the efficiency, like the reduction of CO_2 emissions and fuel consumption. At present, single-core platforms have reached their computing limits due to the power consumption, which is called a *power wall*. The idea is to use multiple processing units with a lower frequency that consume lower power to further increase the computing capability. A so-called *multicore* platform has several cores that can execute the software in parallel and typically share the same memory for communication. If the hardware architecture does not scale with the number of cores, bottlenecks would reduce the effective usable computing capability. A scalable hardware architecture named *manycore* can parallelize the memory and communication accesses among cores by using a Network-on-Chip architecture. A real-time system usually specifies timing requirements e.g. respond within a certain amount of time, called *real-time constraints*, which need to be satisfied. The software of a real-time system comprises tasks characterized by temporal properties to describe the timing requirements of the application. For safety reasons, the scheduling in a real-time system has to be analyzed in terms of its temporal properties to guarantee that the real-time constraints *always* hold for all tasks. The scheduling policy is critical to effectively utilize the multicore and manycore platforms. However, tasks are tightly coupled because controllers exchange a lot of information like measurement values, state information or control parameters. By releasing the tasks in an arbitrary manner, the inter-core communications can cause a large delay because in the worst case all cores may communicate at the same time. Multicore and manycore platforms are used to increase the computing capabilities, although to ensure a safe execution the worst case significantly reduces the computing capabilities. The challenge is to determine a feasible schedule on a multicore or manycore platform for typical industrial real-time applications like an engine control software.

This thesis explores different real-time scheduling approaches to effectively utilize industrial real-time applications on multicore or manycore platforms. The inter-task communications can be explicitly modeled in the so-called *dependent task model*. The proposed scheduling policy is named the *Time-Triggered Constant Phase* scheduler for handling periodic tasks, which determines time windows for each computation and communication in advance by using the dependent task model. These pre-defined time windows can significantly reduce the worst-case behavior, because the communication interference is avoided. In addition, this thesis proposes a Time-Triggered Server to handle urgent sporadic tasks like angle-synchronous tasks of an engine control application. The analysis of approaches to ensure a safe execution of the real-time system is also presented. Experiments confirm that the proposed scheduler is able to highly utilize the platform for typical industrial applications like engine control, while satisfying all real-time constraints.

Zusammenfassung

Zukünftige eingebettete Echtzeitsysteme haben einen steigenden Bedarf an Rechenleistung. Eines der komplexesten und rechenintensivsten Echtzeitsysteme ist das Motorsteuergerät im Kraftfahrzeug, das in Software implementiert ist. Der Rechner eines Motorsteuergerätes regelt verschiedene Teile, wie z.B. die Kraftstoffeinspritzung, die Abgasnachbehandlung oder verschiedene Diagnosesysteme. Neue Funktionalitäten werden erwartet, um die Effizienz noch weiter zu steigern, wie die Reduktion des CO_2 Ausstoßes oder der Kraftstoffverbrauch vom Fahrzeug. Heutzutage stoßen Rechner mit nur einem Rechen-Kern an ihre Leistungsgrenze, da ihre Leistungsaufnahme bei steigender Taktfrequenz nicht mehr abgeführt werden kann. Die Idee ist mehrere Rechnen-Kerne mit einer geringeren Taktfrequenz zu betreiben, um die gesamte Leistungsaufnahme zu reduzieren, jedoch die Rechenleistung weiter zu steigern. Ein sogenannter Mehr-Kern-Rechner hat mehre Rechen-Kerne, die die Software parallel ausführen können und typischerweise über einen geteilten Speicher kommunizieren. Wenn die Hardware nicht mit der Anzahl der Rechen-Kerne skaliert, können Engpässe die effektiv nutzbare Rechenleistung deutlich reduzieren. Eine skalierbare Hardware-Architektur kann sowohl die Speicherzugriffe als auch die Kommunikation zwischen den Rechen-Kernen parallelisieren, indem ein Network-on-Chip verwendet wird. Ein Echtzeitsystem spezifiziert meist Zeitanforderungen, z.B. das Antworten innerhalb einer bestimmten Zeit, die sichergestellt werden müssen. Die Software von Echtzeitsystemen besteht aus sogenannten Tasks, die durch ihre zeitlichen Eigenschaften und den zeitlichen Anforderungen der Anwendung beschrieben werden. Aus Sicherheitsgründen muss das Scheduling der Tasks analysiert werden, und zu garantieren, das die spezifizierten Zeitanforderungen immer erfüllt werden. Dabei spielt die Scheduling Strategie eine wesentliche Rolle, um Mehr-Kern-Architekturen effektiv auslasten zu können. Allerdings sind die Tasks eng miteinander gekoppelt, da die Regler im Steuergerät vieler Informationen untereinander austauschen, z.B. Messdaten, Zustandsinformationen oder Regler Parameter. Wenn die Tasks in einer willkürlichen Art und Weiße aktiviert werden, kann die Kommunikation zwischen den Rechen-Kernen viel Zeit beanspruchen, da im schlimmsten Fall alle Tasks gleichzeitig kommunizieren wollen. Mehr-Kern-Architekturen sollen die Rechenleistung erhöhen, aber um die Zeitanforderungen sicherzustellen, kann der schlimmste Fall die maximale Rechenleistung stark begrenzen. Die Herausforderung besteht darin, einen zulässigen Ausführungsplan aller Tasks für einen Mehr-Kern-Rechner unter Berücksichtigung von industriellen Anwendungen wie einer Motorsteuerung zu bestimmen.

Diese Dissertation erforscht verschiedene Echtzeit-Scheduling-Ansätze, um typische Echtzeitanwendungen auf Mehr-Kern-Rechner effektiv ausführen zu können. Dabei wird die Kommunikation zwischen den verschiedenen Tasks explizit im Task-Modell beschrieben. Als Scheduling Strategie wird ein zeitgetriebenen Scheduler für die periodischen Teile verwendet, der für jeden periodischen Task und deren Datenkommunikation einen festen Zeit-Slot im Voraus bestimmt. Durch diesen festen Zeit-Slots ist der schlimmste Fall deutlich kleiner, da Konflikte in der Kommunikation vermieden werden. Weiterhin erforscht diese Dissertation einen zeit-getriebenen Server-Ansatz, um die nicht periodischen Teile wie der Drehzahl anhängigen Software eine Motorsteuerung ebenfalls effizient auszuführen.

Die Analyse der Ansätze, um die Einhaltung der Zeitanforderungen zu garantieren, wird ebenfalls dargelegt. Experimente haben bestätigt, dass dieser zeitgetriebenen Scheduling Ansatz Mehr-Kern-Architekturen unter Berücksichtigung von typischen industriellen Anwendungen wie einer Motorsteuerung effektiv auslasten kann.

Acknowledgement

This thesis was written during my time as a PhD student at the Corporate Research Department for Advanced Software Systems (CR/AEA) at the Robert Bosch GmbH in Renningen.

My deep graduate goes to Prof. Dr. Jian-Jia Chen for supervising my thesis and for so many fruitful discussions. Even his relocation to the University of Dortmund could not stop our collaboration, which I appreciate a lot.

I would also like to thank Prof. Dr. Wolfgang Karl, who took on my supervision at the Karlsruhe Institut of Technology (KIT). He gave me a lot of helpful advice and helped me to continue my PhD thesis.

My appreciation also extends to my Bosch colleagues especially to Dr. Jochen Härdtlein, Dr. Björn Saballus and Dr. Dirk Ziegenbein for making this thesis possible and their ongoing support. They guided me as a PhD student and helped me with path-breaking decisions.

Thanks also to all members of the Bosch research project *ManyCore* and the Corporate Research Department CR/AEA. They taught me not only technical topics, but also provided useful feedback to improve my skills and personality. I appreciate the friendly and open-minded working atmosphere in our research office, in which I spend most of the time. Especially, I would like to thank to all *ManyCore* PhD students, namely Alexander Biewer, Sören Braunstein, Martin Lowinski, Peter Munk, and Felix Rützel for the open and helpful technical discussions.

Last but not least, I am indebted to my family, who stay by my side trough my entire life. Primarily, my parents and my brother give me uncountable joyful moments in my life.

Ludwigsburg, September 2016 Matthias Freier

Symbols and Abbreviations

Symbols

Symbol	Description
$\mathbf{T}$,$\lvert\mathbf{T}\rvert$	computational task set, number of computational tasks
$\mathbf{K}$,$\lvert\mathbf{K}\rvert$	communication task set, number of communication tasks
$\mathbf{S}$,$\lvert\mathbf{S}\rvert$	sporadic task set, number of sporadic tasks
τ_i	computational task with index i in the set $\mathbf{T}$
κ_j	communication task with index j in the set $\mathbf{K}$
σ_k	sporadic task with index k in the set $\mathbf{S}$
$\mathbf{C}_k$	core with index k on the platform
$\mathbf{R}_m$	switch with index m on the platform
$\mathbf{L}_l$	link with index l on the platform
$\lvert\mathbf{C}\rvert$, $\lvert\mathbf{R}\rvert$, $\lvert\mathbf{L}\rvert$	number of {cores, switches, links } on the platform
$b_{\mathbf{L}}$	common link bandwidth
$d_{\mathbf{L}}$	link delay of the NoC
$d_{\mathbf{R}}$	switch delay of the NoC
P_{τ_i}, P_{κ_j}, P_{σ_k}	period of task $\{\tau_i, \kappa_j, \sigma_k\}$
W_{τ_i}, W_{κ_j}, W_{σ_k}	worst-case execution time (WCET) of $\{\tau_i, \sigma_k\}$ or traversal time of κ_j
D_{τ_i}, D_{κ_j}, D_{σ_k}	relative deadline of task $\{\tau_i, \kappa_j, \sigma_k\}$
Φ_{τ_i}, Φ_{κ_j}	phase of the TTCP scheduler of task $\{\tau_i, \kappa_j\}$
Ψ_{τ_i}, Ψ_{κ_j}	hypothetical phase of the TTCP scheduler of task $\{\tau_i, \kappa_j\}$
$J_{\tau_{i,\ell}}$, $J_{\sigma_{k,\ell}}$	ℓ-th job of task $\{\tau_i, \sigma_k\}$
$a_{\tau_{i,\ell}}$, $a_{\kappa_{j,\ell}}$, $a_{\sigma_{k,\ell}}$	arrival time of the ℓ-th job or packet of task $\{\tau_i, \kappa_j, \sigma_k\}$
$s_{\tau_{i,\ell}}$, $s_{\kappa_{j,\ell}}$, $s_{\sigma_{k,\ell}}$	starting time of the ℓ-th job or packet of task $\{\tau_i, \kappa_j, \sigma_k\}$
$f_{\tau_{i,\ell}}$, $f_{\kappa_{j,\ell}}$, $f_{\sigma_{k,\ell}}$	completion time of the ℓ-th job or packet of task $\{\tau_i, \kappa_j, \sigma_k\}$
r_{κ_j}, $\lvert r_{\kappa_j}\rvert$	route of task κ_j, route length of task κ_j
τ_{SRC_j}, τ_{DST_j}	source or destination computational task of task κ_j
$M_{i,j}$	conflict matrix of two communication tasks κ_i,κ_j
$\pi_{\sigma_k}^{\mathbf{C}_0}$, $\pi_{\sigma_k}^{\mathbf{C}_1}$	priority of the typical- or exceptional-case execution part of σ_k
O_{σ_k}	relative offset of the exceptional-case execution part of σ_k
$\overline{D}_{\sigma_k}$	effective relative deadline
$R_{\sigma_k}^{\mathrm{Typ}}$, $R_{\sigma_k}^{\mathrm{Exc}}$	WCRT of the typical- or exceptional-case execution part of σ_k
R_{κ_j}, R_{σ_k}	WCRT of task $\{\kappa_j, \sigma_k\}$
W_{TTS}, P_{TTS}, Φ_{TTS}	length, period or phase of the TTS
$d_{\mathrm{TTS}}(t)$, $d_{\mathrm{TTS,max}}$	slot-shifting delay, maximum slot-shifting delay
$V(d_{\mathrm{TTS}})$	recovering time for a slot-shifting delay d_{TTS}
U_τ, U_κ, U_σ	utilization of the task set $\{\mathbf{T}, \mathbf{K}, \mathbf{S}\}$
Ω_τ, Ω_κ	task order of the computational $\mathbf{T}$ or communication task set $\mathbf{K}$
H	hyper-period

Abbreviations

AUTOSAR AUTomotive Open System ARchitecture

BCET best-case execution time

CAN Controller Area Network

COTS commercial off-the-shelf

DAG directed acyclic graph

DM Deadline Monotonic

DM-DS Deadline Monotonic with Density Separation

ECU Electronic Control Unit

EDF earliest-deadline-first

EDPA Effective Deadline aware Priority Assignment

FCFS first-come-first-service

FP Fixed-Priority

G-EDF Global Earliest Deadline First

GRMS generalized rate monotonic scheduling

HPF-NB Higher Periods First with Nested Bin-Packing

ILP Integer Linear Programming

LPF Lower Periods First

NI Network Interface

NoC Network-on-Chip

NPS-F Notional Processor Scheduling Fractional capacity

OSEK Open Systems and their Interfaces for the Electronics in Motor Vehicles

PDMS HPTS Partitioned Deadlinemonotonic Scheduling by allowing the Highest-Priority Task on a Processor Core to be Split

REETIC Real-Time Scheduling for Exploiting the Typical- and Worst-Case Execution Times

RM Rate Monotonic

RMnP Rate Monotonic non-Preemptive

RR Round Robin

RTC Real-Time Calculus

SMT Satisfiability Modulo Theories

SQ Sporadic Queue

TDMA Time Division Multiple Access

TTCP Time-Triggered Constant Phase

TTS Time-Triggered Server

WCET worst-case execution time

WCRT worst-case response time

WCTRT worst-case traversal response time

1. Introduction

An embedded system is an information processing system with a particular purpose to interact with the physical environment. In contrast to general purpose computers, embedded systems have additional requirements like reliability, timing or energy consumption. For example, modern vehicles use embedded systems to control features like the Electronic Stabilization Program (ESP) or moving the windshield wiper.

Real-time systems are special embedded systems that need to fulfill their services within a certain amount of time. If a real-time system does not respond within a time limit (deadline), the result is less beneficial or completely useless. For example, an airbag system has to respond within a few milliseconds (e.g. $1ms$). If the system responds too late, a catastrophic consequence could happen, which contradicts the purpose of this system.

1.1. Complex Industrial Applications with an Example of Engine Control Software

This thesis focuses on general real-time systems but considers engine control applications in particular as a demonstrator for a complex industrial application. The engine control application is one of the most complex and computing-intensive real-time systems in modern cars [18]. Other examples of complex real-time applications are autonomous driving or driver assistance systems. Approaches are also assumed to be applicable for less complex systems.

An engine of a vehicle is a complex and sophisticated part of a vehicle, as presented in the literature [81]. Many stakeholders intend to improve the next generation of engines for better fuel efficiencies, less CO_2 emission and other new features. Especially the European Union (EU) releases laws [92] to restrict CO_2 emissions, which have a vast impact on the requirements of new combustion engines. Thus, the engine and its surrounding sub systems become a complex application, which is controlled by an Electronic Control Unit (ECU). Different types of a vehicle model further increase the complexity, because they have different configurations. Figure 1.1 shows an abstract view of a combustion engine, representing a simplified overview of an engine and its components.

For example, a throttle device needs to control the amount of air (oxygen) required for the combustion of fuel. The throttle controller needs to sample the air mass in a fixed rate and calculates the angle for the throttle to set a specific amount of air. All components need to be controlled, although high-level controllers also exist with more sophisticated functions, like adaptive cruse control or a lane-keeping assistant [88]. The challenge is to control these applications to ensure that they meet their requirements.

Engine control applications are dominated by the control theory with a closed or open feedback loop. There exist many nested and inter-dependent controllers to fulfill the requirements for next generation engines. The control paradigm demands sampling, processing and responding at a certain rate whereby the system can be controlled in a stable manner. There are controllers with a fixed rate like a throttle controller and those with a crankshaft-synchronous (angle-synchronous) rate like the fuel injection. Besides the controllers, there are several sensors and actuators that need to be sampled and respond to the engine, as shown in Figure 1.1.

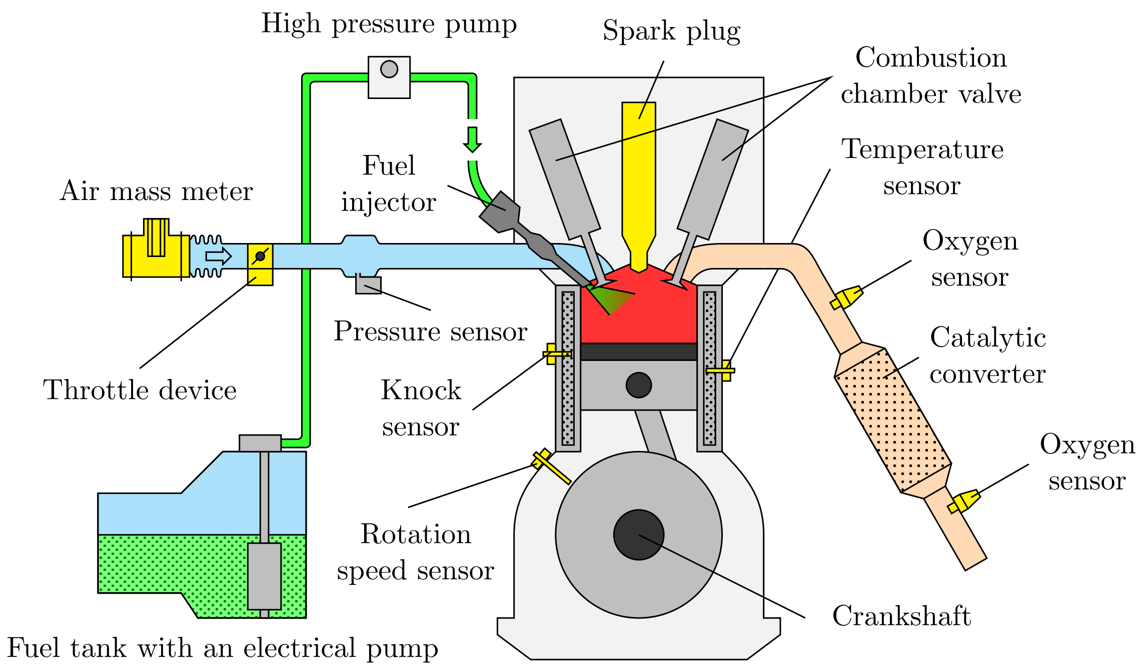


Figure 1.1.: Overview of the combustion engine and its controllable components. All components need to be controlled by an engine control unit. The figure and names are explained in detail in the literature [81].

Due to this complex structure, the controllers of the engine are implemented by real-time embedded software on an ECU. The software of such a control-dominated application is modeled by *real-time tasks*, which represent the different implemented controllers. A *task* is an abstract representation of a part of the software, which is executed in a periodic manner. The control software is tightly coupled because the controllers exchange information like measurement values, state information or control parameters, which are visualized by Figure 1.2. The rate of a controller determines the period of a task. Many tasks have a fixed period, although angle-synchronous tasks also exist. An angle-synchronous task is triggered by a specific angle of the crankshaft, after which the fuel is injected and combustion is ignited, considering the crankshaft rotation speed.

The development of an engine control software is a challenge with respect to new feature and improvement requests. Based upon these requests, more computing capabilities are necessary to implement more advanced features and support more efficient engines [18]. For example, more accurate models to estimate the required fuel or additional components in the catalytic converter could be implemented.

1.2. Real-Time Multicore / Manycore platforms

Another challenge is the development of the platform (ECU) to enable more computing capabilities' for the real-time software. Due to the complexity and their computing capabilities demand, single-core platforms have reached their limits. A limit in the power consumption of single-core platforms prevents satisfying the demand of more computing capabilities, which is called a *power wall* [91]. An increased clock frequency significantly increases the power consumption of a chip, which overheats, if the power cannot drain. Due to the cooling, a chip has a maximum power consumption, which also limits the maximum clock frequency.

The idea is to use multiple processing units with a lower processing frequency, which results in lower power consumption to further increase the computing capabilities. These

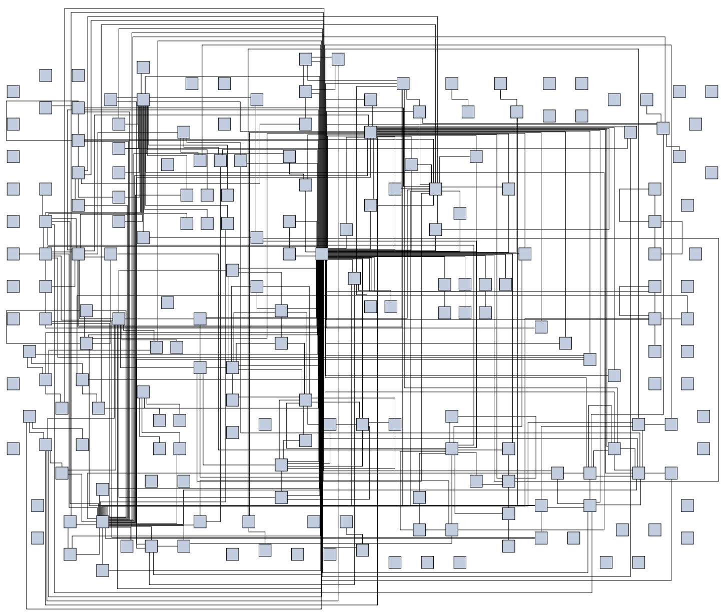


Figure 1.2.: The software on an engine control can be visualized by a graph, where the nodes are tasks and the edges are data communications among tasks. A part of the task graph of a current engine control application [34] with 206 tasks and 334 communications is shown.

platforms are called *multicore*, because they use multiple *cores* to execute the software. A *core* represents the processing unit, although it also has its own local memory and a communication interface to support parallel software execution. According to the common sense definition, a *multicore* is a hardware architecture class that comprises multiple cores, although uncountable implementation possibilities exist. In order to enable more computing capability, the embedded systems industry has begun to develop products with *multicore* platforms [18, 62]. One challenge is to provide a real-time capable multicore hardware architecture that can effectively be used by industrial software applications.

The communication fabric and the memory architecture of a multicore platform are critical to enable more computing capabilities than a single-core and scale with the number of cores. The memory accesses and the communication fabric also need to be parallelized; otherwise, the increase computing capabilities cannot effectively be used. For example, suppose a multicore platform with four cores and one shared memory (flash) to store the program code. If each core is constantly activated, all cores fetch their program code from the shared memory. Simple memories can only be accessed by one core at a time. Thus, each core has to wait on average 75% of its time to obtain the program code for execution. Therefore, the memory access and the communication also need to be parallelized.

A simple approach is that each core has its own local memory to store program code and temporal data. By considering a scalable communication fabric, a *Network-on-Chip (NoC)* architecture is used to parallelize inter-core communications with moderate hardware costs. In this thesis, a scalable hardware platform is named *manycore*. Figure 1.3 shows an example of a multicore and a manycore platform. The challenge is to effectively utilize a manycore platform with a typical industrial application. This thesis focuses on multicore and manycore platforms because these platforms are promising approaches to enable more computing capabilities for real-time applications like an engine control software.

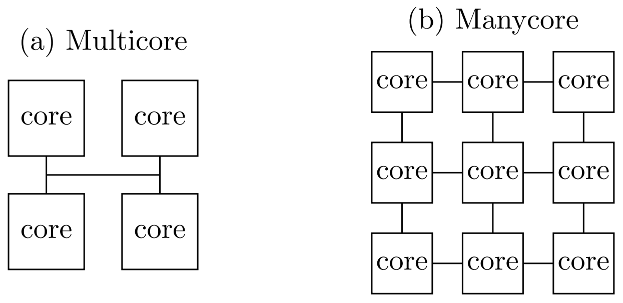


Figure 1.3.: Example of a multicore and a manycore platform. Both platforms have multiple processing units to enable more computing capabilities. By contrast, a manycore platform has a scalable hardware architecture with a Network-on-Chip (NoC) to avoid bottlenecks in the inter-core communications. Thus, the manycore platform is capable of handling a large number of cores.

1.3. Real-Time Scheduling

The software of a real-time system comprises tasks characterized by temporal properties and timing requirements of the application. A platform with multiple cores allows more opportunities to execute tasks. The scheduler is part of the operating system and determines the timing and the execution order of the tasks [5, 73].

A real-time system usually specifies timing requirements e.g. responding within a certain amount of time called *real-time constraints*, which need to be satisfied. These constraints have a vast impact on the scheduling of real-time systems, because tasks need to be scheduled to ensure they can fulfill their real-time constraints. For safety reasons, the temporal properties of the scheduling in a real-time system have to be analyzed to guarantee that real-time constraints *always* hold for all tasks, which is called *scheduling analysis*. One important real-time constraint is a deadline of a task, which defines the maximum tolerable time between the activation and the completion of a task. For example, an airbag system has a deadline of a few milliseconds (e.g. $1ms$), which is a relative definition between the activation and the response to release the airbag.

There exist many principles to schedule tasks, which are called *scheduling policies*. For example, a well-known scheduling policy is the Fixed-Priority (FP) policy, where each task has a priority level and the activated task with the highest priority is executed. On a single-core platform, scheduling policies are well explored [17].

On a multicore platform the scheduling is more complex because the mapping of tasks to cores and interactions between cores also need to be considered. Thus, more advanced scheduling approaches have to be developed to handle multicore and manycore platforms. The scheduling of these tasks is essential to effectively utilize multicore and manycore platforms.

Real-time scheduling is usually divided into an off-line part and an on-line part. In the off-line part, the timing parameters for the scheduling policy are determined and the scheduling analysis is performed to guarantee the correct timing behavior. For real-time systems, this guarantee is important to avoid incidents where human lives are endangered. In the on-line part, the scheduling policy decides the task dispatching, i.e. the task execution decision in time and space.

There exist many different approaches to schedule real-time applications on multiple cores [24]. For example, a global scheduler has a global queue to dispatch tasks to available cores in a global manner, in which tasks can migrate to other cores for improving the platform utilization. Another approach is to use a partitioned scheduler and apply well-known scheduling policies for the single-core platform. Each core has its own local scheduler, although the initial task-to-core mapping has to be computed.

Industrial applications typically use a Rate Monotonic (RM) scheduler based upon the Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK) operating system for single-core platforms. For multicore platforms, proposed scheduling approaches are difficult to adapt for industrial applications like an engine control software. The reason is that most approaches assume independent tasks, which is generally over-simplified in many scenarios. According to Figure 1.2, tasks communicate a lot of data among each other. On a single-core platform, the data dependencies do not result in run-time overhead, because the processing unit can only execute tasks one by one such that the execution order ensures no memory access conflicts. The challenge is to find a feasible schedule on a multicore or manycore platform for typical industrial real-time applications like an engine control software.

The inter-core communications can cause a large delay because in the worst case all cores may communicate at the same time. By releasing the tasks in an arbitrary manner, the scheduling analysis becomes pessimistic because all inter-core communication may interfere with each other. This pessimism can significantly reduce the maximum platform utilization to guarantee a feasible schedule of a real-time application. Multicore and manycore platforms are used to increase the computing capabilities, although to ensure a safe execution the worst case can significantly reduce the computing capabilities. This contraction leads to another scheduling policy, which is based upon the time-triggered principles.

A time-triggered scheduler determines the schedule in advance such that the worst-case interference can be less pessimistic. Regarding typical industrial applications like an engine control software, it is challenging to determine a time-triggered schedule that can satisfy the requirements. Especially in manycore platforms with a NoC, the analysis of interferences between inter-core communication can require a complex algorithm [23]. In the communication fabric, there is a similar contradiction because a NoC is used to allow more communication at the same time, although the worst-case analysis can significantly reduce the maximum utilization of the communication fabric.

1.4. Focus of this Thesis

This section summarizes assumptions and problems of this thesis.

Assumptions of this thesis

In order to find a feasible schedule, this thesis makes some assumptions about the application, which are described in the following. The scheduler can only manage individual tasks, although in real applications a task usually comprises many software units. The AUTomotive Open System ARchitecture (AUTOSAR) standard [5] names the software units as *Runnables* and a task contains several *Runnables*. For single-core platforms, this concept reduces the inter-task dependencies but limits the parallel task executions on a multicore platform. This thesis exploits the parallelism of the individual software units (*Runnables*) by re-defining tasks.

Definition 1. *(Task)* Each software unit is defined as an individual task.

Considering Definition 1, the number of tasks is assumed in the range of 100–$1,000$. This large range is caused by different requirements of the engine control software. For example, a smaller vehicle with a simpler and less powerful engine has fewer tasks than a high-end vehicle with turbo charging and other advanced features [81].

Due to typical industrial applications, Kramer et al. [49] published numbers of generic task sets used in the engine control applications. They state that the maximum number of tasks is $1,000$–$1,500$ and the maximum task has 3% utilization. The small maximum task utilization implies that no heavy-utilized task exists, which simplifies the problem to determine a feasible schedule. Angle-synchronous tasks have a stringent real-time constraint, i.e the relative deadline of an angle-synchronous task is smaller than its period, because fuel injection and combustion ignition need to be undertaken relative to the crankshaft position. The crankshaft rotates around 30 degrees from the angle-synchronous activation until the relative deadline of the angle-synchronous tasks. Such an *a priori* not-known activation pattern is called *sporadic* activation.

The periods of the tasks are assumed to be harmonic, i.e. each period is an integer multiple of all lower periods. From the technical perspective, industrial characteristics [49] show that the periods are modifiable to be harmonic with minor changes, which significantly simplifies the scheduling analysis. In a control application, the period of a task is set by the application of Shannon's sampling theorem [56]. The sampling theorem is implemented such that the period of a controller is set to $(1.5 \dots 15)P_{\max}$, where $P_{\max}$ is the maximum observable period in the real system. The adjustment of periods to be harmonic can add some overhead, which can artificially increase the platform utilization. However, the problem of finding a feasible schedule is simplified for harmonic periods, which can increase the maximum feasible utilization and reduce the run-time complexity of the analysis.

Tasks communicate with each other to exchange data. The scheduler needs to ensure that the data of each task is available before it can be executed. Thus, a data communication can have real-time constraints, because the execution of a task can depend on its receiving data. There exist urgent data communications, which represent a *precedence constraint* between two tasks. A precedence constraint is a real-time constraint, which determines an execution order by considering a certain interval.

The scheduling relevant assumptions are summarized in the following:

- The software comprises tasks that are *a priori* defined.
- Tasks have harmonic periods (activations).
- Rarely, tasks are sporadic-activated like angle-synchronous activated tasks.
- There exists a large number of tasks (100–$1,000$), which have a large number of data communications (300–$3,000$).
- Some data communications are urgent, which represent precedence relations between any two tasks.
- There are only non-heavy-utilized tasks.

Problem statements

The problems addressed by this thesis are stated in the following:

The *scheduling policy selection problem* is to select one scheduler that can schedule a typical industrial application like an engine control application on a multicore/manycore platform with a high platform utilization ($\geq 50\%$) and satisfy all real-time constraints.

The *scheduling analysis problem* is to guarantee that all real-time constraints of typical industrial application like an engine control application on a multicore/manycore platform always hold for a certain scheduling policy, which has to run with polynomial or pseudo-polynomial time complexity.

The *scheduling design problem* is to determine parameters of a scheduling policy such that all real-time constraints of typical industrial applications like an engine control application on a multicore/manycore platform hold, which has to run with polynomial or pseudo-polynomial time complexity.

Note that the run-time complexity has to be polynomial or pseudo-polynomial time complexity, because there exists a large number of tasks ($1,000$).

1.5. Contributions and Thesis Overview

This thesis explores different real-time scheduling approaches to effectively utilize typical industrial real-time applications on multicore or manycore platforms. Figure 1.4 provides an overview of the chapters of this thesis. First, related background knowledge is given in Chapter 2.

The idea is to separate tangled tasks into periodic and sporadic parts, because the schedule of periodic task can be determined *a priori* to simplify the scheduling analysis. The inter-task communications are explicitly modeled in the so-called *dependent task model*, as presented in the system model of Chapter 3. There are plenty of opportunities to realize multicore platforms in hardware. This thesis defines two different platforms namely *multicore* and *manycore* which represent a simple and a scalable platform with a NoC, respectively.

Based upon these models, this thesis proposes a Time-Triggered Constant Phase (TTCP) scheduling approach to handle periodic tasks. A TTCP scheduler determines a periodic time window for each task, in which a task is allowed to be executed. These time windows are *a priori* defined such that the time for the communication can statically be reserved, which significantly reduces the worst-case behavior. This reduces the complexity of the scheduling analysis because the periodic behavior can be exploited. In addition, the TTCP scheduling approach reduces the number of values that need to be stored for the scheduler implementation. Experiments confirm that this scheduler is able to strongly utilize the platform for typical industrial applications like engine control software. The

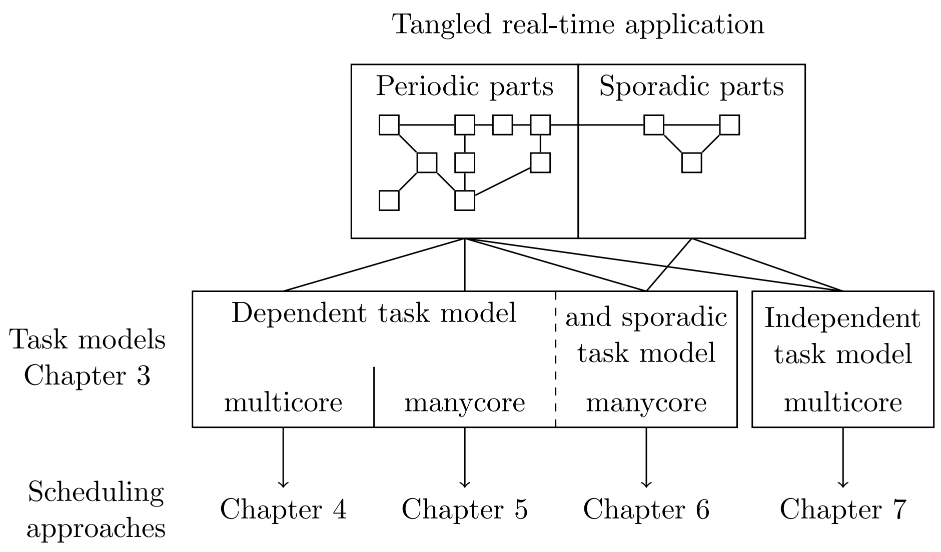


Figure 1.4.: Overview of chapters of this thesis. The applications comprise period and sporadic parts. The tangled tasks can be modeled by the dependent and the independent ask model. Each chapter propose a solution for one of these cases.

TTCP scheduling approach is applied on a multicore platform (Chapter 4) and a manycore platform with a NoC (Chapter 5).

Chapter 6 presents an approach to handle periodic and sporadic parts with the TTCP scheduling approach. The idea is to schedule the sporadic-activated tasks in a periodic time window and delay periodic tasks if an urgent sporadic task arrives. The chapter contains the scheduling analysis of periodic and sporadic tasks to ensure that all real-time constraints hold.

The well-known independent task model can be applied if the inter-task communication is hidden in the execution time of each task, as presented in Chapter 7. On a multicore platform, inter-task communication can interfere with each other in the worst case. The chapter presents a scheme that exploits the typical execution of tasks and reserves a core only to bound the worst-case behavior.

The main contributions of this thesis are described in the following:

- A dependent task model to describe the inter-task communication with timing properties, which represents typical industrial real-time applications.
- A scheduling policy named TTCP scheduler capable of handling tangled tasks on a multicore and manycore platform.
- The scheduling analysis of the TTCP scheduler for a multicore and a manycore platform with a NoC that can be performed in polynomial time complexity.
- A scheduling design approach to determine the parameters of a TTCP scheduler that can be performed in pseudo-polynomial time complexity and can strongly utilize the platform ($\geq 50\%$).
- An approach named *slot-shifting* to handle and increase the responsiveness of sporadically-activated tasks like angle-synchronous tasks from engine control by using a Time-Triggered Server (TTS). The scheduling analysis and design is also presented.
- An approach to exploit typical cases of task executions if the data communication is hidden as part of the execution time of each task.

2. Background

This chapter provides an overview of the state-of-the-art, which is a prerequisite for this thesis. These topics are traditional and multicore-related scheduling approaches (Section 2.1 and 2.2), and an exploration of NoC design space (Section 2.3).

2.1. Traditional Scheduling Concepts for Single-Core Platforms

This section summarizes the common scheduling knowledge [17, 53] used to analyze the real-time capability of a real-time embedded system. These concepts originate from traditional scheduling on single-core platforms or from multicore-related problems [6, 17, 24, 53].

Fundamental scheduling concepts

A *scheduler* is a logic inside of the operating system, which determines the execution order and the timing of the software. Software applications are modeled by tasks, which are handled by the scheduler. In most operation systems, there is a *task queue*, which contains the tasks that are *ready* to be executed. The scheduler decides, which task is dispatched and executed by the processing unit. The procedure to start the execution of a specific tasks is called *task dispatching.* A *scheduling policy* is the theoretical principle of a scheduler to decide upon the dispatching of tasks.

If a very urgent task is added to the task queue, the scheduler can preempt the current running task to execute the urgent task first. After finishing the urgent task, the preempted task is again dispatched to the processing unit. This procedure is called *preemption.*

A real-time system is a system that needs to respond within a guaranteed upper-bounded time. To guarantee an upper-bounded response time, the scheduler needs to ensure specific temporal properties of the application, called *real-time constraints.* The common concept defines *deadlines* as an upper-bounded time limit for each task. If each task completes before its deadline, the real-time constraints of the system are fulfilled. Real-time systems are classified into hard and soft real-time systems. A hard real-time system means that if the system responds too late, human life can be in danger. A soft real-time system means that if the system responds too late, the result loses its utility. Other real-time constraints are precedence or resource constraints.

In order to guarantee the real-time constraints, tasks in a real-time system are usually described by an abstract *task model.* A *task model* is an abstract description of an application software part, which is used to guarantee the real-time constraints of the system. Figure 2.1 shows the basic terms that describe a task. In the literature, there are many task models, like the periodic task model or the sporadic task model [90].

A task is activated in cyclic manner, which is described by realising an infinite number of task instances (jobs). A *task* activation means that the task is added to the task queue at its arrival time for executing a specific job. The activations can be periodic or sporadic, characterized by a period or a minimum-inter-arrival time, respectively. A period or a

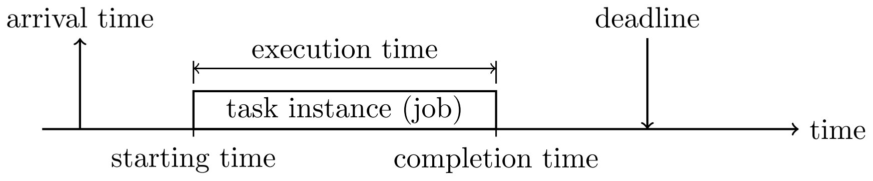


Figure 2.1.: The basic task model for a real-time scheduler

minimum-inter-arrival time describe the time difference between two consecutive released jobs. The deadlines are usually defined relative to the arrival time, called the *relative deadline*. In the literature [53], the deadlines are classified into:

- *implicit deadlines*, for which the relative deadline equals to the period;
- *constrained deadlines*, for which the relative deadline is smaller or equal to the period; and
- *arbitrary deadlines*, for which the relative deadline has no limitation.

Scheduling policies

There exist many principles to schedule tasks and decide the task for the dispatching, called *scheduling policies*. These scheduling policies are classified in the following types:

- *Cyclic Scheduling Policy:* This policy determines all scheduling decisions in advance, which is also-called static scheduling. One simple approach is to store all jobs in a list such that the scheduler executes the job that is the next element in the list, named *list scheduling* [1]. Another approach is to define the starting times of each job *a priori*, named *time-triggered scheduling*. The advantage is the predictable execution pattern, whereby timing properties can be easily derived in advance. The disadvantage is the inflexibility to react to urgent tasks that arrive during run-time.
- *Fixed-Priority (FP) Scheduling Policy:* This policy defines a priority level for each task. Based upon all active tasks that need to be scheduled, the task with the highest-priority level is executed. The different scheduling policies define the priorities with a certain method. For example, a Rate Monotonic scheduler defines the priority levels for each task according to the period (rate) of a task, in which the task with the smallest period has the highest priority [17]. The Rate Monotonic scheduler is often used in industries like automotive applications, which are stated in OSEK operating systems for single-core platforms [73]. This scheduler can react quickly to urgent-arriving tasks like interrupts. If the task with the highest priority has to be scheduled immediately, it can disturb the current executed task, which cases some run-time overhead. The overhead includes the mechanism to *preempt* a task and calculate the new scheduling decision.
- *Dynamic-Priority Scheduling Policy:* The priority levels can be changed during run-time, which is called dynamic-priority scheduling policy. A well-known example is the earliest-deadline-first (EDF) scheduling policy [59]. A EDF scheduler determines the priority level of each task according to its absolute deadline, in which the tasks with the earliest absolute deadline have the highest priority. For a single-core platform, the EDF scheduler is proven to be optimal, i.e. if there exists a feasible schedule for a certain task set, the EDF scheduler provide a feasible schedule by considering some assumption for the task set [59].

Scheduling Analysis

Based upon the task model, the real-time system needs to ensure that the real-time constraints are satisfied (feasibility). The *scheduling analysis* is the procedure to verify that all real-time constraints *always* hold for all tasks, even in the worst case. The problem is that each scheduler which follows a specific scheduling policy needs a customized scheduling analysis.

One possibility is to test all possible scheduling combinations of the application and check whether any of them fail. However, this simple approach is not realizable because even small task sets result in too many possibilities to execute the tasks. Another approach is to prove in a formal way that all real-time constraints hold for all cases. Therefore, often it is possible to define a *critical instant theorem* namely a certain task activation pattern which represents the worst-case timing for a specific task. For example, Liu and Layland [59] present the critical instant theorem for single-core platforms scheduled under a fixed-priority scheduler:

Theorem 1. *(critical instant theorem [59]) A critical instant for any task occurs whenever the task is requested simultaneously with a request for all higher-priority tasks.*

Proof. Proven by Liu and Layland [59]. □

Without a *critical instant theorem* for a certain task for a platform model, it is usually possible to find a safe upper bound. This upper bound might never appear in the system, although it is pessimistic enough to sufficiently guarantee safe the executions of tasks, which hold their real-time constraints.

Typically, the scheduling analysis is performed in the two following steps:

1. *WCET Analysis:* Determine the worst-case execution time (WCET) for each task.
2. *WCRT Analysis:* Determine the worst-case response time (WCRT) for each task, which needs to be smaller than or equal to the relative deadline.

The WCET analysis determines the maximum possible time to execute a certain task, which strongly depends on the platform. In the literature [96], the WCET analysis is often referred as an estimation, because the exact WCET is usually not derivable. First, the binary code a task is disassembled into basic blocks, which compose a graph of all possible program paths. This graph also contains all execution branches of a program, which come from *loops* and *if* statements. Second, with a concrete model of the processing unit, each basic block gains an upper-bounded execution time. During this step, the memory accesses, the instruction set of the processing unit and many other platform-related features influence the time of each basic block. Third, the graph of basic blocks is analyzed to find the longest path in the graph, which determines the WCET. Note that the interference of other tasks is not considered in this WCET analysis. There are several tools for determining the WCET, like aiT [29], boundT [40], Chronos [58] or RapiTime [80].

Subsequently in the WCRT analysis, the WCET and the scheduling policy are used to calculate the maximum time for executing a task by considering the worst-case interference of the other tasks. The worst-case interference occurs in the critical instant. For example, a task scheduled by a fixed-priority preemptive scheduler can be blocked by other higher-priory tasks. This blocking by other tasks increases the WCRT. One important timing constraint is the deadline. If the WCRT for each task is smaller or equal to its corresponding relative deadline, the deadline constraint holds.

2.2. Overview of Multicore-Related Scheduling Concepts

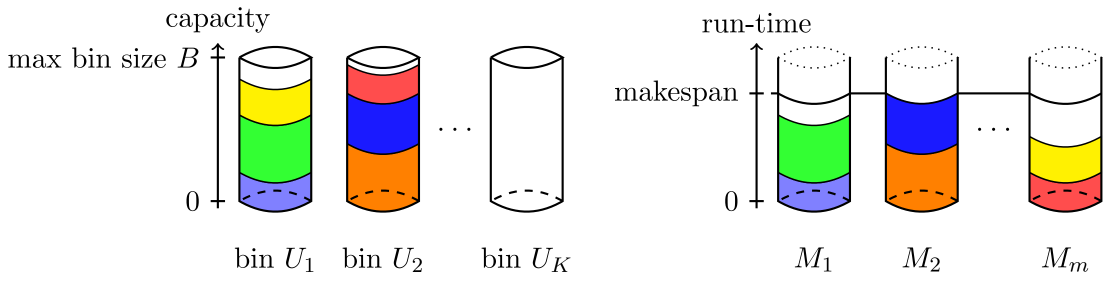
If this two-step approach from the previous section is applied to a multicore or manycore platform, the WCET analysis can become very pessimistic because the memory access time strongly depends on other cores. In the worst-case analysis, this pessimism eliminates the performance improvement for multicore platforms in comparison to a single-core platform. Therefore, this thesis applies the classical two-step approach by making the WCET analysis independent from the inter-core communication and assuming core-local memories. Thus, it is possible to obtain a tight WCET, in which the interference is decoupled by the scheduling policy. Suppose that the memory is not assigned to a dedicated core. In the worst case, all cores may access this memory at the same time, which significantly restricts the parallel execution due to sequential memory accesses. Note that a motivational example in Chapter 7 shows the impact of a multicore platform on the WCET estimation.

There exist many different approaches to schedule real-time applications on multiple cores [24], which are categorized in the following.

Multicore Scheduling Categories

In general, there exist many scheduling approaches for handling the tasks on a multicore platform, which can be classified into the following three scheduling categories [24]. The scheduler is part of the operating system and determines the timing and execution order of the software.

- Global Scheduling: All tasks that need to be executed are collected in one global queue. A global scheduler assigns the jobs dynamically to all available cores based upon a global strategy, like Global Earliest Deadline First (G-EDF) [37] or Deadline Monotonic with Density Separation (DM-DS) [8]. The advantages are the usage of global optimization strategies like load balancing or energy efficiency by turning off some cores, as well as a theoretical higher system utilization. One problem of the global scheduling is the run-time overhead for managing the global queue and migrating the tasks, which may become a bottleneck considering increasingly more cores.
- Partitioned Scheduling: In contrast to a global scheduler, each task is statically mapped to a core. Each core has its own queue for executing its corresponding tasks. A partitioned scheduler can be implemented like a single-core scheduler and thus well-known scheduling strategies can be applied. On the one hand, the fixed task-to-core mapping reduces the maximum possible system utilization and prohibits dynamic adaptations during run-time, because the fixed mapping can cause irregular load balancing. On the other hand, the fixed task-to-core mapping ensures more predictability, which results in less pessimism in the communication analysis, because the communication paths are also static.
- Semi-Partitioned Scheduling: This scheduling category is a mixture of the global scheduling and the partitioned scheduling. Each core has its own task queue, although tasks are allowed to migrate between the cores in a limited way. There exist many semi-partitioned scheduler approaches like Notional Processor Scheduling Fractional capacity (NPS-F) [13], Partitioned Deadlinemonotonic Scheduling by allowing the Highest-Priority Task on a Processor Core to be Split (PDMS HPTS) [50] and Carousel-EDF [89]. The limited migration reduces the run-time overhead and makes the schedule more predictable in comparison to a global scheduler. Tasks can be split into parts, which are run on different cores to increase the maximum possible utilization. In contrast to partitioned scheduling, the task migration significantly increases the effort in the scheduling analysis, especially if communicating tasks are considered.



(a) bin packing problem (b) makespan problem

Figure 2.2.: Bin packing and makespan problem

This thesis presents a partitioned scheduler called a Time-Triggered Constant Phase (TTCP) scheduler, in which the tasks are statically mapped to cores. This scheduler increases the predictability, which helps in the communication analysis to obtain tight communication latencies. The task migration has a vast impact on the scheduling analysis [71], e.g. the analysis needs to be conducted during run-time with a small run-time overhead or the WCET analysis may need to consider different core architectures.

Task partitioning problem and typical approaches

By introducing multicore platforms, the problem of mapping the tasks to a certain core appears. This problem is well known in the literature and can be solved by adapting the solutions for classical problems named the bin packing problem or makespan problem. This thesis identifies different packing-related problems to adapt solutions that are well known in the literature.

The bin packing and makespan problems have evolved from the job machine scheduling problem [39]. It comes from the problem to optimize the production processes in factories. These classical problems are defined in the following:

Definition 2. *(Bin packing problem by [36])*
INSTANCE: Finite Set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, a positive integer bin capacity B, and a positive integer K.
QUESTION: Is there a partition of U into disjoint sets $U_1, U_2, \dots, U_k$ such that the sum of the sizes of the items in each U_i is B or less?

Definition 3. *(makespan scheduling problem by [57, 85])*
Suppose a given set of n jobs $\{J_1, \dots, J_n\}$ and m machines $\{M_1, \dots, M_m\}$. Each job can only run on one machine at same time and each machine can process at most one job at same time. If a job J_j is processed on a machine M_i, it takes a processing time p_{ij}.
The makespan C_{max} is time between the minimum arrival time of all jobs and the maximum completion time of all jobs. The problem is to find a schedule such that the makespan C_{max} is minimized.

These two problems are illustrated in Figure 2.2. The complexity of both problems is proven to be $\mathcal{NP}$-hard in the strong sense [36].

Due to the complexity of the problem, there are several heuristic approaches [4, 15, 85] to obtain good but not optimal solutions in polynomial time complexity, which are listed in the following:

- First-Fit: Pack the item into the first bin in the list in which the item fits.

- Next-Fit: Pack the item into the next bin in a cyclic list in which the item fits.
- Worst-Fit: Iterate over all available bins and pack the item into the bin that has the highest remaining bin capacity.
- Best-Fit: Iterate over all available bins and pack the item into the bin that has the lowest remaining bin capacity and into which the item fits.

Note that each heuristic can sort the items according to its size in order to increase the packing quality. The heuristic approaches can also be analyzed to guarantee a certain performance [15].

This thesis exploits the worst-fit and first-fit heuristics to determine the parameters for the Time-Triggered Constant Phase (TTCP) scheduling approach. In addition, the worst-fit heuristic is used to determine the task-to-core mapping dependent task set generator. By considering one bin for each core on a multicore platform, these heuristics can calculate the task-to-core mapping. This thesis presents solutions for extended bin packing problems, which relies on these traditional solutions.

2.3. Network-on-Chip (NoC) Design Space

This section describes the Network-on-Chip (NoC) assumptions, which are used for the *manycore* platform in this thesis. A NoC is a scalable communication fabric to exchange data among cores. In general, a NoC can be abstracted, represented by a graph in which the nodes are cores or switches and the edges are physical connections between nodes, as shown in Figure 2.3.

The analysis of the inter-core communication strongly depends on the arbitration logic and mechanism of the NoC. Hence, the NoC model significantly matters. This thesis only deals with one particular NoC but examines the different NoC design spaces in the following.

