Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science with Specialization in Embedded Systems 15.0 credits

OPTIMIZING INTER-CORE DATA-PROPAGATION DELAYS IN MULTI-CORE EMBEDDED SYSTEMS

Emir Hasanović
ehc18001@student.mdh.se
Hasan Grošić
hgc18001@student.mdh.se

Examiner: Thomas Nolte
Mälardalen University, Västerås, Sweden

Supervisors: Saad Mubeen
Mälardalen University, Västerås, Sweden

July 2, 2019

Abstract

The demand for computing power and performance in real-time embedded systems is continuously increasing, since new customer requirements and more advanced features are appearing every day. To support these functionalities and handle them in a more efficient way, multi-core computing platforms are introduced. These platforms allow for a parallel execution of tasks on multiple cores, which in addition to its benefits to the systems performance, introduces a major problem regarding the timing predictability of the system. That problem is reflected in unpredictable inter-core interferences, which occur due to shared resources among the cores, such as the system bus. This thesis investigates the application of different optimization techniques for the offline scheduling of tasks on the individual cores, together with a global scheduling policy for the access to the shared bus. The main effort of this thesis focuses on optimizing the inter-core data propagation delays which can provide a new way for creating optimized schedules. For that purpose, Constraint Programming optimization techniques are employed and a Phased Execution Model of the tasks is assumed. Also, in order to enforce end-to-end timing constraints that are imposed on the system, job-level dependencies are generated prior, and subsequently applied during the scheduling procedure. Finally, an experiment with a large number of test cases is conducted to evaluate the performance of the implemented scheduling approach. The obtained results show that the method is applicable for a wide spectrum of abstract systems with variable requirements, but also open for further improvement in several aspects.

Table of Contents

List of Figures

List of Tables

Glossary

Term	Description
Constraint Programming	An optimization technique which solves optimization problems where multiple constraints between variables are present.
Data age constraint	The maximum allowed latency of data in a chain.
End-to-end Delay	The time needed for data to propagate through a chain.
Hyperperiod	The period of the whole application, defined as the least common multiple of all the periods in the application's task set.
Job-Level Dependency	A precedence constraint between a particular instance of one task and a particular instance of another task.
Scheduling	The process of arranging all the instances of the tasks in a task set so that all timing and precedence constraints are met.
Task	A unit of execution consisting of a set of instructions which achieve a particular result.
Task Chain	An arranged sequence of tasks that define the propagation of data.
Task Instance/Job	A particular execution of a task.
Task Set	A collection of all the tasks defined in an application.

Acronyms

Term	Meaning
CP	Constraint Programming
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DMA	Direct Memory Access
FCFS	First-Come-First-Serve
GCD	Greatest Common Divisor
IDE	Integrated Development Environment
ILP	Integer Linear Programming
I/O	Input/Output
IP	Intellectual Property
LCM	Least Common Multiple
RTA	Response-Time Analysis
RTOS	Real-Time Operating System
TDMA	Time Division Multiple Access
WCET	Worst Case Execution Time
XML	eXtensible Markup Language

1 Introduction

Nowadays, the demand for computing power and performance in embedded systems is rapidly increasing, since more advanced features and new customer requirements are appearing every day. To support these functionalities and handle them in a more efficient way, multi-core computing platforms are introduced into the design and implementation of such systems. In practice, many of these systems implement some form of real-time control. It is known that real-time systems do not need to be necessarily fast, but their behaviour needs to be predictable, i.e., it is important that not only the task deadlines, but also all other timing constraints are satisfied in the worst-case scenario [2].

Multi-core computing platforms allow for a parallel execution of tasks on multiple cores, which in addition to its benefits to the system's performance, introduces a major problem regarding the timing predictability of the system. That problem is reflected in unpredictable inter-core interferences, which occur due to shared resources among the cores, e.g., system bus, main memory, caches and I/O modules. As a result of the inter-core interference, the inter-core data-propagation delays can be very large and thus endanger the time predictability of the system's task scheduling. Scheduling, in this case, does not only refer to finding a feasible task schedule of a task set, but it also refers to determining the optimal schedule among all feasible ones.

There are different ways to fulfill all the timing requirements, e.g., introducing a time-based channel access protocol for the system bus [3], like the *Time-Division Multiple Access* (TDMA) protocol or introducing a *Phased Execution Model* for the tasks [4], which ensures non-conflicting memory access requests. However, most of these solutions still heavily rely on various sorts of optimization, mostly regarding the scheduling of the tasks on the individual cores and the scheduling of the data transmission on the system bus. While these solutions lead to more predictable accesses to shared resources and consequently simpler analysis for timing verification, the main trade-off is that they do not fully take advantage of the system's computing resources (i.e. compromising throughput and average case performance) [5]. It remains an open question, whether these techniques and their analyses can be improved to facilitate a higher degree of utilisation of the available hardware resources.

Since this field of research is still active and open for new investigations, the main purpose of this thesis is to explore different approaches to the optimization of inter-core data propagation delays which can provide a new way for creating optimized schedules.