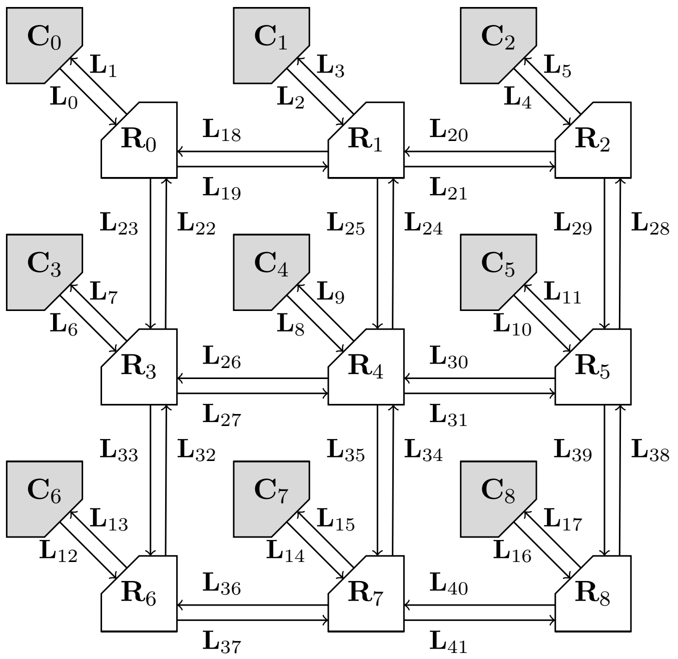
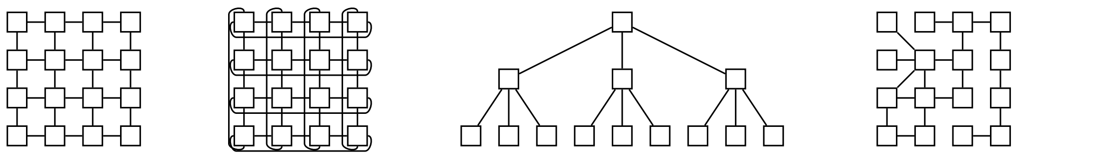


Figure 2.3.: The platform example shows a manycore platform with nine cores connected by a 3 × 3 2D-mesh NoC, which is used in this thesis (Section 3). The NoC comprises cores **C**, switches **R** and links **L**.

Topology

There are different possibilities to connect the nodes on a NoC to each other. Figure 2.4 shows several NoC topologies that are conceivable for a scalable architecture. The mesh and the torus structure are simple to layout on a chip, because the 2D structure can easily be implemented on a chip. Therefore, these topologies already exist in commercial off-the-shelf (COTS) hardware, like the Epiphany® chip [2]. This thesis assumes a mesh structure, because it is commercially available.



(a) Mesh (b) Torus (c) Tree (d) Arbitrary topology

Figure 2.4.: There exists several possible NoC topology types, like a mesh, torus, tree or some arbitrary types. The boxes indicate cores inside the NoC and the edges indicate a physical connection between two cores.

Another topology is a tree structure, which reduces the number of edges in the graph. Considering the motivation in the introduction, the NoC is used to parallelize the communication. In a tree, bottlenecks are likely to appear unless the application can be easily partitioned into different independent branches of the tree. Arbitrary topologies may fit perfectly to application, although they have to be customized to the application, which cost too much. Nowadays, it is unclear which arbitrary topologies are better than a mesh, because different applications are possible.

Routing

Depending on the topology, each communication has to select a path to its desired destination core. In general, there exist many different paths and strategies, particularly on larger-sized NoCs. In a tree structure, the generation of routes is trivial because there exists only one possible path to another core. In a mesh topology, the routes are typically generated by an X-Y routing policy, which ensures a deadlock-free behavior. First, a message is directly transmitted to the node with the desired x-position before it is further transmitted to the correct y-position such that the destination is reached.

Another strategy is the adaptive route generation to avoid high utilized edges in the NoC. For these, it is difficult to guarantee the real-time constraints. In certain cases, the shortest path is possibly not the best strategy to obtain short communication response times, because the routes can be defined such that the messages have less contention. This thesis applies the X-Y routing policy because it is proven to be deadlock-free [26], the routes are deterministic and it is simple to detect contention between two routes.

Switch implementation

The switch architectures can be implemented by different concepts, as shown in Figure 2.5. For a mesh topology, a switch has at most five input and five output ports, namely *Inner* (I), *North* (N), *South* (S), *West* (W) and *East* (E). A relatively simple architecture as shown in Figure 2.5a costs less chip area when implemented but allows fewer parallel message transmissions.

For parallelization of the message transmission, many NoC architectures are designed according to Figure 2.5b [2, 67]. If multiple messages arrive at the switch such that each

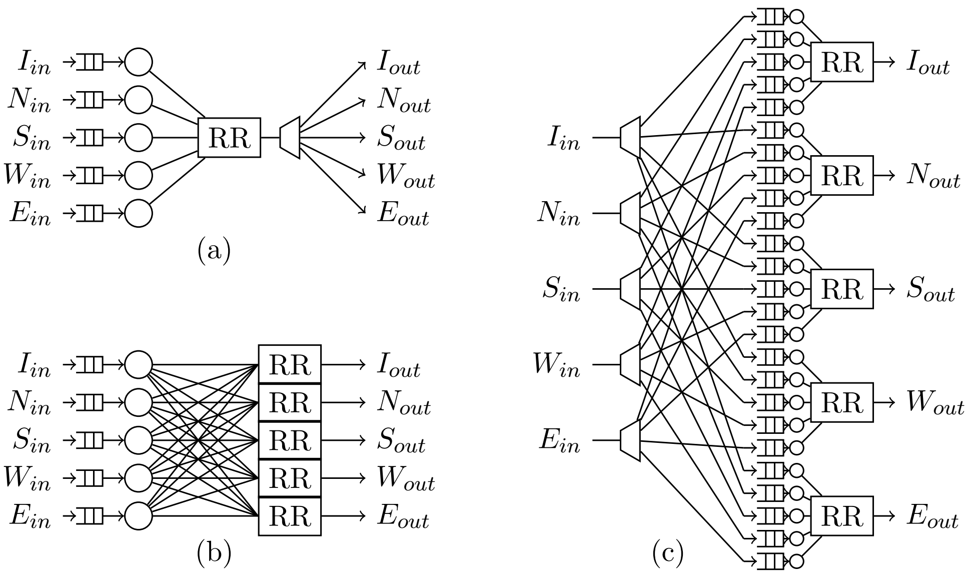


Figure 2.5.: Types of switch architectures: (a) All incoming ports have their own input queue and have a common arbiter. (b) Each output port has its own arbiter and each input port has its own queue. This type is used in DSPIN NoC [67] (c) Each incoming packet will first be switched to its own queue of its desired port to avoid back pressure conflicts. In this figure, each type has a round-robin arbiter (RR), although other arbitration policies are possible.

message has a different destination port, all messages can be forwarded simultaneously. In this architecture (Figure 2.5b), the common input buffer for each port can block other messages, which request another destination. A blocked message can cause back pressure to other preceding switches. Back pressure is interference between messages that are present on different switches.

In order to solve this issue, the architecture in Figure 2.5c has its own buffer for each output and input port. For a mesh topology, 25 buffers are necessary to avoid the blocking of succeeding messages, although this costs a large chip area.

This thesis assumes an architecture according to Figure 2.5b, because the time-triggered scheduling approach (Section 4.2) calculates a contention-free schedule. In a contention-free schedule, there exists no back pressure, although the parallel message transmissions can be exploited. Another reason is that this architecture (Figure 2.5b) is used in most NoC implementations.

Switch arbitration logic

As shown in Figure 2.5, each switch architecture has one or more arbiter for the switching decision. If multiple messages request the same output port, the arbitration logic controls the access to the link. Typical arbitration policies are Round Robin (RR), Time Division Multiple Access (TDMA) or FP.

A Fixed-Priority (FP) arbiter switches the messages with the highest priority. Therefore, each message needs a unique priority for describing its urgency. The problem with this arbiter is the high implementation effort, because each switch needs a queue for each priority. Considering typical industrial applications, this would result in 300–$3,000$ queues for each switch. If fewer queues than unique priorities exist, a higher-priority message could be blocked by a lower priority message. The high number of required queues (300–$3,000$) results in high implementation costs because the switches need to be implemented in hardware. Due to the high implementation effort, this arbiter only supports a small number of priorities. This arbiter is not applicable due to the large number of required queues.

A Time Division Multiple Access (TDMA) arbiter defines several time slots for an exclusive access from a certain input port, which repeats within a time cycle. The definition of the individual TDMA time slots requires a sophisticated method. If the TDMA slots in each switch are not synchronized, a message needs to wait for its slot in each switch in an otherwise-empty NoC. For the number of messages found in typical industrial-sized applications, the TDMA design becomes a complex problem. In addition, the TDMA schedule is stored in memory, which is assigned to the switch. For many messages, the storage of the TDMA slots become a problem.

A simple arbitration policy is Round Robin (RR). In general, a RR arbiter grants access for a message from each input to a specific output port in a cyclic manner. For example, if messages from two input ports S_{in}, E_{in} request the output port I_{out}, the messages are transmitted in the alternating manner $\{S_{in}, E_{in}, S_{in}, E_{in}, S_{in}, E_{in}, \ldots\}$. Thus, arbitration policy is simple to be implemented and is commonly used, e.g. the Epiphany® chip [2] has a Round Robin (RR) arbiter logic.

Another concept to forward messages is a contention-free schedule. The idea is to send messages such that a switch has never to decide which message has to be forwarded, because at most one message arrives at the same time. Therefore, the messages are isolated by reserving the path of each message in advance. For these cases, the arbitration logic is called *early access*, which implies that the one arriving message is immediately forwarded. Note that this *early access* concept can be implemented together with other arbitration logic, e.g. RR with *early access* or TDMA with *early access*. This thesis exploits the *early access* concept, because the time-triggered scheduling approach (Section 4.2) determines the scheduling in the NoC such that no contention occurs.

Message transmission policy

A message can be sent via different transmission policies, namely *Store-and-Forward* and *Worm Whole Switching*. They define how multiple messages are transmitted on a single path. The *Store-and-Forward* policy as shown in Figure 2.6a sends a message to the succeeding switch, until it is completely stored in the buffer of the succeeding switch. Hence, the buffer of each switch needs to store the entire message. The advantage of the *Store-and-Forward* policy is its limited interference with other messages, because a message can occupy at most one link at a time.

The *Worm Whole Switching* policy (Figure 2.6b) transmits a message on the same route, although it does not wait until the message is fully stored in the buffer. The time between the start of sending the message and the complete transmission the so-called network traversal time is shorter. The buffers do not need to store entire messages and thus they

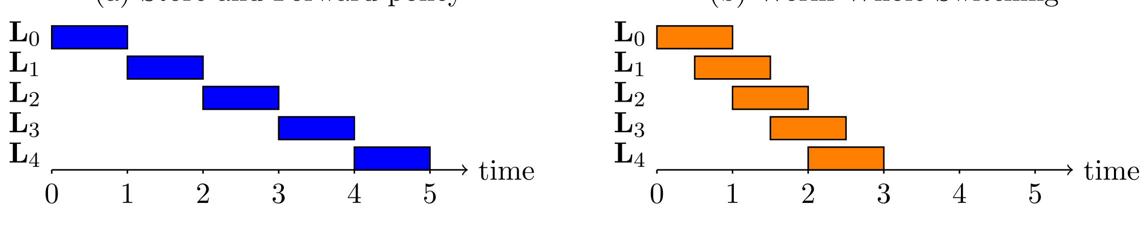


Figure 2.6.: Different message transmission policies are presented. The route r is defined by a list of edges in the NoC $\{\mathbf{L}_0, \mathbf{L}_1, \mathbf{L}_2, \mathbf{L}_3, \mathbf{L}_4\}$.

are usually smaller sized. If back pressure occurs, the *Worm Whole Switching* policy could block several edges of the NoC in a row, which can interfere with much more messages than the *Store-and-Forward* policy.

This thesis uses a *Worm Whole Switching* policy because it is more common in real NoC implementations. Furthermore, the proposed time-triggered scheduling approach (Section 4.2) prevents contention, which results in shorter network traversal times.

Flow control

A flow control logic needs to control contention if a link is too often used by different messages such that the buffer is not sufficiently sized. There are two general methods to prevent such overload situations: (i) messages can be dropped if the buffer is full, which is called *buffer overflow*; or (ii) no more massages are allowed to access the buffer for the link, which can cause *back pressure*. During back pressure, a buffer is needed to store more messages than its capacity allows, whereby this buffer blocks further messages, which can influence messages on other switches. In general, there are several approaches to deal with back pressure. A credit-based flow control ensures that the preceding switch stops sending new messages, although this may lead to a deadlock. Therefore, the system needs to be analyzed to avoid deadlocks.

Due to the time-triggered scheduling approach (Section 4.2), this thesis constructs the schedule such that messages do not interfere with each other. Hence, in a time-triggered system, a flow control unit is not required.

Availability

NoC architectures are an active research field, especially for real-time systems. While there exist only a few COTS NoCs with real-time support, many NoC architectural concepts are published [38, 66, 67, 69], which present different approaches to design a NoC. Note that this thesis focuses only on real-time capable NoCs [12, 23].

As an example of real products, the company Adapteva sells the Epiphany® chip [2], which uses a 2D-Mesh NoC. The company Kalray produces a chip called MPPA™ [25], in which the cores are connected by a 2D-Torus NoC. Due to the intellectual property of the companies, the details of these NoCs are not published.

Other NoC architecture concepts are not manufactured as real chips, although they are often used in simulators, reviewed by the scientific community and are more likely to appear in future chip designs. Examples of these concepts include DSPIN [67], Nostrum [66], Aethereal [38] and HERMES [69]. This thesis assumes the DSPIN NoC architecture because this architecture is close to our previously-mentioned NoC assumptions.

3. System Models

This chapter presents the platform and task models that are used in this thesis. The models include the essential information about the hardware and software required to analyze the scheduling. In the following, two platform models and two task models are presented [32–34].

3.1. Multicore and Manycore Platform Models

This section describes an abstract model of the hardware, named the platform model. In contrast to single-core platforms, multicore and manycore platforms comprise multiple cores, which are connected by a communication fabric. On a single-core platform, the communication between tasks can be modeled by their execution time, although for an increasing number of cores the communication fabric becomes the bottleneck. There exist many different possible hardware architectures to construct a platform. Thus, this thesis distinguishes between two general platforms, namely *multicore* and *manycore*.

In general, each core on a multicore or a manycore platform has a local memory to store its program code and temporal data. To obtain access to the communication fabric, each core has a Network Interface (NI), $\forall k \in \mathbb{Z}_0^+$. The NI can access core-local memory independent from the core. The tasks are executed on a processing unit, as showcased for one core in Figure 3.1.

Multicore

Definition 4. *(multicore platform) A multicore platform comprises identical cores* $\mathbf{C}_k$*, which are connected by one common shared resource, where* $|\mathbf{C}|$ *is the number of cores.*

On a multicore platform, each core $\mathbf{C}_k$ is connected to one common shared resource, represented by a bus, as shown in Figure 3.1.

The bus can be used by at most one NI at a time to send data to other cores. The bus access is regulated by a fixed-priority arbiter, as implemented in a Controller Area Network (CAN) bus [42,51]. The bus architecture is often used for on-chip communication, because the hardware costs less chip area. The NI implements the arbitration policy to gain access to the bus.

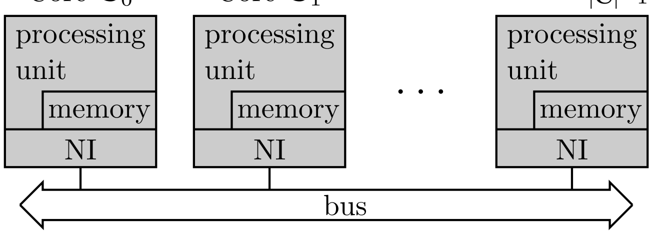


Figure 3.1.: The multicore platform comprises $|\mathbf{C}|$ cores. Each core has a NI, a core-local memory and a processing unit.

Each data transmission is sent under a certain priority and the arbiter decides which data transmission gains access to the communication fabric first. The data transmission with the highest priority gains access.

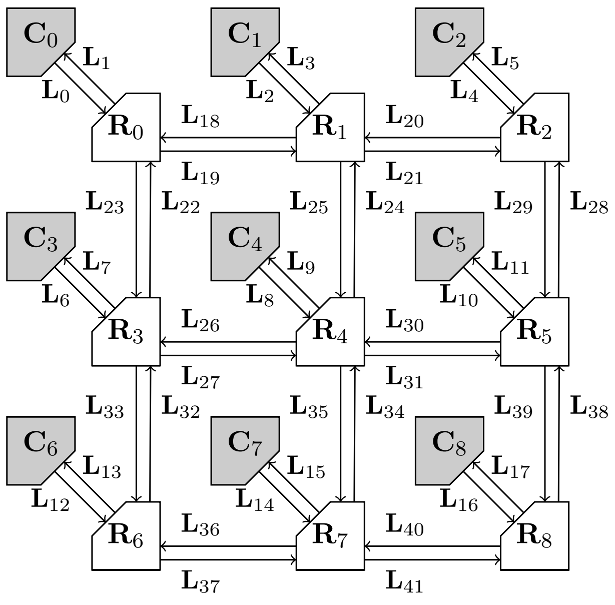
Another advantage is the one-to-all broadcast property, which allows the cores to read all transmitted data. However, if a lot of data has to be sent between the cores, the bus can become a bottleneck by increasing the number of cores. If the number of cores exceeds a certain level, the inter-core communication can limit the computational speed-up. For such cases, the computational tasks have to wait for the communication data to arrive. Therefore, applications with a high communication demand require a parallel communication architecture, which is presented in the following section.

Manycore

Definition 5. *(manycore platform) A manycore platform comprises cores $\mathbf{C}_k$, which are connected by Network-on-Chip (NoC) to allow parallel communication paths, where $|\mathbf{C}|$ is the number of cores.*

The cores of a manycore platform have the same structure as in Figure 3.1, although the communication fabric is different from a multicore platform, defined in Definition 4. A NoC is a communication fabric that scales with the number of cores by allowing parallel communication and is designed to be implementable with moderate hardware costs. Note that methods in this thesis support heterogeneous cores, although the experiments were performed with identical cores. Since the task-to-core mapping is given, the execution time is fixed such that the analysis becomes the same. In the following, NoC assumptions derived in Section 2.3 are described in detail.

The NoC is depicted by a graph, as shown in Figure 3.2. Each node is either a core $\mathbf{C}_k$ for computation or a switch $\mathbf{R}_m$ for communication. Each directed edge is a unidirectional links $\mathbf{L}_l$ to indicate a path for transmitting the data from one core to an other. Note that $|\mathbf{C}|$, $|\mathbf{R}|$, $|\mathbf{L}|$ are denoted as the number of cores, the number of switches and the number of links, respectively. A switch forwards the messages on its path to its desired destination. In case of contention, the switch decides which message is transmitted first. The NIs



$\mathbf{C}_0$
$\mathbf{C}_1$
$\mathbf{C}_2$
$\mathbf{C}_3$
$\mathbf{C}_4$
$\mathbf{C}_5$
$\mathbf{C}_6$
$\mathbf{C}_7$
$\mathbf{C}_8$
$\mathbf{R}_0$
$\mathbf{R}_1$
$\mathbf{R}_2$
$\mathbf{R}_3$
$\mathbf{R}_4$
$\mathbf{R}_5$
$\mathbf{R}_6$
$\mathbf{R}_7$
$\mathbf{R}_8$
$\mathbf{L}_0$ $\mathbf{L}_1$ $\mathbf{L}_2$ $\mathbf{L}_3$ $\mathbf{L}_4$ $\mathbf{L}_5$ $\mathbf{L}_6$ $\mathbf{L}_7$ $\mathbf{L}_8$ $\mathbf{L}_9$ $\mathbf{L}_{10}$ $\mathbf{L}_{11}$ $\mathbf{L}_{12}$ $\mathbf{L}_{13}$ $\mathbf{L}_{14}$ $\mathbf{L}_{15}$ $\mathbf{L}_{16}$ $\mathbf{L}_{17}$
$\mathbf{L}_{18}$ $\mathbf{L}_{19}$ $\mathbf{L}_{20}$ $\mathbf{L}_{21}$ $\mathbf{L}_{22}$ $\mathbf{L}_{23}$ $\mathbf{L}_{24}$ $\mathbf{L}_{25}$ $\mathbf{L}_{26}$ $\mathbf{L}_{27}$ $\mathbf{L}_{28}$ $\mathbf{L}_{29}$ $\mathbf{L}_{30}$ $\mathbf{L}_{31}$ $\mathbf{L}_{32}$ $\mathbf{L}_{33}$ $\mathbf{L}_{34}$ $\mathbf{L}_{35}$ $\mathbf{L}_{36}$ $\mathbf{L}_{37}$ $\mathbf{L}_{38}$ $\mathbf{L}_{39}$ $\mathbf{L}_{40}$ $\mathbf{L}_{41}$

Figure 3.2.: Example of a manycore platform with $|\mathbf{C}| = 9$ cores connected by a 3×3 2D-mesh NoC.

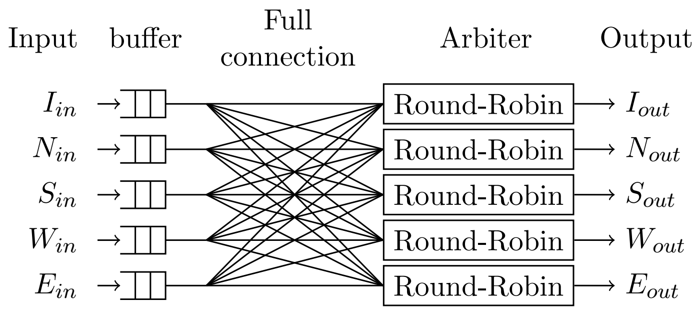


Figure 3.3.: The inner structure of a NoC switch: Each switch output port has its own round robin (RR) arbiter. Each switch input port has its own First-In-First-Out (FIFO) buffer. This represents the DSPIN NoC [67] implementation. The ports are named based upon their direction with Inner (I), North (N), South (S), West (W) and East (E).

handle incoming and outgoing data from the core to the NoC. A NI is able to inject the communication data at a certain time. Thus, a computational task can define the injection time of its sending data.

The topology of the NoC is a 2D-mesh as shown in Figure 3.2 in which the number of cores equals the number of switches $|\mathbf{C}| = |\mathbf{R}|$. A switch $\mathbf{R}_m$ is connected with its corresponding core $\mathbf{C}_m$ and at most four other switches. All connected nodes can transmit data in full duplex with two links $\mathbf{L}$ (one for each direction). Thus, each switch $\mathbf{R}_m$ has at most five input ports and five output ports, which are fully connected, as shown in Figure 3.3.

If more than one data transmission requests a certain output port, the corresponding Round Robin (RR) arbiter decides which one gains access. If the arbiter blocks some messages, these messages are stored in a *sufficiently sized buffer* such that no data becomes lost. A *sufficiently sized buffer* means that the scheduling analysis ensures the maximum amount of data for each buffer, which needs to be less than its real size. If the buffers can never overflow, no flow control is required to handle these cases. For routing the data from switch to switch, the routes are determined according to the dimensional order X-Y routing policy [69]. The X-Y routing policy avoids deadlocks and ensures deterministic routes.

Furthermore, a link can transmit data with an upper-bounded link bandwidth $b_{\mathbf{L}}$. Each link delays the transmitted data by a certain amount of time, called *link delay* $d_{\mathbf{L}}$. Similarly, a switch delays the data by an upper-bounded switch delay $d_{\mathbf{R}}$ on the route to the next link. As an example, the DSPIN NoC [67] supports the assumed NoC model.

3.2. Independent Sporadic Task Model

This section presents the sporadic real-time task model often found in the literature [68]. A sporadic task σ_k is defined by its inter-arrival time P_{σ_k}, its relative deadline D_{σ_k}, its worst-case execution time (WCET) W_{σ_k} and its assigned core c_{σ_k}, $\forall k \in \mathbb{Z}_0^+$. One or more sporadic tasks compose the sporadic task set $\mathbf{S}$, where $|\mathbf{S}|$ is denoted as the number of sporadic tasks in the set. Each task σ_k releases an infinite number of jobs $J_{\sigma_{k,\ell}}$, which arrive at their arrival time $a_{\sigma_{k,\ell}}$ for the ℓ-th job, $\forall \ell \in \mathbb{Z}_0^+$.

A job can arrive at any time, although two consecutive jobs arrive with the minimum time distance P_{σ_k}, and thus $a_{\sigma_{k,\ell+1}} - a_{\sigma_{k,\ell}} \geq P_{\sigma_k}$. The minimum inter-arrival time P_{σ_k} is also named period, because in the worst case the jobs arrive periodically with P_{σ_k}.

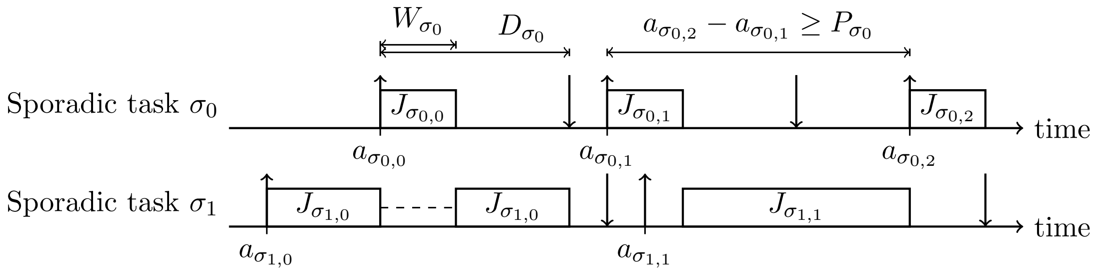


Figure 3.4.: Visualization of the sporadic task model. In this case, the sporadic task σ_0 has a higher priority than σ_1. The arrival times of any two consecutive jobs of a sporadic task σ_k are separated by at least P_{σ_k}. The dashed line represents the preemption of job $J_{\sigma_{1,0}}$ by the higher-priority job $J_{\sigma_{0,0}}$. The arrows indicate the arrival time and the absolute deadline of each job, respectively.

The WCET W_{σ_k} is the maximum amount of time that the task σ_k needs to be executed. However, the sporadic task can complete its execution earlier. In order to hold the real-time constraint, a job needs to be completed before its absolute deadline $a_{\sigma_{k,\ell}} + D_{\sigma_k}$. Otherwise, this job violates its deadline. The assigned core c_{σ_k} defines the core $\mathbf{C}_{c_{\sigma_k}}$, on which the sporadic task is executed. Figure 3.4 visualizes our sporadic task model. Note that sporadic tasks are scheduled by a Fixed-Priority (FP) scheduler.

3.3. Dependent Periodic Task Model

In the dependent task model, the computational tasks communicate with each other. This model separates the computation from the communication, defining computational task set $\mathbf{T}$ and a communication task set $\mathbf{K}$.

Computational task model

Each computational task $\tau_i \in \mathbf{T}$ has a WCET W_{τ_i}, a period P_{τ_i}, a relative deadline D_{τ_i}, an assigned core c_{τ_i} and a list of predecessor tasks $Q_{\tau_i} = \{\tau_{v_1} \dots \tau_{v_q}\} \subset \mathbf{T}$. The WCET W_{τ_i} is an upper bound of the execution time of the computational task τ. In contrast to the sporadic task model, the computational tasks τ are executed in a pure periodic behavior, i.e. once every period P_{τ_i}. The number of tasks is denoted by $|\mathbf{T}|$. There exists a hyper-period H with

$$H = \text{LCM}(P_{\tau_1}, P_{\tau_2}, \dots, P_{\tau_{|\mathbf{T}|}}), \qquad \forall \tau_i \in \mathbf{T}, \tag{3.1}$$

where LCM is the least common multiple. Due to typical industrial characteristics, the periods are harmonic, i.e. the higher periods are integer multiplies of the lower periods. Each task τ_i releases an infinite number of jobs $J_{\tau_{i,\ell}}$, which arrive at their arrival time $a_{\tau_{i,\ell}}$. Due to the periodic releases, the arrival times can be defined as

$$a_{\tau_{i,\ell}} = a_{\tau_{i,0}} + \ell \cdot P_{\tau_i}, \tag{3.2}$$

where $a_{\tau_{i,0}}$ is the arrival time of the first job $J_{\tau_{i,0}}$. The first jobs $J_{\tau_{i,0}}$ of all the tasks arrive at time 0 such that $a_{\tau_{0,0}} = a_{\tau_{1,0}} = \ldots = a_{\tau_{|\mathbf{T}-1|,0}} = 0$. To satisfy the real-time constraints, each computational job needs to be completed before its absolute deadline $a_{\tau_{i,\ell}} + D_{\tau_i}$. Note that the relative deadlines are assumed to be implicit, whereby $D_{\tau_i} = P_{\tau_i}$, $\forall \tau_i$. The assigned core c_{τ_i} defines the core $\mathbf{C}_{c_{\tau_k}}$, on which the computational task is executed.

A computational task can have a set of predecessor tasks Q_{τ_i}. This results in precedence relations between tasks, which is stated in Definition 6. No cycles are allowed in

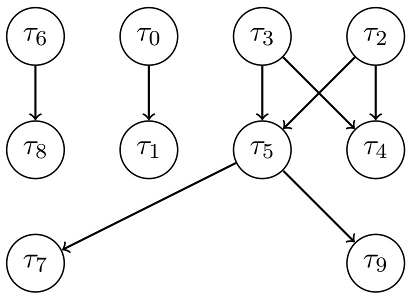


Figure 3.5.: The DAG of a computational task set with $|\mathbf{T}| = 10$ computational tasks. The directed edges represent the precedence relations between the tasks.

the precedence relations, because this lead to an infeasible schedule. Thus, a directed acyclic graph (DAG) can represent the precedence relations of the computational tasks. An example of a DAG is shown in Figure 3.5.

Definition 6. *(task precedence) A precedence constraint between tasks is defined based upon their jobs that they release. Suppose the computational task τ_i is a predecessor of the computational task τ_j, then the l-th job $J_{\tau_i,l}$ of τ_i needs to complete its execution and transmits its data to τ_j, before k-th job $J_{\tau_j,k}$ of τ_j starts, if $a_{\tau_{i,l}} \leq a_{\tau_{j,k}}$, where $a_{\tau_{i,l}}$ and $a_{\tau_{j,k}}$ are the corresponding arrival times of the jobs, $\forall l, k$.*

Note that each computational task comprises two succeeding segments. First the task is executed without any inter-core communication and second the task activates its inter-core communication without execution. Therefore, the computation and inter-core communication can be separated. This model is similar to the acquisition-execution-replication (AER) model proposed by Schranzhofer et al. [84]. By default computational tasks are scheduled by a Time-Triggered Constant Phase (TTCP) scheduling policy, which is explained in Section 6.2.

Communication task model

The communication task set $\mathbf{K}$ comprises communication tasks κ, where $|\mathbf{K}|$ is the number of communication tasks. The communication tasks represent dependencies between two computational tasks that exchange data with each other. Each communication task has a traversal time W_{κ_j}, a period P_{κ_j}, a relative deadline D_{κ_j}, a route r_{κ_j}, a source task τ_{SRC_j} and a destination task τ_{DST_j}.

Each communication task κ releases an infinite number of packets (messages), which traverse via the communication fabric from the source task τ_{SRC_j} to the destination task τ_{DST_j}. If the source and destination tasks are placed on the same core $c_{\tau_{\mathrm{SRC}_j}} = c_{\tau_{\mathrm{DST}_j}}$, the traversal time is $W_{\kappa_j} = 0$. In the general case when $c_{\tau_{\mathrm{SRC}_j}} \neq c_{\tau_{\mathrm{DST}_j}}$, the traversal time W_{κ_j} is defined as the time that it takes to inject the packet into the network fabric.

In every period P_{κ_j}, exactly one packet is injected into the communication fabric. Each packet arrives at its arrival time $a_{\kappa_{j,\ell}}$, which is equal to the completion time of the source job $J_{\tau_{\mathrm{SRC}_{j,\ell}}}$. The packet has to be completely transferred to the destination within its absolute deadline $a_{\kappa_{j,\ell}} + D_{\kappa_j}$ to satisfy the real-time constraints. The maximum time between the complete transmission of a packet and its arrival time is called worst-case traversal response time (WCTRT). The route r_{κ_j} is defined as a list of nodes in which the packet is forwarded by the communication fabric. The route r_{κ_j} contains $|r_{\kappa_j}|$ nodes, which represent the number of hops to reach the destination of the packet. Based upon the network delays $d_{\mathbf{L}}$ and $d_{\mathbf{R}}$, the network traversal time $\overline{W}_{\kappa_j}$ is the time to fully transmit a packet via all nodes of the NoC, with

$$\overline{W}_{\kappa_j} = W_{\kappa_j} + |r_{\kappa_j}|(d_{\mathbf{R}} + d_{\mathbf{L}}). \tag{3.3}$$

The precedence constraints of the computational tasks create two types of communication tasks, namely *κ-precedence* and *κ-non-precedence.* If τ_{SRC_j} is a predecessor of τ_{DST_j} (or not), the related communication task becomes *κ-precedence* (or *κ-non-precedence*). The different types of the communication tasks are shown in Figure 3.6.

Each computational job commonly sends its data after its execution, although some communication is unnecessary for different periods of source and destination job. This thesis defines specific jobs that send and receive the data to avoid futile communications. Only certain jobs communicate data to its destination, which are indexed as the h-th jobs of τ_{SRC_j} with

$$h = \left\lceil \frac{P_{\tau_{\mathrm{DST}_j}}}{P_{\tau_{\mathrm{SRC}_j}}} \right\rceil g, \qquad g \in \mathbb{Z}_0^+. \tag{3.4}$$

Note that the periods P_{τ_i}, P_{κ_j} are harmonic and each computational task τ_i releases their first job $J_{\tau_i,0}$ at the same time 0.

To receive the data from $J_{\tau_{\mathrm{SRC}_j},h}$, only a certain job $J_{\tau_{\mathrm{DST}_j},k}$ has to receive the data. The destination job $J_{\tau_{\mathrm{DST}_j},k}$ is the earliest job that

- starts its execution at time $s_{\tau_{\mathrm{DST}_j},k}$ no earlier than $a_{\tau_{\mathrm{SRC}_j},h}$ if κ_j*-precedence*, or
- starts its execution at time $s_{\tau_{\mathrm{DST}_j},k}$ no earlier than $a_{\tau_{\mathrm{SRC}_j},h} + P_{\tau_{\mathrm{SRC}_j}}$ if κ_j*-non-precedence.*

All computational and communication tasks compose a *task graph*, in which the nodes are computational tasks and the edges are communication tasks. This task graph shows the complexity of the inter-task dependencies and can be used to find less dependent partitions. The communication model only supports sending (non-blocking) communication that does not wait for a response of the destination task. Bi-directional (blocking) communication can be modeled by multiple computational tasks, which consecutively send data to each other. By default, communication tasks are scheduled by a Time-Triggered Constant Phase (TTCP) scheduling policy, which is explained in Section 6.2.

Example of the dependent task model

The dependent task model is more comprehensive than the sporadic task model. Therefore, in the following an example demonstrates this model. The example has $|\mathbf{T}| = 10$

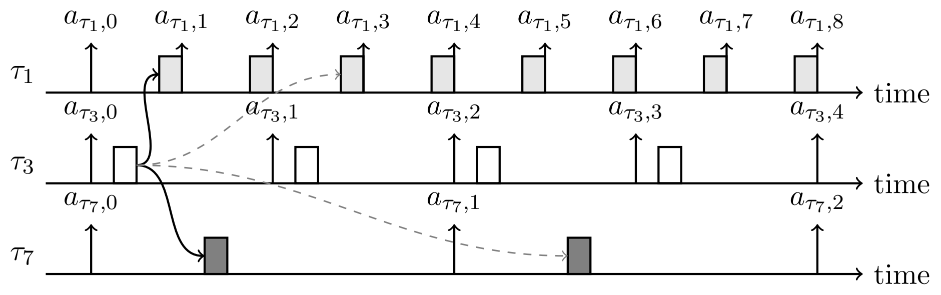


Figure 3.6.: Example of precedence and non-precedence communication tasks. If the communication task is of type κ_j*-precedence*, the edge is a solid line. If the communication task is of type κ_j*-non-precedence*, the edge is a dashed line.

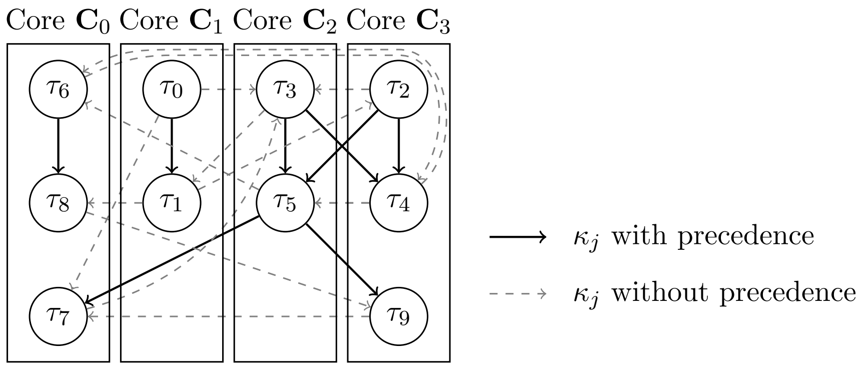


Figure 3.7.: The task graph of the dependent task model example. Solid and dashed lines represent precedence constraints and communication tasks, respectively.

Table 3.1.: The computational tasks of the example of the dependent task model.

Computational task	$W_{\tau_i}/\mu s$	$D_{\tau_i}/\mu s$	$P_{\tau_i}/\mu s$	c_{τ_i}	Q_{τ_i}
τ_0	1432	10000	10000	1	
τ_1	5223	10000	10000	1	τ_0
τ_2	7219	20000	20000	3	
τ_3	12610	20000	20000	2	
τ_4	2138	20000	20000	3	τ_2, τ_3
τ_5	415	20000	20000	2	τ_2, τ_3
τ_6	9994	20000	20000	0	
τ_7	304	40000	40000	0	τ_5
τ_8	6940	40000	40000	0	τ_6
τ_9	1387	40000	40000	3	τ_5

Table 3.2.: The communication tasks of the example of the dependent task model.

Communication task	$W_{\kappa_i}/\mu s$	$D_{\kappa_i}/\mu s$	$P_{\kappa_i}/\mu s$	τ_{SRC_j}	τ_{DST_j}	Type
κ_0	45	12451	20000	τ_1	τ_2	*κ-non-precedence*
κ_1	418	12610	20000	τ_2	τ_5	*κ-precedence*
κ_2	628	20000	20000	τ_3	τ_1	*κ-non-precedence*
κ_3	300	18321	40000	τ_5	τ_7	*κ-precedence*
κ_4	629	19994	20000	τ_1	τ_8	*κ-non-precedence*
κ_5	611	20000	20000	τ_4	τ_6	*κ-non-precedence*
κ_6	85	10000	20000	τ_0	τ_3	*κ-non-precedence*
κ_7	561	15225	20000	τ_3	τ_4	*κ-precedence*
κ_8	221	20000	20000	τ_4	τ_5	*κ-non-precedence*
κ_9	378	20000	20000	τ_5	τ_6	*κ-non-precedence*
κ_{10}	288	20000	20000	τ_9	τ_7	*κ-non-precedence*
κ_{11}	512	20000	20000	τ_2	τ_3	*κ-non-precedence*
κ_{12}	216	16934	40000	τ_5	τ_9	*κ-precedence*
κ_{13}	29	20000	20000	τ_6	τ_4	*κ-non-precedence*
κ_{14}	392	40000	40000	τ_7	τ_3	*κ-non-precedence*
κ_{15}	679	20000	20000	τ_8	τ_9	*κ-non-precedence*
κ_{16}	96	20000	20000	τ_2	τ_3	*κ-non-precedence*
κ_{17}	171	15225	20000	τ_3	τ_4	*κ-precedence*
κ_{18}	20	20000	20000	τ_4	τ_5	*κ-non-precedence*
κ_{19}	546	18321	20000	τ_0	τ_7	*κ-non-precedence*

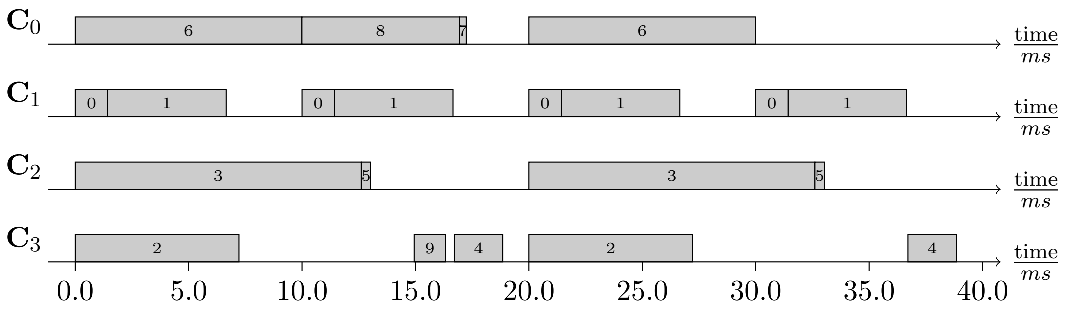


Figure 3.8.: A feasible schedule of an example of the dependent task model. The timeline is shown for each core. The computational tasks are visualized by gray rectangles. The numbers of the rectangles represent the task index i for task τ_i.

computational tasks and $|\mathbf{K}| = 20$ communication tasks. The platform is generic and has four cores, which are connected by a communication fabric (bus or NoC).

All computational tasks are represented by nodes in Figure 3.7. The computational tasks are specified in Table 3.1 by the WCET W_{τ_i}, the period P_{τ_i}, the relative deadline D_{τ_i}, the assigned core c_{τ_i} and the list of predecessor tasks Q_{τ_i}. The arrow at the edges indicates the communication direction, in which the tip means the destination task. The details of the communication tasks are provided in Table 3.2, which specifies the traversal time W_{κ_j}, the period P_{κ_j}, the relative deadline D_{κ_j}, the source task τ_{SRC_j} and the destination task τ_{DST_j}.

All communication tasks of type *κ_j-precedence* compose a DAG of the computational task. The DAG is shown in Figure 3.7 if only *κ_j-precedence* are focused. The precedence constraints only define a sequence of jobs in a certain interval. For example, τ_4 is scheduled after τ_3 and the transmitted packet of κ_7. Figure 3.8 shows a possible feasible schedule for all computational and communication tasks that satisfy the precedence and real-time constraints.

4. Time-Triggered Constant Phase Scheduling Analysis with a Bus Architecture

This chapter presents the approach to schedule and analyze a tangled system for a multicore platform with a bus architecture. The proposed approach uses a TTCP scheduler to schedule the computational tasks. With this scheduler, the scheduling analysis can be efficiently performed, whereas the platform can be highly utilized. The general ideas of this chapter have already been published [33] by the author.

4.1. Introduction

This section introduces the proposed approach to schedule a dependent task model (Section 3.3) with a high platform utilization. In contrast to the dependent task model, this chapter assumes only non-precedence communication tasks. The literature [17,24,59] usually assumes an independent task model, because this model significantly simplifies the scheduling analysis. The dependent task model is more difficult to handle, although this model represents typical industrial applications, where the tasks communicate with each other, better. Especially on multicore and manycore platforms, the inter-core communication occurs in the scheduling analysis, which is the focus of this chapter.

In the proposed platform model (Section 3.1), each core has a local memory for storing its program code and temporally-required data. Thus, communication via the shared communication fabric to fetch the program code is not required. If the tasks are placed on different cores and exchange data, these tasks must use the communication fabric. In our case, this is a shared bus. The problem is that in the worst case all inter core communications may arrive at the same time.

On a single-core platform, the communication time can be modeled as part of the WCET. However, on a multicore platform, the worst case significantly increases this modeled communication time. Even if the communication time to transmit data is short on average, in the worst case the data transmission is blocked by all other communication tasks. One the one hand, if the arrival pattern of the communication tasks is known, the scheduling analysis for fixed-priority scheduling is NP-hard [14]. On the other hand, the scheduling analysis can be performed in pseudo-polynomial time complexity [14] for unknown arrival times. In order to guarantee the real-time capability, the timing analysis assumes the worst case to obtain a reliable result. For example, suppose two tasks. If the arrival pattern is not known, these tasks may interfere with each other such that one task is blocked, which increases its response time. If the arrival pattern is known, the interference is clear, although the arrival pattern has to be designed.

The proposed approach is to improve the worst-case behavior by defining the starting time of the computational tasks τ. If the computational starting times are *a priori* known, the communication arrival times to transmit data are determined by the latest completion time of the computational tasks. By knowing each communication arrival time $a_{\kappa_{j,\ell}}$, the communication schedule can be determined without assuming that all communication tasks may arrive at the same time. Thus, the communication response times can be tightly determined and a feasible solution can accommodate higher possible communication utilization. Communication utilization represents the proportion of the used time and the maximum available time for the usage of communication fabric.

The class of scheduler which determines the schedule *a priori* is called a static scheduler. A typical static scheduler requires exponential time complexity to analyze the timing considering the number of tasks. To overcome the complexity issue, the proposed approach is to schedule the computational task set **T** with a TTCP scheduler. The TTCP scheduler executes each computational task in a certain periodic time window, which starts at a fixed time offset, named a phase.

A real-time scheduler needs its own scheduling analysis to verify whether the schedule is feasibly schedulable. The scheduling analysis is divided into two problems, the computational analysis (Section 4.3) and the communication analysis (Section 4.4). The TTCP approach claims to be analyzable in pseudo-polynomial time complexity, which is also shown.

Related work

The problems with the data communications in a multicore system has been shown in several publications [48, 70, 84]. Schranzhofer et al. [84] evaluate different resource access patterns to a common shared resource and propose separating the execution phase from the communication read-write phases to reduce the completion time. The proposed approach adopts this acquisition-execution-replication (AER) model to achieve a tight WCET. Munk et al. [70] analyze different communication patterns to bound the worst case in on a NoC architecture. In the worst case, all data transmissions collide with each other, which results in large response times or a limited sending rate. By comparison, the time-triggered approach schedules each communication at a certain time to avoid the worst-case pessimism.

The time-triggered approach in general is widely adopted [31, 48, 72, 74]. The classic time-triggered scheduler defines the starting time of each individual job within the hyper-period. The hyper-period is the least common multiple of the periods of a task set. The problem is that a scheduling design and analysis algorithm needs to unfold the schedule to the hyper-period. By contrast, this thesis uses a special time-triggered scheduling approach namely a TTCP, which can significantly reduce the effort to design and analyze the schedule.

Closely related, some researchers also assume the TTCP scheduling concept [11, 45, 61, 64, 82], although they only implicitly present their TTCP approach. Most of these approaches [11, 61, 82] formulate the TTCP scheduling problem as a set of equations and calculate the schedule by a solver-based approach. The problem is that the solvers run in exponential time complexity considering the number of tasks. In contrast to these approaches, this thesis provides an approach to construct the TTCP schedule in pseudo polynomial time complexity and is thus applicable to typical industrial applications. Focusing on the TTCP scheduling analysis, Marouf and Sorel [64] provide a feasibility test that runs in polynomial time complexity. Kermia and Sorel [45] design a heuristic to determine the starting times, although the algorithm in [45] was not presented sufficiently clearly to be adopted. Moreover, their approach is limited to single-core scheduling without considering communications and multicore platforms.

Motivational example

In the following, a motivational example demonstrates the inter-core communication problem and the proposed TTCP approach. In order to keep the example simple, there are

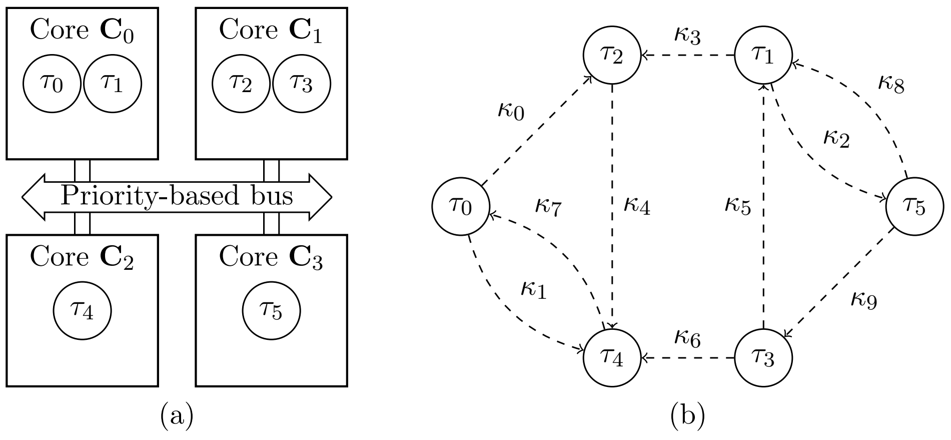


Figure 4.1.: Motivational example: (a) The computational tasks are mapped to a four-core platform, which is connected by a priority-based bus. (b) The task graph shows the communications between the tasks, where each dashed edge represents an inter-core communication.

only $|\mathbf{T}| = 6$ computational tasks τ and $|\mathbf{K}| = 10$ communication tasks κ. The platform comprises four cores, which are connected by a priority-based bus. The computational tasks are mapped to the platform as shown in Figure 4.1a where the tasks τ_0, τ_1 are mapped to the same core and the tasks τ_2, τ_3 are mapped to another core. The task graph shown in Figure 4.1b represents the data transmission dependencies of the tasks. A directed edge means that a task has to send data after its completion to another task (the destination task is depicted by the tip of the edge). The detailed properties of this task model are described in Table 4.1 for the computational task set $\mathbf{T}$ and Table 4.2 for the communication task set $\mathbf{K}$. The communication task set $\mathbf{K}$ is sorted according to the priority-based on the bus, in which each communication task κ_j has a unique priority denoted by the index j. The communication task κ_0 has the highest priority. A higher index j indicates a lower communication priority.

One possibility is to allow the release of a communication task at any time. For the scheduling analysis, this freedom implies that in the worst case all ten communication tasks arrive at the same time. Due to priority-based arbitration, the communication task κ_1 can be blocked by one task, which may already communicate. Thus, if κ_0 is blocked by κ_8, its worst-case traversal time is $8ms$. In comparison to the TTCP scheduling approach, the resulting response time is $1ms$, because the other communication tasks are assigned to a certain time window such that κ_0 is not blocked. An *a priori* defined schedule can significantly reduce the traversal response time for the communication tasks κ. Another advantage is the *a priori* known schedule, which is shown in Figure 4.2. In the schedule

Table 4.1.: The computational task set $\mathbf{T}$ of the motivational example.

Given by the application					To be determined
task	W_{τ_i}/ms	P_{τ_i}/ms	c_{τ_i}	Q_{τ_i}	Φ_{τ_i}/ms
τ_0	10	20	1	κ_0, κ_1	0
τ_1	9	40	1	κ_2, κ_3	10
τ_2	5	20	2	κ_4	1
τ_3	14	80	2	κ_5,κ_6	46
τ_4	25	40	3	κ_7	2
τ_5	46	80	4	κ_8, κ_9	26

Table 4.2.: The communication task set **K** of the motivational example.

Given by the application						To be determined
task	τ_{SRC_j}	τ_{DST_j}	W_{κ_j}/ms	P_{κ_j}/ms	D_{κ_j}/ms	Φ_{κ_j}/ms
κ_0	τ_0	τ_2	1	20	5	10
κ_1	τ_0	τ_4	1	40	3	10
κ_2	τ_1	τ_5	2	80	5	19
κ_3	τ_1	τ_2	1	40	$P\kappa_j$	19
κ_4	τ_2	τ_4	2	40	4	6
κ_5	τ_3	τ_1	1	80	$P\kappa_j$	60
κ_6	τ_3	τ_4	3	80	$P\kappa_j$	60
κ_7	τ_4	τ_0	4	40	$P\kappa_j$	27
κ_8	τ_5	τ_1	7	80	$P\kappa_j$	72
κ_9	τ_5	τ_3	1	80	20	72

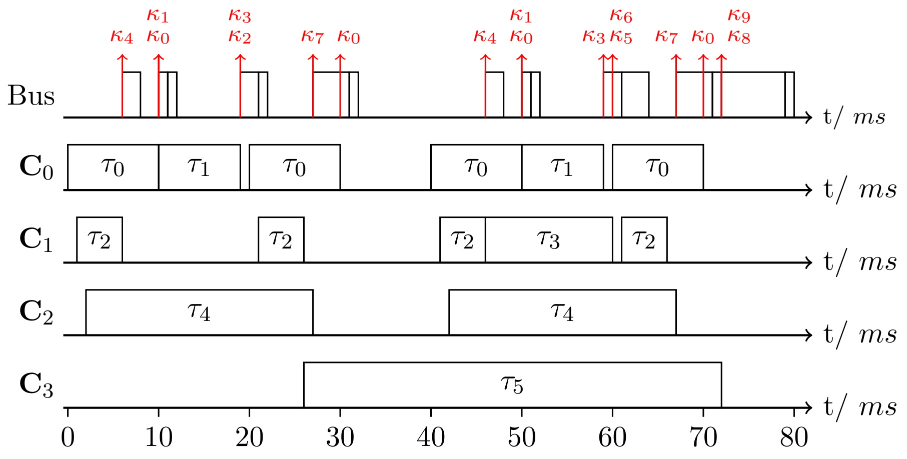


Figure 4.2.: The schedule within one hyper-period $[0 \ldots 80]ms$ for the example task sets, given in Table 4.1 and 4.2.

visualization, the scheduling problem in this chapter is finding a time window for each computational task such that they do not overlap in the schedule.

Problem definition

The assumption is to use the TTCP scheduling approach, which results in a scheduling analysis and design problem. The problem is to define

- an efficient feasibility test running in polynomial time or pseudo polynomial time,
- feasible starting times of the TTCP scheduled computational task set **T**, and
- feasible starting times of the TTCP scheduled communication task set **K**,

while satisfying all real-time constraints on a *multicore* platform.

The following sections of this chapter present the proposed TTCP approach.

4.2. Time-Triggered Constant Phase (TTCP) Approach

The Time-Triggered Constant Phase (TTCP) approach is presented in this section, which is used in proposed approaches in Chapters 4-6. In general, the time-triggered approach is widely adopted [31,47,48,61,72,74]. In contrast to event-based approaches, all tasks are activated based upon a global time. Thus, all cores on a *multicore* or *manycore* platform need to have the same time base. Based upon this time base, the tasks are activated and scheduled in a coordinated and *a priori* known manner. Each periodic activated task is started at an *a priori* known starting time, which needs to be determined in advance.

This thesis uses a special type of time-triggered scheduler, called the TTCP scheduler, which defines a phase Φ for each task rather than defining the starting time for each individual job. The phase Φ is defined as the first starting time of a task. Hence, the starting time s_τ of jobs of the computational task τ can be defined by

$$s_{\tau_i} = \Phi_{\tau_i} + P_{\tau_i} \cdot k, \qquad k \in \mathbb{Z}_0^+. \tag{4.1}$$

Note that each computational and communication task has a phase Φ_{τ_i}, Φ_{κ_i}, respectively, although the equation only shows the computational tasks. Figure 4.3 provides an overview of the relevant timing parameters of the TTCP scheduler.

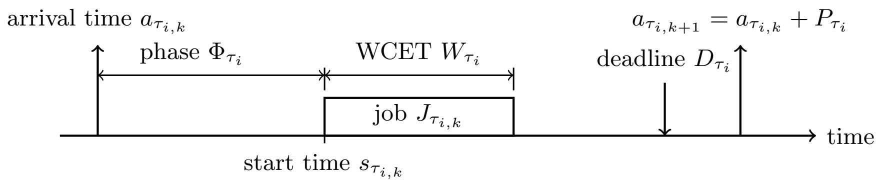

arrival time $a_{\tau_{i,k}}$
phase Φ_{τ_i}
WCET W_{τ_i}
$a_{\tau_{i,k+1}} = a_{\tau_{i,k}} + P_{\tau_i}$
deadline D_{τ_i}
job $J_{\tau_{i,k}}$
time
start time $s_{\tau_{i,k}}$

Figure 4.3.: The parameters of the TTCP scheduler

The advantage of the TTCP scheduler is that all execution windows can be expressed by three parameters, namely Φ_{τ_i},P_{τ_i} and W_{τ_i}. This reduces the complexity of the analysis, because the periodic behavior of the job activations can be exploited. Having less parameters also implies that the scheduler has to store less information, which results in a significant reduction for embedded real-time platforms. The TTCP scheduler is able to dynamically calculate the starting times of the jobs during run-time.

The TTCP scheduler assumes the existence of a hyper-period H. Hence, there also exists a greatest common divisor (GCD), which is required by the analysis. For task sets with harmonic periods, the GCD and LCM are the minimum and maximum periods in the task set, respectively. For modern processing units, the period can be expressed by an integer number of processing units cycles, and thus exists a least common multiple exists.

The TTCP approach applies the TTCP scheduler to each core and the communication fabric. The TTCP schedule can be constructed such that all tasks on all cores satisfy all the real-time constraints such as deadlines, precedence and communication constraints. All cores need to be synchronized to be able to execute the TTCP scheduler. In addition, the communication tasks are also scheduled in the TTCP manner. However, in contrast, the injection time for the communication fabric is determined.

Owing to the *a priori* known time windows for executing a task, two tasks are not allowed to be executed at the same time and the same core. Therefore, the proposed method has to construct a contention-free schedule in which no task time window overlaps another. Therefore, if the execution of a task overlaps with another task, the resulting TTCP schedule is infeasible. This chapter assumes the TTCP approach and deals with TTCP scheduling analysis and design problem, for which the workflow of the proposed solution is presented in the following.

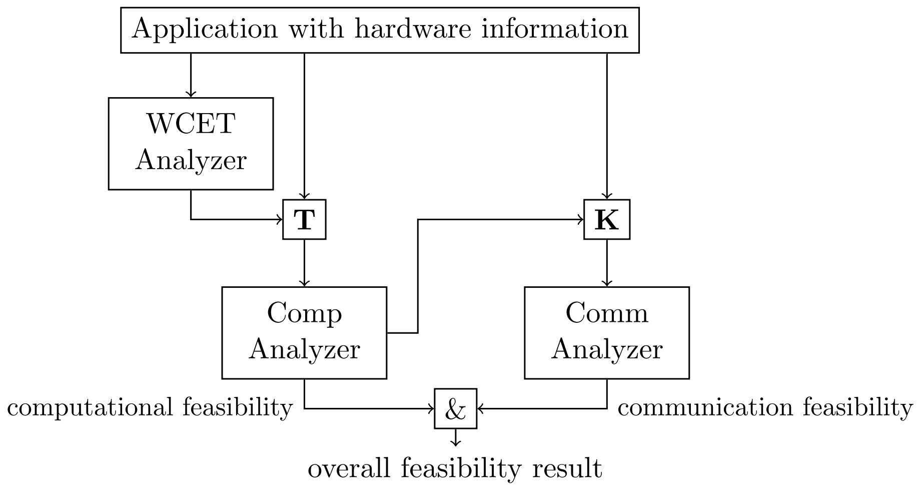

Application with hardware information
WCET Analyzer
T
K
Comp Analyzer
Comm Analyzer
computational feasibility
&
communication feasibility
overall feasibility result

Figure 4.4.: The workflow for determining a feasible set of computational and communication phases $\Phi_\mathcal{T}, \Phi_\mathcal{K}$ is shown. The schedule of the computational tasks is analyzed on each core before the communication analysis can be performed. The system is only feasible if both analyzers return a feasible result.

Workflow for analyzing and designing the TTCP schedule

The workflow describes the approach to solve the previously-defined problem in this chapter. The steps to determine a feasible TTCP schedule are shown in Figure 4.4.

The dependent task model and the *multicore* platform model provide all required parameters. The WCET W_{τ_i} can be determined with a WCET analyzer. In general, the problem of determining the WCET is difficult and an active research area [95]. There are several tools e.g. aiT [29] or Bound-T [40] for extracting the WCET based upon a static binary code analysis. The TTCP approach can simplify the WCET analysis, because the over-approximation of the inter-core communication can be neglected. The communication tasks are scheduled such that they do not interfere with the computational execution. Based upon the system model, each task follows the two segments: first the task is executed; and second, the task activates its inter-core communication.

The proposed approach is to determine the computational schedule first, before the communication schedule is subsequently analyzed. Thus, the arrival time for the communication tasks are known, because the computational execution windows are previously calculated. The computational analyzer (Comp Analyzer) in Figure 4.4 determines and analyzes the computational phases Φ_{τ_i} to schedule the computational tasks with a TTCP scheduling policy. The main idea of the computational analyzer is to define an efficient feasibility test (Section 4.3) and design a greedy heuristic (Section 4.5), which assigns the phases in a feasible manner. The communication analyzer (Comm Analyzer) is similar to the computational analyzer because it also uses a greedy heuristic with an *a priori* defined feasibility test. Of course, the system can only be feasibly scheduled if both analyzers return a feasible solution.

If no feasible result can be found, there may exist no feasible result at all. An alternative is to formulate the scheduling problem based upon equations and let a solver find the feasibility results. The solver finds a solution if a feasible solution exists. However, the solver may take longer than a certain time limit and return no solution. For comparison, this solver-based approach is presented in Section 4.5.

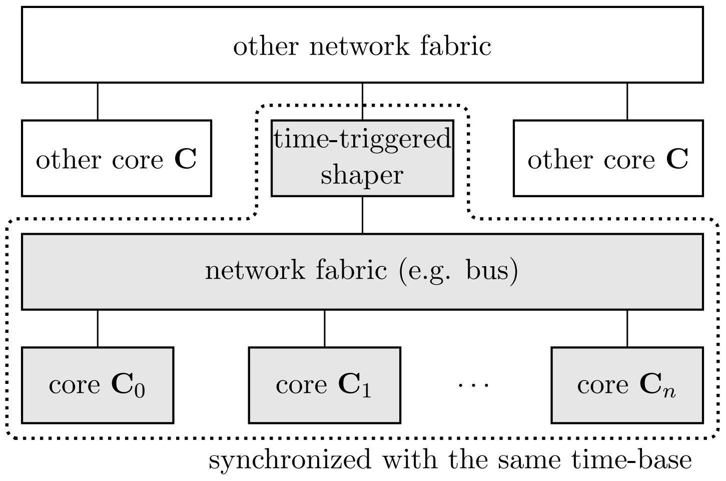


Figure 4.5.: According to the platform model, each core injects traffic to the network fabric in a time-triggered manner. Therefore, the cores need to be synchronized with the same time base for a tight analysis. Unsynchronized cores are connected via another network fabric and through a synchronisation block (time-triggered shaper), which needs to be analyzed separately.

Remark for integration with non-TTCP components

The TTCP approach requires that all cores are synchronized with the same time for executing the *a priori* defined TTCP schedule. This section discusses an alternative if cores are operated with a different time base.

In general, the TTCP approach implies predictable and periodic activations to trigger the computation and communication tasks. Upon first glance, it may seem to be restricted as this excludes the possibility of having cores with a different time base. A component called *time-triggered shaper* is introduced to further schedule the traffic injected from these cores in a time-triggered manner to keep the TTCP schedule feasible. The unsynchronized cores can still send data to the TTCP scheduled core, as in the scenario described in Figure 4.5. The communication analysis of TTCP scheduled cores together with unsynchronized cores (*other cores* in Figure 4.5) is easier than totally unsynchronized cores because the arrival times are partly known.

The time-synchronized cores including the *time-triggered shaper* compose a time-triggered domain. The *time-triggered shaper* can avoid collisions of messages from outside the time-triggered domain to safeguard the TTCP schedule. This method separates the time- and non-time-triggered tasks.

4.3. Computational Analysis for the TTCP Scheduled Tasks

This section presents the feasibility analysis of the computational task set $\mathbf{T}$ for a *multi-core* platform. The computational analysis is part of the computational analyzer, as shown in the workflow in Figure 4.4. In order to determine a feasible set of computational phases $\Phi_\mathcal{T}$, first the feasibility test is defined, which is used for defining the phase assignment algorithm.

The TTCP schedule ensures that each task can be executed in its pre-defined time window. Hence, a task execution window is not allowed to overlap with another window. For the feasibility analysis, a computational task τ_i is legally executed during the time interval

$$[\Phi_{\tau_i} + k \cdot P_{\tau_i}, \Phi_{\tau_i} + k \cdot P_{\tau_i} + W_{\tau_i}) \,, \qquad \forall k \in \mathbb{Z}_0^+, \tag{4.2}$$

which is *a priori* known by the TTCP scheduler. This time-overlap test is performed by first determining the overlap for two computational tasks and iterating this overlap test for all computational task pairs.

In the following, the time-overlap test based upon two computational tasks is presented. Marouf et al. [63] introduced an efficient feasibility test for the TTCP scheduler with the target of single-core scheduling. The time-overlap test is defined slightly differently, whereby it is easier to construct an algorithm.

Lemma 1 states an elementary property for handling non-harmonic periods. Theorems 2 and 3 define the feasibility test for two computational tasks. For harmonic periods, the feasibility test can be simplified to a single overlap test by virtually shifting the execution window of the computational task with a higher period, as shown in Figure 4.6. For arbitrary periods, Figure 4.7 visualizes Theorem 2 as a graphical representation.

Lemma 1. *Suppose that $a, b \in \mathbb{Z}^+$ with gcd$(a, b) = 1$, then for a given integer $c \in \mathbb{Z}$*

$$\exists \alpha \cdot a - \beta \cdot b = c, \tag{4.3}$$

with $\alpha, \beta \in \mathbb{Z}_0^+$ and gcd is the greatest common divisor.

Proof. This Lemma is partly proven by [16] on page 324 (5.198b). The gcd comprises a linear combination of two integers

$$\gcd(a_0, a_1) = c_{n-2}a_0 + d_{n-2}a_1, \tag{4.4}$$

where $a_0, a_1 \in \mathbb{Z}_0^+$ and $c_{n-2}, d_{n-2} \in \mathbb{Z}$. This implies

$$c \cdot \gcd(a, b) = c \cdot (\alpha \cdot a + \beta \cdot b), \tag{4.5}$$

which proves the existence of the linear combination of two integer numbers. □

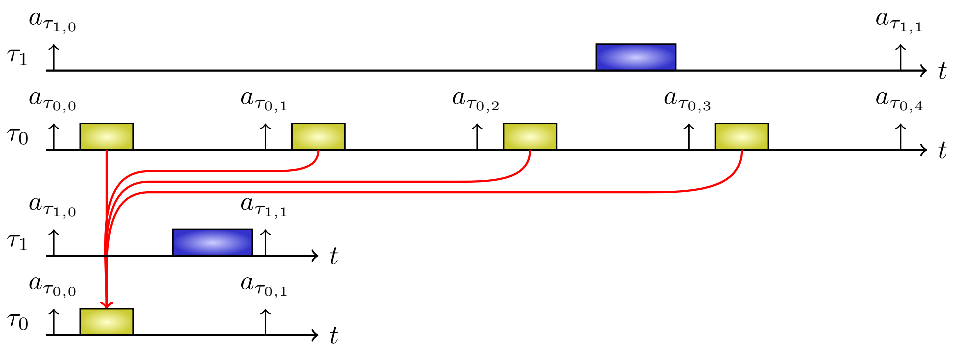


Figure 4.6.: A conflict detection example with harmonic periods between two tasks τ_0 and τ_1: All jobs of τ_0 are periodic, whereby the jobs of τ_1 can be shifted into a common interval $I = (0 \dots P_{\tau_1})$ to check for a conflict.

Theorem 2. *For two periodic time-triggered constant phase scheduled tasks τ_i and τ_j with known Φ_{τ_i} and Φ_{τ_j}, the computational task set is feasible if, and only if, $\forall 0 \leq \delta_i < W_{\tau_i}$, and $0 \leq \delta_j < W_{\tau_j}$:*

$$\Big(\Phi_{\tau_i} + \delta_i\Big) \, mod \; gcd_{i,j} \neq \Big(\Phi_{\tau_j} + \delta_j\Big) \, mod \; gcd_{i,j}, \tag{4.6}$$

where $gcd_{i,j}$ is the greatest common divisor of the periods P_{τ_i} and P_{τ_j}.

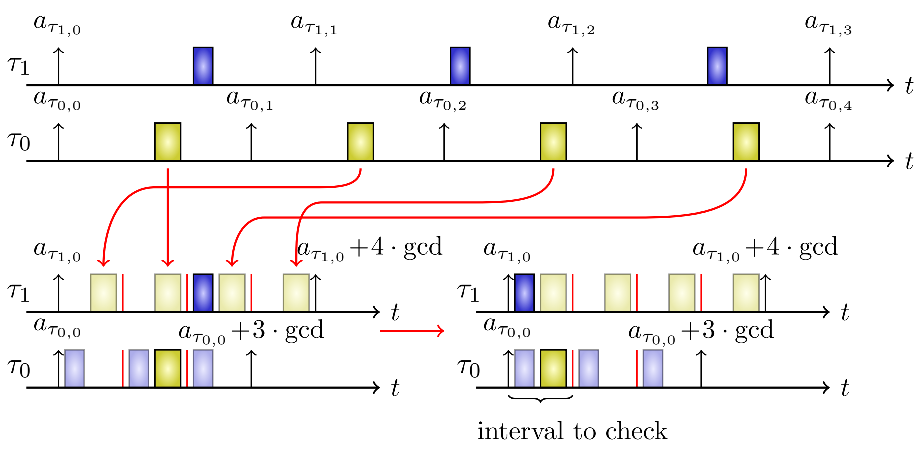


Figure 4.7.: A conflict detection example with arbitrary periods between two tasks τ_0 and τ_1: All jobs of τ_1 can theoretically be shifted into a common interval $I = (0 \ldots \text{gcd}(P_{\tau_0}, P_{\tau_1}))$. The jobs in this interval are periodic, whereby a potential conflict can be detected in this single interval.

Proof. All job executions of a task τ_i can be expressed by

$$t_{\text{execution}} = \Phi_{\tau_i} + \delta_i + k \cdot P_{\tau_i}, \qquad k \in \mathbb{Z}_0^+, \tag{4.7}$$

for any δ_i in the range of $[0, W_{\tau_i})$ represents the execution time. Therefore, the task set is not feasible by using TTCP **if, and only if**, there exist $k, l \in \mathbb{Z}_0^+$, $0 \leq \delta_i < W_{\tau_i}$, and $0 \leq \delta_j < W_{\tau_j}$ such that

$$\Phi_{\tau_i} + \delta_i + k \cdot P_{\tau_i} = \Phi_{\tau_j} + \delta_j + l \cdot P_{\tau_j}. \tag{4.8}$$

By adopting Lemma 1, its known that Equation (4.8) holds **if, and only if**, there exist $q \in \mathbb{Z}$, $0 \leq \delta_i < W_{\tau_i}$, and $0 \leq \delta_j < W_{\tau_j}$ such that

$$\Phi_{\tau_i} + \delta_i = \Phi_{\tau_j} + \delta_j + q \cdot \text{gcd}_{i,j}, \tag{4.9}$$

where q is defined as $\frac{k \cdot P_{\tau_i} - l \cdot P_{\tau_j}}{\text{gcd}_{i,j}}$. Therefore, Equation (4.9) holds **if, and only if**,

$$q = \frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}} - \frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}}. \tag{4.10}$$

Given that q is an integer, Equation (4.10) holds **if, and only if**, the *fractional part* of $\frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}}$ is equal to the fractional part of $\frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}}$. Accordingly, Equation (4.10) has the structure $q = \big(a + r\big) - \big(b + r\big)$, $a, b \in \mathbb{Z}$ and $0 \leq r < 1, r \in \mathbb{R}$. By using the flooring operation, the result is sill the same:

$$\lfloor a + r \rfloor - \lfloor b + r \rfloor = a - b = q = \big(a + r\big) - \big(b + r\big). \tag{4.11}$$

As a result, Equation (4.10) holds **if, and only if**, there exists an integer q with

$$q = \frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}} - \frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}} = \left\lfloor \frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}} \right\rfloor - \left\lfloor \frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}} \right\rfloor \tag{4.12}$$

$$\Leftrightarrow \frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}} - \left\lfloor \frac{\Phi_{\tau_i} + \delta_i}{\text{gcd}_{i,j}} \right\rfloor = \frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}} - \left\lfloor \frac{\Phi_{\tau_j} + \delta_j}{\text{gcd}_{i,j}} \right\rfloor \tag{4.13}$$

$$\Leftrightarrow \Big(\Phi_{\tau_i} + \delta_i\Big) \bmod \text{gcd}_{i,j} = \Big(\Phi_{\tau_j} + \delta_j\Big) \bmod \text{gcd}_{i,j}. \tag{4.14}$$

As a result, it is shown that the theorem holds. □

Theorem 3. *(computational task overlap) For two periodic time-triggered constant phase scheduled tasks τ_i and τ_j with known Φ_{τ_i} and Φ_{τ_j}, suppose that $\Psi_{\tau_i} = \Phi_{\tau_i}$ mod $gcd_{i,j}$, $\Psi_{\tau_j} = \Phi_{\tau_j}$ mod $gcd_{i,j}$, in which $\Psi_{\tau_i} \geq \Psi_{\tau_j}$ without loss of generality. These two tasks are feasibly scheduled by TTCP if, and only if,*

$$\begin{gathered}(W_{\tau_i} < gcd_{i,j} \textit{ and } W_{\tau_j} < gcd_{i,j} \textit{ and} \\ ((\Psi_{\tau_i} > \Psi_{\tau_j} + W_{\tau_j}) \textit{ and } (\Psi_{\tau_j} > \Psi_{\tau_i} + W_{\tau_i} - gcd_{i,j})))\end{gathered} \tag{4.15}$$

Proof. This comes directly from Theorem 2. It is not difficult to prove that there exist $0 \leq \delta_i < W_{\tau_i}$ and $0 \leq \delta_j < W_{\tau_j}$ such that $\left(\Phi_{\tau_i} + \delta_i\right)$mod $\text{gcd}_{i,j}$ is equal to $\left(\Phi_{\tau_j} + \delta_j\right)$mod $\text{gcd}_{i,j}$ **if and only if** all the conditions in Equation (4.15) hold. The proof is focused on the **if** part, because the **only-if** part is similar.

Clearly, if $W_{\tau_i} \geq \text{gcd}_{i,j}$, the operation $\left(\Phi_{\tau_i} + \delta_i\right)$mod $\text{gcd}_{i,j}$ covers any value in the range of $[0, \text{gcd}_{i,j})$, which will overlap with a certain $\left(\Phi_{\tau_j} + \delta_j\right)$ mod $\text{gcd}_{i,j}$. It is similar if $W_{\tau_j} \geq \text{gcd}_{i,j}$.

The other cases are now proven under the conditions $W_{\tau_i} < \text{gcd}_{i,j}$ and $W_{\tau_j} < \text{gcd}_{i,j}$. The proof is achieved through contrapositive. Suppose that tasks τ_i and τ_j cannot be feasibly scheduled by TTCP under the given Φ_i and Φ_j, which implies the existence of $0 \leq \delta_i^* < W_{\tau_i} < \text{gcd}_{i,j}$ and $0 \leq \delta_j^* < W_{\tau_j} < \text{gcd}_{i,j}$ such that

$$\Phi_{\tau_i} + \delta_i^* - \left\lfloor \frac{\Phi_{\tau_i} + \delta_i^*}{\text{gcd}_{i,j}} \right\rfloor \text{gcd}_{i,j} = \Phi_{\tau_j} + \delta_j^* - \left\lfloor \frac{\Phi_{\tau_j} + \delta_j^*}{\text{gcd}_{i,j}} \right\rfloor \text{gcd}_{i,j}. \tag{4.16}$$

The parameter $\ell \in \{i, j\}$ represents the operations when they work both for i and j. Due to $W_{\tau_i} < \text{gcd}_{i,j}$ and $W_{\tau_j} < \text{gcd}_{i,j}$, the fractional part with the flooring operation can result in the two different cases:

$$\left\lfloor \frac{\Phi_{\tau_\ell} + \delta_\ell^*}{\text{gcd}_{i,j}} \right\rfloor = \begin{cases} \left\lfloor \frac{\Phi_{\tau_\ell}}{\text{gcd}_{i,j}} \right\rfloor & \text{if } \Psi_{\tau_\ell} + \delta_\ell^* < \text{gcd}_{i,j} \\ \\ \left\lfloor \frac{\Phi_{\tau_\ell}}{\text{gcd}_{i,j}} \right\rfloor + 1 & \text{otherwise.} \end{cases} \tag{4.17}$$

Therefore, Equation (4.16) can be formulated as

$$\Phi_{\tau_\ell} + \delta_\ell^* - \left\lfloor \frac{\Phi_{\tau_\ell} + \delta_\ell^*}{\text{gcd}_{i,j}} \right\rfloor \text{gcd}_{i,j} = \begin{cases} \Psi_{\tau_\ell} + \delta_\ell^* & \text{if } \Psi_{\tau_\ell} + \delta_\ell^* < \text{gcd} \\ \Psi_{\tau_\ell} + \delta_\ell^* - \text{gcd}_{i,j} & \text{otherwise.} \end{cases} \tag{4.18}$$

By Equation (4.18), the existence of δ_i^* and δ_j^* to make Equation (4.16) hold implies that either one of the following cases occurs:

- case 1: $\Psi_{\tau_i} + \delta_i^* = \Psi_{\tau_j} + \delta_j^*$, if $\Psi_{\tau_i} \leq \Psi_{\tau_j} + W_{\tau_j}$
- case 2: $\Psi_{\tau_i} + \delta_i^* - \text{gcd}_{i,j} = \Psi_{\tau_j} + \delta_j^*$,if $\Psi_{\tau_i} + W_{\tau_i} - \text{gcd}_{i,j} \geq \Psi_{\tau_j}$
- case 3: $\Psi_{\tau_i} + \delta_i^* = \Psi_{\tau_j} + \delta_j^* - \text{gcd}_{i,j}$, true due to $\delta_j^* < \text{gcd}_{i,j}$
- case 4: $\Psi_{\tau_i} + \delta_i^* - \text{gcd}_{i,j} = \Psi_{\tau_j} + \delta_j^* - \text{gcd}_{i,j}$, if $\Psi_{\tau_i} \leq \Psi_{\tau_j} + W_{\tau_j}$

Therefore, it is known that either $(\Psi_{\tau_i} \leq \Psi_{\tau_j} + W_{\tau_j})$ or $(\Psi_{\tau_j} \geq \Psi_{\tau_i} + W_{\tau_i} - \text{gcd}_{i,j})$ should hold, if tasks τ_i and τ_j are not feasibly scheduled by TTCP under the given Φ_i and Φ_j. Hence, the proof can be concluded for the if-part due to contrapositive. □

Algorithm 1 Feasibility test for the computational tasks scheduled by a TTCP scheduler

```
Input: $\mathbf{T}$, $\tau_i \in f(W_{\tau_i}, P_{\tau_i}, D_{\tau_i}, \Phi_{\tau_i}, \Phi_{\tau_i,\min})$;
Output: Feasibility result;
 1: for $i = 0, \cdots, |\mathbf{T}| - 1$ stepped by 1 do
 2:   if $(\Phi_{\tau_i} < \Phi_{\tau_i,\min})$ or $(\Phi_{\tau_i} > D_{\tau_i} - W_{\tau_i})$ then
 3:     return "not feasible";
 4:   for $j = i + 1, \cdots, |\mathbf{T}| - 1$ stepped by 1 do
 5:     Calculate the greatest common divisor ($\mathrm{gcd}_{i,j}$) of $\tau_i$,$\tau_j$;
 6:     $\Psi_{\tau_i} \leftarrow \Phi_{\tau_i} \bmod \mathrm{gcd}_{i,j}$;
 7:     $\Psi_{\tau_j} \leftarrow \Phi_{\tau_j} \bmod \mathrm{gcd}_{i,j}$;
 8:     if $(\Psi_{\tau_i} < \Psi_{\tau_j})$ then
 9:       if $(\Psi_{\tau_j} < \Psi_{\tau_i}+W_{\tau_i})$ or $(\Psi_{\tau_i}+\mathrm{gcd}_{i,j} < \Psi_{\tau_j}+W_{\tau_j})$ then
10:         return "not feasible";
11:     else
12:       if $(\Psi_{\tau_i} < \Psi_{\tau_j}+W_{\tau_j})$ or $(\Psi_{\tau_j}+\mathrm{gcd}_{i,j} < \Psi_{\tau_i}+W_{\tau_i})$ then
13:         return "not feasible";
14: return "feasible";
```

Note that the feasibility test can also be performed by unfolding the time-triggered schedule to all individual jobs and perform the time-overlap test based upon the jobs. For this case, the schedule only needs to be unfolded until the hyper-period, because the time-triggered schedule ensures the periodic behavior in further cycles.

In addition to the feasibility test for the computational tasks, the proposed approach defines a range of valid phases for the communication tasks. A computational task may require data from other tasks before executing. Accordingly, there exists a lower bound for the computational phase called minimum computational phase $\Phi_{\tau_i,\min}$. This lower bound $\Phi_{\tau_i,\min}$ ensures that the incoming data for the tasks already arrived at the starting time s_{τ_i}. For typical industrial applications, the relative deadlines are implicitly defined by the period $D_{\tau_i} = P_{\tau_i}$. In case of constraint deadlines $D_{\tau_i} \leq P_{\tau_i}$, the relative phases also limit the range of valid phases. Feasible computational phases Φ_{τ_i} need to fulfill Theorem 3 and are in the range

$$\Phi_{\tau_i,\min} \leq \Phi_{\tau_i} \leq D_{\tau_i} - W_{\tau_i}. \tag{4.19}$$

If there is a time overlap between any two computational tasks the phases are infeasible.

According to (4.19) and Theorem 3, the time-overlap test between two tasks can be calculated by an algorithm with constant time complexity $O(1)$. Algorithm 1 presents the feasibility test for all computational tasks τ, which includes the time-overlap test. Algorithm 1 iterates over all computational task pairs and performs the time-overlap test based upon (4.15) from Theorem 3. The time complexity of Algorithm 1 is dominated by the two *for-loops*, which loop over all computational tasks $|\mathbf{T}|$ times. Therefore, the time complexity is polynomial with $O(|\mathbf{T}|^2)$, where $|\mathbf{T}|$ is the number of computational tasks.

This feasibility test is used to build our heuristic algorithm to determine a feasible assignment of computational phases. This heuristic algorithm and other approaches are presented in Section 4.5. Another problem is the communication analysis and the corresponding time window assignment, which is presented in the following section.

4.4. Communication Analysis for Given Phases

This section describes the feasibility analysis of the communication tasks κ, which are scheduled on a *multicore* platform with a bus architecture. According to the workflow

(Figure 4.4), the communication analysis is represented by the communication analyzer. In general, many different communication arbitration policies are conceivable to regulate the access to the communication fabric. The system model assumes a priority-based bus, which transmits packets in a non-preemptive manner. This chapter assumes only non-precedence communication tasks. In Chapter 5, communication tasks of type κ_j*-precedence* are taken into consideration.

The TTCP approach injects the communication packets at an *a priori* known time window such that the worst-case injection pattern can be tightly bounded. The communication packets are injected into the communication fabric at the end of its execution window. Thus, the communication arrival time a_{κ_j} is defined as

$$a_{\kappa_{j,l}} = l \cdot P_{\kappa_j} + \Phi_{\tau_{\mathrm{SRC}_j}} + W_{\tau_{\mathrm{SRC}_j}} \qquad l \in \mathbb{Z}_0^+, \tag{4.20}$$

where τ_{SRC_j} is the source computational task for sending this communication task κ_j. Note that the timing definitions of communication tasks are relatively defined based upon the arrival time of the computational jobs.

A common approach to define the period P_{κ_j} of the communication task κ_j is to send the data after each computational job execution. If the periods of the source and destination task are different $P_{\tau_{\mathrm{SRC}_j}} \neq P_{\tau_{\mathrm{DST}_j}}$, some data may not be further processed at the destination. Therefore, in the proposed approach, the period P_{κ_j} is defined based upon the maximum period of the source and destination task with

$$P_{\kappa_j} = max(P_{\tau_{\mathrm{SRC}_j}}, P_{\tau_{\mathrm{DST}_j}}). \tag{4.21}$$

to avoid futile communications. Hence, the communication fabric can be more effectively used by communications. The first job sends data to the destination, although in an industrial application a system designer can choose, which job sends the data. Theoretically, any job could be selected as long as it is always the same job in the hyper-period, given that the communication needs to be predictable.

For a feasible communication task set, each packet has to arrive before its relative deadline D_{κ_j},

$$R_{\kappa_j} \leq D_{\kappa_j} \qquad \forall \kappa_j, \tag{4.22}$$

where the WCTRT R_{κ_j} is the maximum time between the arrival $a_{\kappa_{j,l}}$ of the communication packet and its full transmission to the destination including the contention on the bus. In the following, the analysis for obtaining the WCTRT and the definition of the relative communication deadline is presented.

WCTRT analysis

The benefit of the TTCP scheduler is in knowing the arrival times of the communication tasks *a priori*. Hence, the WCTRT can be tightly bound. There are different possibilities to determine the WCTRT(κ_j), through either a formal upper-bounded approximation or a simulation-based approach. A simple method is to simulate the schedule until the hyper-period and extract the WCTRT(κ_j) based upon the constructed schedule. The property of a predicable schedule by the TTCP principle and the existence of a hyper-period enables this simulation-based approach to be a tight upper bound.

Figure 4.8 presents the constructed bus schedule for the motivational example (Section 4.1). In general, this communication analysis results in $\sum_{j=1}^{|\mathbf{K}|} \frac{H}{P_{\kappa_j}}$ packets, which need to be placed in the schedule. Note that H represents the hyper-period of the communication periods. To construct the communication schedule, tasks are analyzed chronologically

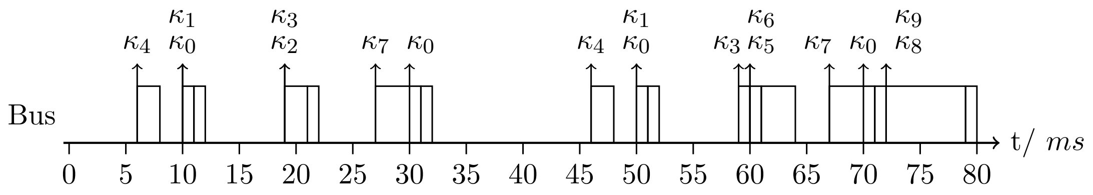


Figure 4.8.: Due to the TTCP scheduler, the bus schedule can be constructed from the motivational example within one hyper-period $[0 \dots 80]ms$. The WCTRT R_{κ_4} of κ_4 is $R_{\kappa_4} = 2 = W_{\kappa_4}$, because this communication task has no interference to other communication tasks. The arrows represent the communication arrival times.

to determine their specific time windows until the hyper-period is reached. During this analysis, the WCTRT R_{κ_j} can be determined by finding the maximum response time of all individual packets of each communication task κ_j. In case of bus contention, the bus arbitration policy decides which packet is transmitted first.

By contrast, the worst-case response time analysis for sporadic communications is more pessimistic. If the communication injection times are not known, all communications may arrive at the same time. Thus, for a fixed-priority non-preemptive scheduled bus, the WCTRT is calculated according to

$$\text{WCTRT}_{\text{non time-triggered}}(\kappa_j) = t_{\text{nP}}(\kappa_j) + t_{\text{HP}}(\kappa_j), \tag{4.23}$$

where t_{nP} and t_{HP} are the blocking times from the non-preemptive and higher-priority communication tasks, respectively. These blocking times are calculated according to

$$t_{\text{nP}}(\kappa_j) = \max_{\forall \kappa_\ell \in \mathbf{K} | \ell \neq j} (W_{\kappa_\ell}) \tag{4.24}$$

$$t_{\text{HP}}(\kappa_j) = W_{\kappa_j} + \sum_{\forall \kappa_\ell \text{ with higher priority}} \left(W_{\kappa_\ell} \left\lceil \frac{t_{\text{HP}}(\kappa_j)}{P_{\kappa_\ell}} \right\rceil \right). \tag{4.25}$$

Note that the arrival times are not *a priori* known for a fixed-priority scheduler, although the periodic release pattern of the tasks ensures no arrival jitter.

For demonstration, Figure 4.9 provides an example of the WCTRT for both methods. This figure shows three different methods to set the bus schedule. The two presented methods of a priority-based bus with and without the known arrival times of the communication tasks are named *TTCP-FPnP* and *FPnP*. Another arbitration policy is Time Division Multiple Access (TDMA), which is also shown for comparison.

Calculation of the relative communication deadline

Another problem related to the communication analysis is the relative communication deadline. The communications tasks of type τ_j*-non-precedence* also request data at least a certain time bound. The TTCP scheduler communicates the data based upon the computational phases and the system model (Section 3.3). There are only two cases for non-precedence communication tasks to define the absolute deadline:

- $a_{\tau_{\text{SRC}_j},h} + \Phi_{\tau_{\text{DST}_j}}$ if $P_{\tau_{\text{SRC}_j}} \leq P_{\tau_{\text{DST}_j}}$ and $P_{\tau_{\text{SRC}_j}} \leq \Phi_{\tau_{\text{DST}_j}}$, or
- $a_{\tau_{\text{SRC}_j},h} + \Phi_{\tau_{\text{DST}_j}} + P_{\tau_{\text{SRC}_j}}$ if $P_{\tau_{\text{SRC}_j}} > P_{\tau_{\text{DST}_j}}$ or $P_{\tau_{\text{SRC}_j}} > \Phi_{\tau_{\text{DST}_j}}$.

To shorten the relative deadline D_{κ_j}, the equation can be represented with respect to the arrival time of a job of its source computational task as follows:

$$D_{\kappa_j} = \Phi_{\tau_{\text{DST}_j}} + P_{\tau_{\text{SRC}_j}} \left\lceil \frac{P_{\tau_{\text{SRC}_j}} - \Phi_{\tau_{\text{DST}_j}}}{P_{\tau_{\text{DST}_j}}} \right\rceil \tag{4.26}$$

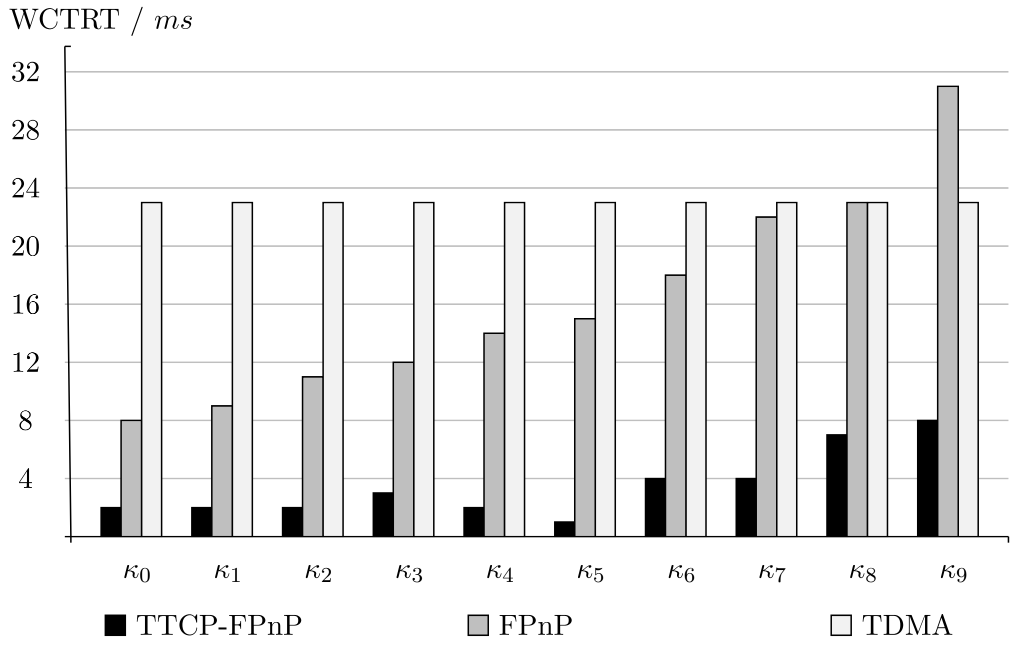


Figure 4.9.: WCTRT from motivational example: The analysis for the WCTRT in comparison of different non-preemptive priority bus arbitration policies from the motivational example from Section 4.1.

4.5. Phase Assignment Methods

Several methods to assign the computational and communication phases are presented in this section. Based upon the previous two sections, the feasibility test for the computational and communication task set $\mathbf{T}$, $\mathbf{K}$ is used to assign the phases Φ_τ, Φ_κ. This section provides an answer to the problem definition, which has to define a feasible phase assignment.

The three presented algorithms are the heuristic Lower Periods First (LPF), the heuristic Higher Periods First with Nested Bin-Packing (HPF-NB) and the Satisfiability Modulo Theories (SMT) solver approach, which are explained in the following. The heuristic algorithms greedily assign the phases Φ_τ, Φ_κ without reassigning. Hence, the heuristics need to define an order for the phase assignment and strategy to assign the phases regarding the already-assigned phases.

Heuristic Lower Periods First (LPF)

This heuristic sorts the computational tasks according to their periods, beginning with the lowest period. If two tasks have the same period, the tasks with a lower minimum computational phase $\Phi_{\tau_i,\min}$ are selected first. Thus, all computational tasks compose a task order Ω_τ.

The phase assignment strategy searches greedily for gaps in the schedule to determine each computational phase Φ_{τ_i}. Therefore, the proposed heuristic algorithm iterates chronologically through time to find a feasible phase regarding other computational tasks. To check, whether the phase is feasible by considering the already-assigned tasks, this heuristic uses the previously-defined feasibility test (Section 4.3). Algorithm 2 presents the proposed heuristic, which is explained in detail in the following.

First, Algorithm 2 orders the computational tasks according to the strategy with lower periods first and lower minimum phases first. The first *for-loop* iterates over all computational tasks τ and assigns the phases step by step. The proposed algorithm assumes a

Algorithm 2 Lower Periods First (LPF)

```
Input: computational task set $\mathbf{T}$;
Output: the computational phases $\Phi_{\tau_i}$;
 1: $\Omega_\tau \leftarrow$ Order $\tau_i$ in a non-descending period, same periods according to lower $\Phi_{\tau_i,\min}$ first;
 2: for $i = 0, \cdots, |\mathbf{T}| - 1$ stepped by 1 according to $\Omega_\tau$ do
 3:     $\Psi_{\tau_i} \leftarrow \Phi_{\tau_i,\min}$;
 4:     t $\leftarrow$ false;
 5:     while $\Psi_{\tau_i} < P_{\tau_i}$ and t = false do
 6:         t $\leftarrow$ true;
 7:         if $D_{\tau_i} - W_{\tau_i} < \Psi_{\tau_i}$ then
 8:             return "not feasible";
 9:         for $j = 0, \cdots, (i-1)$ stepped by 1 do
10:             Calculate the $\text{gcd}_{i,j}$ of $P_{\tau_j}, P_{\tau_i}$;
11:             $\Psi'_{\tau_i} \leftarrow \Psi_{\tau_i}$ mod $\text{gcd}_{i,j}$;
12:             $\Psi'_{\tau_j} \leftarrow \Psi_{\tau_j}$ mod $\text{gcd}_{i,j}$;
13:             if $(\Psi'_{\tau_j} < \Psi'_{\tau_i})$ then
14:                 if $(\Psi'_{\tau_i} < \Psi'_{\tau_j} + W_{\tau_j})$ then
15:                     $\Psi_{\tau_i} \leftarrow \Psi_{\tau_i} + \Psi'_{\tau_j} + W_{\tau_j} - \Psi'_{\tau_i}$;
16:                     t $\leftarrow$ false;
17:                 else if $(\Psi'_{\tau_j} + \text{gcd}_{i,j} < \Psi'_{\tau_i} + W_{\tau_i})$ then
18:                     $\Psi_{\tau_i} \leftarrow \Psi_{\tau_i} + \Psi'_{\tau_j} + \text{gcd}_{i,j} + W_{\tau_j} - \Psi'_{\tau_i}$;
19:                     t $\leftarrow$ false;
20:             else
21:                 if $(\Psi'_{\tau_j} < \Psi'_{\tau_i} + W_{\tau_i})$ then
22:                     $\Psi_{\tau_i} \leftarrow \Psi_{\tau_i} + \Psi'_{\tau_j} + W_{\tau_j} - \Psi'_{\tau_i}$;
23:                     t $\leftarrow$ false;
24:                 else if $(\Psi'_{\tau_i} + \text{gcd}_{i,j} < \Psi'_{\tau_j} + W_{\tau_j})$ then
25:                     $\Psi_{\tau_i} \leftarrow \Psi_{\tau_i} + \Psi'_{\tau_j} + W_{\tau_j} - (\Psi'_{\tau_i} + \text{gcd}_{i,j})$;
26:                     t $\leftarrow$ false;
27:     $\Phi_{\tau_i} \leftarrow \Psi_{\tau_i}$;
28: return all phases $\Phi_{\tau_i}$ and task set is "feasible";
```

hypothetical phase Ψ_τ, which is an intermediate result to find a valid phase. In the first iteration, Ψ_τ is the minimum computational phase $\Phi_{\tau_i,\min}$. Lines 10–31 in Algorithm 2 perform a feasibility test to validate the hypothetical phase with all already-assigned computational tasks. If the hypothetical phase Ψ_{τ_i} is feasible, the computational phase is set by the hypothetical phase $\Phi_{\tau_i} \leftarrow \Psi_{\tau_i}$. If the hypothetical phase Ψ_{τ_i} is invalid, the algorithm increments the hypothetical phase by a value such that the conflict cause by the feasibility test is solved. Lines 16, 19, 24 and 27 resolve the conflict by shifting the phase to the end of this particular conflict. The feasibility test is constructed such that this conflict-resolving is possible with low time complexity.

Due to the chronological search through the time to find a valid hypothetical phase Ψ_{τ_i}, no feasible phase exists for the already-assigned computational phases. In addition, each phase has a unique value, because otherwise it causes a conflict. With these properties, there exists an optimal task ordering for the computational tasks to find a feasible computational phase assignment.

The communication phases are determined based upon the computational phases with

$$\Phi_{\kappa_j} = \Phi_{\tau_{\text{SRC}_j}} + W_{\tau_{\text{SRC}_j}} \tag{4.27}$$

Higher Periods First with Nested Bin-Packing (HPF-NB)

In this heuristic algorithm, all computational tasks are ordered by Ω_τ according to the non-ascending period. In contrast to LPF, this strategy requires another greedy assignment method. Both approaches seem reasonable, although they perform completely differently.

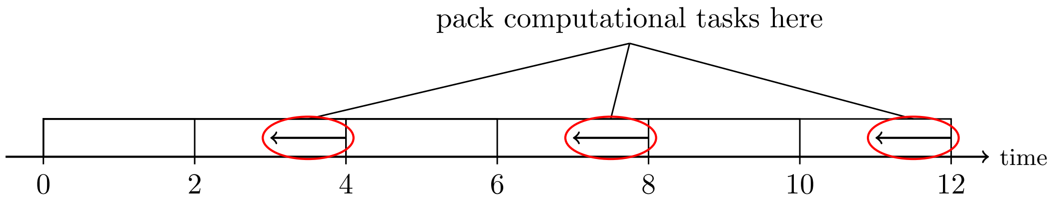


Figure 4.10.: The concept of packing higher periodic tasks into time windows (bins), which prevents an overlap with lower periodic tasks. E.g. twenty tasks with three different periods $\{2, 4, 12\}$ need to be assigned. The red circles represent the time windows, in which the computational tasks are assigned first.

Algorithm 3 Higher Periods First with Nested Bin-Packing (HPF-NB)

```
Input: computational task set $\mathbf{T}$ with harmonic period;
Output: all computational phases $\Phi_{\tau_i}$;
 1: sort task set according to their period, start with highest period;
 2: max size ← 0;
 3: for all different periods $\mathcal{P}_p \in \{P_{\tau_1}, \cdots, P_{\tau_r}\}$ by following $\Omega_\tau$ do
 4:     if $\mathcal{P}_p \neq P_{\tau_r}$ then
 5:         b ← $\frac{\mathcal{P}_p}{\mathcal{P}_{p+1}}$;
 6:     else
 7:         b ← 1;
 8:     bin$_\ell$ ← 0, $\forall \ell$;
 9:     bin$_0$ ← max size;
10:     for all $\tau_\ell$ with $P_{\tau_\ell} = \mathcal{P}_p$ do
11:         selected bin $f$ ← worst-fit bin-packing of $W_{\tau_\ell}$ in one of the $b$ bins;
12:         if $p \neq P_{\tau_r}$ then
13:             $\Phi_{\tau_i}$ ← bin$_f$ + $f \cdot \mathcal{P}_{p+1}$;
14:         else
15:             $\Phi_{\tau_i}$ ← bin$_f$;
16:         bin$_f$ ← bin$_f$ + $W_{\tau_\ell}$;
17:     max size ← max$_{\forall \ell}$(bin$_\ell$);
18: return all phases $\Phi_{\tau_i}$;
```

The concept of this heuristic is illustrated in Figure 4.10. The assignment of the phases Φ_{τ_i} of the higher periodic tasks can obstruct the lower periodic tasks such that no further feasible phase can be found. The computational tasks are assigned into bins, which are periodic time windows in the timeline, as shown in Figure 4.10. Therefore, the phases are packed in periodic time windows (bins) to reserve time for the lower periodic computational tasks.

Algorithm 3 presents the proposed heuristic to determine the phases based upon a bin-packing approach. After ordering the computational tasks into an order Ω_τ, the algorithm iterates over all different periods $\mathcal{P}$ in the task set $\mathbf{T}$. There are usually several tasks with the same periods, although each period $\mathcal{P}_p$ is processed only once. The parameter r represents the number of different periods in the task set $\mathbf{T}$. In each iteration of this *for-loop* (line 3), the heuristic performs a bin packing to a set of selected time windows depicted as red circles in Figure 4.10. The bins represent the timeline inside the hyperperiod H, which is divided into several time windows (bins) to improve the computational phase assignment. The number of bins b is determined by the division of two consecutive periods $\mathcal{P}_p$, $\mathcal{P}_{p+1}$. Line 9 sets up the bins for packing the computational tasks with the same period $P_{\tau_\ell} = \mathcal{P}_p$.

There may already be assigned tasks with a higher period, which can be represented by one bin with an initial capacity of *max size*. The *for-loop* (line 10) performs a worst-fit bin packing and assigns the computational phase Φ_{τ_ℓ} respecting their selected bin. After all tasks with the highest period are packed into their time windows, the maximum bin size (*max size*) is set to the initial bin size for the packing of tasks with the second highest period.

Note that this approach only supports task sets with harmonic periods, because the number of bins to pack needs to be an integer. As with the heuristic LPF, the communication phases are determined based upon the computational phases as shown in (4.27).

The Satisfiability Modulo Theories (SMT)-based approach

In contrast to the heuristic algorithms LPF and HPF-NB, this section presents another complete method for determining the computational phases Φ_{τ}. The Satisfiability Modulo Theories (SMT) solver is an algorithm to check, whether a set of equations is satisfiable by a certain parameter setting. In order to determine valid phases, an SMT problem needs to be formulated, which represents the feasibility test by a set of equations. The SMT approach is widely used in the literature [11, 61, 82].

There exist two possibilities to formulate the SMT problem. On the one hand, the SMT problem can be defined at a task-level by using the feasibility test from Theorem 3 and (4.19). On the other hand, the starting times of all individual jobs are known such that all equations can be defined based upon the job-level within the hyper-period. The experimental results show that formulating the problem at the task-level requires more run-time for the SMT solver than the job-level formulation. This thesis presents only the job-level formulation as a base-line for comparison, although both formulations were implemented in the experiments.

The Algorithm 13 (Appendix A.1) shows the SMT problem formulation for the job-level. If a feasible parameter setting exists that satisfies the SMT problem, the solver theoretically returns one valid parameter setting. In general, the SMT solver requires more time to determine the phases than the heuristic, owing to its exponential run-time complexity. In the experiments, run-time measurements confirm this behavior. Identically to the heuristics, the communication phases are determined based upon the computational phases as shown in (4.27).

4.6. Evaluations

In this section, experiments show the correctness of the TTCP approach and evaluate its performance in contrast to other approaches. The specific settings for the experiments are described in the following.

4.6.1. Experimental Setup

This section summarizes the experimental settings used in the experiments. Each experiment has an individual setting to show a particular effect. In general, the feasibility analysis returns a Boolean result. Synthetic random task sets $\mathbf{T}$, $\mathbf{K}$ are generated to evaluate the scheduling algorithm. The generated task sets have typical industrial characteristics. Tasks in the task sets $\mathbf{T}$, $\mathbf{K}$ have (i) mostly harmonic periods, (ii) are non-heavy tasks and (iii) there exists a large number (100–1,000) of computational and communication tasks.