1.1 Problem Formulation

Multi-core computing platforms consist of several processor cores located on a single chip. The chips also contain some resources that are shared among the cores, such as the on-chip shared memory and shared system bus. These resources can be the cause of additional delays because they are shared by the tasks that are simultaneously running on different cores. The inter-core interference not only affects the response times of individual tasks on the cores, but also impacts significantly the data-propagation delays in the task chains that are distributed over multiple cores. Note that this thesis focuses on multi-core architectures with one shared system bus, private caches for each core and no shared caches among the cores. Offline scheduling of tasks inside the cores and a TDMA protocol to schedule the system bus is one solution to develop predictable embedded systems on multi-core platforms [5]. However, the inter-core data-propagation delays can still be very high if the offline schedules inside the cores and the TDMA schedule on the bus are not optimized. Another way to achieve temporal isolation and facilitate better control of the bus contention is to utilize a *phased task execution* model together with the implementation of a global offline scheduling approach. The term *global scheduling*, in this case, does not refer to the scheduling process where the tasks can be assigned to any core in a system, but it rather means that task-to-core mapping is given beforehand and the activation times and execution times of these tasks need to be exactly determined and known before the system's run-time. In this context, this thesis will seek to answer the following research question:

RQ1: *How can the inter-core data-propagation delays be optimized in multi-core real-time systems that are scheduled using offline global scheduling?*

- **RQ1.1:** *What are the results and achievements of an offline global scheduling approach regarding the schedulability when end-to-end inter-core data propagation constraints are considered?*
- **RQ1.2:** *How does an offline global scheduling approach scale in terms of computational time with regard to the number of task chains defined in the system?*

1.2 Initial Assumptions

With the intention of reducing the complexity of the problem that is analyzed throughout this thesis, some initial assumptions have to be made. These assumptions help narrow down and confine the scope of the research to dimensions that are appropriate for the time period and resources allocated for it.

The main class of systems that are targeted within this research are *hard real-time* systems. By addressing the strict conditions and timing requirements of these systems from the start on, it becomes easier to generalize the findings of the research to systems where the requirements are more loosely defined.

The system is assumed to be scheduled with global scheduling and the task-to-core mapping is assumed to be given beforehand. Otherwise, if this aspect is included in the problem analysis, it substantially increases the problem's complexity. Therefore, throughout this thesis, it is assumed that the mapping of the system's tasks to the individual cores is fixed and given *a priori*. This also implies that inter-core task migrations are not allowed, meaning that tasks, and thus all of their instances, can only execute on the cores they are assigned to.

Processors with various multi-core architectures and different memory hierarchies and shared resources exist and are available commercially. If processor architectures express strong inter-core dependencies caused by the different shared resources present on the chip, the analysis and scheduling of tasks on such platforms becomes much more complicated. Within this thesis, it is assumed that only multi-core processors with a shared system bus, private caches and no shared caches are analyzed, as shown in Figure 1. Thus, in accordance with this assumption it can be concluded that the only possible source of contention in the system is the system bus (main memory).

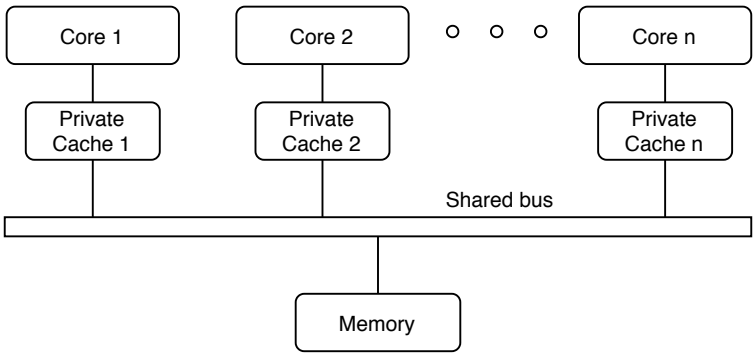

Figure 1: Processor architecture that contains multiple cores with private caches and a shared system bus.

Furthermore, to facilitate task execution on the individual cores, global non-preemptive scheduling is assumed. Only tasks with periodic activation are considered. To limit the interference of co-running tasks that needs to be considered in the analysis, a phased execution model for the tasks is introduced. That means that task execution is divided into three phases: two memory phases (*Read* and *Write*) and an *Execution* phase. This allows us to schedule the tasks so that there are no conflicting memory access requests, and since offline scheduling is employed, all delays that are present in the system are predictable and known beforehand.

Lastly, since the cores are located on the same physical chip and share the same clock, the assumption of total synchronization between them can be made.

1.3 Thesis Outline

The report consists of ten sections. After the introductory section, Section 2 provides insight into theoretical and practical background relevant for the understanding of the topic of this thesis. Further, in Section 3, a brief review of related works is presented and discussed. Section 4 introduces the research methodology followed in this thesis, together with its concrete application throughout the research process. In Section 5, the technical approach that includes a description of all parts of the project's processes and used programs, is presented and followed by the technical description which describes the model of the system in Section 6. Also, in Section 7, the limitations of the approach and the proposed method are highlighted and discussed. After that, the process of testing the devised method is presented in Section 8. The obtained results together with the method's performance and difficulties of its operation are commented. In the last two sections 9, and 10, certain conclusions are drawn and some directions for possible improvement through future work discussed.

2 Background

Since the subject of this thesis covers and connects many different fields, the main features, and their purposes need to be elucidated. Therefore, in the following subsections, the main terms and concepts that are mentioned and used during this thesis, are explained.

2.1 Embedded Systems

An embedded system represents a computing device composed of two basic components: hardware and software, which are usually designed for a certain purpose. Large variations exist in the hardware designed for embedded systems. And unlike the hardware for personal computers, the structure of embedded systems depends on the usage and the purpose of the specific system. However, every embedded system hardware consists of several components that cannot be omitted: a processor (single-core or multi-core), memory (usually divided into Read-Only Memory and Random Access Memory), communication interfaces, and input/output devices (Figure 2) [6][7]. Software for embedded systems also depends on the application of the system, but in many cases, it is executed using a real-time operating system (RTOS) which controls multiple tasks with defined sets of instructions. It can be programmed and adjusted depending on operations that need to be performed and constraints that need to be considered.

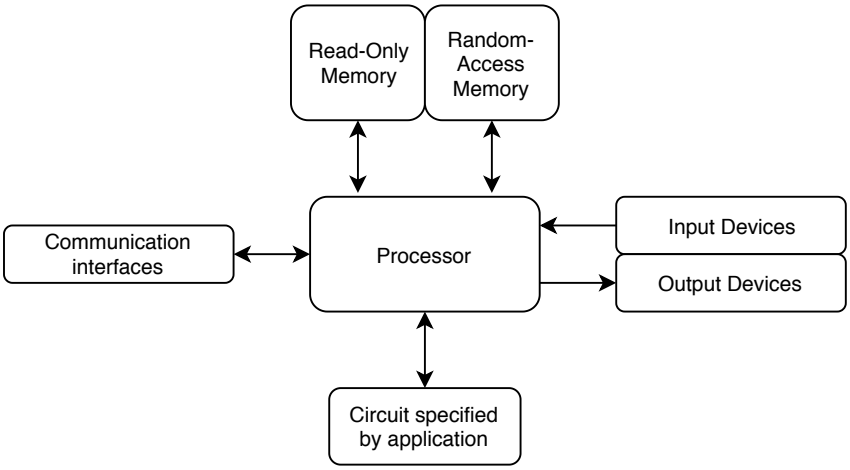

Figure 2: Main structure of embedded systems hardware.

Embedded systems have a widespread applicability. In the modern world, embedded systems find everyday use in different consumer electronics and household appliances (mobile phones, cameras, printers, scanners, microwaves, dishwashers and etc.). Besides that, embedded systems can be found in telecommunications, medical equipment, transportation systems, satellite systems, military equipment and in many other fields. All this explains why 99% of all produced processors are made just for embedded systems [6].

2.2 Single-core and Multi-core Processors

One of the main parts of any embedded system is the processor. A processor is also called a *central processing unit* (CPU) and it corresponds to a logic circuit which executes a set of basic instructions and performs certain calculations in order to manage processes on a computer (or embedded system). A processor has four main functions:

- Fetch – based on the program counter which stores the number of the next instruction, the instruction that needs to be performed is determined.
- Decode – the instruction is interpreted and converted into signals by the instruction decoder.
- Execute – based on the decoded instruction, a sequence of actions is performed.

- Write-through or Write-back – data is written to cache and to main memory (Write-through), or, data is written only to cache and writing to main memory is postponed until a specific condition is fulfilled (Write-back).

The simplest type of a processor is a single-core processor. This kind of processor has one core on a chip which allows only one task at a moment to be run. Therefore, single-core processors require single-threaded code to execute which at one moment, became the bottleneck in embedded systems development. Since the demand for better performance increases every day, single-threaded code could not execute any faster on new single-core processors. The reason for this lies in a fact that it is not possible to achieve more advancement in terms of single-core processors performance by increasing clock rates or by introducing instruction-level parallelism, since every instruction that is being executed requires a certain time for previously mentioned processor functions.

Because of that, many processors nowadays are multi-core processors. A multi-core processor represents a single component composed of two or more cores where each core has similar characteristics as a single-core processor. In this way, it enables the execution of more than one (depends on the number of cores) thread/task at a time. The parallelization of execution of tasks significantly speeds up the execution of a program, allows the implementation of a higher number of tasks and eases the process of satisfying all timing constraints, especially considering real-time systems. Multi-core systems also have some deficiencies, like the appearance of inter-core dependencies and unpredictable data propagation delays which are the main interest of this thesis. Regarding the structure, especially the type of cache memory, there are various multi-core systems. One possible structure, that is taken into consideration within the frame of this thesis, is described in Subsection 1.2 and presented in Figure 1.

2.3 Real-time Systems

A real-time embedded system is a type of system which does not only need to correctly perform its functionality, but it also needs to satisfy certain timing constraints during its run-time. More precisely, "a real-time computer system may be defined as one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time" [8]. This definition may lead someone to the wrong conclusion that real-time systems need to be quick, but in reality, real-time systems need to be predictable and to react to a certain event within a certain time frame. To achieve predictability of a real-time system, it is important to find a feasible schedule for a given set of system tasks. Finding a feasible schedule refers to determining an order of execution of tasks which fulfills all timing constraints. So, the main role in real-time systems is placed upon tasks, which are described by three main parameters: period, execution time and deadline. The deadlines of the tasks represent one set of timing constraints that need to be satisfied, but besides that, there can be several other types of timing requirements.

2.4 Tasks

A task represents an independent process or thread which is controlled by an operating system and which executes a pre-assigned set of instructions. The control of a task set in an operating system is performed in a way where the kernel grants permissions to each task job to run and allowance to occupy a certain CPU core. During its run-time, a task executes an exactly defined instruction set represented by executable codes in order to bring the system into a desired state or to perform the specified actions. Since each execution of a task job requires a certain amount of time to perform its instructions, one of the most important parameters of a task, especially in real-time system applications, is its execution time. In the great majority of the implementations, the execution time that is considered is the Worst-Case Execution Time (WCET) which represents the longest possible time that a task needs to execute under certain conditions. Besides execution time, as it is mentioned in Subsection 2.3, for any real-time system it is of core importance to satisfy all timing constraints and deadlines. Furthermore, the deadline is another very important parameter of a task and it represents the time measured from the activation of the task, before which the task execution must be completed. A deadline as a parameter, in this work, is not explicitly defined, but it is implied that a task's deadline is equal to the period of the task. The remaining parameters