The multicore platform comprises four cores, which are connected by a priority-based bus scheduled by a Rate Monotonic non-Preemptive (RMnP) scheduler [7]. Our strategy is to use a TTCP scheduler on each core and evaluate different phase assignment approaches. In addition, the TTCP approach is compared with a RMnP scheduler on each core. The RM or fixed-priority scheduling approach is often used in industry and is proven to be optimal [59] for harmonic periods and independent tasks. The RMnP scheduler performs similar to the RM scheduler with non-heavy tasks. Different versions of RMnP exist, whereby the experiments use an implementation from Bertogna et al. [7]. The proposed Algorithm 2 is capable of handling task sets with arbitrary periods. Due to the typical industrial characteristics, most experiments are performed with harmonic periods. One experiment examines the capabilities for non-harmonic period.

System set generator

The computational and communication task sets $\mathbf{T}$, $\mathbf{K}$ are randomly generated, which are described by the following by four steps.

1. Generate the four core platform with a bus architecture
2. Generate the computational task set $\mathbf{T}$
3. Task-to-core mapping
4. Generate the communication task set $\mathbf{K}$

These four steps are explained in the following.

Step 1: The platform comprises four individual schedulable cores, which share the same time (synchronization) such that the TTCP approach is applicable. To avoid bottlenecks in the instruction fetching, each core has sufficient local memory to store its program code and temporally-required data.

Step 2: First, the computational utilization $U_{\mathbf{T}}$ and the number of tasks $|\mathbf{T}|$ are defined. Second, the generator defines the first period P_{τ_1}. Due to the condition that a hyper-period exists, each period has an integer value. Based upon the first period, all periods can be calculated according to

$$P_{\tau_{i+1}} = P_{\tau_i} \cdot w_i \qquad w_i \in \{1, 2, 3\} \subset \mathbb{Z}^{|\mathbf{T}|-1}, \tag{4.28}$$

where w_i is a set of randomized numbers that varies in each period P_{τ_i}. In order to generate a random set w_i, the probabilities are calculated with

$$w_i = \min(\left\lceil \frac{2.5}{|\mathbf{T}| \cdot x_i} \right\rceil, 3) \qquad x_i \in (0 \ldots 1) \subset \mathbb{R}^{|\mathbf{T}|-1}, \tag{4.29}$$

which generates approximately five different periods. Third, a uniform distributed set of random numbers y_i is generated with $0 < y_i < 1$ for each task τ_i. Accordingly, the execution time can be defined with

$$W_{\tau_i} = \frac{P_{\tau_i} \cdot U_{\mathbf{T}} \cdot y_i}{\sum_{\tau_j \in \mathbf{T}} y_j}. \tag{4.30}$$

Fourth, the relative deadlines D_τ are calculated with another additional uniform distributed set of randomized numbers z_i with $0 < z_i < \frac{1}{4}$, thus,

$$D_{\tau_i} = P_{\tau_i} \cdot (1 - z_i). \tag{4.31}$$

Step 3: The scheduling analysis and phase assignment methods assume a given task mapping. As the task mapping is not focused in this thesis, a simple bin packing heuristic distributes the task set $\mathbf{T}$ to the cores $\mathbf{C}$. The generator applies the worst-fit bin-packing approach to assign the computational task τ to the cores. Each computational task utilization $\frac{W_{\tau_i}}{P_{\tau_i}}$ is packed into a number of bins $|\mathbf{C}|$ such that the load is balanced.

Step 4: To generate the communication tasks κ_j, the bus utilization U_{bus} and the number of communication tasks $|\mathbf{K}|$ is defined. Each κ_j is randomly mapped to two randomized tasks τ_{SRC_j} and τ_{DST_j} placed on different cores with $c_{\tau_{\text{SRC}_j}} \neq c_{\tau_{\text{DST}_j}}$. The periods P_{κ_j} are calculated according to equation (4.21) and the relative deadlines according to (4.26). The traversal times W_{κ_j} are computed similar to the execution times W_{τ_i} in (4.30) by given U_{bus} and P_{κ_j}.

SMT solver-related settings

The hardware for the experiments is an Intel i5-2520M processor with 2.5GHz and 8GB RAM. There are several SMT solver implementations available, such as STP [19], miniSmt [97], or Z3 [27]. For the evaluation, the Z3 solver implementation (version 4.3.2.0) [27] is chosen for the SMT solver.

The solver is set to stop its execution beyond 10 min as a timeout bound. After a timeout, the solution is neither feasible nor infeasible. The resulting plots show the timeouts as a gray area to indicate the uncertainty in the experiments. It emerges that the SMT solver takes too much time for larger computational task sets. Therefore, the SMT solver experiments are performed with only 100 tasks. Note that for larger computational task sets, the SMT solver also failed due to its memory requirement. The SMT solver problem definition and solver itself have an exponential memory requirement relative to the number of tasks.

4.6.2. Experimental Results

This section presents the experiments with their results to demonstrate the proposed TTCP approach and heuristic phase assignment algorithm. Each experiment uses the default setting described above, unless otherwise specified.

Dynamic scheduling approach

This experiment evaluates the feasibility of computational tasks scheduled by different approaches. The proposed two heuristics LPF and HPF-NB are compared with a dynamic scheduler, namely RMnP. The feasibility test RMnP scheduler is implemented according to the literature [7], which assumes the maximum blocking time and the worst-case interference of all higher-priority tasks. Figure 4.11 shows the comparison, in which the simulator generates $1,000$ randomized system sets, assigns the priorities or phases and plots the feasibility ratio.

If only non-heavy computational tasks exist, RMnP is close to the results of Rate-Monotonic preemptive, which has a very high utilization rate $U_\mathcal{T} \approx 100\%$. The LPF algorithm performs very well and is capable of reaching a utilization above 90%, which reaches almost the same maximal utilization as RMnP. For the HPF-NB algorithm, the relative deadlines are implicitly defined $D_{\tau_i} = P_{\tau_i}$, where the parameter z_i is set to zero only for this algorithm. Even for this simplification, the experiment shows that the HPF-NB algorithm has a lower maximal utilization barrier.

Furthermore, the experiments show that the heuristic LPF algorithm runs approximately $5s$ for large computational task sets with $1,000$ tasks and $U = 75\%$. Considering an offline calculation of the parameters Φ_{τ_i}, a maximum run-time of $5s$ is satisfactory for practical applications.

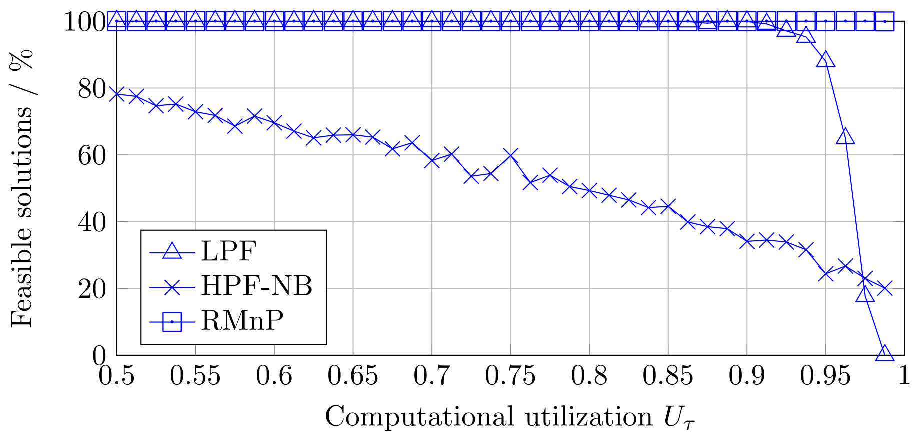


Figure 4.11.: Dynamic scheduling approach: The proposed heuristic approach compared to Rate Monotonic non-Preemptive (RMnP).

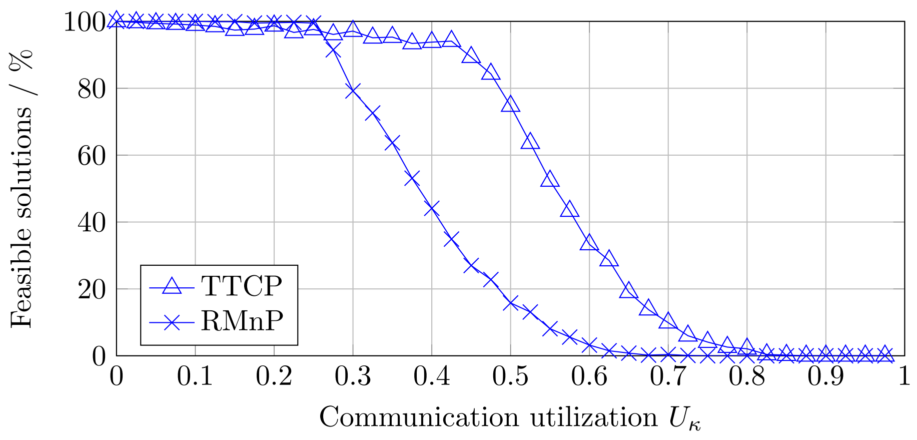


Figure 4.12.: Communication enhancement: Different bus utilizations by constraining the relative communication deadlines D_{κ_j}.

Communication enhancement

Another experiment focuses on the network analysis. Similar to the motivational example, this experiment considers a four-core platform with 250 computational tasks and 500 communication tasks randomly mapped to each other. Figure 4.12 presents a feasibility ratio for the system with constraint deadlines D_{κ_j}, in which different bus utilizations are explored.

Each of the four cores has a 50% task utilization in this experiment. The bus is scheduled with a non-preemptive fixed-priority arbitration e.g. a CAN bus in which the communication arrival time $r_{j,l}$ is calculated with different approaches. Figure 4.12 shows that TTCP reaches a higher maximum bus utilization. The *a priori* knowledge of the arrival times of the communication tasks facilitates a tighter communication analysis, which increases the feasibility ratio of the bus in the experiment; otherwise, the worst case has to be considered, where all communication tasks arrive at the same time.

Harmonic task sets

In the following, the SMT solver is applied to evaluate the efficiency of the proposed LPF heuristic. Figure 4.13 presents the algorithm under harmonic periodic. Note that number of tasks is set to 100 and no heavy task is in the task set. The heuristic can reach over 90% task set utilization, which is close the result of the exhaustive search approach.

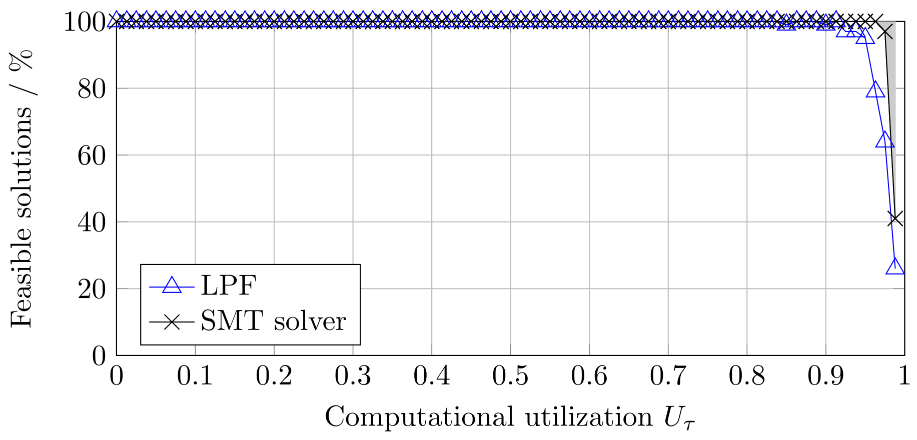


Figure 4.13.: Harmonic task sets: Each task set comprises 100 tasks with harmonic periods.

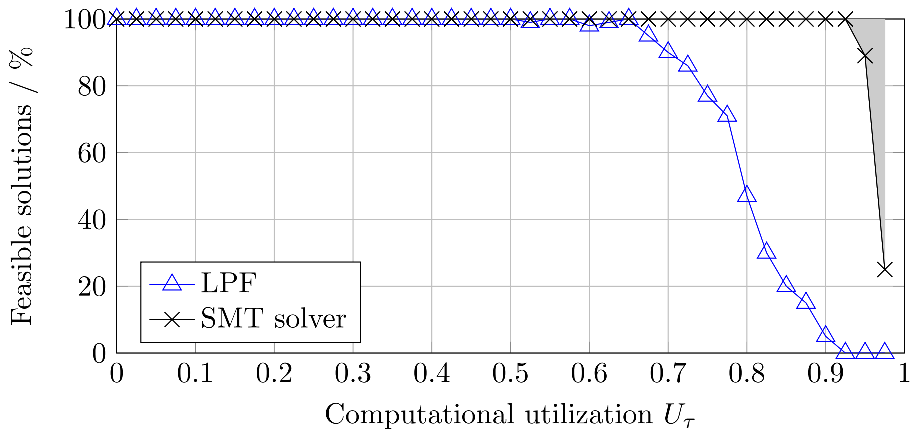


Figure 4.14.: Non-harmonic tasks: All tasks randomly have a period of {1,2,5,10,20,50,100} ms, which are not harmonic.

Non-harmonic tasks

This experiment examines the case of arbitrary periods based upon industrial characteristics, defining a set of non-harmonic periods. 100 tasks have a randomly period of the set {1,2,5,10,20,50,100} ms. The proposed algorithm LPF can now reach only 65% utilization (see Figure 4.14). By contrast, the SMT solver again reaches over 90%. In order to reach high utilization, the heuristic no longer works.

Influence of heavy tasks

The existence of a feasible phase assignment significantly depends on the heavy tasks, namely those with the highest task set utilization. For harmonic periods, the execution time of the heavy task needs to be smaller than the lowest period in the task set

$$\max_{\forall i}(W_{\tau_i}) < \min_{\forall i}(P_{\tau_i}).$$

Otherwise, no feasible assignment exists. As shown in Figure 4.15, increasing the heavy task utilization leads to less feasible solutions. The task set has 100 tasks with an overall 75% utilization and harmonic periods. The LPF algorithm has a slightly lower feasibility rate than the exhaustive search for heavy tasks under 10%. However, for larger heavy tasks, the SMT solver more commonly finds a feasible solution than the heuristic. Furthermore, if the heavy task is assigned to the lowest period of the task set, the probability of finding a feasible phase assignment is very high.

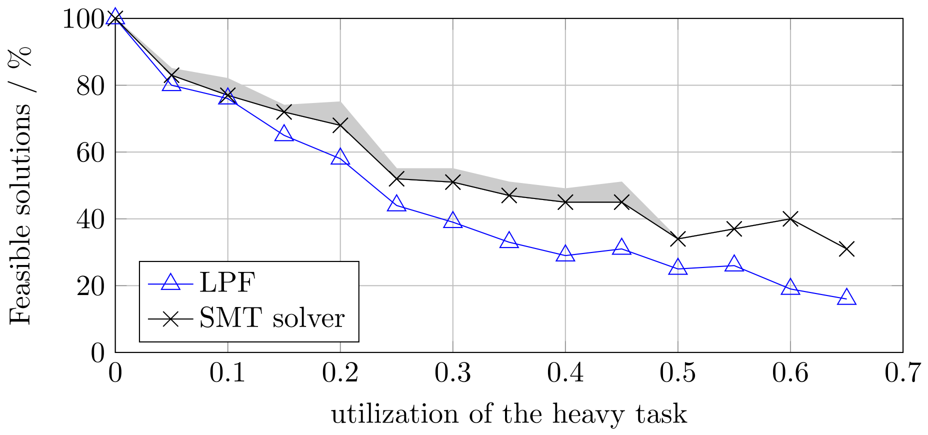


Figure 4.15.: Influence of heavy tasks: The task generation is similar to the harmonic task generation, although a random task has a higher utilization.

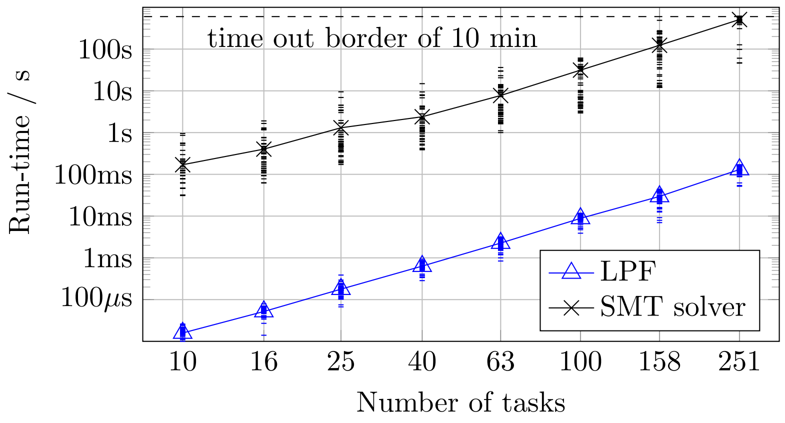


Figure 4.16.: Run-time measurements: The run-time of the phase assignment is measured by using the proposed heuristic or an SMT solver for different sizes of task sets. The linear ramp in the double logarithmic scale indicates an exponential growth considering the number of tasks.

Run-time measurements

This experiment measures the time to find a solution for a certain task set with the SMT solver and the LPF heuristic. Thereby, the feasibility of the solution does not matter. In this experiment, 50 task sets with 75% utilization with harmonic non-heavy tasks are generated to measure the run-time of the phase assignment. The algorithm sometimes terminates earlier, if the task set is infeasible. Note that the run-time also depends on the overall task set utilization, because particularly the SMT solver requires much more time if the solution space is small. Considering a larger hyper-period or an increasing number of tasks, the SMT solver requires more resources. For such cases, the memory or run-time requirements could exceed the limits, whereby the solver is incapable of returning a solution. With such conditions, the heuristic algorithm may still provide a solution using a moderate amount of resources. Figure 4.16 shows that in general the heuristic is approximately three magnitudes (10^3) faster than the SMT solver.

5. Time-Triggered Computation and Communication Analysis by using a NoC

This chapter applies the proposed TTCP approach to a manycore platform with a Network-on-Chip (NoC) structure. An iterative approach can handle the dependent system model in which computational tasks communicate with each other. The schedulability analysis proposed in this chapter provides an efficient heuristic to compute a feasible schedule for both computation and communication tasks that meets its timing constraints. All methods and techniques proposed in this thesis have been published [34] by the author.

5.1. Introduction

The previous chapter presents methods that can be used to compute a feasible schedule for an application, which are executed on a multicore platform and communicate with a shared communication bus. It is also shown that architectures with a shared communication bus tend to be inefficient with an increasing computation and communication demand. Therefore, this chapter proposes methods to schedule a dependent system on a manycore platform with a NoC. This chapter presents an approach to compute a feasible time-triggered schedule for a dependent system model considering a manycore platform with a NoC.

In comparison to a shared bus, the NoC provides a higher bandwidth for communication, because it allows the possibility of simultaneous communication without resource conflicts. However, in order to fully utilize a NoC, communication tasks have to be carefully scheduled. Otherwise, packets can interfere with each other at a NoC switch, which can reduce the effective bandwidth. In addition, there are several side effects like *back pressure* or *indirect contention*, which can increase the run-time complexity of a communication analysis. *Back pressure* occurs if too many packets are sent over one switch such that the packets cannot be stored in the buffer of a switch. The problem is that succeeding packets can be blocked on another switch. This can cause *indirect contention*, where two packets interfere with each other even if they have independent routes. This chapter proposes avoiding contention between packets in the NoC to simplify the feasibility and response time analysis of a communication task set.

In the literature, a large design space for NoC architectures exists, as shown in Section 2.3. However, this chapter focuses on a DSPIN NoC architecture [67], which allows applying the TTCP approach to an effective communication infrastructure. For typical industrial applications, the run-time complexity of the approach becomes critical for its applicability. In particular, an industrial-sized application comprises 300–$3,000$ communication tasks, where the challenge is to find an arbitration policy to handle this number of communication tasks. In addition, this arbitration policy has to be analyzed to design a feasible communication schedule.

If the communication injection times of packets are not carefully defined *a priori*, the worst-case traffic pattern [70] significantly reduces the maximum possible NoC utilization. The main idea is to apply the TTCP approach to the NoC by defining the injection time of each communication task. There are two possibilities to apply the TTCP approach to the NoC structure.

1. Inject packets at the completion of the sending computational task and determine the worst-case traversal response time (WCTRT) by analyzing contention. Based upon the TTCP scheduled computational tasks, the completion times are *a priori* known.
2. Delay the injection of the communication task such that packets never interfere. In this case, the injection times have to be defined such that the communication task can meet its real-time constraints.

This thesis considers the second possibility as the adopted system model allows assigning priorities to communication tasks for the phase assignments, which can be leveraged to avoid contentions of urgent communication tasks. Thus, the proposed approach constructs a contention-free communication task schedule. Additionally, if all communication phases are determined, the WCTRT of each task can be easily determined by injection delay and the network traversal time.

However, there exists a cyclic dependency between the computation and the communication analysis. To determine time windows for the computational tasks, the WCTRT are required. However, the communication analysis determines the WCTRT based upon time windows for the computational task.

As a result, the approach proposed in this chapter performs the computational and communication analysis iteratively to derive a feasible solution. In the first iteration, the WCTRT are estimated, whereas in further iterations, these times are adjusted. The challenge is to define the computational and communication analysis such that the iterations converge to a feasible solution even for highly-utilized systems. The feasibility test presented in Section 4.3 is adapted to the NoC structure to build up a heuristic algorithm. This algorithm computes the TTCP phase for each communication and computational task.

Experiments confirm that large task sets can still be successfully scheduled by the proposed iterative approach. In comparison to a bus architecture, the NoC-based approach allows simultaneous communications, which increases the common bandwidth for all inter-core communications. For typical industrial task sets with $1,000$ computational tasks and $3,000$ communication tasks, the proposed approach can utilize a 3×3 NoC by around 60%. Thus, a core sends on average 60% of its time data to another core without interfering other packets.

Related work

This thesis specifically focuses on the DSPIN NoC [67] architecture [67], while other NoC architectures are summarized in Section 2.3. The communication analysis strongly depends on the given NoC architecture, although techniques already exist for analysing the communication response times [23, 46, 52, 78]. Kiasari et al. [46] summarize different NoC analysis methods.

One method is to apply network calculus [52, 78], which uses the *min-plus algebra* to calculate the interference between different packet streams. *Network Calculus* is a generic mathematical framework for analysing time bounds given a set of arrival and service curves. An arrival curve is a model that describes the arrival pattern of consecutive tasks with different impacts. A service curve is an abstract resource that can be utilized by the arriving tasks. Dasari et al. propose a branch-and-prune algorithm [23] to determine the communication response times in a NoC. By using network calculus, their branch-and-prune algorithm results in tighter models for the arrival and service curves, However, determining and bounding contention between different communication tasks remains difficult. By contrast, by assuming a contention-free TTCP schedule, the analysis can be enhanced to support the dependent system model and target typical industrial-sized applications comprising thousands of computational and communication tasks.

The time-triggered scheduling approach has also been extensively studied in the literature [33, 45, 64, 74]. In the TTCP approach [33, 74], each task is statically scheduled in a predefined time window. The time-triggered scheduling analysis with periodic tasks on a single-core platform is presented by Marouf and Sorel [64]. However, the available approaches support neither the dependent task model nor the NoC communication fabric.

Closely related, there are approaches for constructing the time-triggered schedule for a manycore platform with a NoC communication fabric [10, 22, 61, 82, 83]. However, these approaches rely on mathematical solvers to construct a feasible schedule. Specifically, the time-triggered scheduling problem is formulated as a set of equations to be solved by an Integer Linear Programming (ILP) or Satisfiability Modulo Theories (SMT) solver. For example, Biewer et al. [10] demonstrate the use of an SMT solver for constructing and analysing the time-triggered schedule. The main problem with the solver-based approach is the exponential run-time complexity. For problems with typical industrial-sizes, the solver-based approach is not always applicable.

Problem definition

Chapter 4 highlighted that the TTCP approach is capable of efficiently utilizing a multicore platform, as the communication tasks are known *a priori*. The idea is to apply the TTCP approach to a NoC structure to schedule the communication tasks. Therefore, this chapter assumes that each computational and communication task is scheduled under the TTCP policy with a corresponding time window, which starts at the so-called phase.

For a dependent system model scheduled by a TTCP scheduler on a manycore platform with a NoC, the problem comprises two parts:

1. Assigning a computational phase Φ_{τ_i} to each computational task τ_i, and
2. Assigning a communication phase Φ_{κ_j} to each communication task κ_j

such that all real-time constraints are satisfied.

In the following, this problem is solved within two steps. First, the feasibility analysis is defined to know whether a given set of computational and communication phases are valid. Second, methods are presented to determine the phases of communication and computation tasks based upon the feasibility analysis.

In contrast to a bus architecture, a NoC offers possibilities for simultaneous communication, although the communication analysis requires more advanced methods. The communication analysis on a NoC adapts the computational analysis from the core to schedule and determines the feasibility of the packets.

5.2. Scheduling Analysis on a Manycore Platform with a NoC

In order to solve the above-defined problem, this section defines the feasibility test of the dependent system model for a manycore platform with a NoC. This includes the computational and communication analysis.

Computational feasibility analysis

The basic computational feasibility test for the computational tasks is presented in Chapter 4.3. This chapter adapts the proposed basic feasibility test to handle computational tasks with precedence constraints, as stated in Definition 6 in Section 3.3. Theorem 3 provides an efficient test to determine whether a computational task overlaps in time with another computational task. In addition, the computational phases Φ_{τ_i} need to be in a certain range to be feasible

$$\Phi_{\tau_i,\min} \leq \Phi_{\tau_i} \leq D_{\tau_i} - W_{\tau_i}, \tag{5.1}$$

where $\Phi_{\tau_i,\min}$ is the minimum computational phase. In order to satisfy the precedence constraints, a computational phase Φ_{τ_i} needs to be greater than or equal to the computational phase of the preceding tasks Φ_{τ_ℓ}:

$$\Phi_{\tau_i,\min} = \max_{\forall \tau_\ell \in Q_{\tau_i}} (\Phi_{\tau_\ell} + W_{\tau_\ell}). \tag{5.2}$$

Note that the data from a predecessor task which is mapped to another core needs to be available at a certain time. This time is represented by the relative deadline of the corresponding communication task.

The computational task set is feasible if Equation (5.1) and Theorem 3 which is implemented in Algorithm 1 hold.

Communication feasibility analysis

Due to the proposed TTCP scheduling, the communication tasks are not allowed to overlap with each other. An overlap occurs if any two packets request the same link at the same time. Each communication task has a communication phase Φ_{κ_j}, which defines the offset for injecting a corresponding packet into the NoC. The specific offset ensures that packets do not interfere at a network switch, which is essential for achieving real-time simultaneous communication over a NoC. In particular, the time instants t_{inject}, at which packets of a communication task κ_j are injected into the NoC, are given by:

$$t_{\text{inject}} = \Phi_{\kappa_j} + P_{\kappa_j} \cdot l, \qquad l \in \mathbb{Z}_0^+ \tag{5.3}$$

The time-overlap analysis of the communication tasks is adapted from the time-overlap test in Theorem 3. Consequently, Lemma 2 reveals that there exists at most one switch in the given NoC, where packets from two communication tasks can interfere with each other.

Lemma 2. *(critical switch) Suppose two periodic time-triggered constant phase scheduled communication tasks κ_i and κ_j, which are scheduled by the TTCP approach with in X-Y routing on a 2-D mesh, these two communication tasks can only collide at most once at the switch $\mathbf{R}_{crit}$.*

Proof. Proven by Munk et al. [70]. □

Based upon Lemma 2, if there is no time overlap at the critical switch $\mathbf{R}_{\text{crit}}$, the communication task can be feasibly scheduled. To find the critical switch $\mathbf{R}_{\text{crit}}$ between two communication tasks κ_i, κ_j, the routes r_{κ_i}, r_{κ_j} need to be analyzed. The first shard link of the routes indicates the location of the critical switch $\mathbf{R}_{\text{crit}}$, since packets could be delayed at this switch.

Let $z_{i,j}$ be the number of hops on the routing path r_{κ_i} of task κ_i to the critical switch $\mathbf{R}_{\text{crit}}$ between task κ_j, i.e. κ_i and κ_j can collide at the critical switch $\mathbf{R}_{\text{crit}}$, where κ_i needs $z_{i,j}$ hops to $\mathbf{R}_{\text{crit}}$ via the x-y routing. A packets needs some time $d_{\mathbf{L}}$ to traverse to the critical switch $\mathbf{R}_{\text{crit}}$, while the NoC needs a finite time $d_{\mathbf{R}}$ to forward the packets to the

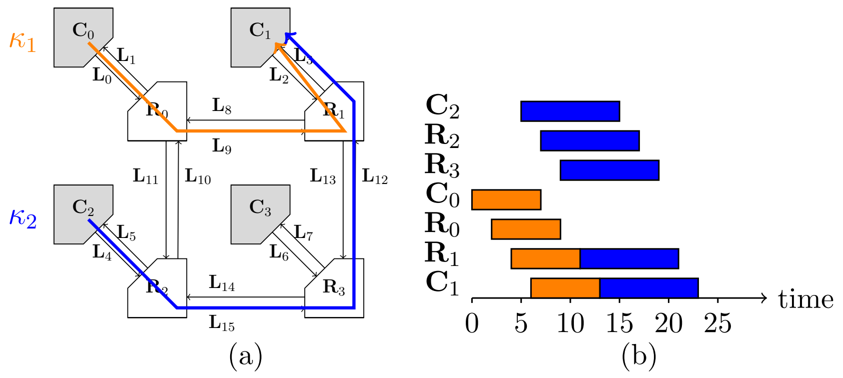


Figure 5.1.: Example of $M_{i,j}$: The routing (a) and the schedule (b) are presented for κ_1 and κ_2 colliding at switch $\mathbf{R}_{\text{crit}} = \mathbf{R}_1$, with route $r_{\kappa_1} = \{\mathbf{R}_0, \mathbf{R}_1, \mathbf{C}_1\}$, route $r_{\kappa_2} = \{\mathbf{R}_2, \mathbf{R}_3, \mathbf{R}_1, \mathbf{C}_1\}$, number of hops $z_{1,2} = 2$, number of hops $z_{2,1} = 3$, the network delays $d_{\mathbf{R}} + d_{\mathbf{L}} = 2ms$, $M_{1,2} = 4ms$, $M_{2,1} = 6ms$, $W_{\kappa_1} = 7ms$, $W_{\kappa_2} = 10ms$, $\Phi_{\kappa_1} = 0$, $\Phi_{\kappa_2} = 5$.

correct output link. For the communication analysis, these additional times in the NoC are described by a *conflict matrix* $M_{i,j}$, which is defined as

$$M_{i,j} = \begin{cases} z_{i,j}(d_{\mathbf{L}} + d_{\mathbf{R}}) & \text{if } r_{\kappa_i} \text{ overlaps with } r_{\kappa_j} \\ \emptyset & \text{otherwise.} \end{cases} \tag{5.4}$$

Figure 5.1 provides an example demonstrating the usage of the conflict matrix $M_{i,j}$. Theorem 4 makes use of the conflict matrix $M_{i,j}$ to formulate the time-overlap test for communication tasks.

Theorem 4. *(communication task overlap) Suppose two periodic time-triggered constant phase scheduled communication tasks κ_i and κ_j with $M_{i,j} \neq \emptyset$ and with known Φ_{κ_i} and Φ_{κ_j}. Let $\Psi_{\kappa_i} = (\Phi_{\kappa_i} + M_{i,j}) \bmod gcd_{i,j}$, $\Psi_{\kappa_j} = (\Phi_{\kappa_j} + M_{j,i}) \bmod gcd_{i,j}$ with $\Psi_{\kappa_i} \geq \Psi_{\kappa_j}$. These two tasks are feasibly scheduled by TTCP (in X-Y routing), if*

$$(\Psi_{\kappa_i} \geq \Psi_{\kappa_j} + W_{\kappa_j}) and \ (\Psi_{\kappa_j} \geq \Psi_{\kappa_i} + W_{\kappa_i} - gcd_{i,j}) \tag{5.5}$$

Proof. With X-Y routing and no back pressure in the NoC, two packets can collide at most once at a particular switch, which is proven in Lemma 2. Otherwise, this particular switch would sequentialize the packets such that no further conflict can occur. The elapsed time to the critical switch is represented by $M_{i,j}$ after the communicating task is started. Therefore, this critical switch can be considered as a resource that may be used by more than one task in the time-triggered manner. The collision detection on this critical switch is hence the same as in Theorem 3. □

Even if communication tasks do not conflict with each other at any $\mathbf{R}_{crit}$, communication phases can still be infeasible because they need to be in a certain range. This feasible range for each communication phase Φ_{κ_j} is given by:

$$\Phi_{\kappa_j,\min} \leq \Phi_{\kappa_j} \leq D_{\kappa_j} - W_{\kappa_j} - |r_{\kappa_j}|(d_{\mathbf{R}} + d_{\mathbf{L}}), \tag{5.6}$$

where $\Phi_{\kappa_j,\min}$ is the minimum communication phase, which equals the end of the sending computational task τ_{SRC_j} with $\Phi_{\kappa_j,\min} = \Phi_{\tau_{\text{SRC}_j}} + W_{\tau_{\text{SRC}_j}}$. Packets of the communication task κ_j cannot be injected earlier than $\Phi_{\kappa_j,\min}$, because the required data is not produced. Note that the network traversal time $\overline{W}_{\kappa_j}$ is defined as

$$\overline{W}_{\kappa_j} = W_{\kappa_j} + |r_{\kappa_j}|(d_{\mathbf{R}} + d_{\mathbf{L}}), \tag{5.7}$$

which is the total time to fully transmit the packet to its desired destination including all network delays without interference from other communication tasks. Each communication task κ_j has a relative deadline D_{κ_j}, at which the data has been fully transmitted to the destination core, where the computational task τ_{DST_j} at the destination requires this data. The relative deadline of a communication task κ_j is defined relative to the arrival time $a_{\tau_{\mathrm{SRC}_j},k}$ of the computational tasks τ_{SRC_j}, which triggers the communication task κ_j.

Due to different periods in the task set, only a subset of jobs of a computational task send their data to avoid futile communications. Thus, this thesis defines specific jobs that send and receive the data. In particular, jobs that communicate data are indexed as the h-th jobs of the task τ_{SRC_j}. According to the definition of computational jobs that need to communicate (Section 3.3), the following three cases determine the absolute deadline of the corresponding packet:

- $a_{\tau_{\mathrm{SRC}_j},h} + \Phi_{\tau_{\mathrm{DST}_j}}$, if κ_j*-precedence*, or
- $a_{\tau_{\mathrm{SRC}_j},h} + \Phi_{\tau_{\mathrm{DST}_j}}$, if κ_j*-non-precedence* and $P_{\tau_{\mathrm{SRC}_j}} \leq P_{\tau_{\mathrm{DST}_j}}$ and $P_{\tau_{\mathrm{SRC}_j}} \leq \Phi_{\tau_{\mathrm{DST}_j}}$, or
- $a_{\tau_{\mathrm{SRC}_j},h} + \Phi_{\tau_{\mathrm{DST}_j}} + P_{\tau_{\mathrm{SRC}_j}}$, if κ_j*-non-precedence* and $P_{\tau_{\mathrm{SRC}_j}} > P_{\tau_{\mathrm{DST}_j}}$ or $P_{\tau_{\mathrm{SRC}_j}} > \Phi_{\tau_{\mathrm{DST}_j}}$.

The calculation of the relative communication deadline D_{κ_j} can be reformulated with respect to the arrival time $a_{\tau_{\mathrm{SRC}_j},h}$ of a job of its source task τ_{SRC_j} as follows:

$$D_{\kappa_j} = \begin{cases} \Phi_{\tau_{\mathrm{DST}_j}} & \text{if } \kappa_j \textit{ precedence} \\ \Phi_{\tau_{\mathrm{DST}_j}} + P_{\tau_{\mathrm{SRC}_j}} \left\lceil \frac{P_{\tau_{\mathrm{SRC}_j}} - \Phi_{\tau_{\mathrm{DST}_j}}}{P_{\tau_{\mathrm{DST}_j}}} \right\rceil & \text{otherwise.} \end{cases} \tag{5.8}$$

By applying Theorem 4 to all possible pairs of two communication tasks, the communication task set is feasible if no time overlap exists. This feasibility test can be easily implemented, as shown in Algorithm 4, which calculates all the above-mentioned equations required for a feasible communication task set.

5.3. Approaches for the Computational and Communication Phase Assignment

In the first part, this section proposes a heuristic approach to determine the computational and communication phases of a TTCP scheduled system. The approach uses the feasibility test presented in Section 5.2 to determine a heuristic algorithm. In the second part, a solver based approach [27] is presented.

Overview of the iterative approach to determine phases of a TTCP scheduler

The computational and communication phase assignments are performed in two consecutive steps. As mentioned earlier, there exists a cyclic dependency between these two steps, which needs to be taken into account. In the computational phase assignment, the heuristic reserves time windows for the communication tasks to construct a feasible communication phase assignment. These reserved time windows equal the worst-case traversal response time (WCTRT), which are determined in the communication phase assignment. In the communication phase assignment, the communication arrival times have to be known, which also depends on the computational phase assignment.

Algorithm 4 Feasibility test for the communication task set

```
Input: Communication task set $\mathbf{K}$
Output: Feasibility result;
1: Calculate the route conflict Matrix $M$;
2: feasibility ← true;
3: for $a = 0, \cdots, |\mathbf{K}| - 1$ stepped by 1 do
4:   if ($\Phi_{\kappa_a} < \Phi_{\kappa_a,\min}$) or ($\Phi_{\kappa_a} > D_{\kappa_a} - \overline{W}_{\kappa_a}$) then
5:     return "not feasible";
6:   for $b = a + 1, \cdots, |\mathbf{K}| - 1$ stepped by 1 do
7:     if ∃ conflict between Route $r_{\kappa_a}$ and $r_{\kappa_b}$ via $M$ then
8:       $t_a \leftarrow M_{a,b} \cdot (d_{\mathbf{L}} + d_{\mathbf{R}})$;
9:       $t_b \leftarrow M_{b,a} \cdot (d_{\mathbf{L}} + d_{\mathbf{R}})$;
10:      Calculate $\text{gcd}_{a,b}$ of $P_{\kappa_a}$ and $P_{\kappa_b}$;
11:      $\Psi_{\kappa_a} \leftarrow (\Phi_{\kappa_a} + t_a) \bmod \text{gcd}_{a,b}$;
12:      $\Psi_{\kappa_b} \leftarrow (\Phi_{\kappa_b} + t_b) \bmod \text{gcd}_{a,b}$;
13:      if ($\Psi_{\kappa_a} < \Psi_{\kappa_b}$) then
14:        if ($\Psi_{\kappa_b} < \Psi_{\kappa_a} + W_{\kappa_a}$) or ($\Psi_{\kappa_a} + gcd_{a,b} < \Psi_{\kappa_b} + W_{\kappa_b}$) then
15:          feasibility ← false;
16:      else
17:        if ($\Psi_{\kappa_a} < \Psi_{\kappa_b} + W_{\kappa_b}$) or ($\Psi_{\kappa_b} + gcd_{a,b} < \Psi_{\kappa_a} + W_{\kappa_a}$) then
18:          feasibility ← false;
19: return feasibility;
```

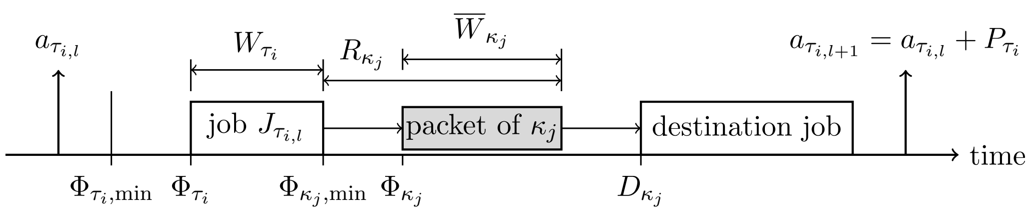


Figure 5.2.: Overview of the parameters for TTCP scheduled computational and communication tasks τ_i, κ_j.

In view of such circular dependency, the idea is to use an iterative approach that performs the computational and communication phase assignment repeatedly, as shown in Figure 5.3. The iterations are represented by the iteration counter q, where q is incremented in each iteration. The iterative process is repeated for a predetermined number of cycles I. In the first iteration, each worst-case traversal response time (WCTRT) is optimistically estimated, such that each communication task κ_j responds within its network traversal time $\overline{W}_{\kappa_j}$. Due to the constructed contention-free communication schedule, the WCTRT is only influenced by the delay for injecting packets into the communication network, i.e. the communication phase. Let R_{κ_j} be the worst-case traversal response time (WCTRT) of the communication taskκ_j, which is influenced by the injection delay, defined as

$$R_{\kappa_j} = \Phi_{\kappa_j} + \overline{W}_{\kappa_j} - \Phi_{\kappa_j,\min}. \tag{5.9}$$

Initially, the WCTRT R_{κ_j} is estimated as the network traversal time $\forall R_{\kappa_j} = \overline{W}_{\kappa_j}$. Figure 5.2 provides an overview of the parameters of the computational and communication analysis.

Referring to Figure 5.2, the block "Analyze **T**" represents the computational phase assignment, which is described later in this section. The block "Analyze **K**" computes the communication phase assignment Φ_{κ_j} based upon the computation phase assignment Φ_{τ_i}

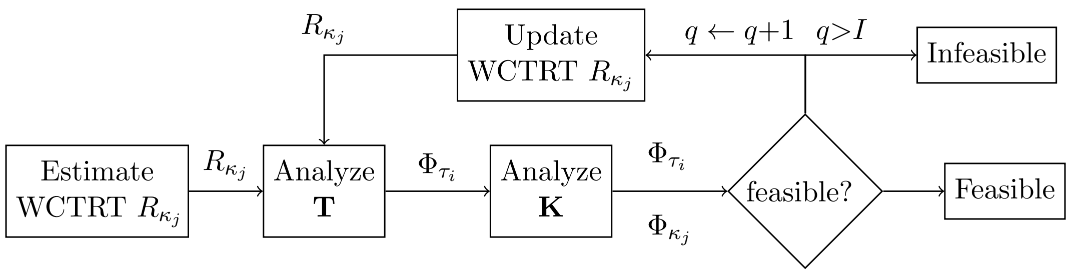


Figure 5.3.: The workflow for handling the cyclic dependencies among each **T** and **K**.

given by the block "Analyze **T**".

Based upon the calculated phases Φ_{τ_i} and Φ_{κ_j}, the feasibility test from Section 5.2 determines whether both the communication and computation task set **T** and **K** are feasible. In case of an infeasible result, another iteration is performed with an updated WCTRT, which leads to improved computation and communication phases Φ_{τ_i}, Φ_{κ_j}. The block *Update WCTRT* R_{κ_j} prepares the next iterations by calculating the WCTRT for each communication task κ_j based upon (5.9).

Further iterations tend to a fixed point because there is a specific order of computational and communication phase assignment. However, in case that no feasible solution after I iterations can be found, the algorithm terminates. Experiments show that usually a few iterations are sufficient to determine a feasible phase assignment and thus the parameter is set to $I = 10$.

Another approach is to formulate the phase assignment into an SMT problem and use a solver to find a feasible phase assignment. In the following, the two phase assignment parts "Analyze **T**" and "Analyze **K**" from Figure 5.3 are described in detail.

Computational phase assignment under given WCTRT R_{κ_j}

The computational phase assignment requires the WCTRT R_{κ_j} to reserve time for the communication in the TTCP schedule. The communication response times R_{κ_j} are determined according to (5.9).

The proposed heuristic algorithm greedily assigns computational phases without reassigning a phase. All tasks are sorted in a topological order $\Omega_\mathcal{T}$ such that precedence constrained tasks are ordered first. In this step, communication tasks of type κ_j*-non-precedence* are ignored.

According to the topological order $\Omega_\mathcal{T}$, the computational phases are assigned step by step. Suppose that task τ_i is the i-th task in $\Omega_\mathcal{T}$. The algorithm searches chronologically through the time to find a feasible phase Φ_{τ_i} for the computational task τ_i regarding the already-assigned phases. The feasibility is determined by the time-overlap test from Theorem 4. The heuristic proposed in the Algorithm 2 LPF in Section 4.5 is used to determine all computational phases.

A problem is to define the minimum computational phase $\Phi_{\tau_i,\min}$, because the precedence and non-precedence communication tasks may limit the feasible range for the computational phase τ_i. To derive $\Phi_{\tau_i,\min}$, communication tasks are divided into two sets depending on their associated type. Let E_{prec,τ_i} be the set of communication tasks κ_j, which comprises κ_j*-precedence* with $\tau_i = \tau_{\mathrm{DST}_j}$, i.e.

$$E_{prec,\tau_i} = \left\{\kappa_j \mid \kappa_j\text{-}\textit{precedence} \text{ and } \tau_{\mathrm{DST}_j} = \tau_i\right\}. \tag{5.10}$$

Algorithm 5 Communication task ordering algorithm

```
Input: T, K;
Output: Communication phase assignment order $\Omega_\kappa$;
1: $\Omega_\kappa \leftarrow \emptyset$;
2: for $a = 0, \cdots, |\mathbf{K}| - 1$ stepped by 1 do
3:     Calculate $D_{\kappa_j}$ according to Equation 5.8;
4:     $\Omega_{\kappa_a} \leftarrow \forall\ \kappa_j \notin \Omega_\kappa$ assign $\kappa_j$ lowest relative deadline $D_{\kappa_j}$;
5: return $\Omega_\kappa$;
```

Similarly, E_{np,τ_i} only considers κ_j*-non-precedence.* Accordingly, let E_{np,τ_i} be the set of communication tasks κ_j, which comprises κ_j*-non-precedence* with $\tau_i = \tau_{\mathrm{DST}_j}$, i.e.

$$E_{np,\tau_i} = \left\{\kappa_j \mid \kappa_j\textit{-non-precedence} \text{ and } \tau_{\mathrm{DST}_j} = \tau_i\right\}. \tag{5.11}$$

For κ_j*-precedence*, computational phases Φ_{τ_i} cannot be assigned before their predecessor tasks finish and their corresponding packets are received. Thus, τ_i cannot start earlier than:

$$b_{prec,i} = \max_{\kappa_j \in E_{prec,\tau_i}} (\Phi_{\tau_{\mathrm{SRC}_j}} + W_{\tau_{\mathrm{SRC}_j}} + R_{\kappa_j}). \tag{5.12}$$

If communication tasks are of type κ_j*-precedence*, computational phases cannot legally be assigned before receiving the packet of the previous job. Therefore, τ_i cannot start before

$$b_{np,i} = \max_{\kappa_j \in E_{np,\tau_i}} (\Phi_{\kappa_j,\min} - P_{\tau_{\mathrm{SRC}_j}} + R_{\kappa_j}). \tag{5.13}$$

By respecting both κ_j*-precedence* and κ_j*-non-precedence*, the minimum computational phase can be calculated by

$$\Phi_{\tau_i,\min} = \max(0, b_{np,i}, b_{prec,i}). \tag{5.14}$$

Communication phase assignment under given computational phases Φ_{τ_i}

The communication phases are determined by the proposed heuristic algorithm, which is described in the following. The proposed algorithm defines a communication phase assignment order Ω_κ. The communication tasks are ordered according to non-decreasing relative deadlines, i.e. deadline monotonic, which are computed according to (5.8). The order Ω_κ expresses the urgency of communication tasks. The proposed Algorithm 5 calculates the communication phase assignment order Ω_κ.

Suppose κ_j is the j-th task in the communication order Ω_κ. The minimum communication phase is determined based upon the completion time of the source computational task, whereby $\Phi_{\kappa_j,\min} \leftarrow \Phi_{\tau_{\mathrm{SRC}_j}} + W_{\tau_{\mathrm{SRC}_j}}$. Algorithm 6 implements the proposed approach to assign phases Φ_{κ_j} for communication tasks κ_j. The algorithm searches chronologically through time to find a feasible phase Φ_{κ_j} for κ_j with respect to already-assigned communication tasks.

First, Algorithm 6 calculates minimum communication phases $\Phi_{\kappa_j,\min}$ and conflict matrix according to (5.4). The outer *for-loop* iterates over all communication tasks in the topological order Ω_κ to determine communication phases step by step. The *while-loop* performs the chronologically search through the time, in which the parameter *feasible* indicates, whether the proposed communication phase is valid. The inner *for-loop* iterates over each other communication task and checks, whether there is a time overlap between another task. The function *ResolveCommConflict()* checks the time overlap and returns a

Algorithm 6 Heuristic algorithm for assigning the communication phases Φ_{κ_j} of the communication task set $\mathbf{K}$

```
Input: $\mathbf{T}$, $\mathbf{K}$, $\Omega_\kappa$, and platform;
Output: Phases $\Phi_{\kappa_j}$, packet delay $R_{\kappa_j}$ and feasibility;
1: $\forall \kappa_j\ \Phi_{\kappa_j,\min} \leftarrow \Phi_{\tau_{\mathrm{SRC}_j}} + W_{\tau_{\mathrm{SRC}_j}}$;
2: Calculate a route conflict matrix $M$ for given platform;
3: for $\ell = 0, \cdots, |\mathbf{K}| - 1$ stepped by 1 according to $\Omega_\kappa$ do
4:     $\Phi_{\kappa_\ell} \leftarrow \Phi_{\kappa_\ell,\min}$;
5:     feasible ← false;
6:     while ($\Phi_{\kappa_\ell} < P_{\kappa_\ell}$) and (feasible = false) do
7:         feasible ← true;
8:         for each $\kappa_j$ with $j < \ell$ do
9:             $\delta_\ell \leftarrow$ ResolveCommConflict($\kappa_\ell$,$\kappa_j$,$M$);
10:            if $\delta_\ell \neq 0$ then
11:                $\Phi_{\kappa_\ell} \leftarrow \Phi_{\kappa_\ell} + \delta_\ell$;
12:                feasible ← false;
13:    if (feasible=false) then
14:        return "not feasible";
15:    $R_{\kappa_\ell} \leftarrow \Phi_{\kappa_\ell} + \overline{W}_{\kappa_\ell} - \Phi_{\kappa_j,\min}$;
16: return "feasible";
```

time delay δ_ℓ for κ_j to resolve the time overlap with this communication task. If $\delta_\ell = 0$, there is no time overlap to the other task.

Algorithm 7 presents the function *ResolveCommConflict()*, which implements the time-overlap test of Theorem 4. In addition, it calculates for different cases the time delay to resolve the time overlap between another communication task.

The algorithm can decide which communication task becomes a short response time. The reason is the contention-free schedule, because it guarantees a specific response time for a fixed communication phase.

Run-time complexity of the proposed iterative algorithm

In the following, the complexity of the proposed iterative approach is analyzed, as shown in Figure 5.3. The overall run-time complexity is dominated by the computational and communication phase assignment.

First, Algorithm 6 is analyzed to determine communication phases. The time complexity of determining the conflict matrix M is $O(|\mathbf{K}|^2 \cdot |\mathbf{R}|)$, where $|\mathbf{R}|$ is the number of switches in the NoC and $|\mathbf{K}|$ is the number of communication tasks. Algorithm 7 for determining the time-overlap test has $O(1)$ run-time complexity. The run-time complexity for sorting communication tasks into the topological order Ω_κ according to Algorithm 5 is $O(|\mathbf{K}| \log |\mathbf{K}|)$. The outer *for-loop* has at most $|\mathbf{K}|$ iterations. The *while-loop* iterates $|\mathbf{K}|$ times for the inner *for-loop*, which results in $O(|\mathbf{K}|)$ to execute one *while-loop*. Let ν be the number of iterations of the *while-loop*, thus

$$\nu = |\mathbf{K}|(\left\lceil \frac{P_{\kappa_\ell}}{P_{\kappa_0}} \right\rceil + \left\lceil \frac{P_{\kappa_\ell}}{P_{\kappa_1}} \right\rceil + \ldots + \left\lceil \frac{P_{\kappa_\ell}}{P_{\kappa_{(\ell-1)}}} \right\rceil, \tag{5.15}$$

because each packet of κ_ℓ collides at most once with the $\left\lceil \frac{P_{\kappa_\ell}}{P_{\kappa_j}} \right\rceil$ packets of κ_j. Hence, $\nu \leq \left\lceil \frac{P_{\max}}{P_{\min}} \right\rceil |\mathbf{K}|^2$. As a result, the run-time complexity of Algorithm 6 is $O(\left\lceil \frac{P_{max}}{P_{min}} \right\rceil |\mathbf{K}|^3 |\mathbf{R}|)$.