that a task representation consists of, depend on the type of the task. Usually, tasks are divided into three main groups: *periodic tasks*, *aperiodic tasks* and *sporadic tasks*. Aperiodic and sporadic tasks are also called event-triggered tasks so the activation time of these tasks is not known in advance and their behavior is unpredictable. The difference between aperiodic and sporadic tasks is that all requests for execution that come from aperiodic tasks are accepted and the request for execution of a sporadic task job can be rejected since the requests need to be separated with a time interval of certain length and shall not cause a periodic task or another accepted sporadic task job to miss its deadline. Aperiodic and sporadic tasks are not considered in this work and all considerations are aimed at periodic tasks. Periodic tasks are also known as *time-triggered* tasks and as the term *periodic* implies, the jobs of the tasks are periodically repeated and their behavior is very static. All this makes periodic tasks and systems composed of them very predictable, since the schedule for all tasks can be generated in advance. Therefore, by taking everything mentioned above into account, tasks that are considered in this work can be represented by three main parameters: period, WCET and deadline.

2.5 Offline Scheduling

One of the ways to find a feasible schedule and to satisfy all timing constraints is to make an offline schedule. Offline scheduling refers to finding a convenient order of the tasks' executions before the system starts running. This means that during run-time, the clock-driven scheduler uses a predetermined schedule of all real-time tasks. Because of that, offline scheduling is known as a predictable method since it is always known which task is the next to execute. The downside of offline scheduling is its inflexibility, since it requires all the information about the system (number of tasks, release times, execution times, etc.) and does not allow any deviations of the values.

2.6 Timing Verification

The defining characteristic of real-time systems is that they need to fulfill strict requirements, regarding both correct functionality and exact timing, as described in Subsection 2.3. Most of the timing constraints that are specified in such systems are in the form of deadlines, within which certain system operations must be carried out. To guarantee that all timing constraints will be met, the timing behaviour of the system is verified in a process called *timing verification* [5]. This process consists of two steps:

- *Worst-Case Execution Time (WCET) Analysis*: The upper bound on each task's execution time is determined. During this analysis, it is assumed that the analyzed task runs in *full isolation*, i.e., with no preemption, interruption or other interference caused by other tasks in the system. The upper bound that is obtained during this process is called the *worst-case execution time* of the task.
- *Response-Time Analysis (RTA)*: This step determines the Worst-Case Response Time (WCRT) of each task, taking into account its execution context. The execution context depends on the scheduling policy employed by the underlying Real-Time Operating System (RTOS). The analysis considers the WCET of each task determined in the previous step, the preemptions by higher priority tasks and system interrupts, the blocking by lower priority tasks due to resource locking, the scheduling overheads of the RTOS and the delays due to communication and access to shared resources.

The two-step approach presented above can be typically applied out-of-the-box for traditional single-core systems. In case that the target system is multi-core, the direct applicability of the approach diminishes due to challenges that multi-core systems inherently introduce into their analysis. The main issue with multi-core systems is the presence of contention over shared resources. The effects of the contention over shared resources can be integrated into the response time analysis, but then a perceptible circularity problem arises. Specifically, the response time of the tasks depend on the amount of contention, while on the other hand, the resource contention depends on the interval considered, which in this case, is the response time of the task that needs to be determined.

2.7 Data-propagation delays

In this work, data-propagation delays are considered as end-to-end delays meaning that the delay time is measured from the start to the end of the observed chain. Usually, end-to-end delays, also known as one-way delays, are closely connected to networks and they represent the time needed for a packet to be transferred from a source to a destination point. The same principle can be mapped to scheduling, so that an end-to-end delay can be considered as the time needed for some information to be propagated through a certain chain, where the starting point is the start of the first task in the chain, while the endpoint is associated to the end of the last task in the chain. Calculating the end-to-end delay in that way takes into consideration and includes in its final value all the eventual interferences that have happened between the start and the end of the chain and affected its execution.

The calculation of end-to-end delays is of crucial importance for a real-time system since the impact of these delays determines if the data in a system is temporally valid and usable or outdated and useless. Therefore, the end-to-end constraints are equally important for real-time systems as all other timing constraints that need to be fulfilled (i.e. meeting deadlines of the tasks) so that the system works properly. If the system cannot meet all end-to-end timing constraints, then it is considered that the system does not have a feasible schedule for the tasks.

To deal with the usability and validity of the data in a chain, as it is already mentioned, all possible end-to-end latencies need to be calculated and verified. There exist four different types of end-to-end delays [9]. However, within the scope of this thesis, solely data age delays are considered. The implemented approach introduces additional timing constraints to the process of scheduling which extends the computational time needed for finding a feasible schedule, but it is also the only proper way to keep a real-time system operative in a manner that its environment, current events, and data are relevant and valid. Besides data age constraints, heuristics such as job-level dependencies [10] are utilized in this work. This is one possible option to ensure the valid and timely propagation of data through the instances of the tasks in a task-chain, but also introduces additional precedence constraints to the optimization problem.

To show the importance of end-to-end delays on an example, the process of avoiding an obstacle with an autonomous car can be taken. If one of the tasks in a task-chain reads the distance to an obstacle, and then this information propagates through the chain and based on it certain decisions need to be made, it is of utmost importance that the data is read before it becomes outdated so the right decisions can be made. An example of a valid data propagation regarding the end-to-end delay is presented in Figure 3, where it can be seen that the propagation of data through a chain of three tasks finishes before the data age constraint. On the other hand, an example where the data age constraint is not fulfilled is shown in Figure 4. It can be seen that the second instance of task τ_3 reads outdated data since it is activated after the data age constraint.

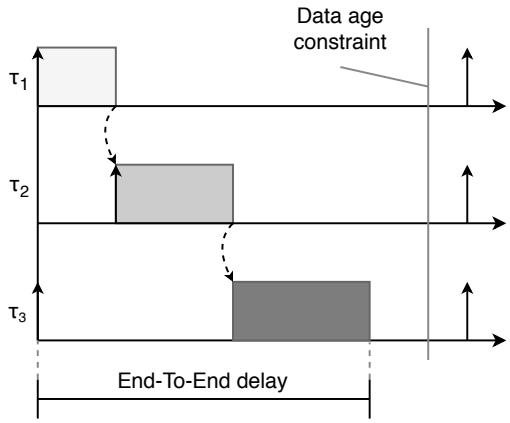

Figure 3: Visual representation of an end-to-end delay that fulfils the data age constraint.

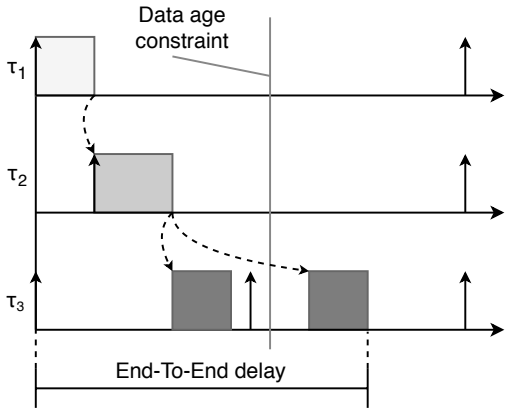

Figure 4: Visual representation of non-valid data since the maximum end-to-end delay is higher than data age constraint.

2.7.1 Job-Level Dependencies

One of the ways to fulfil data age constraints and ensure the validity of data inside the chains is by introducing job-level dependencies between tasks [10]. Job-level dependencies together with the deadlines of the system's tasks represent the vital scheduling requirements needed for the creation of an offline schedule. Therefore, in this work, job-level dependencies play a huge role in finding feasible schedules by targeting the maximum allowed data age inside different task-chains. The job-level dependencies are introduced as precedence constraints between individual instances of the tasks in a way that guarantees that data age constraints are met. That way, the restrictions on the ordering of the task jobs are made so that the combinations that do not fulfil the needed requirements are eliminated. If there is at least one combination created while satisfying the job-level dependencies, it is said that the system can meet all specified end-to-end constraints. It is important to distinguish between satisfying end-to-end constraints and finding feasible schedules, since the fulfilled data age constraints do not necessarily imply that the obtained schedule is feasible. The reason for that is the existence of other requirements that need to be satisfied.

2.8 Optimization Techniques

Generally, mathematical optimization refers to a group of methods which have the purpose of finding the best element inside a given data set by applying some criteria. Almost all of these methods come down to finding a maximum or minimum of a real function by computing its value using the given data. Therefore, an optimization problem can be described as:

- **Given**: Function $f : A \to \mathbb{R}$ where A is the set of provided data (data set) and $\mathbb{R}$ is the set of real numbers;
- **Find**: Element $\mathbf{y} \in A$ such that $f(\mathbf{y}) \leq f(\mathbf{x})$ for all $\mathbf{x} \in A$.

This can also be written in the canonical form of an optimization problem that these methods solve [11]:

- **Minimize** $f(\mathbf{x})$,
- **Subject to**:

$$\begin{aligned} \mathbf{g}_j(\mathbf{x}) &\leq \mathbf{0}, & j &= 1, \ldots, m; \\ \mathbf{h}_k(\mathbf{x}) &= \mathbf{0}, & k &= 1, \ldots, p; \\ x_{i,min} \;\leq\; x_i \;\leq\; x_{i,max}, & & i &= 1, \ldots, n. \end{aligned}$$

where $f(\mathbf{x})$ represents the objective function, $\mathbf{g}_j(\mathbf{x})$ represents the vector of inequality constraints and $\mathbf{h}_k(\mathbf{x})$ the vector of equality constraints.

There are different optimization techniques and they employ different calculi for finding the optimal solution. In this thesis, a classical optimization technique, known as *constraint programming*, is considered. This optimization technique is the basis for the implementation of the approach for finding the optimal schedules.

2.8.1 Constraint Programming

The presented standard form of optimization problems is commonly solved by *Integer Linear Programming* (ILP) which introduces the restriction that the variables are integers, and the objective function and the constraint functions are linear. Limiting the variables to just integers does not impair usage of this method since, in the real world, most problems can be represented by integer values. On the other hand, the method employed in this work, called *Constraint Programming* (CP) is a method for solving optimization problems where relations between system variables are defined by certain constraints which restrict the values of these system variables. The major difference between constraint programming and integer programming is that the variable range in constraint programming is defined as a set of elements, while in integer programming, the variable

range is defined as an interval. The main goal of this programming paradigm is to determine a valid solution that fulfills a defined set of conditions. This method is most effective on highly combinatorial problem domains and it is widely used in problems such as resource-constrained scheduling. There are different algorithms implemented and employed in different programs which are created for solving constraint programming problems. Considering the purpose of this work, two tools were appropriate candidates to select from: *IBM ILOG CP Optimizer* and *Google OR-Tools*. In the end, *IBM ILOG CP Optimizer* was chosen, since it provides a more extensive example base and an elaborate collection of user guidelines.