Algorithm 7 ResolveCommConflict() algorithm to determine the time difference until the next possible phase

```
Input: $\kappa_a$, $\kappa_b$, $M$;
Output: time shift $t$;
 1: $t \leftarrow 0$;
 2: if $\exists$ conflict between route $r_{\kappa_a}$ and $r_{\kappa_b}$ via $M$ then
 3:     Calculate $\text{gcd}_{a,b}$ of $P_{\kappa_a}$ and $P_{\kappa_b}$;
 4:     $\Psi_{\tau_a} \leftarrow (\Phi_{\kappa_a} + M_{a,b} \cdot (d_{\mathbf{L}} + d_{\mathbf{R}})) \bmod \text{gcd}_{a,b}$;
 5:     $\Psi_{\tau_b} \leftarrow (\Phi_{\kappa_b} + M_{b,a} \cdot (d_{\mathbf{L}} + d_{\mathbf{R}})) \bmod \text{gcd}_{a,b}$;
 6:     if $(\Psi_{\tau_a} < \Psi_{\tau_b})$ then
 7:         if $\Psi_{\tau_b} < \Psi_{\tau_a} + \overline{W}_{\kappa_a}$ then
 8:             $t \leftarrow \Psi_{\tau_a} + \overline{W}_{\kappa_a} - \Psi_{\tau_b}$
 9:         else if $\Psi_{\tau_a} + \text{gcd}_{a,b} < \Psi_{\tau_b} + \overline{W}_{\kappa_b}$ then
10:             $t \leftarrow \Psi_{\tau_a} + \text{gcd}_{a,b} + \overline{W}_{\kappa_a} - \Psi_{\tau_b}$;
11:     else
12:         if $\Psi_{\tau_a} < \Psi_{\tau_b} + \overline{W}_{\kappa_b}$ then
13:             $t \leftarrow \Psi_{\tau_a} + \overline{W}_{\kappa_a} - \Psi_{\tau_b}$;
14:         else if $\Psi_b + \text{gcd}_{a,b} < \Psi_a + \overline{W}_{\kappa_a}$ then
15:             $t \leftarrow \Psi_{\tau_a} + \overline{W}_{\kappa_a} - \Psi_{\tau_b} - \text{gcd}_{a,b}$;
16: return $t$;
```

Similar to the communication phase assignment, the computational phase assignment has run-time complexity of $O(\left\lceil \frac{P_{max}}{P_{min}} \right\rceil |\mathbf{T}|^3)$. As a conclusion, the overall run-time complexity of the iterative approach as depicted in Figure 5.3 is

$$O\Big(I \left\lceil \frac{P_{max}}{P_{min}} \right\rceil \Big(|\mathbf{T}|^3 + |\mathbf{K}|^3 |\mathbf{R}|\Big)\Big), \tag{5.16}$$

where I is the maximum number of iterations.

Solver-based approach by using SMT

In contrast to the heuristic algorithm for defining the computational and communication phases, another approach is to use a Satisfiability Modulo Theories (SMT) solver [27]. As mentioned in Section 4.5, the phase assignment problem can be formulated into a set of equations that can be solved by SMT. Both computational and communication phases Φ_{τ_i}, Φ_{κ_j} need to be determined by the SMT solver. The detailed Algorithm 14 is found in the Appendix A.2, which describes the implemented SMT problem formulation.

Due to typical industrial applications, a large number of communication tasks (300–3,000) is assumed to be present in the system. This is a challenge for solver-based approaches, in which the time requirement of a solver (e.g. Z3 SMT solver) increases exponentially with the number of tasks. With $|\mathbf{T}| = 100$ computational and $|\mathbf{K}| = 300$ communication tasks, the SMT solver was able to provide a feasible solution, although larger problem sizes exceed its limit. For instance, with $|\mathbf{T}| = 1{,}000$ and $|\mathbf{K}| = 3{,}000$, the common SMT problem definition (e.g. Z3 SMT solver) requires ˜$8 \cdot 10^9$ *ASSERT* statements to describe the SMT problem. Such a large number of *ASSERT* statements exceeds the solver capacity and no feasible solution could be returned. Thus, solver-based approaches are not applicable to typical industrial-sized problems. By contrast, the proposed heuristic is able to successfully solve scheduling problems associated with industrial-sized applications. Nevertheless, the SMT solver is used in the evaluations as a comparison to the heuristic approach.

5.4. Evaluations

This section presents an evaluation of different approaches to determine computational and communication phases of a TTCP scheduled system. The experiments highlight the advantages and limitations of the iterative approach in comparison to a solver-based approach [10], which is proposed in the previous Section 5.3.

5.4.1. Experimental Setup

This subsection describes the configuration used for the experiments. A generator uses the dependent system model to create synthetic task sets, which are described in the following.

Dependent system model generator

In order to test the proposed iterative approach for different variations of the generated dependent system model, the generator produces randomized synthetic system sets with characteristics of typical industrial applications. The generator performs several steps to determine all parameters of the system model, as described in the following:

1. Platform: Build a $2 \times 2 \ldots 8 \times 8$ manycore platform with a NoC.
2. Computational task set: Generate 100–1,000 computational tasks with no heavy task and only harmonic periods.
3. Commutation task set: Generate $|\mathbf{K}| = 3|\mathbf{T}|$ communication tasks, in which 20% are κ_j*-precedence.*
4. Mapping: Calculate the task-to-core mapping for each computational task τ_i based upon the communications.
5. Routing: Generate the route for each κ_j according to the X-Y routing policy.
6. Clean up: Remove the communication tasks κ, which are no inter-core communications.

In the following, these steps are explained in detail.

Step 1 Platform: The generator builds quadratic 2D-mesh NoC structures, for which the 3×3 NoC with $|\mathbf{R}| = |\mathbf{C}| = 9$ cores is the default platform. Each core is connected to its corresponding switch. Each connection between two nodes in the NoC used two links, one for transmitting packets in each direction. In the generated NoC, there are a certain number of links $|\mathbf{L}|$, which depend on the NoC size with $|\mathbf{L}| = 6|\mathbf{C}| - 4\sqrt{|\mathbf{C}|}$. Figure 3.2 shows the default manycore platform. Each core has sufficient local memory to store its program code and temporal required data.

Step 2 Computational task set: First, the computational utilization $U_{\mathbf{T}}$ and the number of computational tasks $|\mathbf{T}|$ are defined. Subsequently, the generator defines the first period P_{τ_1}. Due to the assumption that a hyper-period exists, all periods are integer values, i.e. periods are integer multiples of the time unit of $1\mu s$. Based upon the first period, all periods can be calculated as:

$$P_{\tau_{i+1}} = P_{\tau_i} \cdot k \qquad k \in \{1, 2, 3\}, \tag{5.17}$$

where k is a randomized number, which varies in each period P_{τ_i}. In order to generate a random variable k, the probability is calculated with

$$k = \min(\left\lceil \frac{2.5}{|\mathbf{T}| \cdot k_r} \right\rceil, 3) \qquad k_r \in (0 \ldots 1) \subset \mathbb{R}^{|\mathbf{T}|-1}, \tag{5.18}$$

which generates approximately five different periods in $\mathbf{T}$. Third, a uniform distributed random number ρ_i is generated with $0 < \rho_i < 1$ for each task τ_i. Accordingly, the execution time can be defined with

$$W_{\tau_i} = \frac{P_{\tau_i} \cdot U_{\mathbf{T}} \cdot \rho_i}{\sum_{\tau_\ell \in \mathbf{T}} \rho_\ell}. \tag{5.19}$$

The relative deadlines are implicit, whereby

$$D_{\tau_i} = W_{\tau_i}, \qquad \forall \tau_i. \tag{5.20}$$

Note that the precedence constraints are calculated within the communication task set generation in step 3 and the task-to-core mapping is calculated in step 4.

Step 3 Commutation task set: First, the input parameters are defined, which are (i) the communication utilization $U_{\mathbf{K}}$, (ii) the number of communication tasks $|\mathbf{K}|$ and (iii) the precedence rate p. The precedence rate is in the range of $[0\% \dots 100\%]$, which indicates the probability that a communication task has *κ_j-precedence*.

Next, the generator determines the source τ_{SRC_j} and destination tasks τ_{DST_j} for each communication task. Therefore, the generator defines an order $\Omega_{\mathrm{precedence}}$ of all computational tasks, which is used to build a consistent DAG. Based upon this computational task order $\Omega_{\mathrm{precedence}}$, only tasks with a lower order can be a predecessor of computational task τ_{DST_j}. The generator uses two randomly-mixed strategies to define the source and destination tasks. On the one hand, a list of computational tasks τ_i which are sorted according to $\Omega_{\mathrm{precedence}}$ defines consecutive communication tasks to define the source τ_{SRC_j} and destination τ_{DST_j}. One the other hand, single randomized pairs of computational tasks are used to define the source and destination computational tasks τ_{SRC_j}, τ_{DST_j}. Each communication task has a precedence constraint, which is quantified by the precedence rate p, if this communication task is valid ordered according to $\Omega_{\mathrm{precedence}}$.

Periods are generated based upon the source τ_{SRC_j} and destination τ_{DST_j} task with

$$P_{\kappa_i} = \max\left(P_{\tau_{\mathrm{SRC}_j}}, P_{\tau_{\mathrm{DST}_j}}\right), \tag{5.21}$$

and the relative deadlines according to (5.8). The traversal times W_{κ_j} are computed similar to the execution times W_{τ_i} in (5.19) by given $U_{\mathbf{K}}$ and P_{κ_j}.

Step 4 Mapping: Although the mapping of computational tasks to cores is not the focus of this thesis, our generator uses a heuristic for task-to-core mapping. The overall mapping procedure is depicted in Figure 5.4. In the dependent task model, the DAG has to be considered in the mapping (Figure 5.4a). The proposed heuristic exploits parallel

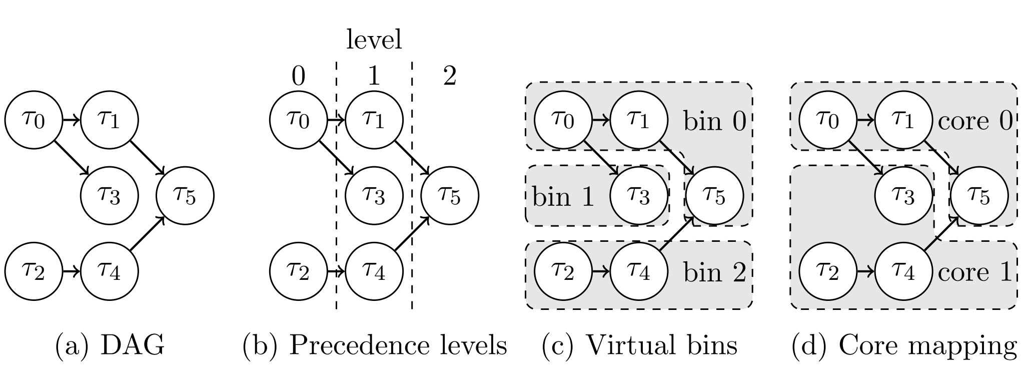


Figure 5.4.: An example of the workflow for the proposed task-to-core mapping heuristic of the dependent system model generator.

executions by defining precedence levels based upon the DAG (Figure 5.4b). Each precedence level ℓ contains a set of computational tasks τ_i, which have a precedence constraint to the previous level. Based upon these levels, the maximum communication utilization determines, whether the succeeding task is assigned to the same virtual bin. Thus, the objective is to minimize the inter-core communication while keeping parallel tasks executions. In one precedence level, each task has a different virtual bin to maximize parallel task executions (Figure 5.4c). The generator applies the worst-fit bin packing approach to assign the virtual bin to the cores (Figure 5.4d), wherein the task utilization determines the bin quantity.

Step 5 Routing: With a given task-to-core mapping, the generator determines the routes r_{κ_j} for each communication task κ_j according to the X-Y routing policy. Note that a route contains all nodes on the path from the source core $c_{\tau_{\mathrm{SRC}_j}}$ to its destination core $c_{\tau_{\mathrm{DST}_j}}$ except the source core $c_{\tau_{\mathrm{SRC}_j}}$.

Step 6 Clean up: Based upon the task-to-core mapping, some communication tasks κ_j possibly communicate within the same core, whereby $c_{\tau_{\mathrm{SRC}_j}} = c_{\tau_{\mathrm{DST}_j}}$. These communication tasks κ_j can be removed from the communication task set $\mathbf{K}$, because only inter-core communication is considered. Such communications can be implemented with writing and reading the data to the local memory, which is a well-known method from the single-core architecture.

Default parameter settings

In the experiment, the generator creates 100 randomized system sets to evaluate the different approaches and algorithms. The NoC comprises a 3×3 2D-mesh in most experiments. The computational $U_{\mathbf{T}}$ and communication utilization $U_{\mathbf{K}}$ are varied separately in the range $[0 \dots |\mathbf{C}|)$. Note that $U_{\mathbf{T}} = |\mathbf{C}|$ means that each core is 100% utilized and $U_{\mathbf{K}} = |\mathbf{C}|$ means that each core alway sends data. The smallest platform has $|\mathbf{C}| = 4$ cores and the largest platform has up to $|\mathbf{C}| = 64$ cores. The precedence rate p is 20%, i.e. 20% of all communication tasks are of type *κ_j-precedence* and 80% are *κ_j-non-precedence.*

The Z3 solver (version 4.3.2.0) [27] is used to solve the SMT problem, which is generated for each system set. After a 10 min timeout, the solver is terminated and the result is counted as *unknown* marked as gray areas in the plots. For a large number of computational and communication tasks ($|\mathbf{T}| > 150$), the solver often failed due to its memory requirement. Such results are also counted as *unknown.*

5.4.2. Experimental Results

This subsection presents experiment exploring the capabilities of the proposed iterative approach to determine the TTCP schedule.

Computational and communication utilization

This experiment determines the reachable utilization of the TTCP approach. The experiment examines different computational $U_{\mathbf{T}}$ and communication $U_{\mathbf{K}}$ utilization of the system set to determine a limit on the feasibility rate. The parameters $U_{\mathbf{T}}$ and $U_{\mathbf{K}}$ increase the WCET and traversal times of the system set such that the scheduling problem is more difficult. The computational and communication utilization $U_{\mathbf{T}}$, $U_{\mathbf{K}}$ are defined as

$$U_{\mathbf{T}} = \sum_{\forall \tau_i} (\frac{W_{\tau_i}}{P_{\tau_i}}) \qquad \text{and} \qquad U_{\mathbf{K}} = \sum_{\forall \kappa_i} (\frac{W_{\kappa_i}}{P_{\kappa_i}}). \tag{5.22}$$

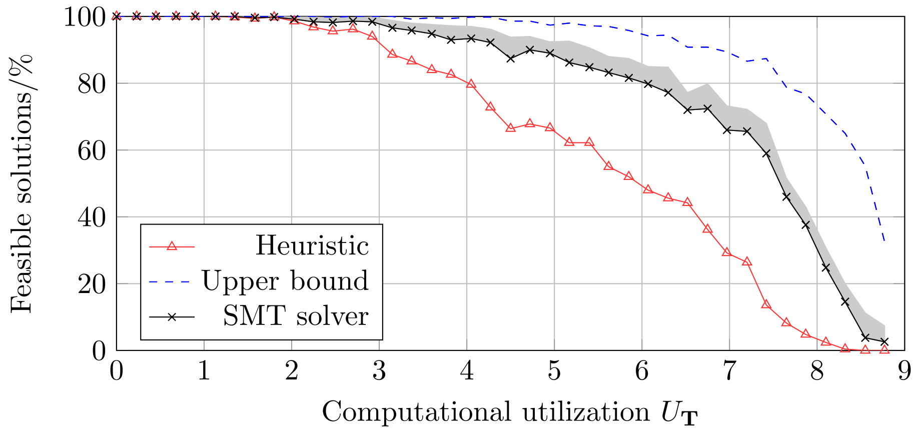


Figure 5.5.: Computational and communication utilization: Different computational utilization on a 3×3 NoC

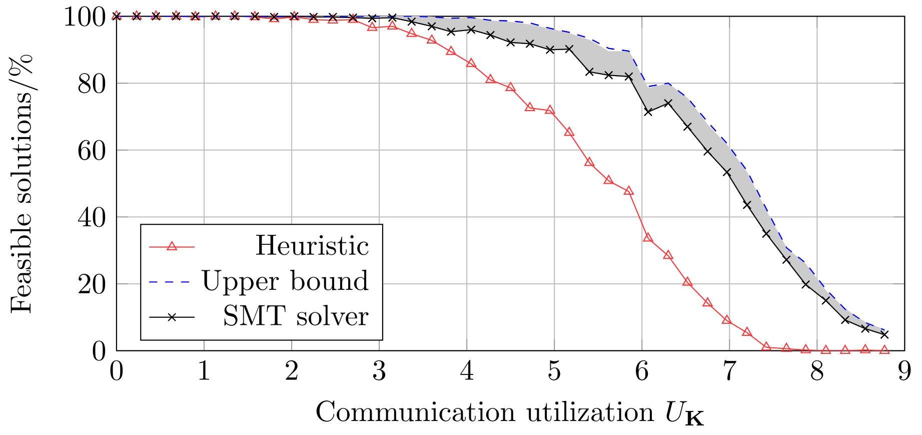


Figure 5.6.: Computational and communication utilization: Different communication utilization on a 3×3 NoC

Note that $U_{\mathbf{T}} = 1$ represents the maximum single-core platform utilization and $U_{\mathbf{K}} = 1$ represents the maximum bus utilization. Figures 5.5 and 5.6 show the plots of different methods.

The approaches namely *Heuristic*, *Upper bound* and *SMT solver* determine the phases of the generated system. Based upon the determined phases for 100 generated system sets, the test counts the number of feasible solutions, which results in a feasibility rate called *Feasible solutions*. The number of computational and communication tasks are set to $|\tau| = 100$ and $|\kappa| = 300$.

The curve *Heuristic* represents the proposed iterative approach described in Section 5.3. For comparison, the *SMT solver* formulates the SMT problem based upon the generated system set and uses the Z3 solver to determine the phases. If the solver does not provide any result called *unknown*, e.g. time-out or out-of-memory error, the plots are marked for such results as gray areas. The curve *Upper bound* represents an upper bound on the feasibility rate, which is determined by each individual core and link utilization. If any link or core in the system is utilized with more than 100%, there cannot exist a feasible schedule. If the curve *Upper bound* returns an infeasible result, there cannot exist a feasible result because the system generator creates an infeasible system.

The figures show that both the solver and the proposed iterative algorithm are capable of efficiently utilizing the manycore platform. The SMT solver has a higher success rate

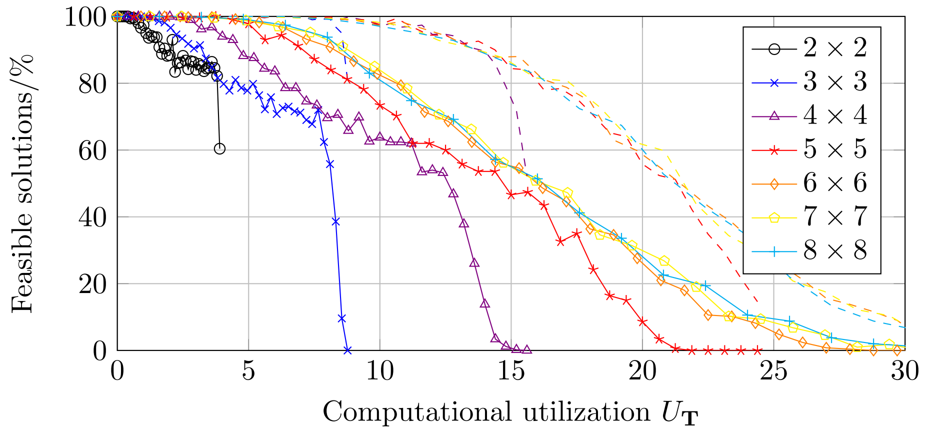


Figure 5.7.: NoC scalability: Computational utilization for different-sized mesh NoCs

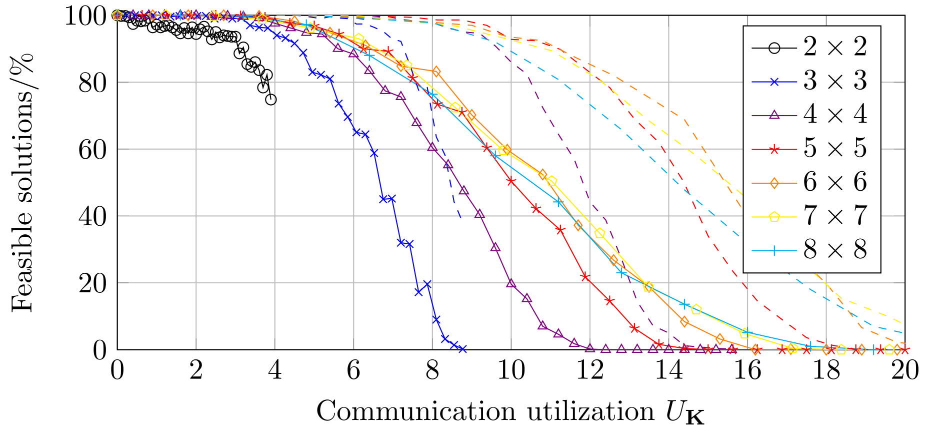


Figure 5.8.: NoC scalability: Communication utilization for different-sized mesh NoCs

(feasibility rate) in terms of the proportion of feasible solutions computed, although it also requires more time for its calculation. This experiment only uses sets with $|\tau| = 100$ computational tasks to compare the proposed iterative approach with an SMT solver.

However, the solver fails to find feasible solutions for an application with more computational and communication tasks. The curve *Upper bound* shows that the generator only rarely creates feasible system sets with a high utilization $U_\mathbf{T} > 8.5$ and $U_\mathbf{K} > 7$. Another interesting aspect is that for more computational tasks like $|\tau| = 1,000$ the feasibility rate of the proposed iterative approach is higher.

NoC scalability

This experiment explores different-sized NoCs to determine the scalability of the approach. The NoC sizes are always quadratic 2D-meshed and in range of 4–64 cores. Figures 5.7 and 5.8 show the results for different computational $U_\mathbf{T}$ and communication utilizations $U_\mathbf{K}$. The generator creates $|\tau| = 250$ computational and $|\tau| = 750$ communication tasks for each system set. The curves only visualize the proposed iterative algorithm and the upper bound, because the SMT does not provide feasible results.

Smaller NoC platforms can be highly utilized. With $|\mathbf{C}| \geq 36$ cores, the curves have only minor improvements because the platform cannot be further utilized due its sequential software parts. The curve *upper bound* drops down $U_\mathbf{T} > 25$ under less than 30%, i.e. the system generator is unable to generate feasible system sets.

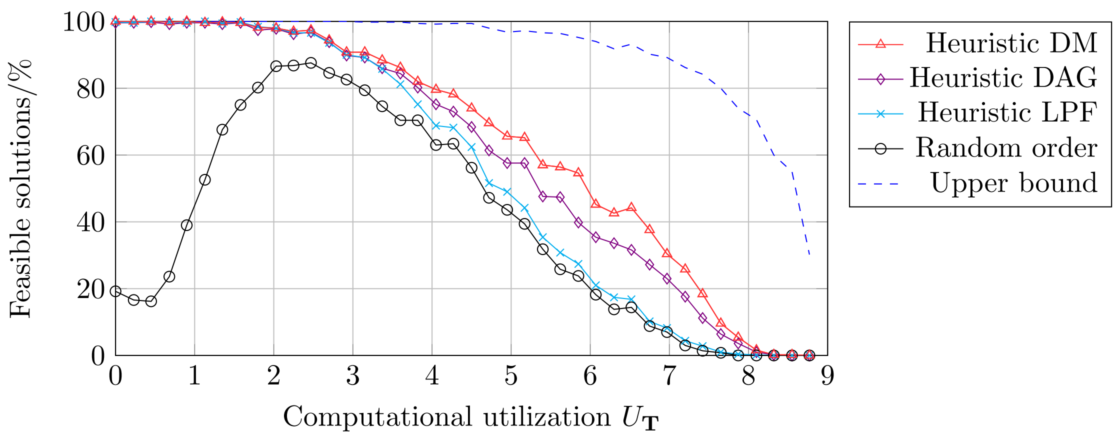


Figure 5.9.: Communication task ordering: Different methods to define the order Ω_κ are compared to each other.

Communication task ordering

This experiment varies the communication order Ω_κ to determine the performance of competitive ordering approaches. The proposed iterative approach in the previous Section 5.3 uses a Deadline Monotonic (DM) ordering of the communication tasks κ_j. Figure 5.9 shows the performance of other approaches namely *DAG*, *LPF* and *random order*.

The ordering according to the *DAG* of the computational tasks τ_i sorts the communication task κ_j on critical path in the DAG first. All paths in the DAG are scored based upon the execution and traversal time to determine the urgency of each path. Based upon these scores, the communication tasks with a high scored path are ordered before those with a lower scored path. Another possibility is to determine the order based upon the Lower Periods First (LPF) approach, in which communication tasks with the same period are ordered with lower minimum communication phases $\Phi_{\kappa_i,\min}$ first. The *random order* sorts the communications tasks κ_j in a randomized order, in which each task could be ordered first. Note that the curve *Upper bound* represents an upper bound to the feasibility rate, which is based upon a utilization test for each core and link in the system.

The experiment highlights that the proposed DM communication task ordering outperforms other approaches. The DAG ordering has exponential run-time complexity.

Precedence rate influence

This experiment evaluates the parameter precedence rate p, reflecting is the probability that a communication task becomes a κ_j*-precedence*. A precedence constraint defines a sequence of computational tasks that cannot be parallelized. Therefore, precedence constraints can limit parallel executions of the system set. Figure 5.10 shows the feasibility rates for different precedence rates from $p = 0\%$ to $p = 80\%$. The generator creates $|\tau| = 250$ computational and $|\tau| = 750$ communication tasks for each system set.

The experiments highlights that the possible computational utilization strongly depends upon the precedence constraints. With $p = 0$, the system set can be utilized up to $U_{\mathbf{T}} = 8.5$, even if NoC with $|\tau| = 750$ communication tasks are used. For high precedence rates, the possible computational utilization is significantly lower, although a parallel platform can still be utilized around $U_{\mathbf{T}} = 3.5$. This means that dependent tasks can be highly parallelized, although only certain dependencies called precedence constraints limit parallelization.

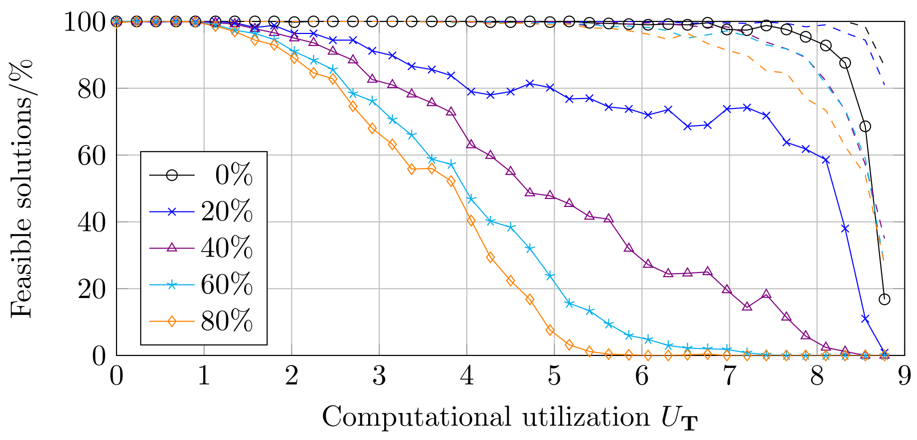


Figure 5.10.: Precedence rate influence: Different precedence rates significantly determine the reachable computational utilization $U_{\mathbf{T}}$.

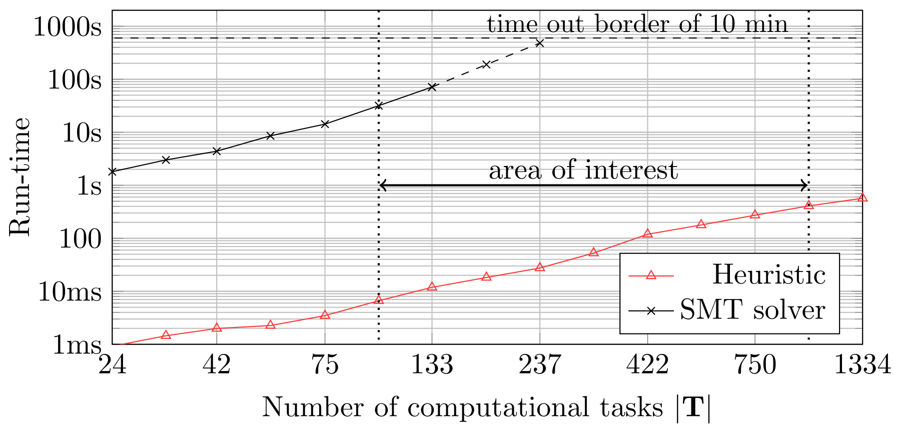


Figure 5.11.: Run-time and solver feasibility: Double logarithmic plot of the run-time for different phase assignment methods.

Run-time and solver feasibility

This experiment shows the run-time for different-sized system sets, which significantly influences the SMT solver approach. In the *computational and communication utilization* experiment (Figure 5.5), the SMT solver has a higher feasibility rate than the iterative algorithm. The exponential run-time complexity of the SMT solver limits its scalability to a large application comprising thousands of computational and communication tasks.

Figure 5.11 presents a run-time measurement for the proposed iterative algorithm (*Heuristic*) and the Z3 solver (SMT solver). The experiment uses different-sized system sets with $|\mathbf{K}| = 3|\mathbf{T}|$ and a 3×3 NoC structure. For this platform, the computational utilization is set to $U_{\mathbf{T}} = 4.5$ and the communication utilization is set to $U_{\mathbf{K}} = 1$. The SMT solver sets a timeout after 10 min. The iterative algorithm is around three magnitudes (10^3) faster, as shown in Figure 5.11.

For smaller system sets $|\mathbf{T}| = 100$, the SMT solver is able to compute a feasible phase assignment, although for larger system sets $|\mathbf{T}| > 200$, the solver usually returns errors or timeout, as shown in Figure 5.12. If the SMT problem size is too large, it takes too much time to obtain a solution. Another issue is the memory requirement. For larger system sets, the SMT solver often returns an *out-of-memory* error because the solver or the SMT problem formulation exceeds the physical memory of the simulation computer.

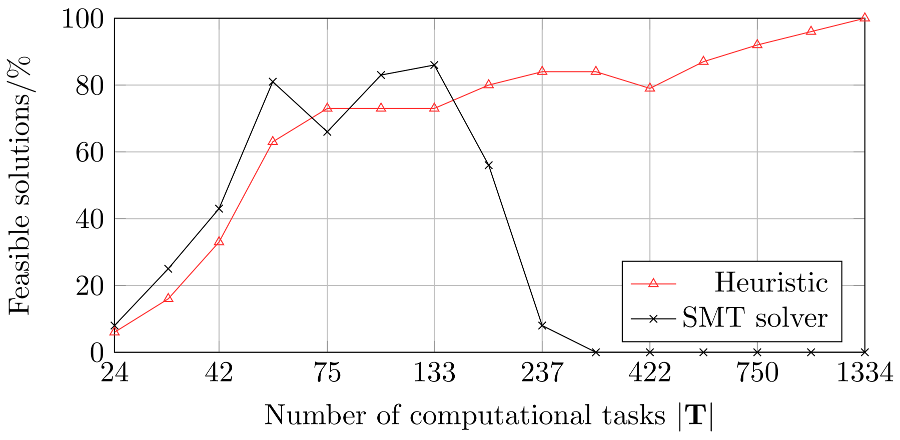


Figure 5.12.: Run-time and solver feasibility: Semi-logarithmic plot of the feasibility of different phase assignment methods. With more than $|\mathbf{T}| > 133$ tasks, the solver reaches its limits depending on the hyper-period, such that no solution can be found.

For instance, for $|\mathbf{T}| = 1,000$ computational tasks, the SMT problem formulation results in approximately $8 \cdot 10^9$ *ASSERT* statements. Considering system sets with typical industrial-sizes, the SMT approach is not applicable for large system sets, while the proposed iterative algorithm scales with the number of computational and communication tasks.

6. Integration of Sporadic Tasks into Pure Time-Triggered Systems

This chapter describes concepts to handle sporadic tasks in a Time-Triggered Constant Phase (TTCP) scheduled system. The TTCP scheduler is extended to additionally support sporadic tasks of typical industrial applications. The general ideas of this chapter have already been published [35] by the author.

6.1. Introduction

This section introduces approaches to handle sporadic tasks in a TTCP scheduled system. The previous chapter has highlighted that the TTCP approach is capable of effectively utilizing a manycore platform with a NoC. However, sporadic tasks are not considered in the TTCP approach. The approach is incapable of handling sporadic tasks, because schedule is determined *a priori*. The sporadic events are represented by sporadic tasks, for which the arrival time is not *a priori* known.

In general, a periodic task represents a special case of a sporadic task. In contrast to a periodic task, a sporadic task can have a large response time on a manycore platform, because the communication interference cannot be avoided by design. Following the example as described in the thesis introduction, in an engine control application exists a so-called *angle-synchronous task* [49], which is activated at a specific rotation angle of a crankshaft of an engine. After an additional rotation of 30 degrees, the angle-synchronous task needs to output its calculated results to control the engine in a safe manner. Therefore, this sporadic task has a stringent timing constraint. This chapter assumes that each sporadic tasks has a constraint relative deadline, i.e. the relative deadline is smaller than the period of the task. Note that the period of a sporadic task is defined as the minimum time difference between two consecutive arrival times of a task (Section 3.2). The problem is to schedule the time-triggered and the sporadic tasks in the same system such that a manycore platform can be highly utilized.

The time-triggered tasks are the computational τ_i and communication tasks κ_j, which are activated in a time-triggered manner. If a sporadic task arrives in a TTCP scheduled system, a time-triggered task could be delayed. This delay can cause a missed deadline, because the TTCP schedule is precisely designed to fulfill all real-time constraints without additional interference. Without applying correct mechanisms to handle the sporadic tasks, the sporadic tasks would disrupt the predefined TTCP schedule. In addition, a sporadic task communicates with other tasks, which may interfere with time-triggered communication tasks.

The proposed approach is to define a Time-Triggered Server (TTS) that can handle the sporadic tasks. The TTS is a resource reservation method to handle and execute sporadic tasks. It is represented by periodic time-triggered time slots (similar to a computational task), in which no computational or communication task is allowed to execute or communicate. These reserved time windows (slots) are used to execute sporadic tasks including their related communications. Considering the constrained relative deadline of the angle-synchronous task, the response time could be too large to satisfy its real-time constraint.

For example, suppose a sporadic task with a WCET of $2ms$, a relative deadline of $5ms$ and a period (minimum inter-arrival time) of $10ms$. If the sporadic task responds within 5–$10ms$ due to interference of time-triggered tasks, the deadline is violated. Hence, the proposed *slot-shifting* method improves the responsiveness of the sporadic tasks by exploiting the TTS. In slot-shifting, the time-triggered tasks are delayed to temporally accomplish more computing time to execute the sporadic task. After executing the sporadic tasks, the delayed time-triggered tasks are executed to recover the delay.

The slot-shifting approach was initially introduced by Fohler [30, 43] for the purpose of utilizing idle times on single-core platforms. By contrast, the proposed approach explicitly reserves time through the TTS for the sporadic tasks and is able to shift both TTCP scheduled computational and communication tasks. The delay of the time-triggered tasks is recovered by skipping the time slots of the TTS. Moreover, this thesis presents an algorithm to optimize the phase assignment of the time-triggered tasks to maximize the capacity of the TTS, because there is a trade-off between the capacity and the period of the TTS. Experiments confirm that the proposed slot-shifting approach can enhance the responsiveness of a sporadic task and thus reduce the worst-case response time by 25% of a typical industrial application.

Related work

The problem that time-triggered and sporadic tasks may coexist in a real-time system has also been discussed in the literature [3, 21, 76]. Claesson and Suri present an approach to insert a virtual event-triggered channel in a time-triggered bus. Albert [3] highlights the advantages and disadvantages of using a time-triggered or sporadic communication arbitration. The Flex Ray protocol [76] separates time-triggered parts from sporadic parts, in which both parts have a guaranteed utilization and latency. By considering a NoC, these approaches cannot be applied because they only assume a single resource, namely a bus architecture.

In contrast to the TTS, another possibility is to use idle times from the TTCP scheduler to handle the sporadic tasks [30,43,93]. In general, time-triggered tasks may communicate in idle times such that the sporadic task could disrupt the time-triggered communication schedule. Thus, idle times are not applicable for handling sporadic tasks with constraint relative deadlines. Another approach is to separate time-triggered from sporadic tasks in time or space [76], although a sporadic task may need to be scheduled immediately to improve the run-time performance.

Closely related to this chapter, the slot-shifting approach has already been presented [30, 43, 93]. Fohler [30] introduces the slot-shifting approach to delay the time-triggered scheduled jobs for some time to handle periodic and aperiodic events. This approach has been extended by Isovic et al. [43] and van den Heuvel et al. [93] to additionally support *firm* task deadlines and limited preemptive tasks, respectively. Note that a firm deadline must be met once if the task is accepted by an on-line scheduler. The presented slot-shifting approach is capable of handing sporadic tasks in a time-triggered system, which is extended in this chapter. By contrast, their approaches neither support a NoC nor shift the communications between the tasks to additionally allow the sporadic tasks to safely communicate.

Motivational example

The following example demonstrates a TTS and the proposed slot-shifting approach. Suppose five time-triggered computational tasks $\{\tau_0, \dots, \tau_4\}$ and one sporadic task σ_0.

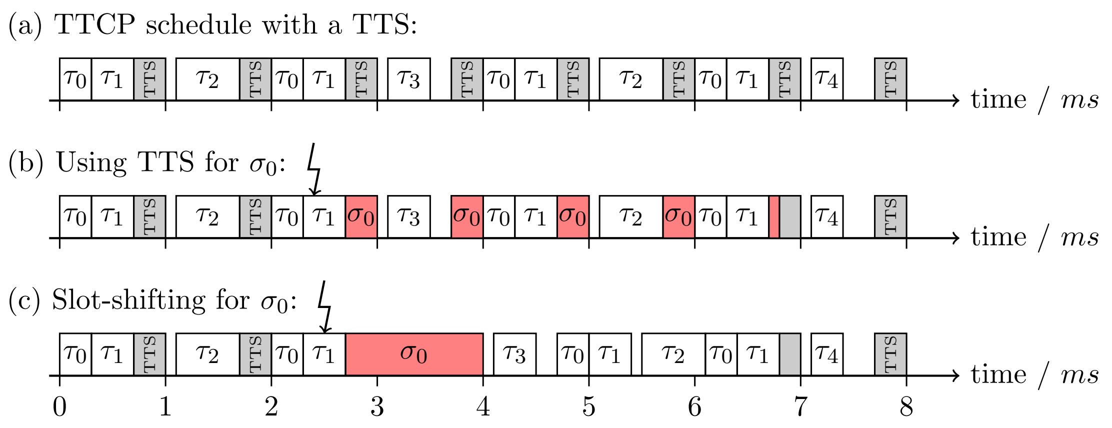

(a) TTCP schedule with a TTS:
(b) Using TTS for σ_0:
(c) Slot-shifting for σ_0:
τ_0
τ_1
τ_2
τ_3
τ_4
TTS
σ_0
time / ms
0
1
2
3
4
5
6
7
8

Figure 6.1.: Motivational example: Five tasks are scheduled by a normal TTCP scheduler (a). The gray boxes represent spare time, which is reserved for the Time-Triggered Server (TTS) and is not used by the time-triggered tasks scheduler. The arrow indicates the arrival of a sporadic task σ_0. On the one side (b), σ_0 can be scheduled by the TTS. One the other side (c), the time-triggered tasks can be delayed to temporally enable more computational resources for σ_0. After σ_0 is executed, the scheduler uses the spare times (gray boxes) to recover delayed time-triggered tasks.

These tasks are scheduled by a TTCP scheduler, as shown in Figure 6.1a. The TTS can be represented by the time slots of an additional time-triggered task. The TTS is represented by the gray boxes, in which the time-triggered tasks are not allowed to be executed.

One approach is to use the time slots of the TTS to execute the sporadic task σ_0, as shown in Figure 6.1b. The sporadic tasks are only allowed to be executed in the time slots of the TTS. The TTS slot is arranged into the TTCP schedule in a way that no time-triggered task is disturbed, although the response time for σ_0 is large. The TTS fits into the TTCP schedule such that no time-triggered task is disturbed, although the response time for σ_0 is large. Figure 6.1c depicts another approach, which delays time-triggered tasks to temporally allow more execution time for σ_0, called *slot-shifting*. After the completion of σ_0, the time slots of the TTS are used to *recover* the delayed time-triggered tasks, where recover means the reduction of the delay of time-triggered tasks. The disadvantage of the slot-shifting is the disruption of the time-triggered tasks, because time slots are not executed in *a priori* determined slots. The advantage is a shorter response time of the sporadic task σ_0, which increases the schedulability of sporadic task with constrained relative deadlines.

Problem definition

This chapter solves two problems, namely *slot-shifting* and *TTS phase assignment* problem. The *slot-shifting* problem is to determine an on-line scheduling algorithm that is able to shift time-triggered tasks and handle sporadic tasks, such that all real-time constraints hold. The *TTS phase assignment* problem is to define TTS by periodic time slots and determine the phases of the time-triggered tasks such that:

- the period of the TTS is minimized to shorten the response times of the sporadic tasks;
- the TTS utilization is maximized to increase the service for the sporadic tasks; and
- the entire system satisfies its real-time constraints.

Note that the time-triggered task utilization is fixed. The maximization of the TTS utilization uses the remaining time to obtain a high platform utilization.

6.2. Time-Triggered Server (TTS) and Slot-Shifting Approach

The solution of the *slot-shifting* problem is divided in two parts. The first part, presented in this section, defines the proposed slot-shifting approach and our on-line scheduler implementation. The second part deals with the related timing analysis, and provides proofs of the correctness of the proposed approach, which is presented in the subsequent section.

Concept of the TTS and the slot-shifting approach

The proposed slot-shifting approach can delay the time-triggered tasks to improve the responsiveness of sporadic tasks σ_k. The slot-shifting approach was originally introduced by Fohler [30] with the purpose of exploiting the idle times while guaranteeing the real-time constraints of the sporadic tasks. In contrast to his approach, a TTS reserves additional time for sporadic tasks and the proposed slot-shifting approach is also capable of shifting the inter-core communication in manycore platforms.

A Time-Triggered Server (TTS) is a periodic time slot like a time-triggered task that handles sporadic tasks such that they do not interfere with time-triggered tasks. In the time slot of a TTS, no computational or communication task is allowed to execute or communicate. For the integration to the TTCP schedule, the TTS has a slot length W_{TTS}, a period P_{TTS} and a phase Φ_{TTS}. With these parameters, time slots of the TTS are determined by

$$[\Phi_{\text{TTS}} + \ell \cdot P_{\text{TTS}}, \Phi_{\text{TTS}} + \ell \cdot P_{\text{TTS}} + W_{\text{TTS}}) \qquad \forall \ell \in \mathbb{Z}_0^+. \tag{6.1}$$

All time-triggered tasks arrive at time synchronized at time 0. In difference to the computational task model, the TTS has neither a deadline nor a list of predecessor tasks. Each core has a TTS for handling sporadic tasks.

In addition to the TTS definition, the proposed slot-shifting approach can be used to reduce the response time of a sporadic task σ_k. The slot-shifting delay $d_{\text{TTS}}(t)$ defines the delay between the currently-scheduled time-triggered tasks and its corresponding statically-defined version. Without using the proposed slot-shifting approach, the slot-shifting delay is $d_{\text{TTS}}(t) = 0$. To guarantee the feasibility of time-triggered computational and communication tasks, there exists a maximum tolerable slot-shifting delay $d_{\text{TTS,max}}$, which is defined as

$$d_{\text{TTS,max}} = \min_{\forall \tau_i} (\text{slack}_{\tau_i}) \tag{6.2}$$

where slack_{τ_i} is the slack of the computational tasks. The slack is defined as

$$\text{slack}_{\tau_i} = D_{\tau_i} - W_{\tau_i} - \Phi_{\tau_i}, \tag{6.3}$$

where D_{τ_i} is the relative deadline, W_{τ_i} is the WCET and Φ_{τ_i} is the phase of the computational task τ_i.

The principles of the slot-shifting approach are visualized in Figure 6.2. A system always operates in one of four states:

- *Normal*: Time-triggered tasks are executed at their statically-defined time slots. The TTS does not exceed its time slot to execute a sporadic job. The slot-shifting delay is $d_{\text{TTS}}(t) = 0$.
- *Shift*: The sporadic job of σ_k is executed until it finishes or the maximum delay $d_{\text{TTS,max}}$ of the TTS is reached. Only the TTS can start a sporadic job.

- *Retain*: If the sporadic job is not finished but the maximum tolerable delay is reached $d_{\text{TTS}}(t) = d_{\text{TTS,max}}$, the delayed time-triggered tasks is executed. In the retain state, the sporadic task is only allowed to be executed in the corresponding periodic time slots of the TTS.
- *Recover*: The TTS does not execute a sporadic job such that its time slots are used to execute the time-triggered tasks. Thus, skipping of TTS slots reduces the slot-shifting delay $d_{\text{TTS}}(t)$. Each skipped time slot of the TTS reduces the delay by $d_{\text{TTS}}(t) \leftarrow d_{\text{TTS}}(t) - W_{\text{TTS}}$.

Considering a manycore platform, the slot-shifting approach requires a TTS on each core, which are synchronized. Otherwise the communications of a sporadic job could disrupt the time-triggered schedule, i.e. a time-triggered task becomes delayed such that it miss its deadline. Therefore, each core needs to be in the same state. This can be implemented by sending the synchronization information at the beginning of the sporadic job, because at this time no time-triggered task is allowed to communicate. If the slot-shifting approach is not used but the TTS is used to handle the sporadic tasks, the configuration of each TTS can be different for each core, unless the sporadic task needs to communicate via the same communication fabric.

According to the sporadic task model, a sporadic job can arrive at any time, as shown in Figure 6.2. Thus, an arriving sporadic job will be delayed until the time slot of the TTS. Time-triggered tasks are only delayed if a sporadic task requires more time, which is determined based upon its WCET W_{σ_k}. The TTS determines the delay $d_{\text{TTS}}(t)$, which is communicated at the beginning of the sporadic job during a time slot of the TTS. Each core delays its time-triggered tasks, regardless whether a sporadic task is executed on this core, to keep all cores at the same state. This allows the sporadic job to occupy the entire platform at is execution. If a sporadic job only needs one core for its execution, the other cores also have to wait for this job to be completed.

If a sporadic job completes before the maximum tolerable delay $d_{\text{TTS}}(t) < d_{\text{TTS,max}}$, the delay is recovered without entering the *Retain* state. The *Retain* state prevents time-triggered tasks from missing their deadline. This state uses the slots of the TTS to continue executing the sporadic job. In contrast to immediately recovering the delay $d_{\text{TTS}}(t)$, the

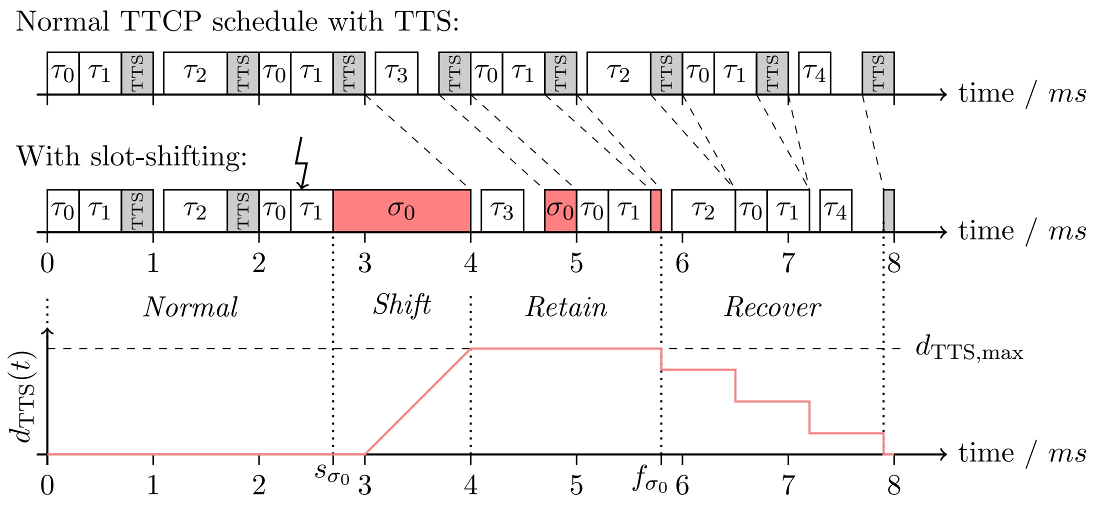

Normal TTCP schedule with TTS:
τ0 τ1 TTS τ2 TTS τ0 τ1 TTS τ3 TTS τ0 τ1 TTS τ2 TTS τ0 τ1 TTS τ4 TTS
time / ms
With slot-shifting:
τ0 τ1 TTS τ2 TTS τ0 τ1 σ0 τ3 σ0 τ0 τ1 τ2 τ0 τ1 τ4
time / ms
0 1 2 3 4 5 6 7 8
Normal
Shift
Retain
Recover
d_TTS(t)
d_TTS,max
time / ms
s_σ0
f_σ0

Figure 6.2.: The concept of shifting the time-triggered schedule comprises four states, namely *Normal*, *Shift*, *Retain* and *Recover*. The delay $d_{\text{TTS}}(t)$ is upper-bounded by $d_{\text{TTS,max}}$ to ensure a feasible time-triggered schedule. s_{σ_0} and f_{σ_0} are the starting and completion time, respectively.

Retain can reduce the response time of the sporadic job because the job is executed in this state.

Figure 6.2 shows the completion time f_{σ_0} of the sporadic task σ_0. In general, the *Recover* state begins at f_{σ_0}. Tasks are executed in the same order as they appear in the statically-defined TTCP schedule, i.e. the most delayed time-triggered tasks are executed first. The slot-shifting delay $d_{\mathrm{TTS}}(t)$ is is reduced by skipping the time reserved for the TTS.

Slot-shifting on-line scheduler implementation

In the following, a scheduler is presented, which implements the proposed slot-shifting approach. This algorithm is analyzed in the subsequent section.

A usual scheduler operates on a task queue for handling incoming tasks. By contrast, a TTCP scheduler determines timer interrupts, which activate the corresponding time-triggered jobs without requiring a task queue. The idea is to use a dedicated queue for handling the sporadic jobs called Sporadic Queue (SQ). Furthermore, the timer interrupts of time-triggered tasks are updated if the delay $d_{\mathrm{TTS}}(t)$ changes. The approach is presented in Algorithm 8.

The scheduler initialization (init) sets up SQ. In a normal TTCP scheduler, each time-triggered computational task τ_i triggers a timer interrupt to start the execution of its jobs. The TTCP scheduler ensures a continuously time-triggered job execution (uninterrupted), before the next timer interrupt triggers another job.

The timer interrupt (also referred as an alarm) for τ_i has the same period P_{τ_i} as its corresponding time-triggered task τ_i. The phase Φ_{τ_i} determines the time of the first timer interrupt. If a time-triggered task needs to be delayed due to the proposed slot-shifting approach, the timer interrupts are shifted according to the slot-shifting delay $d_{\mathrm{TTS}}(t)$. Therefore, timer interrupts need to be dynamically configurable to adjust the delay $d_{\mathrm{TTS}}(t)$.