Conditional Time Intervals: To successfully solve a constraint programming problem, the most important step is to appropriately define all variables and constraints present in the modeled system. A scheduling problem can be defined in many ways, but the most intuitive approach to this type of problems is to define each job that needs to execute in a schedule as an *interval variable*. Interval variables represent the definition of time intervals inside which it is possible to insert the execution of a certain part of the schedule. In that purpose, *Conditional Time Intervals*, which allow for an easy creation of interval variables, are employed [12][13]. An interval is characterized by a start value, an end value and its size. The length of an interval is the difference of its end time and its start time. All three interval parameters can be constrained by a minimum and a maximum value.

For conditional interval variables there exists a special set of constraints which model possible precedence relations imposed on two or more intervals. That way, constraints on the ordering on the individual schedule elements can be set. If two interval variables A and B are assumed, the possible precedence constraints that can be set between them, are:

startBeforeStart(A, B, c):	Defines a constraint where A has to start at least c time units before B is allowed to start.
startBeforeEnd(A, B, c):	Defines a constraint where A has to start at least c time units before B ends.
endBeforeStart(A, B, c):	Defines a constraint where A has to end at least c time units before B is allowed to start.
endBeforeEnd(A, B, c):	Defines a constraint where A has to end at least c time units before B ends.
startAtStart(A, B):	Defines a constraint where A has to start together with B.
startAtEnd(A, B):	Defines a constraint where A has to start when B ends.
endAtStart(A, B):	Defines a constraint where A has to end when B starts.
endAtEnd(A, B):	Defines a constraint where A has to end together with B.

Alongside these precedence constraints, a **noOverlap(L)** constraint can also be specified over a set of interval variables. The **noOverlap** constraint restricts the intervals included in the set L in a way that they cannot overlap. Using the previously listed constraints, the application's timing requirements can be fully defined as a set of interval variables and constraints.

3 Related Work

Since this thesis deals with multi-core systems, which is a broad area and a very active field of research, there is a multitude of related topics that need to be considered and addressed. This section contains a brief summary of all the current trends in multi-core systems research that are relevant to this work.

Firstly, in Subsection 3.1, different timing verification frameworks for multi-core systems are presented and the difficulties regarding the application of this analysis to multi-core systems are highlighted. Subsection 3.2 summarizes currently available approaches to minimizing the extent of these difficulties, mainly focusing on approaches based on temporal isolation. Furthermore, Subsection 3.3 focuses on the application of optimization techniques for the generation of offline schedules for multi-core systems.

3.1 Timing Verification for Multi-core Systems

As highlighted in Subsection 2.6, multi-core systems suffer from an apparent circular dependency problem, where the response time of the tasks depend on the amount of contention over shared resources and vice versa.

Several authors have addressed this problem by pioneering and developing new response time analysis (RTA) frameworks that are specifically oriented at multi-core systems. For instance, Schliecker et al. [14], proposed a methodology that integrates the effects of contention into the RTA. The WCRT of each task is calculated based on the tasks WCET (which is determined for the task in full isolation), preemptions due to higher priority tasks that are mapped to the same core and delays due to contention over the shared resources. To deal with the circular dependency of the WCRTs of tasks on different cores, the framework utilizes a fixed point iteration approach.

A number of papers is based on the *superblock* model [4][15][16][17], where each task is modeled as a sequence of superblocks, which can contain branches and/or loops. The majority among them introduces various arbitration policies for the access to shared resources. Some of the arbitration policies considered are Round-Robin, *First-Come-First-Serve* (FCFS) and TDMA. The papers show that by introducing given arbitration policies, the level of contention can be upper-bounded and, in turn, schedulability can be improved. A further review of papers that rely on arbitration policies to upper-bound the delays caused by contention over shared resources, or more specifically, over the system bus, is presented in Subsection 3.2

Finally, Dasari et al. [18] present a response time analysis framework aimed at multi-core systems with partitioned fixed-priority non-preemptive scheduling. The only shared resource that is assumed is the shared system bus employing a work-conserving arbitration policy. The authors introduce a *request function* into the schedulability analysis to account for the bus contention. The function returns the maximum possible number of access requests that a task can make in a given time period. In this paper, a fixed-point iteration approach is utilized to handle the circular dependency, i.e., the dependency of the number of requests upon the task's WCRT. The iterative process is continued, until all response times converge to constant values or until a deadline is surpassed, implying non-schedulability.

The latter paper, analyzes a multi-core architecture that is similar to the architecture assumed within this thesis. Therefore, this paper is a prolific reference for the timing analysis that is implemented in the thesis.

3.2 Temporal Isolation

The circular dependency between the tasks' WCRTs and the number of shared resource accesses, which is described in Subsection 2.6, can be alleviated in many ways. One of the most prominent approaches to this problem that is relevant to this thesis is *temporal isolation* [5]. The main source of the problematic inter-core interference causing the dependencies is typically the system bus. Temporal isolation revolves around the idea that by introducing controlled access to the shared bus, the effect of the inter-core interferences can be bounded, hence simplifying their integration into the schedulability analysis. Controlled access to the shared bus can be achieved in multiple ways [5]. In this review, the focus is set on two different techniques relevant for this thesis: temporal

isolation achieved via software, more precisely by utilizing a *phased execution model*, and temporal isolation achieved via hardware, by utilizing *TDMA arbitration policies*.

One of the approaches which turned out to be successful in reducing inter-core interferences is the introduction of a *phased execution model*. The main idea behind this technique is to divide tasks' execution into phases which require access to the bus and shared memory, and phases that are called *computation phases* and which do not require access to the shared memory. Usually, each task is divided into three parts: two memory phases and one computational phase, so that the data is collected or written during the memory phases and between these memory phases a computation phase is executed.

In their paper [19], Kim et al. provide an example of the phased execution model implementation. In this work, the target are single-core processors and the problem of transferring and using the safety-critical embedded applications designed for single-core processors, to multi-core processors. Therefore, authors proposed a heuristic algorithm for solving a partitioned scheduling problem which serializes I/O (Input/Output) partitions with a goal of preventing conflicts between I/O transactions from applications (tasks) executing on different cores. During the work, I/O transactions are firstly classified into two groups called *Physical-I/O* and *Device-I/O*, where *Physical-I/O* is used for communication between the physical environment and the I/O device. More importance lies on the implementation of *Device-I/O*, since this process is executed on the core and it is divided into three sub-processes, called partitions: *Device-Input Partition*, *Device-Processing Partition*, and *Device-Output Partition*. This way, a process that happens between the physical input and physical output of data, is divided in two phases obligated for collecting and sending data to *Physical-I/O* and one phase which has the purpose off handling data computing. This allows *Device-I/O* processes to be separated into smaller time-consuming parts and executed at any time as long as the data is valid and usable.

Yao et al. created a Memory Centric scheduling approach where a TDMA schedule is utilized to restrict non-preemptive memory phases to be executed in certain time slots [20]. The main feature of this approach lies in a method of arranging executions of tasks' phases where memory phases provide the tasks to have higher priorities than the tasks in execution phases. Also, by using the TDMA scheduling, the multi-core systems can be observed as single-core systems since full isolation is provided, which eases the response time analysis. Further, a few years later, Yao et al. extended the Memory Centric scheduling approach to global scheduling [21]. It is shown that this new approach gives better results considering contention than the original Memory Centric approach. These results are achieved by dropping out the usage of TDMA scheduling for memory phases and by introducing a global fixed priority scheduler where the promotion of memory phases over execution phases is kept.

Regarding the utilization of TDMA arbitration policies, in a paper [3], Rosen et al. propose an approach based on optimizing the periodic TDMA schedule of the system bus in a step-by-step procedure, where each step corresponds to a single segment of the bus schedule. Based on the optimized TDMA schedule, a system level offline task schedule is constructed.

In two papers that are more relevant to the context of this specific thesis [22][23], Kelter et al. describe a comprehensive approach to multi-core WCET analysis. The papers are assuming a shared TDMA bus. The main characteristic of the analysis is that it bounds the memory access delays caused by the waiting for the succeeding TDMA slot. To determine the bound of the delays, the analysis considers the exact time offsets between task execution and assigned TDMA slot.

Although, the presented approaches bring in distinct improvements to the predictability of multi-core systems, the main trade-off is that the system's computing resources are not utilized to full extent, due to the non-work conserving nature of the TDMA protocol. To increase the utilization of available bandwidth, different optimization techniques on the task scheduling layer can be employed. Such approaches are presented in Subsection 3.3

3.3 Optimization based Offline Scheduling

The research on the topic of offline task scheduling using optimization techniques has been contributed to by multiple authors [24][25][26]. Many of the proposed approaches are based on constraint programming (CP) and integer linear programming (ILP). Although, they are mainly aimed at single-core systems, the authors presented clear formulations which are a helpful starting refer-

ence, since partitioned scheduling with dedicated schedules for each individual core was considered as one possible solution to the problem in focus.

In his PhD thesis [27], Tompkins investigates the applicability of mixed-integer linear programming for task allocation and scheduling in distributed multi-agent operations in general. Within his framework, a problem is formulated, divided into a precedence-constrained set of sub-problems and then the optimal allocation of these sub-problems/tasks to individual agents is determined so that they are completed in optimal time. The resulting schedule is constrained by the execution time of each task, job release times and precedence relations, as well as communication delays between agents. Additionally, the author presents an extension of the framework which facilitates multiple objective optimization. The proposed framework, with slight modifications in line with the context of the problem, can be a useful guide for the application of ILP methods in this thesis.

Within the context of multi-core real-time systems optimization based offline scheduling has been studied by Puffitsch et al. [28]. In their paper, the authors present an approach to execute safety-critical applications on multi- and many-core processors in a predictable manner. The paper details the approach to automatically generate a feasible schedule based on constraint programming, by applying it to several task sets that are derived from industrial applications.

Becker et. al. proposed the *Memory Aware Contention-Free Framework* (MCEF) [29] to facilitate the scheduling of multi-rate real-time applications on clustered many-core architectures while taking into account the memory constraints imposed by the underlying platform. The authors devised an offline scheduling approach based on constraint programming and job-level dependencies [10], in which a non-preemptive time-triggered schedule is utilized to orchestrate the access to the shared memory. The work presented in the above paper serves as an important starting point for the development of the method proposed in this thesis.

4 Research method

This section presents an overview of the scientific research method employed within this thesis, together with a discussion on the applicability of it onto the problem under investigation. Further, the application of the selected method is described and connections with the steps of the thesis processes are drawn.

4.1 System Development Research Method

The main focus of this thesis is directed at developing new approaches to creating optimized schedules for multi-core systems. Considering the main problems of multi-core systems (i.e. inter-core data-propagation delays), the main approach with its specific optimization objectives that will be investigated is defined as:

- Devising an offline scheduling approach to optimize global scheduling of tasks by utilizing a phased task execution model and job-level dependencies.

Since it is expected that this thesis delivers a fully functional approach, as an appropriate research method the *System Development Research Method* is taken into consideration. The reason for selecting this method lies in a fact that the *System Development Research Method* is usually used in processes of creation of new systems, approaches and any other product that requires systematic development. This method is presented and described by Nunamaker and Chen [1]. It is composed of four main phases: *theory building*, *system development*, *experimentation* and *observation*. Figure 5 shows the interconnection of all phases, which means that transitions from each of the processes to any of the other processes is allowed. Therefore, in every stage of the system development, it is possible to proceed or return and restore previously obtained information. This way of problem observation can significantly improve the process of development since the flexibility of the method and its ability to backtrack facilitate an easier problem detection.