In the implementation presented in Algorithm 8, the TTS is activated and scheduled like a time-triggered task. An arriving sporadic job is added the SQ. There are two interrupts, namely `TTS-begin-timer-interrupt` and `TTS-end-timer-interrupt`, which trigger each other and represent the begin or end of the TTS slot. When a `TTS-begin-timer-interrupt` is triggered, the scheduler checks the slot-shifting state and reacts appropriately. An `TTS-end-timer-interrupt` suspends the running sporadic job if its time capacity is depleted, by disabling the `sporadic flag`. The `sporadic flag` indicates wether the TTS slot is active and determines whether an arriving sporadic job can be immediately executed.

If the delay $d_{\mathrm{TTS}}(t)$ reaches its maximum tolerable value i.e. $d_{\mathrm{TTS}}(t) = d_{\mathrm{TTS,max}}$ the execution time capacity for the sporadic job of τ_k is limited to the slot length W_{TTS} of the TTS. If no sporadic task needs to be executed, the `TTS-begin-timer-interrupt` routine starts to recover the delay $d_{\mathrm{TTS}}(t)$ by skipping the slots of TTS. The *Recover* state is implemented by defining the next `TTS-begin-timer-interrupt` without defining the `TTS-end-timer-interrupt`. The interrupt `Sporadic task finish` is activated if a sporadic job of τ_k completes its execution. This interrupt triggers the `TTS-begin-timer-interrupt` to continue the execution of delayed time-triggered tasks. The function `SetTimers` sets the timer interrupts for all computational and communication tasks based upon the current delay $d_{\mathrm{TTS}}(t)$.

A normal TTCP scheduler without slot-shifting has a run-time complexity of $O(1)$ because the timer interrupts are configured once according to the *a priori* determined phases. The run-time complexity of scheduler in Algorithm 8 is dominated by the function

Algorithm 8 On-line slot-shifting scheduler

```
Input: T, run-time interrupts;
Output: On-line scheduling decisions;
 1: Init:
 2: Set a timer interrupt for each computational task $\tau_i$ to $\Phi_{\tau_i}$ with period $P_{\tau_i}$;
 3: Set a timer interrupt for each communication task $\kappa_j$ to $\Phi_{\kappa_j}$ with period $P_{\kappa_j}$;
 4: Set the task TTS-begin-timer-interrupt in $\Phi_{\text{TTS}}$;
 5: Set Sporadic Queue (SQ) $\leftarrow \emptyset$;
 6: $d_{\text{TTS}}(t) \leftarrow 0$; sporadic flag $\leftarrow$ false;
 ---------------------------------------------------------------
 7: Regular timer interrupt for $\tau_i$:
 8: Execute the time-triggered task $\tau_i$;
 ---------------------------------------------------------------
 9: Sporadic task interrupt:
10: add sporadic job in SQ;
11: if sporadic flag = true then
12:    execute sporadic task;
 ---------------------------------------------------------------
13: TTS-begin-timer-interrupt:
14: if SQ $\neq \emptyset$ then                                        Retain
15:    if $d_{\text{TTS}}(t) = d_{\text{TTS,max}}$ then
16:       Set TTS-end-timer-interrupt in $W_{\text{TTS}}$;
17:       sporadic flag $\leftarrow$ true; continue Sporadic task in SQ;
18:    else                                                          Shift
19:       $d_{\text{TTS}}(t) \leftarrow \min(d_{\text{TTS}}(t) + W_{\sigma_k}, d_{\text{TTS,max}})$; SetTimers();
20:       Set TTS-begin-timer-interrupt in $d_{\text{TTS,max}} - d_{\text{TTS}}(t)$;
21:       sporadic flag $\leftarrow$ true; continue Sporadic task in SQ;
22: else                                                             Normal
23:    if $d_{\text{TTS}}(t) = 0$ then
24:       Set TTS-end-timer-interrupt in $W_{\text{TTS}}$;
25:       sporadic flag $\leftarrow$ true;
26:    else                                                          Recover
27:       Set the task TTS-begin-timer-interrupt in $\Phi_{\text{TTS}}$;
28:       $d_{\text{TTS}}(t) \leftarrow \max(d_{\text{TTS}}(t) - W_{\text{TTS}}, 0)$; SetTimers();
 ---------------------------------------------------------------
29: TTS-end-timer-interrupt:
30: sporadic flag $\leftarrow$ false;
31: Suspend the sporadic task, if still running;
32: Set the task TTS-begin-timer-interrupt in $\Phi_{\text{TTS}}$;
 ---------------------------------------------------------------
33: Sporadic task finish interrupt:
34: Remove sporadic task from SQ;
35: Activate TTS-begin-timer-interrupt;
 ---------------------------------------------------------------
36: function SetTimers():
37: for each time-triggered computational task interrupt do
38:    Set timer interrupt to $\Phi_{\tau_i} + d_{\text{TTS}}(t)$ with period $P_{\tau_i}$;
39: for each time-triggered communication task interrupt do
40:    Set timer interrupt to $\Phi_{\tau_j} + d_{\text{TTS}}(t)$ with period $P_{\kappa_i}$;
```

`SetTimers`. The TTCP scheduler with slot-shifting has a run-time complexity $O(|\mathbf{T}|+|\mathbf{K}|)$, where $|\mathbf{T}|$ is the number of computational tasks and $|\mathbf{K}|$ is the number of communication tasks. Moreover, Algorithm 8 requires additional memory to store additional interrupt service routines and store the sporadic jobs in SQ.

6.3. Scheduling Analysis of Time-Triggered Tasks by Using Slot-Shifting

This section presents the scheduling analysis of the proposed slot-shifting approach by regarding time-triggered tasks. In order to feasibly shift time-triggered tasks, the scheduler needs to ensure several properties. Algorithm 8 is analyzed in this section to guarantee the feasibility of the proposed slot-shifting approach, which defines an upper bound of the slot-shifting delay $d_{\mathrm{TTS}}(t)$ for the delay according to (6.2). This upper bound $d_{\mathrm{TTS,max}}$ is necessary to ensure that computational tasks do not miss their deadline.

In a TTCP schedule, each computational task completes its execution at a certain time, which is smaller than its relative deadline. Otherwise, the given TTCP schedule is infeasible. The time between its completion and its deadline is called slack slack_{τ_i} (6.3). If the slot-shifting delay exceeds the slack $d_{\mathrm{TTS}}(t) > \mathrm{slack}_{\tau_i}$ of task τ_i, this task misses its deadline. Lemma 3 proves that the bound $d_{\mathrm{TTS,max}}$ is not exceeded.

Lemma 3. *(Upper-bounded delay) In a statically-feasible TTCP scheduled system with a Time-Triggered Server (TTS) and a many core platform with synchronized cores by following the proposed slot-shifting approach (Algorithm 8), the time delay $d_{TTS}(t)$ between the shifted time-triggered schedule and a never-shifted (normal) schedule cannot exceed $d_{TTS,max}$, thus*

$$d_{TTS}(t) \leq d_{TTS,max}. \tag{6.4}$$

Proof. If all cores are synchronized, each core operates in one of the four states. If the delay $d_{\mathrm{TTS}}(t)$ cannot exceed $d_{\mathrm{TTS,max}}$ in any state, then (6.4) holds. The condition holds in all four states, as shown in the following.

- In the *Normal* state, the slot-shifting delay is by definition $d_{\mathrm{TTS}}(t) = 0$. Therefore, the equation $d_{\mathrm{TTS}}(t) \leq d_{\mathrm{TTS,max}}$ holds.
- In the *Shift* state, Algorithm 8 preempts the sporadic task and changes to the *Retain* state if the delay reaches its upper bound $d_{\mathrm{TTS,max}}$.
- The *Retain* state is only entered if the *Shift* state reaches $d_{\mathrm{TTS}}(t) = d_{\mathrm{TTS,max}}$. In the *Retain* state, the delay does not further increase. Thus, the delay remains at the the upper bound $d_{\mathrm{TTS}}(t) = d_{\mathrm{TTS,max}}$. Therefore, $d_{\mathrm{TTS}}(t) \leq d_{\mathrm{TTS,max}}$ holds.
- The *Recover* state only reduces the delay $d_{\mathrm{TTS}}(t)$ by executing the time-triggered tasks and skipping the time reserved for the TTS. Thus, if the other states fulfill $d_{\mathrm{TTS}}(t) \leq d_{\mathrm{TTS,max}}$, then $d_{\mathrm{TTS}}(t) \leq d_{\mathrm{TTS,max}}$ also holds for the *Recover* state.

□

The slot-shifting approach is feasible if the time-triggered tasks fulfill all of their real-time constraints. Feasible time-triggered tasks satisfy the following conditions:

Definition 7. *(feasibility of time-triggered tasks) The shifted schedule is feasible if*

1. *no pair of jobs or packets overlap (interfere) with each other,*
2. *no relative computation deadline D_{τ_i} is violated, and*
3. *no relative communication deadline D_{κ_j} is violated.*

Note that the precedence constraints are implicitly ensured by the communication deadlines considered in (5.8). Based upon Definition 7, the slot-shifting approach needs to ensure that the TTCP schedule remains feasible. Theorem 5 proves the feasibility of the time-triggered tasks by using the proposed slot-shifting method.

Theorem 5. *(time-triggered feasibility) In a statically-feasible TTCP scheduled system with a TTS and a many core platform with synchronized cores by following the proposed slot-shifting approach (Algorithm* 8*), the time-triggered tasks can be feasibly scheduled if*

$$d_{TTS}(t) \leq d_{TTS,max} = \min_{\forall \tau_i} (s_{\tau_i}). \tag{6.5}$$

Proof. Lemma 3 proves that $d_{\text{TTS}}(t) \leq d_{\text{TTS,max}}$. According to Definition 7, each of the three conditions need to be proven.

Condition 1. The time-triggered jobs or packets do no overlap with each over if they have no overlap in each of the four slot-shifting states, represented by the `TTS-begin-timer-interrupt` in Algorithm 8. Note that all time-triggered task interrupts have the same delay due to the synchronized cores.

- In the *Normal* state, no time-triggered task is delayed. Thus, the time-triggered jobs do not overlap to each other since the given time-triggered schedule (without slot-shifting) is feasible.
- The *Shift* state only executes sporadic tasks. Therefore, it has no overlap with the time-triggered tasks. The end of this state is ensured by the `TTS-end-timer-interrupt`.
- The *Retain* state is similar to the normal state, except that all the time-triggered tasks are delayed by $d_{\text{TTS,max}}$. Thus, the equally delayed time-triggered jobs do not overlap with each other if the given time-triggered schedule is feasible. The `TTS-end-timer-interrupt` ensures that the sporadic task stops its execution if a time-triggered task has to be executed.
- In the *Recover* state no sporadic tasks are present. Accordingly, there is no overlap between the sporadic tasks and the time-triggered tasks. The statically-given time-triggered schedule is feasible if the slots of the TTS are not used for computation or communication, which is ensured by design. The delay of the timer interrupts is reduced by W_{TTS} and thus the schedule skips the time reserved for TTS. According to this, the time-triggered tasks cannot have any overlap if the given time-triggered schedule is feasible.

Due to the on-line slot-shifting scheduler (Algorithm 8), time-triggered tasks cannot be preempted by any other sporadic or time-triggered tasks. Thus, time-triggered tasks cannot overlap with each other.

Condition 2. Proof by contradiction: Suppose a time-triggered computational task τ_v can violate its deadline D_{τ_v}. Due to the non-preemptive scheduler, the time-triggered task τ_v can only violate its relative deadline if the k^*-th job of task τ_v is started too late with

$$s_{\tau_v,k^*} + W_{\tau_v} > a_{\tau_v,k^*} + D_{\tau_v}, \tag{6.6}$$

where s_{τ_v,k^*} is the starting time and a_{τ_v,k^*} is the arrival time of the k^*-th job of task τ_v. According to Lemma 3, this job can be delayed at most by $d_{\text{TTS,max}}$. Therefore, the starting time is at most $s_{\tau_v,k^*} = a_{\tau_v,k^*} + \Phi_{\tau_v} + d_{\text{TTS,max}}$, which gives

$$d_{\text{TTS,max}} + \Phi_{\tau_v} + W_{\tau_v} > D_{\tau_v}. \tag{6.7}$$

Due to the definition of $d_{\mathrm{TTS,max}}$ in (6.2), this results in

$$\min_{\forall \tau_i} (D_{\tau_i} - W_{\tau_i} - \Phi_{\tau_i}) > D_{\tau_v} - W_{\tau_v} - \Phi_{\tau_v}. \tag{6.8}$$

(6.8) cannot be fulfilled and thus the deadline cannot be violated.

Condition 3. Proof by contradiction: Suppose a communication task κ_v can violate its deadline,

$$s_{\kappa_v,k^*} + \overline{W}_{\kappa_v} > a_{\kappa_v,k^*} + D_{\kappa_v}. \tag{6.9}$$

The absolute communication deadline is determined based upon the starting time of the destination task τ_{DST_v}. If there is no relative variation (time shift) between sending and receiving of κ_v, the deadline violation is only possible if the given TTCP schedule is infeasible. Due to the feasibility of the TTCP schedule and the definition of the TTS, the communication task κ_v is not allowed to transmit any data within the time slot of the TTS.

In case of a time shift of the destination task τ_{DST_v} by $d_{\mathrm{TTS}}(t) \geq 0$, the absolute communication deadline is also shifted. With $s_{\kappa_v,k^*} = a_{\kappa_v,k^*} + \Phi_{\kappa_v} + d_{\mathrm{TTS}}(t_1)$, (6.9) results in

$$\Phi_{\kappa_v} + \overline{W}_{\kappa_v} + d_{\mathrm{TTS}}(t_1) > D_{\kappa_v} + d_{\mathrm{TTS}}(t_2), \tag{6.10}$$

where t_1 and t_2 are the injection time of the communication packet and the starting time of the successor job of τ_{DST_v}. A feasible TTCP schedule ensures $\Phi_{\kappa_v} + \overline{W}_{\kappa_v} \leq D_{\kappa_v}$. Therefore, the deadline violation of κ_v with $d_{\mathrm{TTS}}(t_1) \leq d_{\mathrm{TTS}}(t_1)$ is only possible, if the already-given time-triggered schedule is infeasible.

For $d_{\mathrm{TTS}}(t_1) > d_{\mathrm{TTS}}(t_1)$, the delay is only reduced step-wise by $d_{\mathrm{TTS}}(t) \leftarrow d_{\mathrm{TTS}}(t) - W_{\mathrm{TTS}}$ at the starting times of the TTS. If the $d_{\mathrm{TTS}}(t)$ is reduced, it skips the statically-defined time reserved for theTTS, which is not to be used by the communication tasks unless the given TTCP schedule is infeasible. Therefore, κ_v can only violate its deadline if an infeasible time-triggered schedule is given. □

According to Theorem 5, the feasibility of the slot-shifting depends on (6.5). The feasibility test of the given time-triggered schedule is presented in Section 5.2.

6.4. Scheduling Analysis of Sporadic Tasks by Using Slot-Shifting

This section deals with the feasibility of sporadic tasks in a time-triggered system. Two approaches are considered in this section. On the one hand, the TTS can be used to handle sporadic tasks without slot-shifting named *only using TTS*. On the other hand, the proposed slot-shifting approach can handle sporadic tasks by using the TTS slot, which shortens the WCRT.

First, this section presents the scheduling analysis of one sporadic task σ_0. In a typical industrial application like an engine control, all sporadic tasks (angle-synchronous tasks) are activated by one sporadic interrupt, which depends on the crankshaft position. Therefore, the sporadic software units of the angle-synchronous tasks can be modeled by one sporadic task, which consecutively executes all sporadic software units. The scheduling analysis can be performed using a tool like Real-Time Calculus (RTC) [20] with given service curves or by a given minimum inter-arrival time, which represents a simplified timing analysis. Second, the problem of handling multiple sporadic tasks is discussed.

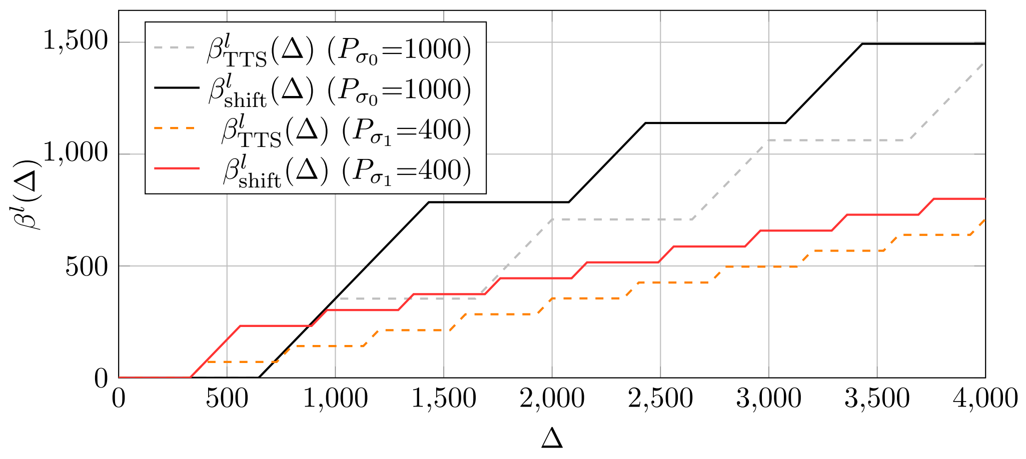


Figure 6.3.: The service curves for handling of sporadic tasks are compared between the slot-shifting approach and the non-shifting TTS. The gap between the curves for the same period is the improvement of the slot-shifting approach.

Feasibility analysis of one sporadic task for given arrival curves

One sporadic task σ_0 could be the angle-synchronous task from an engine control unit. There exist tools to perform the scheduling analysis of a sporadic task called Real-Time Calculus (RTC). RTC models sporadic tasks with arrival cures and uses services curves for modeling the available resources. An arrival curve is the maximum computational demand in a defined time interval Δ. The lower service curve $\beta^l(\Delta)$ of TTS is the minimum accumulated service when the TTS is active (in the *Shift* and *Retain* states) in an interval length Δ. The sporadic task can be modeled by arrival curves. In the following, the service curves of the different approaches are presented.

RTC [20] requires a lower-bounded service curve $\beta^l(\Delta)$ and an upper-bounded arrival curve. If the sporadic task only uses the TTS, the service of the periodic TTS is only provided in time slots of the TTS, which results in

$$\beta^l_{\text{TTS}}(\Delta) = \max\left(\left\lfloor \frac{\Delta}{P_{\text{TTS}}} \right\rfloor W_{\text{TTS}}, \Delta - \left\lceil \frac{\Delta}{P_{\text{TTS}}} \right\rceil (P_{\text{TTS}} - W_{\text{TTS}})\right). \tag{6.11}$$

In comparison to $\beta^l_{\text{TTS}}(\Delta)$, the proposed slot-shifting approach provides more service in the time by delaying time-triggered tasks. In the *Retain* state, a sporadic task has as much service as in the *Normal* state. The service curve $\beta^l_{\text{shift}}(\Delta)$ for the proposed slot-shifting approach starts with a sporadic job of σ_0, thus

$$\beta^l_{\text{shift}}(\Delta) = \begin{cases} \max(\Delta - P_{\text{TTS}} + W_{\text{TTS}}, 0) & \text{if } \Delta < P_{\text{TTS}} + d_{\text{TTS,max}} \\ \beta^l_{\text{TTS}}(\Delta - d_{\text{TTS,max}}) + d_{\text{TTS,max}} & \text{otherwise.} \end{cases} \tag{6.12}$$

Figure 6.3 shows different service curves of $\beta^l_{\text{TTS}}(\Delta)$ and $\beta^l_{\text{shift}}(\Delta)$.

Feasibility analysis of one sporadic task for a given minimum inter-arrival time

Another way to analyze the feasibility of one sporadic task σ_0 is to assume a minimum inter-arrival time P_{σ_0}. The idea is to ensure a minimum time difference between two sporadic jobs of a sporadic task σ_0. Accordingly, this minimum time difference has to be larger than the time to execute the sporadic job and recover delayed time-triggered tasks, which is represented by d_{TTS}. Thus, the situation to schedule another sporadic job is the same.

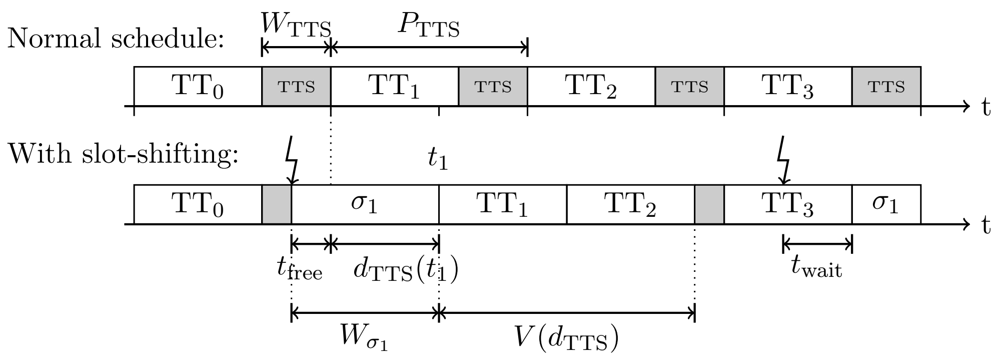


Figure 6.4.: Parameters for analysing the slot-shifting approach. For simplicity, TT represents the time reserved for the time-triggered tasks σ_1.

The worst case for σ_0 is determined by the maximum response time of a sporadic job. This happens, if the arrival time of a job of σ_0 is briefly after the end of the time slot for the TTS. For this arrival time, the sporadic jobs are blocked the maximum possible time, before time-triggered tasks are delayed by the slot-shifting algorithm. The worst case maximizes the worst-case response time (WCRT) R_{σ_0} of the sporadic task σ_0, called R_{σ_0}.

If only the TTS is used to handle the sporadic task σ_0 without slot-shifting, the response time $R_{\sigma_0}^{\text{TTS}}$ is similar to the TDMA response time analysis [94] and results in

$$R_{\sigma_0}^{\text{TTS}} = W_{\sigma_0} + (P_{\text{TTS}} - W_{\text{TTS}}) \left\lceil \frac{W_{\sigma_0}}{W_{\text{TTS}}} \right\rceil . \tag{6.13}$$

Equation (6.13) can be extended to the proposed slot-shifting. The parameters t_{wait} and t_{free} represent additional time intervals, which depend on the arrival time of the sporadic job, as shown in Figure 6.4. t_{wait} is the delay between the starting time and the arrival time of the sporadic task σ_0 if the sporadic task is blocked by time-triggered tasks. t_{free} is the time interval between the arrival time of a sporadic job and the completion of the TTS time slot if the sporadic task arrives in the time slot of TTS. During t_{free} the sporadic job is executed, although the time-triggered tasks that are not started are delayed. The parameters t_{wait} and t_{free} are interdependent. The time interval is $t_{\text{wait}} = 0$ if, and only if, $t_{\text{free}} < W_{\text{TTS}}$. The time interval is $t_{\text{free}} = W_{\text{TTS}}$ if, and only if, $t_{\text{wait}} > 0$. By further assuming that the sporadic job arrives in the *Normal* state, this results in

$$R_{\sigma_0} = t_{\text{wait}} + W_{\sigma_0} + (P_{\text{TTS}} - W_{\text{TTS}}) \left\lceil \frac{\max(W_{\sigma_0} - d_{\text{TTS,max}} - t_{\text{free}}, 0)}{W_{\text{TTS}}} \right\rceil . \tag{6.14}$$

In the worst case, the parameters t_{wait} and t_{free} are defined as $t_{\text{wait}} = P_{\text{TTS}} - W_{\text{TTS}}$ and $t_{\text{free}} = W_{\text{TTS}}$. Thus, response time R_{σ_0} results in the WCRT

$$R_{\sigma_0}^{\text{WC}} = W_{\sigma_0} + (P_{\text{TTS}} - W_{\text{TTS}}) \left\lceil \frac{\max(W_{\sigma_0} - d_{\text{TTS,max}}, 0)}{W_{\text{TTS}}} \right\rceil \tag{6.15}$$

In addition to the response time R_{σ_0}, the recovering time $V(d_{\text{TTS}})$ matters in the scheduling analysis. The recovering time $V(d_{\text{TTS}})$ is the time interval between the completion of a sporadic job and the time when time-triggered tasks are not further delayed $d_{\text{TTS}}(t) = 0$, as shown in Figure 6.4. If a sporadic job delays a time-triggered task by d_{TTS}, it takes a certain amount of time to recover from the delay, called *Recover* state. $V(d_{\text{TTS}})$ is an integer multiple of the of the time-triggered slot, because the slot for the TTS is skipped during the *Recover* state, which results in

$$V(d_{\text{TTS}}) = (P_{\text{TTS}} - W_{\text{TTS}}) \left\lceil \frac{d_{\text{TTS}}(t)}{W_{\text{TTS}}} \right\rceil . \tag{6.16}$$

The maximum recovering time $V_{\max}$ is required if the slot-shifting delay reaches its upper bound $d_{\text{TTS,max}}$. Hence, the maximum recovering time $V_{\max} = V(d_{\text{TTS}} = d_{\text{TTS,max}})$ is calculated according to

$$V_{\max} = (P_{\text{TTS}} - W_{\text{TTS}}) \left\lceil \frac{d_{\text{TTS,max}}}{W_{\text{TTS}}} \right\rceil . \tag{6.17}$$

Based upon the response time R_{σ_0} of the sporadic job and the recovering time $V(d_{\text{TTS}})$, the feasibility of the approach can be proven by Theorem 6.

Theorem 6. *(feasibility of sporadic task) In a statically-feasible TTCP scheduled system with a TTS and a many core platform with synchronized cores by following the proposed slot-shifting approach (Algorithm* 8*), a single sporadic task σ_0 is feasibly schedulable if*

$$(R_{\sigma_0} \leq D_{\sigma_0}) \textit{ and } (R_{\sigma_0} + V(d_{TTS}) \leq P_{\sigma_0}) \tag{6.18}$$

and if the sporadic task starts in the Normal *state, where $V(d_{TTS})$ is the recovering time and R_{σ_0} the response time of σ_0.*

Proof. The sporadic task is feasibly scheduled if the real-time constraints are fulfilled $R_{\sigma_0} \leq D_{\sigma_0}$ and the delayed schedule is fully recovered such that the state is *Normal*.

Suppose a sporadic task σ_0, which generates sporadic jobs arriving at time $a_{\sigma_0,k}$, $\forall k \in \mathbb{Z}_0^+$. The real-time constraints are fulfilled if the worst-case response time R_{σ_0} is upper-bounded by D_{σ_0}. The worst-case response time R_{σ_0} is upper-bounded by (6.15), if the sporadic job arrives in the *Normal* state.

The delayed schedule is fully recovered if two consecutive jobs arrive with a minimum inter-arrival time larger than the time that it takes to fully execute a sporadic job and the time to recover the schedule, thus $R_{\sigma_0} + V(d_{\text{TTS}}) \leq P_{\sigma_0}$ must hold. □

The feasibility of the sporadic task σ_0 is primarily influenced by its relative deadline $R_{\sigma_0} \leq D_{\sigma_0}$. A sporadic task with a shorter response time is more likely feasible. Theorem 6 assumes that the sporadic job starts in the *Normal* state. This property is fulfilled if the entire system starts in the *Normal* state, which is given by a TTCP scheduler.

Feasibility analysis of $|\mathbf{S}| \geq 2$ sporadic tasks for given minimum inter-arrival times

In contrast to the feasibility of one sporadic task, a system may have $|\mathbf{S}| \geq 2$ sporadic tasks with real-time constraints, where $|\mathbf{S}|$ is the number of sporadic tasks. Each sporadic task σ_k needs to be scheduled on the time-triggered system. The proposed slot-shifting approach is capable of handling several sporadic tasks if all tasks are activated by the same trigger like angle-synchronous tasks in typical industrial applications. In general, sporadic tasks are activated by different independent triggers.

These sporadic tasks compete with each other for the time reserved for the sporadic tasks (slot-shifting capacity), which can be expressed by the delay $d_{\text{TTS}}(t)$. By default, each sporadic task can use the slot sifting capacity in a first-come-first-service (FCFS) policy. This results in the worst case that all other sporadic tasks have already depleted the capacity immediately before a certain sporadic task arrives. This is extremely pessimistic such that the worst-case response time (WCRT) could result as long as the response times of using only the TTS for handling the sporadic tasks. For such cases, the slot-shifting approach cannot shorten the worst-case response time (WCRT) of the sporadic tasks.

One approach to deal with multiple sporadic tasks is to select one sporadic task in the ready queue, which is allowed to occupy the slot-shifting capacity. An engine control application would select the angle-synchronous task, which has a high impact in contrast to other sporadic tasks. The other sporadic tasks would be handled by the TTS without shifting the time-triggered schedule. This ensures that the slot-shifting capacity is not empty, if the selected sporadic task arrives such that the response time can be shortened.

Another approach is to establish an arbiter like a FP arbiter to limit the maximum usable shifting capacity to certain tasks. Depending on the slot-shifting access policy, the scheduling analysis needs to consider the maximum blocking time by other sporadic tasks, which is presented by the literature [17, 24]. Another possibility is to establish a generalized rate monotonic scheduling (GRMS) [86] to use the TTS.

6.5. TTS Aware Phase Assignment

This section presents the proposed approach to determine the time-triggered schedule with respect to the Time-Triggered Server (TTS). In general, there exist approaches to determine the phases of the TTCP schedule, as shown in Section 5.3. The TTS has additional requirements because in this time slot no computational or communication task is allowed to compute or communicate on any core. This section provides a computational and communication phase assignment method that constructs a maximized time slot for the TTS in terms of utilization, in which no computational or communication task τ_i, κ_j occurs, respectively.

Overview of the phase assignment approach considering the TTS

The Time-Triggered Server (TTS) is not a usual time-triggered task because the TTS is present on all cores of the manycore platform and during this time slot no communication task is allowed to communicate. In addition, the TTS has to be defined such that:

- its period is minimized to increase the responsiveness of sporadic tasks;
- its utilization is maximized; and
- the other time-triggered tasks still hold their real-time constraints.

The known phase assignment methods requires a period P_{τ_i},P_{κ_j} for each time-triggered task τ_i, κ_j. The constraints allow different periods P_{TTS} for the TTS. The idea is to use a selection of periods and determine the provided service for the sporadic tasks for each period P_{TTS}. If a period fulfills the real-time constrains of time-triggered and sporadic tasks, the period can be selected. Thus, the proposed phase assignment algorithm assumes a given period P_{TTS}.

One simple solution is to use an existing phase assignment approach (Section 5.3) and define the TTS as a time-triggered task. The slot length W_{TTS} of the TTS can be determined with a binary search by applying such a phase assignment algorithm. Such an approach can easily be implemented, although the time complexity is high. Therefore, this section presents a more efficient method.

The idea is to insert the TTS time slot in the TTCP schedule and avoid assigning time-triggered tasks in this slot. The workflow uses an iterative algorithm to determine

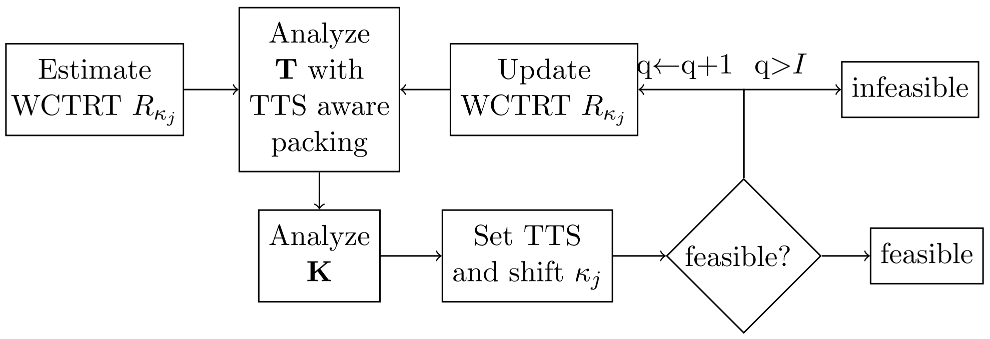

Estimate WCTRT R_{κ_j}
Analyze **T** with TTS aware packing
Update WCTRT R_{κ_j}
q←q+1
q>I
infeasible
Analyze **K**
Set TTS and shift κ_j
feasible?
feasible

Figure 6.5.: The workflow for assigning the phases of time-triggered tasks τ_i, κ_j by handling the cyclic dependencies among **T** and **K** with respect to the TTS.

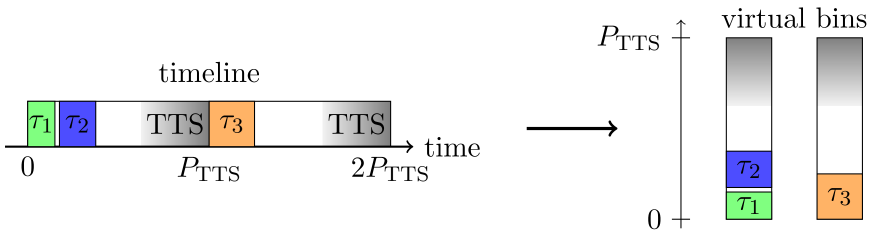


Figure 6.6.: The phases of the time-triggered tasks are assigned in the timeline (left). The timeline can be split into virtual bins (right). If the time-triggered tasks are only placed at the beginning of these time intervals $[k \cdot P_{\text{TTS}}, (k+1) \cdot P_{\text{TTS}})$, $k \in \mathbb{Z}_0^+$, the TTS is maximized, which is equivalent to the *makespan* problem.

computational phases, communication phases and the TTS time slots, as shown in Figure 6.5. In the first iteration the worst-case traversal response time (WCTRT) R_{κ_j} of each communication task κ_j is estimated based upon the network traversal times $\overline{W}_{\kappa_j}$. Within several iterations, the cyclic dependencies between the computational and communication phases assignment converge to fixed point. During the computational phase assignment, the TTS slot is not directly assigned. Instead, the phases Φ_{τ_i} are assigned with a packing strategy, which implicitly optimizes the slot length W_{TTS} for the TTS.

The communication phases Φ_{κ_j} are assigned according to Algorithm 6 from Section 5.3. After the communication phase assignment, time slots for TTS are determined based upon the unused time of the computational and communication tasks. In order to maximize this time slot, communication phases Φ_{κ_j} are partly reassigned to increase the slot length W_{TTS} for the TTS. If phases and TTS slot assignment is feasible, the algorithm stops the iteration and returns the phase assignment with the definition of the TTS. Based upon the approach from Section 5.3, in the following only the two modified steps *Analyze* **T** *with TTS aware packing* and *Set TTS and shift* κ_j are presented.

Analyze T with TTS aware packing

Computational phases Φ_{τ_i} are calculated such that the slot length W_{TTS} of the TTS is maximized. Due to the TTCP approach, each computational task τ_i reserves a time window for its execution. The idea is to insert the TTS time slot in the TTCP schedule and avoid assigning time-triggered tasks in this slot, as shown in Figure 6.6. The TTS is greedily placed at the end of each interval $[k \cdot P_{\text{TTS}}, (k+1) \cdot P_{\text{TTS}})$, $k \in \mathbb{Z}_0^+$, which can be represented by virtual bins. Thus, the phase assignment is related to the well-known *makespan* problem, in which our objective here is to assign the phases of the time-trigger tasks so that they are only executed in the interval $[k \cdot P_{\text{TTS}}, (k+1) \cdot P_{\text{TTS}})$. Accordingly, if the virtual bins have available space, the time slot for the TTS is maximized. To solve

Algorithm 9 Heuristic algorithm for assigning the phases of the computational task set **T** with TTS aware packing

```
Input: T, $\Omega_\tau$;
Output: Phases $\Phi_{\tau_i}$ and feasibility;
 1: for each $\tau_i$ by following $\Omega_\tau$ do
                                                              Preparation of
                                                               virtual bins
 2:     feasible ← false; $U_\Sigma \leftarrow 0$;
 3:     $\text{bin}_b \leftarrow 0, \forall b \in \{0, 1, \ldots, \frac{P_{\tau_i}}{P_{\text{TTS}}}\}$;
 4:     for each $\tau_k$ with an allready assigned phase do
 5:         $\ell \leftarrow \lfloor \Phi_{\tau_k} / P_{\text{TTS}} \rfloor$;
 6:         $\text{bin}_\ell \leftarrow \max(\Phi_{\tau_k} \bmod P_{\text{TTS}} + W_{\tau_k}, \text{bin}_\ell)$;
 7:         $U_\Sigma \leftarrow U_\Sigma + \frac{W_{\tau_k}}{P_{\tau_k}}$;
 8:     $\Psi \leftarrow \Phi_{\tau_i,\min}$;
 9:     $\text{bin}_b \leftarrow \max\Big(\min(\Psi - b \cdot P_{\text{TTS}}, P_{\text{TTS}}), 0\Big), \forall b$;
                                                          Find valid phases by
                                                              packing tasks
                                                            to virtual bins
10:     while $(\Psi < P_{\tau_i})$ and (feasible = false) do
11:         feasible ← true;
12:         FF_size ← $U_\Sigma \cdot P_{\text{TTS}} / |\mathbf{C}|$;
13:         $s$ ← select bin with First-Fit bin packing to FF_size;
14:         if First-Fit bin packing fails then
15:             $s$ ← select bin with Worst-Fit bin packing;
16:         $\Psi \leftarrow \text{bin}_s + s \cdot P_{\text{TTS}}$;
17:         $\delta_\Psi$ ← Resolve conflicts to already-assigned tasks with $\Phi_{\tau_i} \leftarrow \Psi$;
18:         if $\delta_\Psi \neq 0$ then
19:             $\Psi \leftarrow \Psi + \delta_\Psi$; feasible ← false;
20:             $\text{bin}_s \leftarrow \text{bin}_s + \delta_\Psi$;
21:     if $\Psi > P_{\tau_i}$ then
22:         return "not feasible"
23: return "feasible";
```

the makespan problem, heuristics exist to perform the packing, e.g. a worst-fit or first-fit packing strategy.

Algorithm 9 packs the computational task into virtual bins. The feasibility analysis in Section 5.2 determines whether a phase is Φ_{τ_i} valid. Our algorithm uses topological ordering Ω_τ for assigning the computational phases of **T** in a specific sequence. The order Ω_τ is sorted such that it satisfies the following conditions.

1. Tasks with a precedence relation are ordered first. (high priority)
2. Tasks with a lower period are ordered first.
3. Tasks with a larger number of predecessor tasks are ordered first.
4. Tasks with a larger WCET are ordered first. (low priority)

A condition could be violated to satisfy a condition with a higher priority, which are indicated by smaller numbers.

Algorithm 9 assigns the phase for each computational task τ_i step by step. Overall, the algorithm performs two consecutive segments named *preparation of virtual bins* and *find valid phases by packing the task to virtual bins.*

In the first segment *preparation of virtual bins*, the algorithm defines some common variables and sets up the virtual bins considering the already-assigned tasks. The number of bins $|\text{bin}_b|$ is determined based upon periods of computational tasks, with $|\text{bin}_b| = P_{\tau_i}/P_{\text{TTS}}$. The *for-loop* iterates over all computational tasks τ_k, which are already assigned.

Algorithm 10 Set up TTS and reassign the phases of the communication task set **K** to maximize the TTS utilization

Input: **T**, **K** and platform;
Output: TTS with Φ_{TTS}, W_{TTS} and changed Φ_{κ_j}

```
1: $\Theta_\tau \leftarrow \max_{\forall \tau_i}\Big((\Phi_{\tau_i} \bmod P_{\mathrm{TTS}}) + W_{\tau_i}\Big)$;
2: **repeat**
3:     $\Theta_\kappa \leftarrow \max_{\forall \kappa_j}\Big((\Phi_{\kappa_j} \bmod P_{\mathrm{TTS}}) + \overline{W}_{\kappa_j}\Big)$;
4:     Determine $\kappa_\ell$ with maximum $\Theta_\kappa$;
5:     $\Psi \leftarrow \lceil \Phi_{\kappa_\ell}/P_{\mathrm{TTS}} \rceil P_{\mathrm{TTS}}$; $b \leftarrow \Phi_{\kappa_\ell}$;
6:     **while** $(\Psi < P_{\kappa_j})$**and**$(\delta_\kappa \neq 0)$ **do**
7:         $\delta_\kappa \leftarrow$ Resolve conflicts $\forall \kappa_j$ with $\Phi_{\kappa_\ell} = \Psi$; {see Algorithm 7 in Section 5.3}
8:         $\Psi \leftarrow \Psi + \delta_\kappa$;
9:     **if** $\Psi \geq P_{\kappa_j}$ **then**
10:        $\Phi_{\kappa_\ell} \leftarrow b$; shifting $\leftarrow$ **false**;
11:    **else**
12:        shifting $\leftarrow$ **true**;
13: **until** (shifting= **true**)**and**$(\Theta_\kappa > \Theta_\tau)$
14: $\Theta_\kappa \leftarrow \max_{\forall \kappa_j}\Big((\Phi_{\kappa_j} \bmod P_{\mathrm{TTS}}) + \overline{W}_{\kappa_j}\Big)$;
15: $\Phi_{\mathrm{TTS}} \leftarrow \min(\max(\Theta_\tau, \Theta_\kappa), P_{\mathrm{TTS}})$;
16: $W_{\mathrm{TTS}} \leftarrow P_{\mathrm{TTS}} - \Phi_{\mathrm{TTS}}$;
```

The parameter ℓ determines the bin, in which a task is assigned. The corresponding bin size is set by the completion time of the assigned task τ_k to indicate that this time window is already occupied. The parameter U_Σ indicates the summation of the utilization of already-assigned tasks. Considering the minimum computational phase $\Phi_{\tau_i,\min}$ from (5.14), the computational task τ_i can only be feasibly be packed after that time. These bins are set to their maximum bin size P_{TTS}.

In the second segment *find valid phases by packing the task to virtual bins*, the algorithm searches for a feasible phase Φ_{τ_i} of task τ_i. Based upon the prepared bins, a makespan heuristic can propose a phase for Φ_{τ_i}, which is verified by the feasibility test. The parameter *FF_size* determines a limit for the first-fit heuristic to verify whether the computational phase Φ_{τ_i} fit into a certain bin. If the first-fit heuristic does not find a valid phase due to the limit *FF_size*, the worst-fit heuristic is used to select one. The selected bin bin_s determines the hypothetical phase Ψ, which is used to verify the computational phase $\Phi_{\tau_i} = \Psi$. The parameter δ_Ψ represents the time interval between the next free time window and the hypothetical phase Ψ. If $\delta_\Psi = 0$, a valid phase is found; otherwise, the selected bin bin_s is artificially filled by δ_Ψ such that the next iteration can propose another bin. This iteration with the *while-loop* continues until a valid computational phase is found or the range for a valid phase is exceeded.

Set TTS and shift κ_j

The definition of the TTS slot and the refinement of Φ_{κ_j} is a step after the computational and communication phase assignment, as shown in Figure 6.5. The problem is that communication tasks could be placed at any time, although during the TTS slot, no communication task is allowed to communicate. The idea is to define the TTS slot based upon computational tasks and shift the most limiting communication tasks κ_j to increase the slot length W_{TTS}. Algorithm 10 determines the TTS slot and is able to refine the communication phases Φ_{κ_j}. The time slot of the TTS is at the end of its period P_{TTS}, so $W_{\mathrm{TTS}} + \Phi_{\mathrm{TTS}} = P_{\mathrm{TTS}}$.

Furthermore, two parameters are defined named Θ_τ and Θ_κ, which represent a lower bound for determining Φ_{TTS} with respect to the computational or communication phases Φ_{τ_i},Φ_{κ_j}. These are calculated with

$$\Theta_\tau = \max_{\forall \tau_i} \Big((\Phi_{\tau_i} \bmod P_{\text{TTS}}) + W_{\tau_i} \Big), \tag{6.19}$$

$$\Theta_\kappa = \max_{\forall \kappa_j} \Big((\Phi_{\kappa_j} \bmod P_{\text{TTS}}) + \overline{W}_{\kappa_j} \Big). \tag{6.20}$$

Due to the TTS aware computational phases assignment, Θ_τ is already optimized. Typically, communication tasks limit the slot length W_{TTS} of the TTS such that $\Theta_\kappa > \Theta_\tau$. Note that the makespan approach is not applicable to the communication phase assignment, because communication tasks need to be scheduled immediately to reduce the WCTRT R_{κ_j}. Communication phases are only refined if they limit the slot length W_{TTS} of the TTS.

Algorithm 10 first determines Θ_τ. The *repeat-until-loop* performs the communication phase refinement. With Θ_κ, the *repeat-until-loop* selects a communication task κ_ℓ, which limits the TTS at most. For this critical communication task κ_ℓ, a hypothetical phase Ψ is the temporary phase used to find a valid communication phase Φ_{κ_ℓ}.

The *while-loop* represents the iteration to find a valid communication phase Φ_{κ_ℓ}. If the hypothetical phase Ψ is invalid, the delay δ_κ is added to Ψ to continue the searching (see Algorithm 7 in Section 5.3), where δ_κ is the time between the hypothetical phase and the next potential free time window.

In case of a valid communication phase Φ_{κ_ℓ}, the slot for this communication task κ_ℓ is shifted and another critical communication task is investigated. The communication phase refinement is complete if a critical communication task cannot be validly shifted or, if $\Theta_\kappa \leq \Theta_\tau$. After the *repeat-until-loop*, the parameters of the TTS are determined by

$$\Phi_{\text{TTS}} = \min(\max(\Theta_\tau, \Theta_\kappa), P_{\text{TTS}}) \tag{6.21}$$

$$W_{\text{TTS}} = P_{\text{TTS}} - \Phi_{\text{TTS}} \tag{6.22}$$

6.6. Evaluations

This section presents experiments to determine the limits of the slot-shifting approach by considering industrial settings. The reduction of the response time R_{σ_0} by using the proposed slot-shifting approach is quantified in this section.

6.6.1. Experimental Setup

This subsection describes the environment for running the experiments of one sporadic task σ_0. Considering typical industrial applications, sporadic tasks are activated by the same sporadic interrupt. For the experiments, these tasks are modeled to one sporadic task σ_0.

The system generator uses the dependent system model generator from Section 5.4.1 as a base for further extensions. In addition, the sporadic task σ_0 is defined as one of three template sporadic tasks that represent typical industrial characteristics. In each experiment, 100 randomized dependent system sets are generated. The TTS aware phase assignment algorithm is applied to each of the randomized system sets. Based upon the definition of the TTS slot, the WCTRT R_{σ_0} of the sporadic task is determined with and without the slot-shifting approach.

For the definition of the TTS, there are several possibilities to generate the period P_{TTS} of the TTS. The algorithm sets the period P_{TTS} such that the response time is minimized.

Table 6.1.: Sporadic Service Selection: A selection of different configurations of the TTS depending on the period P_{TTS}. Only one period of the TTS slot is selected.

Period P_{TTS} /μs	W_{TTS} /μs	U_{TTS}	$d_{\text{TTS,max}}/\mu s$	$V_{\text{max}}/\mu s$
200	56	28.0%	1581	4032
250	79	31.6%	216	342
400	232	58.0%	318	168
500	297	59.4%	409	203
625	412	65.9%	455	213
1000	708	70.8%	790	292
1250	906	72.5%	1035	344
2500	1934	77.4%	2044	566

This could result in different periods for the slot-shifting approach and the sporadic task handling by only using the TTS.

By default, the generator uses a 3×3 manycore platform with a NoC. There are $|\mathbf{T}| = 500$ computational tasks and $|\mathbf{K}| = 1,500$ communication tasks. The precedence rate is set to $p = 20\%$. The default computational utilization and communication utilization is $U_\tau = 1$ and $U_\kappa = 1$, which represent the maximum single-core or a bus utilization, respectively.

6.6.2. Experimental Results

This subsection presents the experiments handling sporadic tasks by defining a TTS slot for the TTCP scheduler.

Sporadic service curve selection

This experiment examines the problem to define the period P_{TTS} for the TTS. In addition, it visualizes the improvement of the slot-shifting approach. To define P_{TTS}, there is a trade-off between a short blocking time to start the sporadic execution and a large slot length W_{TTS} to have more time for executing the sporadic task σ_0. Table 6.1 and Figure 6.7 show the possible configurations of P_{TTS} and their resulting parameters.

Note that U_{TTS} represents the TTS utilization with $U_{\text{TTS}} = W_{\text{TTS}}/P_{\text{TTS}}$. For smaller periods P_{TTS}, the utilization U_{TTS} is smaller because the packing heuristic adds more pessimism for unused time windows of the time-triggered task. Due to the trade-off, no period P_{TTS} dominates. Thus, the selection of the period P_{TTS} depends on the setting of the sporadic task σ_0.

In addition, Figure 6.7 visualizes our two approaches for handling the sporadic task, namely *slot-shifting* and *only TTS*. The service curves are visualized for different periods indicated by the number in the legend. The *slot-shifting* approach is marked with *S*. As shown in Figure 6.7, the service curve of the proposed *slot-shifting* approach is greater than or equal to the *only TTS* approach. The experiment illustrates that the slot-shifting approach can increase the service for a certain amount of time, resulting in a shorter response time R_{σ_0} for the sporadic task.

Example of TTCP schedule with the TTS

The TTS aware phase assignment influences the TTCP schedule. The TTS is like a computational task, although it run synchronously on each core. During the time slot of the TTS, no computational or communication tasks are allowed to compute or communicate.

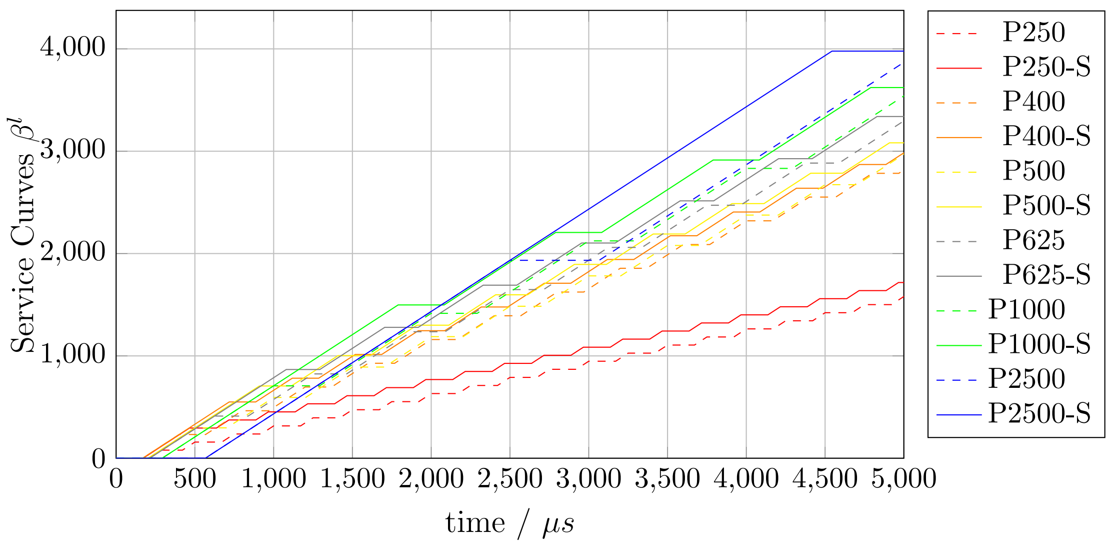


Figure 6.7.: Sporadic Service Selection: For a specific system set, the configuration of the TTS with different periods results in different service curves β^l. The number indicates the period. *S* and the solid lines indicate that the proposed slot-shifting approach is used. The dashed lines represent the approach by only using the TTS for handling the sporadic task.

Table 6.2.: Load Examples: Different sporadic tasks to examine the slot-shifting approach

Task name	WCET / μs	Relative deadline / μs	Inter-arrival time / μs	Worst-case response time reduction by
Fast	100	1000	5000	⌀ 10.9%
Async	1500	3000	30000	⌀ 25.4%
Heavy	3000	20000	20000	⌀ 20.7%

Figure 6.8 provides an example of the TTCP schedule, where the yellow area represents the time slots for the TTS.

In this example, there are $|\mathbf{T}| = 100$ computational tasks and $|\mathbf{K}| = 300$ communication tasks. The manycore platform comprises $|\mathbf{C}| = 9$ cores with a NoC, in which the time reservation of in the $|\mathbf{L}| = 42$ links are visualized. The computational utilization is set to $U_\mathcal{T} = 4.5$ and the communication utilization is $U_\mathcal{T} = 2.5$. Figure 6.8 shows the schedule of each core $\mathbf{C}_k$ and link $\mathbf{L}_l$ in the system within the hyper-period $H = 80ms$.

Load examples

For this experiment, there are three different sporadic tasks named *Fast*, *Async* and *Heavy*. The sporadic task represents different load example that could be used in industrial application. Table 6.2 gives the corresponding timing parameters of the three sporadic tasks. Note that the minimum period of time-triggered tasks is $10ms$.

The sporadic task *Fast* has a short WCET and a short minimum inter-arrival time, which represents a short but urgent sporadic task. The sporadic task *Async* represents the angle-synchronous task from an engine control unit. This task has a short relative deadline and a moderate WCET. The sporadic task *Heavy* has a large WCET, which represents a high computing demanding task.

This experiment determines the WCTRT for each sporadic task, as shown in Figure 6.9. Considering the selection period P_{TTS} of the TTS, the shortest WCTRT is used in the

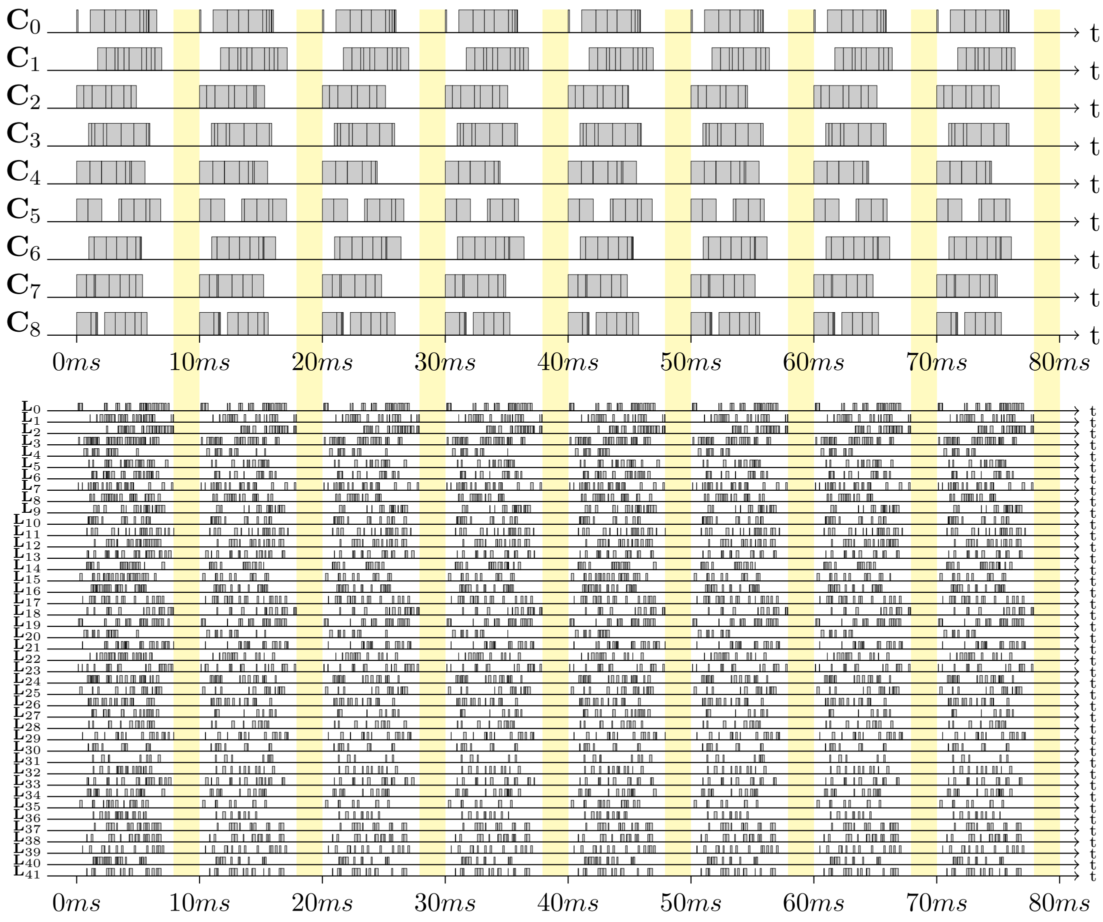


Figure 6.8.: Example of the TTS aware TTCP schedule. The yellow areas in each timeline represent the time slots for the TTS on the core $\mathbf{C}_k$ and link $\mathbf{L}_l$ schedule, in which no time-triggered task is allowed to compute or communicate.

evaluation. Note that the plot in Figure 6.9 shows normalized response times to compare the different load examples. The corresponding relative deadlines are marked as dashed lines to show whether the sporadic task can reach its deadline. The proposed slot-shifting approach always has a shorter response time than only using the TTS.

Given that the slot sifting approach always increases the responsiveness of the sporadic task, it can better utilize the system. In general, the worst-case response time (WCRT) R_σ of the sporadic task could be reduced by up to 25% depending on the load, as shown in Table 6.2. Note that the average reduction of the response time depends on the definition of the sporadic task.

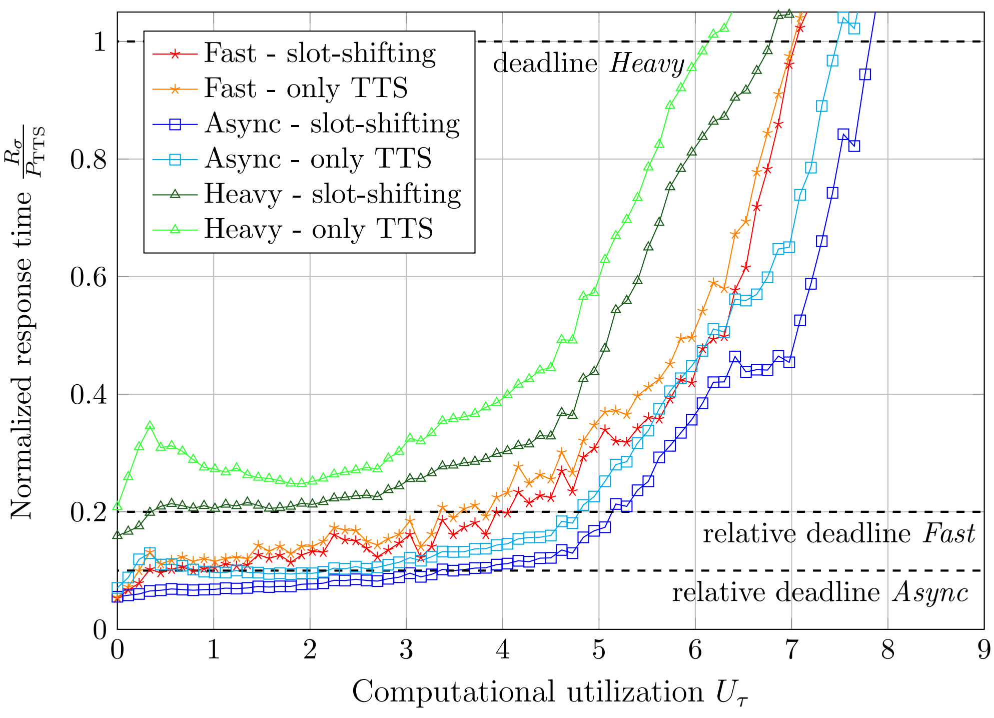


Figure 6.9.: Load Examples: The worst-case response times are presented for different sporadic tasks *Fast*, *Async* and *Heavy*, as shown in Table 6.2. The response times are normalized to their respective period. The computational utilization of the time-triggered part of the system is varied. The dashed lines represent the relative deadlines.

7. Exploring the Typical Execution Time Scheduling Approach

In contrast to the TTCP scheduling approach, this chapter assumes that the inter-task communication can be modeled as a part of the execution time, whereby tasks can be scheduled independently. The proposed approach exploits different execution times of the same task, which are scheduled on a multicore platform. The author has already published the primary results of this chapter [32].

7.1. Introduction

Typical industrial applications comprise tasks that communicate data among each other. Communication requires additional time, which needs to be considered. The literature [17, 24] usually includes data communication in the worst-case execution time (WCET) to derive independently schedulable tasks. The WCET is a safe upper bound on the execution time of a particular task, which takes the hardware effects into account. Thus, the scheduler is able to schedule independent tasks without considering the communication. This chapter uses the independent sporadic task model, as given in Section 3.2.

For the scheduling analysis, it is essential to know a safe upper bound on the execution time, although it is a complex problem to derive a tight WCET. There exist different tools and approaches to derive the WCET of a specific task scheduled on a specific platform. The basic approach is to decompose a task into basic blocks, which are atomic operations of the software, for which an execution time can be derived. A graph describes the execution order of basic blocks, in which for example *if* or *while* statements define branches and loops in the graph. This graph is analyzed such that all possible paths are examined, while the path that requires the most execution time defines the WCET. In addition, the WCET analysis requires a hardware model to determine the execution times for each basic block, the latencies of memory requests and other hardware-dependent times. For instance, tools like aiT [29] or BoundT [40] can derive the WCET as an upper bound to the maximum execution time.

For a multicore platform, the WCET estimation becomes more pessimistic because the communication to other cores also needs to be modeled in the WCET. In the worst case, this communication becomes interfered by communications of all other cores, which results in a larger WCET. The best-case execution time (BCET) remains the same, because communications are assumed to interfere with each other. Another possibility to determine the WCET of a task is to run the given task on the desired platform and measure its execution time. The problem with run-time measurements is that no guarantee exists for the worst case, because usually too many combinations of software and hardware states exist for a comprehensive test. Therefore, measured execution times do not represent a safe upper bound. Figure 7.1 visualizes the problem of determining the WCET of a task.

If the gap between the best-case and the worst-case execution time is large for a task, it typically completes its execution much earlier than its WCET. A typical execution can be defined based upon a percentile, e.g. 90%, 99% or up to 100%, of the meassured execution times. However, the platform reserves resources (like processing time) for tasks

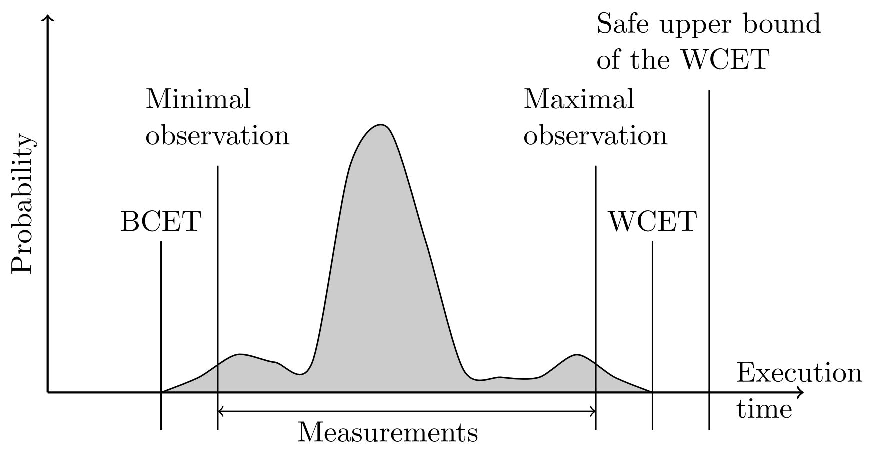


Figure 7.1.: The difficulty in obtaining the worst-case execution time (WCET). Note that there also exists a lower bound for the execution time called best-case execution time (BCET). Image adapted from [96].

that are only used in the worst case. In typical executions, these reserved resources are not used, which results in low resource utilization. For example, suppose a multicore platform with $|\mathbf{C}| = 2$ cores with a bus architecture to handle inter-core communication. In order to ensure a safe system behavior in the worst case, both cores are required, although in typical cases (e.g. 90%) of executions only one core is required. Thus, both cores are mostly used with a low utilization. This is a waste of energy, because two cores are regularly on power.

Multiple cores open a new possibility to deal with such behavior. The proposed scheme is to use mostly one core and use another core to deal with the worst case, which is only rarely used. This approach can also handle industrial applications, which exceeds a single-core platform in the worst case. However, considering typical cases, a single-core platform is sufficient to handle the application.

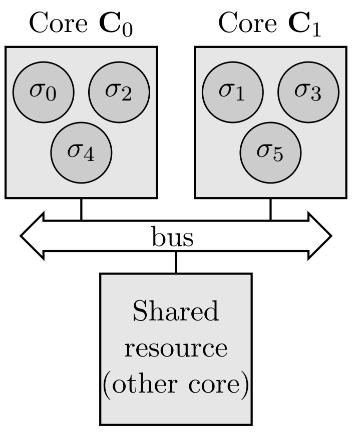
This chapter introduces a scheme that exploits the typical-case execution times of a task on two cores. The idea is to use separated cores for the typical-case execution part and the worst-case execution part of a task such that one core can handle most task executions. Thus, one core is highly utilized to handle the typical cases of the task execution, whereas another core is used to handle with the rare worst-case workloads.

Based upon this scheme, the problem is to derive a schedule by following the scheme for each core such that the real-time constraints of the independent tasks hold. Industrial applications often use a Fixed-Priority (FP) scheduler. The FP scheduling policy schedules the task with the highest priority first.

Example of the WCET estimation on a multicore platform

Suppose a multicore platform with two cores, a bus with a priority-based arbitration and a common shared resource like an external communication controller or a shared memory, as shown in Figure 7.2. In this example, each task comprises two parts. The first part is a computation for $2ms$ and the second part is a communication using the shared resource for $1ms$. During the computation, the cores clearly do not interfere with each other. During the communication part, a task can be blocked by another higher-priority communication.

In the worst case, the communication is blocked by all other higher-priority communications from the other core. Under the assumption that the communication is a part of the WCET, this blocking can increase the WCET on a multicore platform. In contrast to a single-core platform, the communications are performed in a sequential manner such that



Task	Bus priority	Core	P_{σ_k} $/ms$	WCET (single-core)	WCET (multicore)
σ_0	0 (high)	$\mathbf{C}_0$	15	$3ms$	$3ms$
σ_1	1	$\mathbf{C}_1$	15	$3ms$	$4ms$
σ_2	2	$\mathbf{C}_0$	15	$3ms$	$4ms$
σ_3	3	$\mathbf{C}_1$	15	$3ms$	$5ms$
σ_4	4	$\mathbf{C}_0$	15	$3ms$	$5ms$
σ_5	5 (low)	$\mathbf{C}_1$	15	$3ms$	$6ms$

Figure 7.2.: WCET estimation on a multicore platform: Several sporadic tasks run on different cores. Each sporadic task computes for $2ms$ and then communicates for $1ms$ with the shared resource. For independent tasks, the communication is part of the WCET. On a single-core platform, the sequential nature of a task execution ensures no access conflicts to the shared resource. On a multicore platform, the access to the shared resource can be blocked by all higher-priority messages in the worst case. Thus, if the the communication is modeled as part of the WCET, it is increased on a multicore platform.

no interference in the communication is possible. Therefore, the gap between the BCET and the WCET increases on a multicore platform such that typically the reserved time for the worst case is only rarely needed. The proposed approach exploits this gap and schedules the typical- and the exceptional-case execution parts on different cores.

Related work

The model of independent tasks is commonly used in the literature [17, 24, 59]. The proposed approach of separating typical-case executions from worst-case executions is to our knowledge the first one, where multiple cores are exploited to separate these two parts of a task. Quinton et al. [79] exploit the gap between the typical-case execution time and the worst-case execution time by considering overloaded situations. Rather than reserving another core for handling the worst case, they do not provide a safe guarantee for the worst case such that all deadlines can be satisfied. In general, the scheduling analysis can use statistical or probabilistic methods [28, 60], although these methods cannot provide a guarantee satisfying real-time constraints.

The proposed approach can be modeled as a special case of a real-time system that needs to meet its end-to-end deadlines [9,41]. The two parts of typical- and worst-case executions can be modeled as two tasks with a sequencing constraint and different dedicated core assignments. For such a problem, the execution of a task instance has to follow the given sequence of processors. Bettati and Liu [9] propose a strategy to handle a sequence of tasks under a Fixed-Priority (FP) scheduler, in which they calculate the relative deadline as a fixed portion of the period for each task. This approach is not effective in our case, which is also presented in this chapter. Hong et al. [41] use an EDF scheduling policy and calculate relative deadlines by an on-line algorithm. Considering industrial application, the fixed priory scheduling approach is widely used [5], because it has a low on-line overhead.

Melani et al. [65] propose separating the memory access phase and the computing phase to different cores. Thus, the worst case can be improved by avoiding contention on shared resources. Their proposed approach cannot handle tasks with different computing demands, in which a single-core platform lack sufficient computing capabilities for the worst-case execution of tasks.

7.2. Scheme for Exploiting the Typical- and Worst-Case Execution Time

In this section, a scheme is introduced to handle different execution parts of a task. The idea to split each sporadic task σ_k into two parts, namely the *typical-case execution part* and *exceptional-case execution part.* Thus, the WCET W_{σ_k} of a sporadic task can also be split into two parts, named the typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$ and exceptional-case execution time $W_{\sigma_k}^{\mathrm{Exc}}$ with

$$W_{\sigma_k}^{\mathrm{Typ}} + W_{\sigma_k}^{\mathrm{Exc}} = W_{\sigma_k}. \tag{7.1}$$

There are several possibilities to define the typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$. One possibility is to split the task according to a percentile (e.g. 95%) of its messured execution times that would handle the most cases. Another possibility is to define the typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$, based upon the maximum observed execution time. Thus, the difference between the maximum measured execution time and the safe upper bound by a WCET analyzer can be handled by the proposed approach.

After defining the typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$, this time becomes a fixed time bound, at which the task σ_k stops its execution on core $\mathbf{C}_0$ and continues on core $\mathbf{C}_1$. Therefore, in most cases a task completes its execution before its typical-case execution time. Only in rare cases, a task σ_k need additional execution time, although this time is upper-bounded by its WCET W_{σ_k}.

Considering a multicore platform with $|\mathbf{C}| = 2$ cores and a bus architecture, the scheme is to execute all *typical-case execution parts* on one core and all *worst-case execution part* on another core. In addition, each core schedules its assigned sporadic tasks σ_k according to a Fixed-Priority (FP) scheduling policy. Thus, *typical-case execution parts* are executed on core $\mathbf{C}_0$ under a Fixed-Priority (FP) scheduler. Core $\mathbf{C}_1$ executes *exceptional-case execution parts* of sporadic tasks σ_k, which rarely occur due to the definition of the typical cases.

If a task is split into two parts executed on different cores, the system needs to support task migration. In particular, a job of a sporadic task σ_k migrates from core $\mathbf{C}_0$ to core $\mathbf{C}_1$ if the time exceeds its typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$. The migration overhead can be modeled as part of the exceptional-case execution time.

In order to guarantee a feasible execution, the time to start the exceptional-case execution part is defined. The time between the arrival of the typical-case execution part and the exceptional-case execution part of a sporadic task σ_k is referred to the relative offset O_{σ_k}. Figure 7.3 provides an example of two sporadic tasks visualizing the migration overhead and the relative offset O_{σ_k}. If *exceptional-case execution parts* of a sporadic task σ_k are greedily activated, the inter-arrival time of two consecutive executions could be smaller than its period. This could cause an incorrect response time analysis on core $\mathbf{C}_1$. Therefore, *exceptional-case execution part* arrive at $a_{\sigma_{k,\ell}} + O_{\sigma_k}$, where $a_{\sigma_{k,\ell}}$ is the arrival time of the ℓ-th job of sporadic task σ_k.

Thus, the relative offset O_{σ_k} represents a fixed offset to the arrival time of a sporadic job to activate the *exceptional-case execution part.* The offset is defined by the worst-case response time (WCRT) $R_{\sigma_k}^{\mathrm{Typ}}$ of the *typical-case execution part* of σ_k, thus

$$O_{\sigma_k} = R_{\sigma_k}^{\mathrm{Typ}}. \tag{7.2}$$

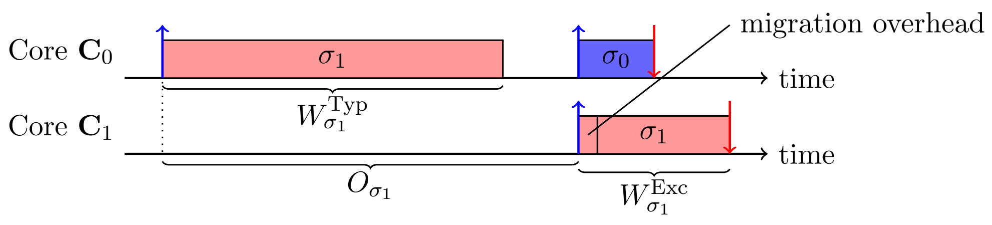


Figure 7.3.: An example of two tasks: σ_1 needs unusual plenty of time for its execution. After exceeding the typical-case execution time $W^{\text{Typ}}_{\sigma_1}$, σ_1 migrates to core $\mathbf{C}_1$. Accordingly, σ_0 is able to be executed on $\mathbf{C}_0$. Note that the exceptional part of task σ_1 is not allowed to start before the relative offset O_{σ_1}. The relative offset is the time between the arrival of the typical and the exceptional part of the task. The arrows represent the arrival and the deadlines, respectively.

For a feasible schedule, the relative offset O_{σ_k} could also be defined with larger values, although this would result in larger response times for the sporadic task σ_k. Hence, the task set would be harder to be feasible.

Due to the FP scheduling policy of each core, each sporadic task part has a unique priority, i.e. no task has the same priority as another one. Thus, the typical-case execution part and the exceptional-case execution part can have different priorities. On core $\mathbf{C}_0$, the priority for a typical-case execution part is defined as $\pi^{\mathbf{C}_0}_{\sigma_k}$ of a sporadic task σ_k. Similarly, $\pi^{\mathbf{C}_1}_{\sigma_k}$ represents the priority for an exceptional-case execution part of σ_k. The unique priorities are indicated by integer numbers, where each number exists only once in the $\pi^{\mathbf{C}_0}_{\sigma_k}$ and $\pi^{\mathbf{C}_1}_{\sigma_k}$. A lower number $\pi^{\mathbf{C}_0}_{\sigma_k} < \pi^{\mathbf{C}_0}_{\sigma_l}$ indicates a higher priority, whereby task σ_k has a higher priority than task σ_l.

7.3. Priority Assignment Problem

This section defines the problem of this chapter. In addition, a straight forward approach is shown that handles typical- and exceptional-case execution parts of sporadic tasks ineffectively.

Problem

Under a Fixed-Priority (FP) scheduler on each of the two cores and an independent sporadic task set $\mathbf{S}$ with $|\mathbf{S}|$ sporadic tasks σ_k, the so-called Real-Time Scheduling for Exploiting the Typical- and Worst-Case Execution Times (REETIC) problem is to

- define the priority of each task part $\pi^{\mathbf{C}_0}_{\sigma_k}$, $\pi^{\mathbf{C}_1}_{\sigma_k}$, and
- define a relative offset O_{σ_k} to start the exceptional-case execution part of each task σ_k

such that all real-time constraints are satisfied.

Considered tasks models for determining the priority assignment

Based upon this scheme, the major problem is to define the priority assignment of each task part. This chapter uses three sightly different task models namely:

- *Frame-based tasks* are independent sporadic tasks, where periods are identical $P_{\sigma_k} = P_\sigma$ and relative deadlines are implicit $D_{\sigma_k} = P_\sigma$, $\forall \sigma_k \in \mathbf{S}$.
- *Frame-based tasks with constraint deadlines* are independent sporadic tasks, where periods are identical $P_{\sigma_k} = P_\sigma$ and relative deadlines are constraint $D_{\sigma_k} \leq P_\sigma$, $\forall \sigma_k \in \mathbf{S}$.

- *Independent sporadic tasks* are defined in Section 3.2.

These different task models are use to find methods for the priority assignment.

Same priority assignments on both cores

In this paragraph, first frame-based tasks are assumed. A simple and straight forward approach is to assign the priorities of typical- and exceptional-case execution parts with same priority $\pi_{\sigma_k}^{\mathbf{C}_0} = \pi_{\sigma_k}^{\mathbf{C}_1}$, $\forall \sigma_k \in \mathbf{S}$. The priorities are assigned according to the index $k = \pi_{\sigma_k}^{\mathbf{C}_0} = \pi_{\sigma_k}^{\mathbf{C}_1}$ of the frame-based task σ_k. Thus, frame-based tasks with a larger index k have a lower priority. Clearly, if the relative deadline exceeds the typical- or exceptional-case execution part, there cannot exist a feasible priority assignment, whereby

$$\sum_{l=0}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Typ}} \leq D_\sigma \qquad \text{and} \qquad \sum_{l=0}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq D_\sigma \qquad \text{must hold,} \tag{7.3}$$

where $|\mathbf{S}|$ is the number of tasks in the set $\mathbf{S}$ and D_σ is the common relative deadline for the frame-based tasks.

The worst case occurs if all tasks require their WCET W_{σ_k}. Thus, each task executes its exceptional-case execution part completely. In addition, by considering the lowest priority task $\sigma_{|\mathbf{S}|-1}$, the worst-case response time (WCRT) occurs if all tasks release their exceptional-case execution part at almost the same time, as shown in Figure 7.4. For *frame-based tasks*, the same priority assignment for each core causes a large WCRT for the task with the lowest priority $\sigma_{|\mathbf{S}|-1}$. Figure 7.4 provides an example of three frame-based tasks, where tasks are preempted shortly before they complete their typical-case execution part. Therefore, the WCRT $R_{\sigma_k}^{\text{Typ}}$ with the same priority $\pi_{\sigma_k}^{\mathbf{C}_0} = \pi_{\sigma_k}^{\mathbf{C}_1}$ of a frame-based task σ_k can be calculated according to

$$R_{\sigma_k} = \sum_{l=0}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Typ}} + \sum_{l=0}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} = \sum_{\forall \sigma_l \in \mathbf{S}} (W_{\sigma_l}^{\text{Typ}} + W_{\sigma_l}^{\text{Exc}}) \tag{7.4}$$

Based upon this response time analysis, the specific priority assignment does not matter, because the summation also represents the worst case for a single-core platform. Thus, for this priority assignment, a multicore platform cannot effectively use more than one core.

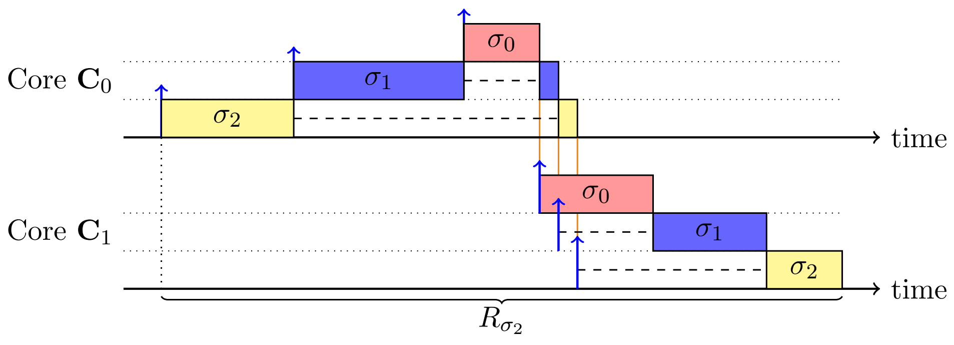


Figure 7.4.: Example of the worst case if typical- and exceptional-case execution parts are scheduled with the same priority. Task σ_0 has the highest priority and task σ_2 has the lowest priority. Thus, the resulting WCRT R_{σ_2} for σ_0 is large. Note that a dashed line represents a preemption of a sporadic task with a higher priority.

In addition, this response time analysis is tight, i.e. if the analysis returns a feasible result, no feasible solution exists.

Another possibility is to apply the approach from Bettati et al. [9] to solve the REETIC problem. Suppose sporadic tasks with implicit deadlines $D_{\sigma_k} = P_{\sigma_k}$, $\forall \sigma_k \in \mathbf{S}$. For this approach, further parameters are defined, namely a minimum utilization $U_{\min}$, a maximum utilization $U_{\max}$ and a constant λ. The minimum and maximum utilization are defined as $U_{\min} = \min\{\sum_{\sigma_k \in \mathbf{S}} \frac{W^{\text{Typ}}_{\sigma_k}}{P_{\sigma_k}}, \sum_{\sigma_k \in \mathbf{S}} \frac{W^{\text{Exc}}_{\sigma_k}}{P_{\sigma_k}}\}$ and $U_{\max} = \max\{\sum_{\sigma_k \in \mathbf{S}} \frac{W^{\text{Typ}}_{\sigma_k}}{P_{\sigma_k}}, \sum_{\sigma_k \in \mathbf{S}} \frac{W^{\text{Exc}}_{\sigma_k}}{P_{\sigma_k}}\}$. The constant λ is used to define the relative offset for releasing the exceptional-case execution part with $O_{\sigma_k} = \lambda P_{\sigma_k}$. They use a Rate Monotonic (RM) scheduler to define the priorities of the sporadic tasks and hence a task with a lower period has a higher priority. The schedulability test can be performed based upon the sporadic task utilization.

If $U_{\min} \leq 0.5$, $U_{\max} \leq |\mathbf{S}|((2(1-U_{\min}))^{1/|\mathbf{S}|} - 1) + U_{\min}$ and by setting λ or $(1-\lambda)$ to $U_{\min}$, the resulting schedule is feasible for the REETIC problem. These two inequalities can be enforced by

$$\sum_{\sigma_k \in \mathbf{S}} \frac{W^{\text{Typ}}_{\sigma_k} + W^{\text{Exc}}_{\sigma_k}}{P_{\sigma_k}} \leq |\mathbf{S}|((2(1-U_{\min}))^{1/|\mathbf{S}|} - 1) + 2U_{\min} \leq 1, \tag{7.5}$$

where the ≤ 1 comes from the fact that $|\mathbf{S}|((2(1-U_{\min}))^{1/|\mathbf{S}|} - 1) + 2U_{\min}$ is a decreasing function with respect to $|\mathbf{S}|$ and an increasing function with respect to $U_{\min}$.

As a conclusion, their approach [9] to set the relative offset O_{σ_k} of the sporadic task σ_k cannot solve the REETIC problem. In the following, a different priority for each part of a task σ_k is assumed to improve the WCRT. The approach employing the same priorities for the typical- and exceptional-case execution part of the task is used as a comparison in later sections.

7.4. Approaches for Different Task Models

As previously defined, this chapter explores three different task models namely *Frame-based tasks*, *Frame-based tasks with constraint deadlines* and *sporadic tasks.* The idea is to increase the complexity of the problem step by step to derive solutions for a complex problem. For each task model, the scheduling analysis and different priority assignment methods are presented.

7.4.1. Priority Assignment for Frame-Based Tasks

First, this chapter considers a frame-based task model in which all tasks have the same period $P_{\sigma_k} = P_\sigma$ and implicit releative deadlines $D_{\sigma_k} = P_\sigma$, $\forall \sigma_k \in \mathbf{S}$. For this task model, the limitation of the feasibility comes by the maximum WCRT of all frame-based tasks. Considering only frame-based tasks, the REETIC problem represents a so-called two-staged flow-shop problem, which is a well-known scheduling problem [75]. Therefore, the priority assignment problem can be solved by applying methods of flow-shop problems.

Two-stage flow-shop problem

A flow-shop problem is a well-known problem [75] that comes from the traditional production problem where different machines in a manufactures need to be utilized [77]. Thus, a flow-shop is an abstract name to schedule different jobs on different machines. A staged flow-shop describes when a job has to be scheduled in a specified order for different machines. Therefore, these jobs can be considered as a sequence of sub-jobs, which need to be scheduled in a predefined order on given machines. A *two-staged flow-shop problem* has two given machines upon which jobs have to be processed, first on one machine and then on another.

The REETIC problem can be represented by a two-staged flow-shop problem, where the two machines are the cores $\mathbf{C}_0$, $\mathbf{C}_1$ and frame-based tasks represent flow-shop jobs. The first stage of a job represents the typical-case execution time $W_{\sigma_k}^{\mathrm{Typ}}$ and the second stage represents the exceptional-case execution time $W_{\sigma_k}^{\mathrm{Exc}}$. The difference from the two-staged flow-shop problem is the periodicity of frame-based tasks, although it is simple to assume that the execution order from flow-shop jobs can be used to define priorities of frame-based tasks. Thus, tasks that are ordered first, have a higher priority.

The objective in a flow-shop problem is to minimize the *makespan*, which is the maximum completion time of all jobs if all jobs arrive at the same time. Thus, the *makespan* represents the maximum WCRT of all tasks in $\mathbf{S}$. The optimal approach to solve this two-staged flow-shop problem by using frame-based tasks is to use an approach called Johnson's sequence [44]. In the following, this approach is rephrased to the notation of the REETIC problem. Johnson's sequence [44] is defined as an order of jobs of a two-stage flow-shop problem. For frame-based tasks, all jobs are assumed to arrive at the same time. Thus, Johnson's sequence can be defined as follows:

The frame-based tasks are split into two sets $\mathbf{S}_1$ and $\mathbf{S}_2$. The first set $\mathbf{S}_1$ contains all frame-based tasks in which the typical-case execution time is smaller than or equal to the exceptional-case execution time $W_{\sigma_k}^{\mathrm{Typ}} \leq W_{\sigma_k}^{\mathrm{Exc}}$. The second set $\mathbf{S}_2$ contains all frame-based tasks in which the typical-case execution time is larger than the exceptional-case execution time $W_{\sigma_k}^{\mathrm{Typ}} > W_{\sigma_k}^{\mathrm{Exc}}$. In Johnson's sequence, all frame-based tasks from the first set $\mathbf{S}_1$ are ordered before the tasks in the second set $\mathbf{S}_2$. In set $\mathbf{S}_1$, the tasks are ordered in a non-decreasing manner of $W_{\sigma_k}^{\mathrm{Typ}}$. In the other set $\mathbf{S}_2$, the tasks are ordered in a non-increasing manner according to $W_{\sigma_k}^{\mathrm{Exc}}$.

$$\mathbf{S}_1 = \{\sigma_k \in \mathbf{S} | W_{\sigma_k}^{\mathrm{Typ}} \leq W_{\sigma_k}^{\mathrm{Exc}}\}$$
$$\mathbf{S}_2 = \{\sigma_k \in \mathbf{S} | W_{\sigma_k}^{\mathrm{Typ}} > W_{\sigma_k}^{\mathrm{Exc}}\}.$$

Note that the original flow-shop scheduling problem was focused on the scheduling of the first machine and thus the scheduling policy for the second machine can be any kind of workload-conserving policy.

Lemma 4. *(Optimality Johnson's sequence) Johnson's sequence is optimal for the two-stage flow-shop problem to minimize the makespan.*

Proof. This has been proven in [44]. □

Priority ordering and schedulability analysis

Based upon Johnson's sequence, the priorities of the frame-based tasks can be defined. The idea is to assign tasks that are earlier ordered with higher priority on the first core $\mathbf{C}_0$, which handles the typical-case execution parts. The priorities of the exceptional-case execution parts for the second core $\mathbf{C}_1$ are inverse to the first core $\mathbf{C}_0$.

Suppose the frame-based tasks are ordered according to Johnson's sequence, i.e. the index k of a task σ_k is its order in Johnson's sequence. Thus, the proposed approach to determine the priorities is as follows:

$$\pi_{\sigma_k}^{\mathbf{C}_0} = k, \qquad \forall \sigma_k \in \mathbf{S} \tag{7.6}$$
$$\pi_{\sigma_k}^{\mathbf{C}_1} = |\mathbf{S}| + 1 - k, \qquad \forall \sigma_k \in \mathbf{S}. \tag{7.7}$$

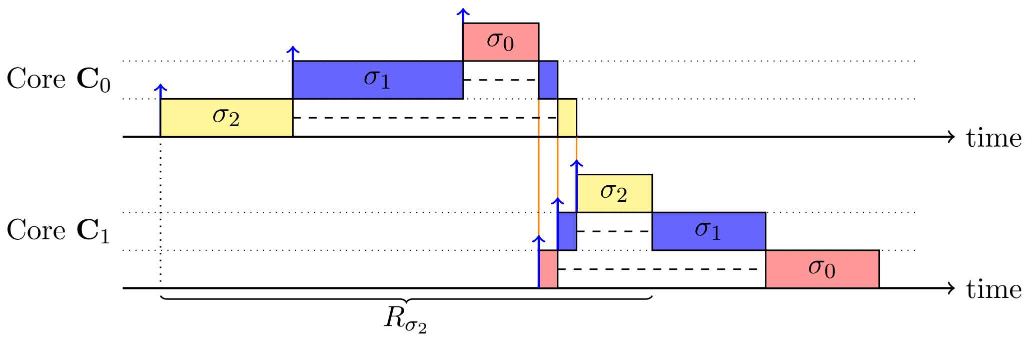


Figure 7.5.: Example of the worst case, if typical- and exceptional-case execution parts are scheduled with different priorities. Task σ_0 has the highest priority and task σ_2 has the lowest priority. The resulting WCRT R_{σ_2} for of the improved priority assignment is smaller.

This priority assignment can easily be implemented by an algorithm with a run-time complexity of $O(|\mathbf{S}| \log |\mathbf{S}|)$, where $|\mathbf{S}|$ is the number of frame-based tasks. This run-time complexity is dominated by the sorting of frame-based tasks. Figure 7.5 shows an example of the resulting WCRT R_{σ_k}, which are smaller in comparison to the simple approach if the typical- and exceptional case execution parts have the same priority.

With Lemma 4, the necessary condition for the scheduling analysis can be defined.

Lemma 5. *(Necessary condition frame-based tasks) Suppose that the tasks are indexed according to the Johnson's sequence for the corresponding two-stage flow-shop problem. If*

$$\exists \sigma_k \in \mathbf{S}, \qquad \text{such that} \sum_{l=0}^{k} W_{\sigma_l}^{Typ} + \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{Exc} > D_{\sigma}, \tag{7.8}$$

then there is no feasible solution for the REETIC problem, where D_σ is the common relative deadline.

Proof. A special case is considered, in which all the tasks arrive at time 0, whereby the proof shows that the condition in (7.8) leads to the non-existence of feasible solutions. This proof is achieved by contradiction. Suppose a feasible solution exists. Based upon Lemma 4, it means that the resulting makespan of the Johnson's sequence for the corresponding two-stage flow-shop problem is less than or equal to D_σ. Suppose that the second machine continues to execute after σ_{k^*} finishes its execution on the first machine in the schedule based upon the Johnson's sequence for of the two-stage flow-shop problem. The makespan is defined by task σ_{k^*}. Accordingly,

$$\sum_{l=0}^{k^*} W_{\sigma_l}^{\text{Typ}} + \sum_{l=k^*}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq D_\sigma.$$

Moreover, for any task σ_k in $\mathbf{S}$, this is

$$\sum_{l=0}^{k} W_{\sigma_l}^{\text{Typ}} + \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq \sum_{l=0}^{k^*} W_{\sigma_l}^{\text{Typ}} + \sum_{l=k^*}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq D_\sigma.$$

Therefore, this contradicts to (7.8). As a result, there is no feasible solution for such a case. □

Based upon the defined priority assignment (7.6) and (7.7), the WCRT $R_{\sigma_k}^{\text{Typ}}$ for the typical-case execution part can be defined as

$$R_{\sigma_k}^{\text{Typ}} = \sum_{l=0}^{k} W_{\sigma_l}^{\text{Typ}}. \tag{7.9}$$

With this response time, the relative offset $O_{\sigma_k} = R_{\sigma_k}^{\text{Typ}}$ can be defined according to (7.2). The following lemma presents the feasibility test by using the proposed priority assignment approach.

Lemma 6. *(Feasibility test frame-based tasks) Suppose that the priorities of the frame-based tasks on $\mathbf{C}_0$ are defined to Johnson's sequence and $\pi_{\sigma_k}^{\mathbf{C}_1} = |\mathbf{S}| + 1 - k$ for all $\sigma_k \in \mathbf{S}$. Moreover, let O_{σ_k} be $\sum_{l=0}^{k} W_{\sigma_l}^{Typ}$. If*

$$\sum_{l=0}^{k} W_{\sigma_l}^{Typ} + \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{Exc} \leq D_\sigma, \qquad \forall \sigma_k \in \mathbf{S}, \tag{7.10}$$

then the priority assignment is guaranteed to be feasible for the REETIC problem.

Proof. According to the critical instant theorem in [59], releasing all the higher-priority tasks at time 0 together with task σ_k results in the longest (worst-case) response time for task σ_k. The worst-case response time (WCRT) $R_{\sigma_k}^{\text{Typ}}$ of task σ_k on $\mathbf{C}_0$ is

$$R_{\sigma_k}^{\text{Typ}} = \sum_{l=0}^{k} W_{\sigma_l}^{\text{Typ}}, \tag{7.11}$$

as $\sum_{l=0}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Typ}} \leq D_\sigma$ has been implicitly assumed.

Based upon the proposed scheme in Section 7.2, if task σ_k has an instance arriving at time t and it requires the second core for executing the exceptional case, the corresponding instance for the task on the second core will be activated on $t + R_{\sigma_k}^{\text{Typ}}$. Moreover, according to the scheme, the resulting job arrivals of task σ_k on $\mathbf{C}_1$ will still have a minimum inter-arrival time equals to $P_\sigma \geq D_\sigma$. Therefore, again according to the critical instant theorem and the definition $\pi_{\sigma_k}^{\mathbf{C}_1} = |\mathbf{S}| + 1 - k$, the worst-case response time $R_{\sigma_k}^{\text{Exc}}$ of task σ_k on the second core (the finishing time minus the arrival time of the exceptional case on the second core) is

$$R_{\sigma_k}^{\text{Exc}} = \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}}. \tag{7.12}$$

As a result, the overall worst-case response time $WCRT(\sigma_k)$ of task σ_k by considering the exceptional case on both cores is

$$R_{\sigma_k} = R_{\sigma_k}^{\text{Typ}} + R_{\sigma_k}^{\text{Exc}} = \sum_{l=0}^{k} W_{\sigma_l}^{\text{Typ}} + \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq D_\sigma, \tag{7.13}$$

where the inequality comes from the statement in (7.10) in the theorem. Therefore, if (7.10) holds, the priority assignment provides a feasible solution under the proposed scheme in Section 7.2. □

Based upon Lemma 6, it can be proven that this priority assignment is optimal i.e. if there exists a feasible priority assignment the proposed approach determines a feasible solution.

Theorem 7. *For frame-based real-time tasks, it is optimal for the REETIC problem to assign the priorities, i.e.* $\pi_{\sigma_k}^{\mathbf{C}_0} = k$*, on* $\mathbf{C}_0$ *based upon the corresponding Johnson's sequence,* $\pi_{\sigma_k}^{\mathbf{C}_1} = |\mathbf{S}| + 1 - k$ *for* $\sigma_k \in \mathbf{S}$*, and* $O_{\sigma_k} = \sum_{l=0}^{k} W_{\sigma_l}^{Typ}$*. Together with the priority ordering, the schedulability test in (7.10) requires* $O(|\mathbf{S}| \log |\mathbf{S}|)$ *time complexity.*

Proof. The first part comes directly from Lemmas 5 and 6. After the priority ordering is done, to verify whether (7.10) holds with run-time complexity of $O(|\mathbf{S}|)$. Therefore, the time complexity is dominated by the complexity $O(|\mathbf{S}| \log |\mathbf{S}|)$ for priority ordering. □

7.4.2. Priority Assignment for Frame-Based Tasks with Constraint Deadlines

This section deals with the priority assignment problem of frame-based tasks with constraint deadlines $P_{\sigma_k} = P_\sigma$, $D_{\sigma_k} \leq P_\sigma$, $\forall \sigma_k \in \mathbf{S}$.

One approach is to use the same strategy as for frame-based tasks with implicit deadlines, as presented in the previous section. The priority assignment according to Johnson's sequence can be modified for constrained deadlines. Consequently, the schedulability test (7.10) from Lemma 6 needs to be modified to

$$\sum_{l=0}^{k} W_{\sigma_l}^{\text{Typ}} + \sum_{l=k}^{|\mathbf{S}|-1} W_{\sigma_l}^{\text{Exc}} \leq D_{\sigma_k}, \qquad \forall \sigma_k \in \mathbf{S}, \tag{7.14}$$

Based upon (7.14), constraint relative deadlines D_{σ_k} influence the feasibility. Thus, another priority assignment can improve the feasibility, e.g. suppose an urgent task with a short relative deadline D_{σ_k}, which should have a higher priority. The idea is to first assign the priority of each typical-case execution part of a task σ_k on core $\mathbf{C}_0$. The priority assignment on core $\mathbf{C}_0$ changes the worst-case response time behavior on the second core $\mathbf{C}_1$. Hence, the relative offset O_{σ_k} depends on the priority assignment of the first core $\mathbf{C}_0$.

The proposed approach defines an *effective relative deadline* $\overline{D}_{\sigma_k}$ for the typical-case execution parts of each task σ_k with respect to the relative offset O_{σ_k}, as defined with

$$\overline{D}_{\sigma_k} = D_{\sigma_k} - O_{\sigma_k} \tag{7.15}$$

With $\overline{D}_{\sigma_k}$, the exceptional-case execution part of σ_k on core $\mathbf{C}_0$ can use an optimal priority assignment approach, i.e. the Deadline Monotonic (DM) scheduling policy [55]. Thus, the proposed approach is to use the DM scheduling policy to determine priorities of exceptional-case execution parts on core $\mathbf{C}_1$ with the proposed effective relative deadlines $\overline{D}_{\sigma_k}$. For all task pairs σ_k, σ_l, the priorities are defined such that $\pi_{\sigma_k}^{\mathbf{C}_1} < \pi_{\sigma_l}^{\mathbf{C}_1}$ holds, if $\overline{D}_{\sigma_k} < \overline{D}_{\sigma_l}$.

The relative offset $\overline{D}_{\sigma_k}$ depends on the priority assignment of core $\mathbf{C}_0$. Accordingly, our problem is to define the priorities for the typical-case execution parts on core $\mathbf{C}_0$. One approach is to go through all priority assignments with an exhaustive search, although this takes factorial time complexity of $O(|\mathbf{S}|!)$, where $|\mathbf{S}|$ is the number of sporadic tasks. Thus, the proposed approach uses a heuristic algorithm to determine the priority assignment. Algorithm 11 is called Effective Deadline aware Priority Assignment (EDPA), which iteratively determines the priorities of typical-case execution parts on core $\mathbf{C}_0$. The algorithm assigns the priorities step by step from the lowest to the highest priority.

Algorithm 11 iterates over all tasks $\sigma_k \in \mathbf{S}$. Each iteration assigns one of the priorities with a descending manner, starting with the index $k = |\mathbf{S}| - 1$, which indicates the lowest

Algorithm 11 Effective Deadline aware Priority Assignment (EDPA)

```
Input: frame-based task set $\mathbf{S}$ with constrained deadlines;
Output: priority ordering for $\mathbf{S}$ on $\mathbf{C}_0$ and $\mathbf{C}_1$;
 1: $\mathbf{G} \leftarrow \emptyset$;
 2: for $k = |\mathbf{S}| - 1, \cdots, 0$ stepped by 1 do
 3:   for each $\sigma_j$ in $\mathbf{S} \setminus \mathbf{G}$ do
 4:     $\overline{D}'_{\sigma_j} \leftarrow D_{\sigma_j} - \sum_{\sigma_k \in \mathbf{S} \setminus \mathbf{G}} W^{\text{Typ}}_{\sigma_k}$;
 5:     calculate slack$_{j,j}$ and slack$_{j,\ell}$ according to (7.17) and (7.18);
 6:     slack$_j \leftarrow \min \left\{ \text{slack}_{j,j}, \min_{\sigma_\ell \in \mathbf{G} \text{ with } \overline{D}_{\sigma_\ell} \geq \overline{D}'_{\sigma_j}} \{\text{slack}_{j,\ell}\} \right\}$;
 7:   if there exists $\sigma_j$ in $\mathbf{S} \setminus \mathbf{G}$ with slack$_j \geq 0$ then
 8:     choose task $\sigma_{j^*} \in \mathbf{S} \setminus \mathbf{G}$ with the max slack$_j$;
 9:     for all $\sigma_\ell \in \mathbf{G}$ do
10:       update $r_\ell$ with $r_\ell \leftarrow r_\ell + W^{\text{Exc}}_{\sigma_{j^*}}$ if $\overline{D}_{\sigma_\ell} \geq \overline{D}'_{\sigma_{j^*}}$;
11:     $r_{j^*} \leftarrow W^{\text{Exc}}_{\sigma_{j^*}} + \sum_{\sigma_\ell \in \mathbf{G} \text{ with } \overline{D}_{\sigma_\ell} < \overline{D}'_{\sigma_{j^*}}} W^{\text{Exc}}_{\sigma_\ell}$;
12:     set $\pi^{\mathbf{C}_0}_{\sigma_{j^*}}$ to $k$; put $\sigma_{j^*}$ to $\mathbf{G}$; set $\overline{D}_{\sigma_{j^*}}$ to $\overline{D}'_{\sigma_{j^*}}$;
13:   else
14:     return "no feasible solution is found";
15: set $\pi^{\mathbf{C}_1}_{\sigma_k} \forall \sigma_k \in \mathbf{S}$ with the DM policy with effective relative deadlines $\overline{D}_{\sigma_k}$ (in case of
    the same effective relative deadline $\overline{D}_{\sigma_k} = \overline{D}_{\sigma_\ell}$, assign $\sigma_k$ with the higher priority
    $\pi^{\mathbf{C}_0}_{\sigma_k} < \pi^{\mathbf{C}_0}_{\sigma_\ell}$);
16: return the resulting priority orderings $\pi^{\mathbf{C}_0}_{\sigma_k}$, $\pi^{\mathbf{C}_1}_{\sigma_k}$;
```

priority $\sigma_{|\mathbf{S}|-1}$. In each iteration, the proposed algorithm greedily assigns the priority $\pi^{\mathbf{C}_0}_{\sigma_{j^*}}$ to k without re-assigning any priority. The temporary task set $\mathbf{G}$ is used to track the already-assigned priorities and thus $\mathbf{G}$ contains tasks with an already-assigned priority. First, the effective relative deadline $\overline{D}_{\sigma_j}$ of each task σ_j is calculated by their WCRT $\sum_{\sigma_k \in \mathbf{S} \setminus \mathbf{G}} W^{\text{Typ}}_{\sigma_k}$ on core $\mathbf{C}_0$. If the task σ_j would be assigned by the current priority k, Step 4 in Algorithm 11 calculates $\overline{D}'_{\sigma_j}$ to represent the relative deadline for the exceptional-case execution part in the worst-case priority assignment.

Suppose task σ_j is assigned with the priority k, this assignment can also affect the worst case of the already-assigned priorities in $\mathbf{G}$ because the priorities on core $\mathbf{C}_1$ are still unknown. Those tasks $\sigma_\ell \in \mathbf{G}$ that can affect the WCRT of σ_j, have a larger effective relative deadline $\overline{D}'_{\sigma_j} > \overline{D}_{\sigma_\ell}$. Tasks σ_ℓ could miss their deadline by assigning the priority k to σ_j. Thus, algorithm EDPA calculates the impact of σ_j with priority k to σ_ℓ, which is denoted as r_ℓ. r_ℓ represents the WCRT on core $\mathbf{C}_1$ by excluding the tasks that are not already assigned with a priority with

$$r_\ell = \sum_{\sigma_\ell \in \mathbf{G} \text{ with } \overline{D}_{\sigma_\ell} < \overline{D}'_{\sigma_j}} W^{\text{Exc}}_{\sigma_\ell}. \tag{7.16}$$

By knowing this impact r_ℓ, the *slack* can be defined.

In general, the slack is the time between the absolute deadline and the completion time of a specific task σ_j. In our case, the slack depends on the priority assignment of the already-assigned tasks and tasks $\sigma_\ell \in \mathbf{G}$ with $\overline{D}_{\sigma_\ell} < \overline{D}'_{\sigma_j}$. Thus, the slack for a task σ_j can be defined as

$$\text{slack}_{j,j} = \overline{D}'_{\sigma_j} - W^{\text{Exc}}_{\sigma_j} - \sum_{\sigma_\ell \in \mathbf{G} \text{ with } \overline{D}_{\sigma_\ell} < \overline{D}'_{\sigma_j}} W^{\text{Exc}}_{\sigma_\ell}. \tag{7.17}$$

Table 7.1.: Example frame-based task set with constrained deadlines.

task	$W_{\sigma_k}^{\text{Typ}}/ms$	$W_{\sigma_k}^{\text{Exc}}/ms$	D_{σ_k}/ms	P_{σ_k}/ms	$\pi_{\sigma_k}^{\mathbf{C}_0}$	$\pi_{\sigma_k}^{\mathbf{C}_1}$
σ_0	2	4	12	25	0 (high)	3 (low)
σ_1	1	2	10	25	1	0 (high)
σ_2	3	1	15	25	2	2
σ_3	7	2	20	25	3 (low)	1

Considering tasks $\sigma_\ell \in \mathbf{G}$, the slack is defined with respect to the impact r_ℓ with

$$\text{slack}_{j,\ell} = \overline{D}_{\sigma_\ell} - W_{\sigma_j}^{\text{Exc}} - r_\ell. \tag{7.18}$$

In Step 6 in Algorithm 11, the slack for each task σ_j can be calculated as the minimum of (7.17) and (7.18) to ensure a safe assignment. If the calculated slack$_j$ is negative, the task σ_j cannot be feasibly be set by the priority $\pi_{\sigma_j}^{\mathbf{C}_0} = k$ because another task would miss its deadline. The proposed heuristic approach is to set the priority $\pi_{\sigma_j}^{\mathbf{C}_0}$ of the task with the maximum slack slack$_j$ to k. However, if the calculated slack$_j$ is negative, there cannot exist a feasible priority assignment. From Step 9 to 12, the parameters are updated for the next iteration, which determines the priority $k+1$ of another task.

Finally, the priorities of exceptional-case execution parts $\pi_{\sigma_k}^{\mathbf{C}_1}$ can be determined based upon effective relative deadlines $\overline{D}_{\sigma_k}$. Algorithm Effective Deadline aware Priority Assignment (EDPA) uses a DM scheduling policy to determine the priority of each exceptional-case execution part of σ_k. In contrast to the normal DM approach, relative deadlines are replaced by the proposed effective relative deadlines $\overline{D}_{\sigma_k}$.

Regarding the time complexity of Algorithm 11, the slack calculation is the domination part. The slack calculation is performed for each task σ_j and requires time complexity of $O(k(|\mathbf{S}| - k + 1))$. Thus, the time complexity of algorithm EDPA is $O(|\mathbf{S}|^3)$, where $|\mathbf{S}|$ is the number of frame-based tasks.

Theorem 8. *For frame-based real-time tasks with different deadlines, the priority assignments $\pi_{\sigma_k}^{\mathbf{C}_0}$ and $\pi_{\sigma_k}^{\mathbf{C}_1}$ derived by Algorithm EDPA (with time complexity $O(n^3)$) can meet the timing constraints for the REETIC problem.*

Proof. This comes directly from the sufficient condition for the schedulability analysis in each iteration. Accordingly, at the end of each iteration i, the non-negative slack$_{j^*}$ ensures that the timing constraints will be satisfied on both $\mathbf{C}_0$ and $\mathbf{C}_1$. □

Example of algorithm EDPA

In the following, an example demonstrates the usage with their intermediate steps of algorithm Effective Deadline aware Priority Assignment (EDPA). Table 7.1 provides an example of $|\mathbf{S}| = 4$ frame-based tasks, for which the priorities for their typical- and exceptional-case execution part need to be determined. Note that the initial task ordering is artificially prepared according to the priority ordering. In the following, each individual iteration of the *outer-for loop* of Algorithm 11 is presented.

1. $k = 3$ (first iteration): The variables of the algorithm are defined as follows: $\overline{D}'_{\sigma_j} = (-1ms, -3ms, 2ms, 7ms)$, slack$_j$ $= (-5ms, -5ms, 1ms, \mathbf{5ms})$ for σ_0 to σ_3. Based upon the maximum slack$_j$, σ_3 gains priority $\pi_{\sigma_3}^{\mathbf{C}_0} = 3$ on core $\mathbf{C}_0$ and σ_3 is added to $\mathbf{G}$. With this assignment, the remaining parameters are $\overline{D}_{\sigma_3} = 7ms$ and $r_3 = 2ms$.

2. $i = 2$ (second iteration): The variables of the algorithm are defined as follows: $\overline{D}'_{\sigma_j} = (6ms, 4ms, 9ms)$ (σ_3 is not any more considered) for σ_0 to σ_2 and

$$\begin{aligned}\text{slack}_0 =& \min\{6ms - 4ms, 7ms - (4ms + 2ms)\} = 1ms\\ \text{slack}_1 =& \min\{4ms - 2ms, 7ms - (2ms + 2ms)\} = 2ms\\ \text{slack}_2 =& \min\{9ms - (1ms + 2ms)\} = \mathbf{6ms}.\end{aligned}$$

   Therefore, task σ_2 has priority $\pi^{\mathbf{C}_0}_{\sigma_2} = 2$. Now, $\mathbf{G}$ contains $\{\sigma_2, \sigma_3\}$, $\overline{D}_{\sigma_2} = 9ms$, $r_2 = 3ms$, and $r_3 = 2ms$.

3. $i = 1$ (third iteration): The variables of the algorithm are defined as follows: $\overline{D}'_{\sigma_j} = (9ms, 7ms)$ (σ_2 and σ_3 are no longer considered) for σ_0 to σ_1 and

$$\begin{aligned}\text{slack}_0 =& \min\{9ms - (4ms + 2ms + 1ms), 9ms - (4ms + 3ms)\} = 2ms\\ \text{slack}_1 =& \min\{7ms - (2ms + 2ms), 9ms - (2ms + 3ms)\} = \mathbf{3ms}.\end{aligned}$$

   Therefore, task σ_1 has priority $\pi^{\mathbf{C}_0}_{\sigma_1} = 1$. Now, $\mathbf{G}$ contains $\{\sigma_1, \sigma_2, \sigma_3\}$, where $\overline{D}_{\sigma_1} = 7ms$, $r_1 = 2ms$, $r_2 = 5ms$, and $r_3 = 4ms$.

4. $i = 0$ (last iteration): The variables of the algorithm are defined as follows: $\overline{D}'_{\sigma_0} = 10ms$, $\text{slack}_0 = 1ms$. Task σ_0 has priority $\pi^{\mathbf{C}_0}_{\sigma_0} = 0$ and the resulting priorities are feasible for feasible for the REETIC problem.

Based upon the effective relative deadlines $\overline{D}_{\sigma_k} = (10ms, 7ms, 9ms, 7ms)$, the priorities for the second core $\mathbf{C}_1$ can be defined. The relative offset O_{σ_k} defines arrival times of the exceptional-case execution parts with $O_{\sigma_k} = R^{\text{Typ}}_{\sigma_k} = (2ms, 3ms, 6ms, 13ms)$. Note that the relative deadlines for σ_1 and σ_3 are equal $\overline{D}_{\sigma_1} = \overline{D}_{\sigma_3}$, whereby the priority on the first core $\pi^{\mathbf{C}_0}_{\sigma_k}$ decide about the higher priority.

7.4.3. Priority Assignment for Sporadic Tasks

In contrast to frame-based tasks, this section assumes sporadic tasks $\sigma_k \in \mathbf{S}$ as described in Section 3.2. The idea is to extend the approach of the previous section and use an improved heuristic algorithm called Effective Deadline aware Priority Assignment (EDPA)*. The structure and concept of the algorithm are similar to EDPA, although this section needs to handle different periods. Thus, the worst-case response time (WCRT) analysis from the literature [54] needs to be adapted to our algorithm for the sporadic task model. Note that the most variables from EDPA are redefined to determine the EDPA* algorithm.

The WCRT analysis has to consider all higher-priority tasks. Suppose a sporadic task σ_j with a priority k, the WCRT $R^{\text{Typ}}_{\sigma_j}$ on core $\mathbf{C}_0$ has to be determined. In addition, let $\mathbf{G}$ be a set of sporadic tasks that have a lower priority than σ_j. Note that $\mathbf{G}$ is defined in such a manner, because algorithm EDPA* assigns the priorities from the lowest to the highest.

According to the literature [17], on one core $\mathbf{C}_0$, the WCRT $R^{\text{Typ}}_{\sigma_j}$ is implicitly defined by

$$R^{\text{Typ}}_{\sigma_j} = W^{\text{Typ}}_{\sigma_j} + \sum_{\sigma_\ell \in \mathbf{S} \setminus (\mathbf{G} \cup \{\sigma_j\})} \left\lceil \frac{R^{\text{Typ}}_{\sigma_j}}{P_{\sigma_\ell}} \right\rceil W^{\text{Typ}}_{\sigma_\ell}, \tag{7.19}$$

where $\mathbf{S} \setminus (\mathbf{G} \cup \{\sigma_j\})$ is the set of higher-priority tasks. This implicit definition of $R^{\text{Typ}}_{\sigma_j}$ on core $\mathbf{C}_0$ is only valid in range $0 < R^{\text{Typ}}_{\sigma_j} \leq D_{\sigma_j}$. If (7.19) cannot find a result, the WCRT $R^{\text{Typ}}_{\sigma_j}$ must be larger than the relative deadline D_{σ_j}.

The effective relative deadline $\overline{D}_{\sigma_j}$ of a sporadic task σ_j is defined according to

$$\overline{D}_{\sigma_j} = D_{\sigma_j} - R_{\sigma_j}^{\text{Typ}} \tag{7.20}$$

In algorithm EDPA*, all priorities for the typical-case execution parts are assigned before the priorities of the exceptional-case execution parts. The priorities of the exceptional-case execution parts $\pi_{\sigma_j}^{\mathbf{C}_1}$ are assigned according to the DM scheduling policy [55] by using the effective relative deadlines $\overline{D}_{\sigma_j}$.

The priorities for the typical-case execution parts $\pi_{\sigma_j}^{\mathbf{C}_0}$ on core $\mathbf{C}_0$ need to consider the impact of the lower priority task, because the priorities for core $\mathbf{C}_1$ are not known. Suppose a task $\sigma_\ell \in \mathbf{G}$, which has a lower priority than σ_j. This task σ_ℓ can influence the WCRT of the task, for which the priority needs to be decided. Thus, this impact can be represented by the temporary WCRT $r_{j,\ell}^*$ of σ_j with respect to the lower priority tasks $\sigma_\ell \in \mathbf{G}$.

Note that $r_{j,\ell}^*$ is differently defined as $r_{j,\ell}$ in the previous section. In contrast to $r_{j,\ell}$, $r_{j,\ell}^*$ already includes the impact of assigning the priority $\pi_{\sigma_j}^{\mathbf{C}_0} = k$ to task σ_j. In case $\overline{D}_{\sigma_\ell} < \overline{D}'_{\sigma_j}$, there is no impact to the temporary WCRT $r_{j,\ell}^*$. If $\overline{D}_{\sigma_\ell} \geq \overline{D}'_{\sigma_j}$, there exists a variable temp with $0 < \text{temp} \leq \overline{D}_{\sigma_\ell}$ such that

$$\text{temp} = W_{\sigma_\ell}^{\text{Exc}} + \left\lceil \frac{\text{temp}}{P_{\sigma_j}} \right\rceil W_{\sigma_j}^{\text{Exc}} + \sum_{\sigma_k \in \mathbf{G} \text{ with } \overline{D}_{\sigma_k} \leq \overline{D}_{\sigma_\ell}} \left\lceil \frac{\text{temp}}{P_{\sigma_k}} \right\rceil W_{\sigma_k}^{\text{Exc}}, \tag{7.21}$$

where the minimum temp $< D_{\sigma_j}$ is the temporary WCRT $r_{j,\ell}^*$. If (7.21) returns no result, temp does not exist, for which $r_{j,\ell}^*$ is defined by infinity. Formally, $r_{j,\ell}^*$ is defined as

$$r_{j,\ell}^* = \begin{cases} \min\{\text{temp} |\ (7.21) \text{ holds}\} & \text{if } \exists \text{temp} \leq \overline{D}_{\sigma_\ell} \text{ such that (7.21) holds,} \\ \infty & \text{otherwise.} \end{cases} \tag{7.22}$$

Based upon the temporary WCRT $r_{j,\ell}^*$, the two slack types slack$_{j,\ell}$ and slack$_{j,j}$ can be defined. The slack$_{j,\ell}$ represents the slack considering the lower priority tasks $\sigma_\ell \in \mathbf{G}$, which can influence the task σ_j, if $\overline{D}_{\sigma_\ell} \geq \overline{D}'_{\sigma_j}$, and is defined as

$$\text{slack}_{j,\ell} = \overline{D}_{\sigma_\ell} - r_{j,\ell}^*. \tag{7.23}$$

Similarly, the slack$_{j,j}$ is defined as

$$\text{slack}_{j,j} = \overline{D}'_{\sigma_j} - r_{j,j}^*, \tag{7.24}$$

where $r_{j,j}^*$ is the WCRT of σ_j with respect the the priority assignment of core $\mathbf{C}_1$.

Based upon this adaption, a heuristic algorithm can be used to determine the priority assignment. Algorithm 12 shows the heuristic algorithm EDPA*. This algorithm is similar to Algorithm 11, although it is able to handle sporadic tasks. Considering the above equations, the description of the algorithm is given in the previous section.

Regarding the time complexity of Algorithm12, it is dominated by the response time analysis and the slack calculation. The calculation of (7.21) to determine the temporary WCRT $r_{j,\ell}^*$ can be a complex function. Therefore, let $O(\gamma)$ be the time complexity of determining the temporary WCRT. As a result, the time complexity of Algorithm 12 is $O(|\mathbf{S}|^3\gamma)$, where $|\mathbf{S}|$ is the number of sporadic tasks.

Based upon the analysis to determine the priorities of all sporadic tasks $\sigma_k \in \mathbf{S}$, Theorem 8 concludes this result. Note that if the heuristic algorithm EDPA* returns a priority assignment, this assignment is feasible. If the algorithm returns no feasible solution, there might exist a feasible priority assignment.

Algorithm 12 EDPA*

```
Input: sporadic task set $\mathbf{S}$;
Output: priority ordering for $\mathbf{S}$ on $\mathbf{C}_0$ and $\mathbf{C}_1$;
 1: $\mathbf{G} \leftarrow \emptyset$;
 2: for $k = |\mathbf{S}| - 1, \cdots, 0$ stepped by 1 do
 3:     for each $\sigma_j$ in $\mathbf{S} \setminus \mathbf{G}$ do
 4:         $\overline{D}'_{\sigma_j} \leftarrow D_{\sigma_j} - R^{\text{Typ}}_{\sigma_j}$;
 5:         calculate $\text{slack}_{j,j}$ and $\text{slack}_{j,\ell}$ according to (7.24) and (7.23);
 6:         $\text{slack}_j \leftarrow \min \left\{ \text{slack}_{j,j}, \min_{\sigma_\ell \in \mathbf{G} \text{ with } \overline{D}_{\sigma_\ell} \geq \overline{D}'_{\sigma_j}} \{\text{slack}_{j,\ell}\} \right\}$;
 7:     if there exists $\sigma_j$ in $\mathbf{S} \setminus \mathbf{G}$ with $\text{slack}_j \geq 0$ then
 8:         choose task $\sigma_{j^*} \in \mathbf{S} \setminus \mathbf{G}$ with the max $\text{slack}_j$;
 9:         set $\pi^{\mathbf{C}_0}_{\sigma_{j^*}}$ to $k$; put $\sigma_{j^*}$ to $\mathbf{G}$; set $\overline{D}_{\sigma_{j^*}}$ to $\overline{D}'_{\sigma_{j^*}}$;
10:     else
11:         return "no feasible solution is found";
12: set $\pi^{\mathbf{C}_1}_{\sigma_k} \forall \sigma_k \in \mathbf{S}$ with the DM policy with effective relative deadlines $\overline{D}_{\sigma_k}$ (in case of
    the same effective relative deadline $\overline{D}_{\sigma_k} = \overline{D}_{\sigma_\ell}$, assign $\sigma_k$ with the higher priority
    $\pi^{\mathbf{C}_0}_{\sigma_k} < \pi^{\mathbf{C}_0}_{\sigma_\ell}$);
13: return the resulting priority orderings $\pi^{\mathbf{C}_0}_{\sigma_k}$, $\pi^{\mathbf{C}_1}_{\sigma_k}$;
```

Theorem 9. *For sporadic real-time tasks, if Algorithm EDPA* gives a priority level assignment for each task in* $\mathbf{S}$*, the resulting schedule can meet all of the timing constraints for the REETIC problem.*

Proof. This comes from Theorem 8. The sufficient condition for a feasible task set is checked in each iteration. The non-negative slack_{j^*} ensures that the timing constraints will be satisfied on both $\mathbf{C}_0$ and $\mathbf{C}_1$. □

7.5. Evaluations

This section presents experiments to determine the performance of the proposed priority assignment approaches. All experiments use our scheme to split the task into a typical- and an exceptional-case execution part.

Experimental setup

In order to evaluate the proposed approach, a sporadic task generator defines randomized task sets that are evaluated in different experiments. Based upon these randomized task sets, different approaches calculate the priority assignment. With this priority assignment, the feasibility of the task set is determined. The generator counts the number of feasible solutions and calculates a feasibility rate for different task set utilizations. For each task set utilization, 10^5 randomized task sets are evaluated.

In the following, this section describes the sporadic task set generator. Note that frame-based tasks are generated as a special case of sporadic tasks. First, the generator defines the utilization U_σ and the number of sporadic tasks $|\mathbf{S}|$. The utilization U_σ determines the utilization on each core as $U_{\mathbf{C}_0} = U_{\mathbf{C}_1} = U_\sigma$. As a default, there are $|\mathbf{S}| = 5$ sporadic tasks such that an exhaustive search is possible.

Second, the generator determines the period P_{σ_k} of each sporadic tasks σ_k. Note that the frame-based tasks have equal periods $P_{\sigma_0} = P_{\sigma_1} = \ldots = P_{\sigma_{|\mathbf{S}|-1}} = 10$. For sporadic

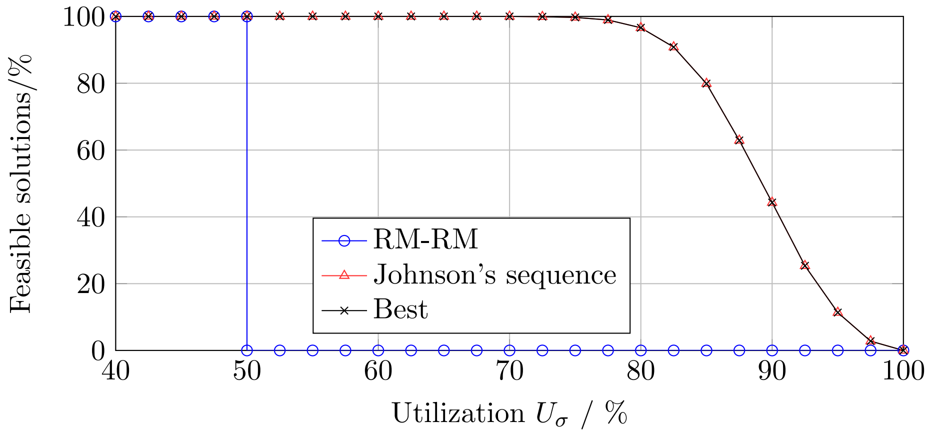


Figure 7.6.: The evaluation results for a frame-based task set with identical deadlines.

tasks, the period $P_{\sigma_0} = 10$ of the first task σ_0 is defined and the subsequent periods are defined by a randomized variable with

$$P_{\sigma_{k+1}} = P_{\sigma_k} \cdot w \qquad w \in [1, 2, 3] \subset \mathbb{Z}^+, \tag{7.25}$$

where w is randomized variable with a uniform distribution.

Third, the generator determines the typical- and exceptional-case execution times for each sporadic task σ_k. Let x_k, y_k and z_k be sets of random numbers with $0 < x_k < 1$, $0 < y_k < 1$, $0 < z_k < 1$ with a uniform distribution. The typical-case execution times are defined as

$$W^{\mathrm{Typ}}_{\sigma_k} = \frac{P_{\sigma_k} \cdot U_{\mathbf{C}_0} \cdot x_k}{\sum_{\sigma_j \in \mathbf{S}} x_j}. \tag{7.26}$$

and the exceptional-case execution times are defined as

$$W^{\mathrm{Exc}}_{\sigma_k} = \frac{P_{\sigma_k} \cdot U_{\mathbf{C}_1} \cdot y_k}{\sum_{\sigma_j \in \mathbf{S}} y_j}. \tag{7.27}$$

Fourth, for the constrained relative deadlines, the generator sets the deadlines according to

$$D_{\sigma_k} = P_{\sigma_k} \cdot z_k. \tag{7.28}$$

For the implicit deadlines, the relative deadlines equal their period $D_{\sigma_k} = P_{\sigma_k}$, $\forall \sigma_k$.

Note that the generator is able to generate task sets that are infeasible for any priority assignment. For thias reason, each experiments uses an approach called *Best* for a comparison with all the other approaches. *Best* represents an exhaustive search for all possible priority assignment to determine whether a feasible priority assignment exists. The calculation of *Best* requires factorial run-time complexity $O(|\mathbf{S}|!)$.

Frame-based tasks with implicit relative deadlines

This experiment uses frame-based tasks with implicit relative deadlines $P_{\sigma_0} = P_{\sigma_1} = \ldots = P_\sigma$, $D_{\sigma_k} = P_{\sigma_k}$, $\forall \sigma_k$. The proposed approach is to set the priorities on the first core $\mathbf{C}_0$ according to Johnson's sequence and the priorities on the second core inverse to Johnson's sequence. This approach is named *Johnsons's sequence*. As a comparison, *RM-RM* represents the approach where both task parts are assigned with the same priority.

Figure 7.6 shows the results for different approaches. If the utilization is larger than

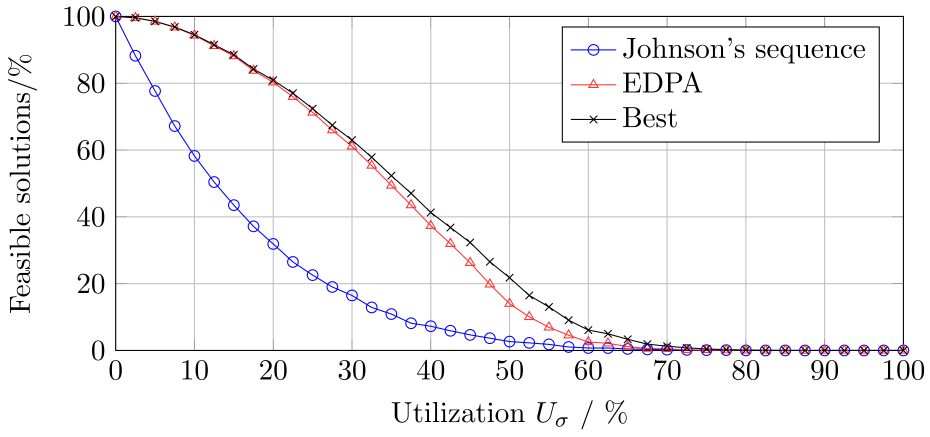


Figure 7.7.: The evaluation results for a frame-based task set with different deadlines.

$U_\sigma = U_{\mathbf{C}_0} = U_{\mathbf{C}_1} = 50\%$, *RM-RM* is unable to provide any feasible solution. At this utilization, a single-core platform would be fully utilized. Typical- and exceptional-case execution parts should have different priorities to improve the worst case, as shown in the worst-case response time analysis from Section 7.3. Furthermore, this experiment confirms that the proposed approach by exploiting Johnson's sequence is equal to the *Best*, because this approach is optimal for frame-based tasks with implicit relative deadlines.

Frame-based tasks with constraint relative deadlines

In this experiment, the generator determines sets of frame-based tasks with constraint deadlines $P_{\sigma_0} = P_{\sigma_1} = \ldots = P_\sigma$, $D_{\sigma_k} \leq P_{\sigma_k}$, $\forall \sigma_k$. For this setting, the proposed approach is algorithm EDPA. This algorithm is a heuristic to define the priority assignment based upon the maximum slack to consider tasks with a short relative deadline. Another approach to assign the priorities is the approach called *Johnsons's sequence*, as described in the previous experiment.

The results are visualized in Figure 7.7. Due to constrained deadlines, the generator determines only rare feasible settings with higher utilizations, owing to the high probability that one task cannot satisfy its deadline. The proposed approach is not optimal, although it performs significantly better than the *Johnsons's sequence* approach. The gap between the algorithm EDPA and the *Best* priority assignment is small.

Sporadic Tasks

This experiment evaluates sporadic tasks that can be used to model industrial applications. The proposed approach is to use the algorithm EDPA*, which is an adaptation of algorithm EDPA to the sporadic task model. As a comparison, two approach namely *RM-RM* and *RM-DM* are used to evaluate the proposed approach. *RM-RM* assigns priorities on each task part according to the Rate Monotonic (RM) scheduling policy. Thus, the typical- and exceptional-case execution part have the same priority. *RM-DM* assign the priorities of the typical-case execution parts with the Rate Monotonic (RM) scheduling policy. The priorities for the exceptional-case execution parts of each task σ_k are calculated with a Deadline Monotonic (DM) scheduling policy by using the proposed calculated effective relative deadlines $\overline{D}_{\sigma_k}$.

As shown in Figure 7.8, the algorithm EDPA* is more effective than other approaches. By applying the effective relative deadlines $\overline{D}_{\sigma_k}$ with *RM-DM*, the priority assignment on core $\mathbf{C}_1$ is significantly improved. Another interesting issue is that for frame-based tasks the utilization bound is $U_\sigma = U_{\mathbf{C}_0} = U_{\mathbf{C}_1} = 50\%$, at which *RM-RM* becomes infeasible.

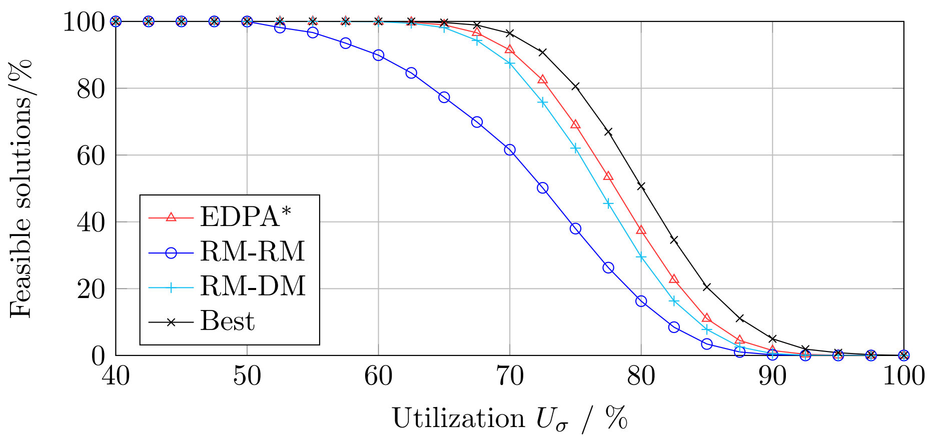


Figure 7.8.: The evaluation results for a periodic task set with implicit deadlines.

As shown in Figure 7.8, *RM-RM* is able to find feasible solution with $U_\sigma > 50\%$, because tasks with a larger period P_{σ_k} could sometimes handle a large WCRT R_{σ_k}.

8. Conclusion

This chapter summarizes this thesis and provides an outlook about further research directions.

8.1. Summary

This thesis studies scheduling problems for multicore and manycore platforms by considering industrial applications, which are summarized in the following.

On multicore or manycore platforms, the inter-core communications are critical to effectively schedule real-time tasks. Therefore, the proposed dependent task model is capable of modeling the communications and their impact to the scheduling analysis. This model comprises computational and communication tasks that have to be considered in the schedule. In real applications, there exist precedence relations among the tasks, which result in shorter end-to-end deadlines. These precedence relations are considered in the model to improve the applicability for industrial applications. By contrast, the independent task model would determine a more pessimistic worst-case execution time (WCET), because inter-core communications can cause a large delay in the worst case.

The proposed approach to handle the dependent task model is to use a Time-Triggered Constant Phase (TTCP) scheduler. This scheduling approach reserves a certain time window in advance for each computational and communication task such that the task can be executed or communicate exclusively. The reservation is designed without contention, i.e. the time windows are timely isolated such that no contention for any computational or communication task occurs. Based upon these *a priori* defined time windows, the scheduling analysis can be simplified to an overlap test between time intervals of these time windows. This thesis applies the TTCP scheduling approach to a multicore platform with a bus communication architecture. The TTCP scheduling approach exploits the periodicity of the reserved time windows such that these windows can be efficiently stored and analyzed. A heuristic algorithm to determine these time windows is presented, which can highly utilize the platform, while the feasibility of the schedule is proven to be correct. Thus, the *scheduling policy selection problem* is solved by proposing the TTCP scheduling approach. The scheduling analysis and the heuristic algorithm can be performed with pseudo polynomial time complexity. Experiments show that a core can be utilized over 90% by using industrial characteristics.

With more cores, the demand for more communication bandwidth is expected because tangled tasks are further distributed on different cores, which communicate more data with each other. The most promising approach is a scalable hardware architecture with a scalable communication fabric, which leads to a manycore platform with a NoC. The idea is to parallelize the communication with moderate hardware costs. Each inter-core communication which is modeled as a communication task is injected at a certain time in the network fabric. Only by defining the injection time can the entire path be reserved such that no communication task interferes with another one. This thesis provides the scheduling analysis of these injection times to ensure a contention-free communication schedule. In addition, different approaches are presented to determine injection times of communication tasks and time windows for the computational task such that all real-time constraints hold. The proposed iterative approach is able to find a feasible schedule by considering all tasks and their inter-core communication, which can highly utilize a manycore platform. In comparison to a single-core platform, this approach can find a feasible schedule for higher utilized dependent task sets. For typical industrial task sets with $1,000$ computational tasks and $3,000$ communication tasks, the proposed approach can utilize a 3×3 NoC by around 60%. Experiments highlight that communications among the tasks can be handled on a manycore platform by the TTCP scheduling approach, which does not limit parallel executions. Only specific communication tasks that define a precedence relation between tasks limit the parallelizability.

The TTCP scheduling approach can only handle periodic tasks, although more general sporadic tasks are usually present in the system. Sporadic tasks are more difficult to be scheduled because their interference in the communication needs to be considered for the worst case. The idea is to schedule periodic (time-triggered) and sporadic tasks separately to exploit the high platform utilization of periodic task and support sporadic tasks. Sporadic tasks are often represented by urgent requests that have to be handled with a short response time. Upon first glance, this may seem contradictory because the arrival times of sporadic tasks are *a priori* not known, although typically periodic and sporadic tasks are present in the system. The simple approach is to reserve a periodic time slot for a Time-Triggered Server (TTS). A TTS can handle and execute sporadic tasks in its assigned periodic time slots, which fit to the TTCP scheduling policy. To increase the responsiveness of sporadic tasks, the idea is to temporally shift time-triggered tasks by a certain amount of time. After a sporadic task is executed, the time reserved for the TTS can be used to reduce the delay of time-triggered tasks. This thesis provides the feasibility analysis of the proposed *slot-shifting* approach and a heuristic algorithm to determine the time reserved for the TTS. For typical industrial applications, the proposed approach can reduce the worst-case response times by 25% for the sporadic tasks.

In contrast to the dependent task model, inter-core communications can be modeled as part of the WCET [32] such that all tasks are modeled by independent sporadic tasks. In the worst case, all inter-core communications interfere with each other because all sporadic tasks may communicate at the same time. Thus, the WCET of each task can be increased in the worst case, although the execution time in the best case remains the same [87]. In addition, the WCET estimation introduces some uncertainty, which further increases the gap between the best case and the worst case of a sporadic independent task. This thesis proposes a scheme to exploit this gap by defining a typical-case execution time, which could be based upon measurements. Each sporadic task is split into two parts, which are scheduled on different cores, for the typical- and worst-case executions of a task. A typical execution of a task can be defined as a percentile (e.g. 90%, 95%) of the execution's time measurements. In an exceptional case when a job of a task exceeds its typical-case execution time, this job is migrated to another core. Thus, the other core is used as a backup to guarantee a safe worst-case execution behavior. Such a backup core is rarely activated to execute a task that saves energy rather than regularly activating two cores. A motivational example shows that a fixed-priority scheduler should not assign the same priority level to each part, because the worst-case response time would be too large. For different properties of the sporadic task, this thesis proposes the scheduling analysis and an efficient priority assignment algorithm. Experiments confirm the effectiveness of the proposed priority assignment methods.

8.2. Outlook

This section presents possible research directions for future investigations.

General support for the TTCP scheduling approach

The Time-Triggered Constant Phase (TTCP) approach is capable of highly utilizing a manycore platform, although currently neither manycore platforms nor operation systems with a TTCP scheduler exist. On the one hand, there exists a large design space for future hardware architectures with multiple cores. For example, cores can be heterogeneous with a special purpose or hardware accelerators. Another approach is to process the sequential dominating parts on a few powerful cores (e.g. 4 cores) and the parallel dominating parts on a grid of many cores (e.g. 64 cores). For hard real-time applications, architecture-dependent and hardware-dependent features need to be explored or considered for the proposed TTCP scheduling approach. Especially the NoC could provide hardware support for the proposed TTCP scheduling approach, because the messages have to be send at a pre-defined time, which could be achieved by a memory controller.

Nowadays, industrial applications are designed according to standards like AUTOSAR [5] or OSEK [73], which also defines a scheduling policy. For an easy integration of the TTCP scheduling approach into the industrial design processes, the TTCP scheduling approach can be adapted to be compatible with such standards. Another issue is the adaptation from existing scheduling policies to the TTCP. For example, suppose a software project with legacy parts that want to use the TTCP scheduler but cannot start from scratch. It has to be proven that a step-wise change to the TTCP scheduler is possible; otherwise, this project is unable to adapt this approach. The problem is that scheduling has a huge impact on the software.

Task model extensions

The dependent task model is closer to industrial applications than the independent sporadic task model because tasks exchange data among each other. Additional feature can be added to the dependent task model to allow more accurate system description. For example, a memory model could be added, which describes the location of the program code and temporary data. Thus, task migration can be described and analyzed in further detail because the program code may need to transmitted among the cores. An often-used approach is to define a function (service), which is often called to save memory for storing the program code. An extended task model can investigate different strategies to implement such functions. In the current model, such functions would be present on all cores to be accessible by any task.

Another feature is the expression of read commands, which are also often used in industrial applications. In the model, each task sends the data required by another task at its end. In some cases, a task needs to request the data, called reading. In addition, a case study with a real application can highlight another problem, namely which features have to be added to improve the task model and develop the corresponding scheduling analysis.

Handling non-synchronized time domains by the TTCP approach

The assumption of time-synchronized cores is difficult to achieve, because the clock signal jitters such that different clocks have to be synchronized. One possibility is to add overhead parameters in the task model to consider clock synchronization. By adding overhead for core synchronization, all cores could be run with the same time base. The TTCP scheduling approach could be extended to further support a clock synchronization overhead.

Another possibility is to define different time domains (clocks), which mostly run independently. Thus, each time domain could be scheduled by a TTCP scheduler. The problem is defining a proper way to exchange data among these time domains such that the tasks are not disrupted by the incoming data from another time domain. These data communications require other communication arbitration policies like Time Division Multiple Access (TDMA) or Fixed-Priority (FP), which need to be adapted and analyzed.

A. SMT Problem Formulations

A.1. SMT Problem Formulations for TTCP Schedule on a Single-Core

Algorithm 13 Formulating computational phase assignment into an SMT problem

```
Input: computational task set $\mathbf{T}$;
Output: SMT problem;
 1: job_number ← 0;
 2: for $i = 0, \cdots, |\mathbf{T}| - 1$ stepped by 1 do
 3:     DEFINE t(job_number);
 4:     ASSERT t(job_number)$\geq \Phi_{\tau_i,\min}$;
 5:     ASSERT t(job_number)$\leq D_{\tau_i}$;
 6:     job_number ← job_number + 1;
 7:     for $j = 1, \cdots \frac{H}{P_{\tau_i}}$ stepped by 1 do
 8:         DEFINE t(job_number);
 9:         ASSERT t(job_number) = t(job_number-j)·j·$P_{\tau_i}$;
10:         job_number ← job_number + 1;
11: for $i = 0, \cdots$ ,job_number stepped by 1 do
12:     for $j = 0, \cdots$ ,job_number stepped by 1 ($j \neq i$) do
13:         ASSERT (t(i) ≥ t(j) + $W_{\tau_j}$) or (t(j) ≥ t(i) + $W_{\tau_i}$);
```

A.2. SMT Problem Formulations for the Dependent System Model

Algorithm 14 SMT formulating a TTCP scheduling problem of the dependent task model including communication and computational tasks

```
Input: computational task set $\mathbf{T}$, communication task set $\mathbf{K}$;
Output: SMT problem;
 1: for each $\tau_i \in \mathbf{T}$ do
 2:   DEFINE t(i,0);
 3:   ASSERT t(i,0)$\geq$ 0;
 4:   for each predecessor $\tau_p$ of $\tau_i$ do
 5:     if $c_{\tau_i} = c_{\tau_p}$ then
 6:       ASSERT t(i,0)$\geq$t(p,0);
 7:     else
 8:       for all $\kappa_\ell$ with $\kappa_{\mathrm{DST}_\ell} = \tau_i$ do
 9:         ASSERT t(i,0)$\geq$c($\ell$)+$\overline{W}_{\kappa_\ell}$;
10:   ASSERT t(i,0)$< D_{\tau_i} - W_{\tau_i}$;
11:   for each job $J_{\tau_i,k}$, $k = 1, \cdots \frac{H}{P_{\tau_i}}$ stepped by 1 do
12:     DEFINE t(i,k);
13:     ASSERT t(i,k) = t(i,0)$\cdot$k$\cdot P_{\tau_i}$;
―――――――――――――――― packet limits ――――――――――――――――
14: for each $\kappa_j \in \mathbf{K}$ do
15:   DEFINE c(j,0)
16:   ASSERT c(j,0)$\geq$t(SRC$_j$,0);
17:   if $\kappa_j$ precedence then
18:     ASSERT c(j,0)$\leq$t(DST$_j$,0)$-\overline{W}_{\kappa_j}$;
19:   else
20:     if $P_{\tau_{\mathrm{SRC}_j}} \geq P_{\tau_{\mathrm{DST}_j}}$ then
21:       ASSERT c(j,0)$\geq$t(DST$_j$,0)+$P_{\tau_{\mathrm{SRC}_j}} - \overline{W}_{\kappa_j}$;
22:     else
23:       ASSERT (c(j,0)$\leq$ t(DST$_j$,0) $-\overline{W}_{\kappa_j}$and t(DST$_j$,0) $\geq P_{\tau_{\mathrm{SRC}_j}}$ )
                or (c(j,0)$\leq$ t(DST$_j$,0)+$P_{\tau_{\mathrm{SRC}_j}} - \overline{W}_{\kappa_j}$ and t(DST$_j$,0) $< P_{\tau_{\mathrm{SRC}_j}}$);
24:     ASSERT c(j,0)$\geq$t(SRC$_j$,0);
25:   for each $l$-th packet of $\kappa_j$, $l = 1, \cdots \frac{H}{P_{\kappa_j}}$ stepped by 1 do
26:     DEFINE c(j,l);
27:     ASSERT c(j,l) = c(j,0)$\cdot$l$\cdot P_{\kappa_j}$;
―――――――――――――――― time-overlap test ――――――――――――――――
28: for $i = 1, \cdots, |t|$ stepped by 1 do
29:   for $k = 1, \cdots, |t|$ stepped by 1 $(k \neq i)$ do
30:     ASSERT (t(i) $\geq$ t(k) + $W_{\tau_k}$) or (t(k) $\geq$ t(i) + $W_{\tau_i}$);
31: for $j = 1, \cdots, |c|$ stepped by 1 do
32:   for $l = 1, \cdots, |c|$ stepped by 1 $(l \neq j)$ do
33:     ASSERT (c(j) $\geq$ c(l) + $W_{\kappa_l}$) or (c(l) $\geq$ c(j) + $W_{\kappa_j}$);
```

Bibliography

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, Dec. 1974.

[2] Adapteva. Epiphany architecture reference. Technical report, 2015. [Online] http://www.adapteva.com/.

[3] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Embedded Worls*, 2004.

[4] S. Anily, J. Bramel, and D. Simchi-levi. Worst-case analysis of heuristics for the bin packing problem with general cost structures. *Oper. Res.*, 42(2):287–298, Apr. 1994.

[5] AUTOSAR. Specification of operating system autosar release 4.2.1. Technical report, 2015. [Online] http://www.autosar.org/.

[6] S. Baruah, M. Bertogna, and G. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer International Publishing, 2015.

[7] M. Bertogna, G. Buttazzo, and G. Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 251–260, Nov 2011.

[8] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 209–218, Washington, DC, USA, 2005. IEEE Computer Society.

[9] R. Bettati and J. W.-S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *ICDCS*, pages 452–459, 1992.

[10] A. Biewer, J. Gladigau, and C. Haubelt. A novel model for system-level decision making with combined ASP and SMT solving. In *DATE*, 2014.

[11] A. Biewer, J. Gladigau, and C. Haubelt. Towards tight interaction of asp and smt solving for system-level decision making. In *Architecture of Computing Systems (ARCS)*, pages 1–7, Feb 2014.

[12] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1, June 2006.

[13] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 447–456, Washington, DC, USA, 2009. IEEE Computer Society.

[14] V. Bonifaci, H. Chan, A. Marchetti-Spaccamela, and N. Megow. Algorithms and complexity for periodic real-time scheduling. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1350–1359, 2010.

[15] J. Boyar, L. Epstein, and A. Levin. Tight results for next fit and worst fit with resource augmentation. *Theoretical Computer Science*, 411(26-28):2572 – 2580, 2010.

[16] I. Bronshtein, K. Semendyayev, G. Musiol, and H. Mühlig. *Handbook of Mathematics.* Springer Berlin Heidelberg, 2007.

[17] G. Buttazzo. *Hard Real-Time Computing Systems.* Springer Verlag, 2011.

[18] D. Buttle. Real-time in the prime-time. Keynote on ECRTS, 2012.

[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[20] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE'03*, pages 190–195, 2003.

[21] V. Claesson and N. Suri. TTET: event-triggered channels on a time-triggered base. In *ICECCS'04*, pages 39–46, 2004.

[22] S. S. Craciunas and R. S. Oliver. Smt-based task-and network-level static schedule generation for time-triggered networked systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 45. ACM, 2014.

[23] D. Dasari, B. Nikoli'c, V. N'elis, and S. M. Petters. Noc contention analysis using a branch-and-prune algorithm. *ACM Trans. Embed. Comput. Syst.*, 13(3s):113:1–113:26, Mar. 2014.

[24] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43:44, Oct 2011.

[25] B. de Dinechin. Dataflow language compilation for a single chip massively parallel processor. In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–1, Sept 2013.

[26] G. De Micheli and L. Benini. *Networks on Chips: Technology and Tools.* Elsevier Science, 2006.

[27] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[28] S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 215–224, Dec 2001.

[29] C. Ferdinand. AbsInt Angewandte Informatik GmbH. aiT: worst-case execution time analyzers. *http://www.absint.com/ait*, 2012.

[30] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 152–161, Dec 1995.

[31] G. Fohler. Predictably flexible real-time scheduling. In *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years).*, pages 207–221, 2012.

[32] M. Freier and J.-J. Chen. Prioritization for real-time embedded systems on dual-core platforms by exploiting the typical- and worst-case execution times. In *8th IEEE International Symposium on Industrial Embedded Systems*, pages 21–29, June 2013.

[33] M. Freier and J.-J. Chen. Time triggered scheduling analysis for real-time applications on multicore platforms. In *RTSS workshop on REACTION*, pages 48–53, 2014.

[34] M. Freier and J.-J. Chen. Time-triggered communication scheduling analysis for real-time multicore systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2015.

[35] M. Freier and J.-J. Chen. Sporadic task handling in time-triggered systems. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 135–144, New York, NY, USA, 2016. ACM.

[36] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[37] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, Sept. 2003.

[38] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE D & T*, 2005.

[39] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[40] N. Holsti. Tidorum Ltd. Bound-T time and stack analyzer. *http://www.bound-t.com*, 2013.

[41] S. Hong, T. Chantem, and X. S. Hu. Meeting end-to-end deadlines through distributed local deadline assignments. In *RTSS*, pages 183–192, 2011.

[42] ISO 11898: Road vehicles – Controller area network (CAN), 2015.

[43] D. Isovic and G. Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems*, 43(3):296–325, 2009.

[44] S. Johnson. Optimal two- and three stage-production schedules with setup time included. *Naval Research Logistics Quarterly*, 1:61 – 68, 1954.

[45] O. Kermia and Y. Sorel. A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In *ISCA PDCS*, 2007.

[46] A. E. Kiasari, A. Jantsch, and Z. Lu. Mathematical formalisms for performance evaluation of networks-on-chip. *ACM Comput. Surv.*, 45(3):38:1–38:41, July 2013.

[47] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer Science + Business Media,, 2011.

[48] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[49] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[50] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 239–248, Washington, DC, USA, 2009. IEEE Computer Society.

[51] W. Lawrenz and N. Obermöller. *CAN: Controller Area Network: Grundlagen, Design, Anwendungen, Testtechnik.* Vde Verlag, 2011.

[52] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer-Verlag, Berlin, Heidelberg, 2001.

[53] I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems.* Chapman & Hall/CRC, 1st edition, 2007.

[54] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[55] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982.

[56] W. Levine. *The Control Handbook.* Electrical Engineering Handbook. Taylor & Francis, 1996.

[57] K. Li and S.-C. Zhang. Heuristics for uniform parallel machine scheduling problem with minimizing makespan. In *Automation and Logistics, 2008. ICAL 2008. IEEE International Conference on*, pages 273–278, Sept 2008.

[58] X. Li, Y. Liang, T. Mitra, and A. Roychoudury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007. http://www.comp.nus.edu.sg/˜rpembed/chronos.

[59] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[60] Y. Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean. A statistical response-time analysis of real-time embedded systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 351–362, Dec 2012.

[61] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *ASP-DAC'12*, pages 665–670, 2012.

[62] H. Mackamul. Amalthea - an open tool platform for embedded multicore systems. Talk on EclipseCon Europe 2013, 2013.

[63] M. Marouf, L. George, and Y. Sorel. Schedulability analysis for a combination of non-preemptive strict periodic tasks and preemptive sporadic tasks. In *ETFA'12*, pages 1–8, Sept 2012.

[64] M. Marouf and Y. Sorel. Schedulability conditions for non-preemptive hard real-time tasks with strict period. In *RTNS'10*, 2010.

[65] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *23rd International Conference on Real-Time Networks and Systems (RTNS)*, 2015.

[66] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The nostrum backbone-a communication protocol stack for networks on chip. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 693–696, 2004.

[67] I. Miro Panades, A. Greiner, and A. Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the gals approach. In *NanoNet*, 2006.

[68] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[69] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 2004.

[70] P. Munk, M. Freier, J. Richling, and J.-J. Chen. Dynamic guaranteed service communication on best-effort networks-on-chip. In *PDP'15*, 2015.

[71] P. Munk, B. Saballus, J. Richling, and H.-U. Heiss. Position paper: Real-time task migration on many-core processors. In *Architecture of Computing Systems. Proceedings, ARCS 2015 - The 28th International Conference on*, pages 1–4, March 2015.

[72] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. Multi-source and multicore automotive ecus - os protection mechanisms and scheduling. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pages 3734–3741, July 2010.

[73] OSEK group. Osek/vdx operating system. Technical report, 2005. [Online] http://www.osek-vdx.org/.

[74] C. Paukovits and H. Kopetz. Concepts of switching in the time-triggered network-on-chip. In *RTCSA*, 2008.

[75] M. Pinedo. *Scheduling : Theory, Algorithms, and Systems*. Springer New York, third edition edition, 2008.

[76] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. In *ECRTS'06*, 2006.

[77] C. N. Potts and V. A. Strusevich. Fifty years of scheduling: a survey of milestones. *JORS*, 60(S1), 2009.

[78] Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 44–53, May 2009.

[79] S. Quinton, M. Hanke, and R. Ernst. Formal analysis of sporadic overload in real-time systems. In *DATE*, pages 515–520, 2012.

[80] Rapita System Ltd. RapiTime Explained. *http://www.rapitasystems.com*, 2015.

[81] K. Reif. *Gasoline Engine Management: Systems and Components*. Bosch Professional Automotive Information. Springer Fachmedien Wiesbaden, 2014.

[82] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Stähle, S. Chakraborty, and A. Knoll. Schedule integration framework for time-triggered automotive architectures. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 20:1–20:6, New York, NY, USA, 2014. ACM.

[83] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 2000.

[84] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *DAC '10*, pages 332–337, NY, USA, 2010. ACM.

[85] P. Senthilkumar and S. Narayanan. Literature review of single machine scheduling problem with uniform parallel machines. *Intelligent Information Management*, 2(8):457–474, 2010.

[86] L. Sha. Real-time virtual machines for avionics software porting and development. In J. Chen and S. Hong, editors, *RTCSA*, volume 2968 of *Lecture Notes in Computer Science*, pages 123–135. Springer, 2003.

[87] H. Shah, A. Raabe, and A. Knoll. Challenges of wcet analysis in cots multi-core due to different levels of abstraction. In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013.

[88] D. Shang, E. Eyisi, Z. Zhang, X. Koutsoukos, J. Porter, G. Karsai, and J. Sztipanovits. A case study on the model-based design and integration of automotive cyber-physical systems. In *Control Automation (MED), 2013 21st Mediterranean Conference on*, pages 483–492, June 2013.

[89] P. B. Sousa. The carousel-edf scheduling algorithm for multiprocessor systems. In *RTCSA*, 2013.

[90] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 71–80, April 2011.

[91] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[92] The European Parliament and the Council of the European Union. Regulation (ec) no 715/2007 of the european parliament and of the council. Technical report, 2007. [Online] http://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32007R0715.

[93] M. van den Heuvel, R. Bril, X. Zhang, S. Md Jakaria Abdullah, and D. Isovic. Limited preemptive scheduling of mixed time-triggered and event-triggered tasks. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–9, Sept 2013.

[94] E. Wandeler and L. Thiele. Optimal tdma time slot and cycle length allocation for hard real-time systems. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6 pp.–, Jan 2006.

[95] 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. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[96] S. Wilhelm. *Symbolic Representations in WCET Analysis.* PhD thesis, Saarland University, 2012.

[97] H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2010.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Ludwigsburg, 1st of September 2016

. . . .
(**Matthias Freier**)

List of Publications

These are the publications of the author, which relate to this thesis.

1. Matthias Freier and Jian-Jia Chen. *Prioritization for Real-Time Embedded Systems on Dual-Core Platforms by Exploiting the Typical- and Worst-Case Execution Times.* In IEEE Symposium on Industrial Embedded Systems (SIES), Porto, Portugal, pages 21–29, June 2013.

2. Matthias Freier and Jian-Jia Chen. *Time triggered scheduling analysis for real-time applications on multicore platforms.* In RTSS workshop on REACTION, pages 48–53, 2014.

3. Peter Munk, Matthias Freier, Jan Richling and Jian-Jia Chen. *Dynamic Guaranteed Service Communication on Best-Effort Networks-on-Chip.* In 23rd Euromicro International Conference on Parallel, Distributed And Network-based Processing (PDP), pages 353–360, March 2015.

4. Matthias Freier and Jian-Jia Chen. *Time-triggered communication scheduling analysis for real-time multicore systems.* In 10th IEEE International Symposium on Industrial EmbeddedSystems (SIES), 2015.

5. Matthias Freier and Jian-Jia Chen. *Sporadic Task Handling in Time-Triggered Systems.* In Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES), pages 135–144, 2016.