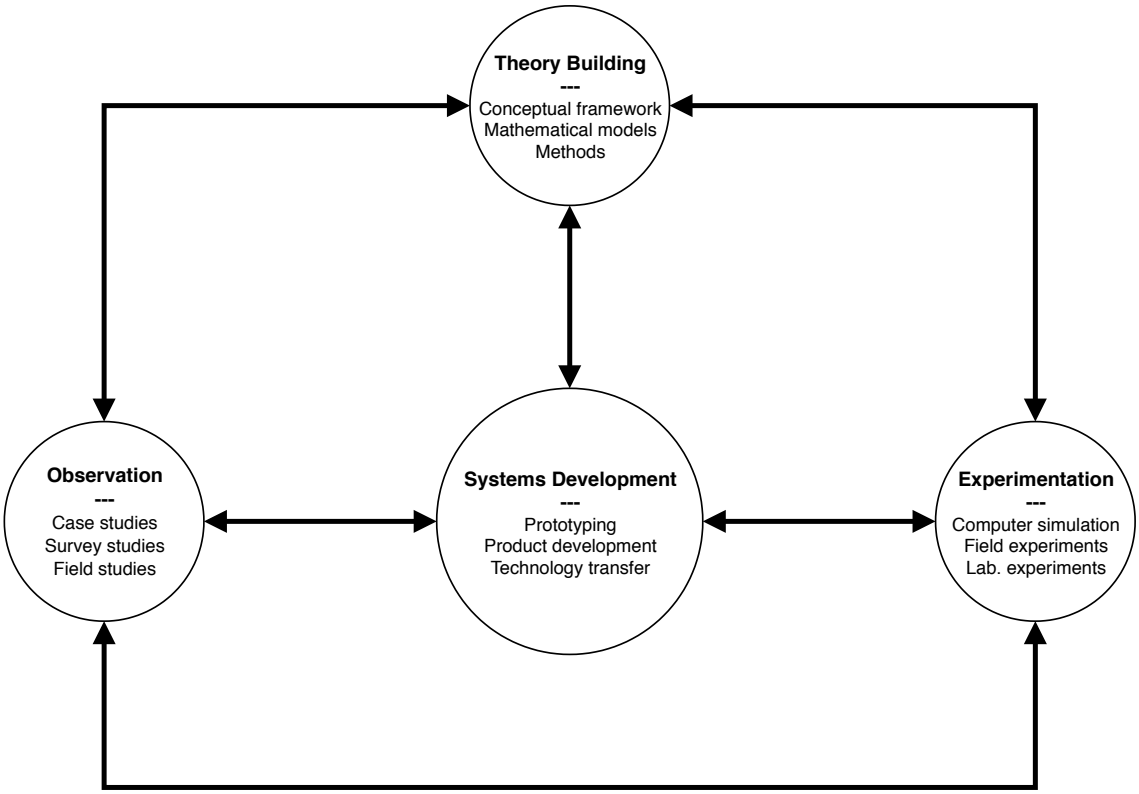

Figure 5: Overview of a multi-methodological research approach [1].

4.2 Application of the Research Method

The whole research procedure is based on the process of *System Development Method* which is a multi-methodological approach to research consisting of several steps (Figure 6). Based on the described work and exploration of this thesis, the steps of conducting the *System Development Method* can be modified and the planned research can be summarized with the following set of steps:

1) Qualitative analysis of the system and its components and modelling.

2) Investigation of existing scheduling approaches that utilize optimization algorithms.

3) Selection of the most suitable optimization algorithms. The selection is narrowed down to CP-based algorithms. Convenient optimization toolsets are picked based on this selection.

4) Implementation of the suggested offline scheduling approach and testing of the selected optimization algorithm.

5) Results analysis and report writing.

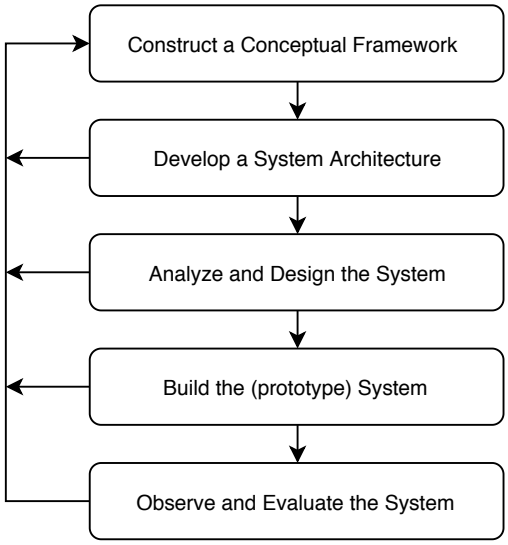

Figure 6: Process of the *System development research* method [1].

Furthermore, the development process and the way of implementing it are described below. To enable the optimization process, it is necessary to analyze the multi-core system in the context of an optimization problem, i.e., to evaluate all timing and precedence constraints and communication properties, as well as to model the system with an appropriate set of variables upon which the optimization can be carried out. Clearly, the principal objective of the optimization is to minimize the inter-core data propagation delays.

Additionally, the optimization algorithm and its corresponding outputs are investigated, with a special focus on the efficiency of *constraint programming* (CP) implementations. The metrics across which the algorithm is observed are mainly related to the algorithm execution complexity and scheduling performance. The overall goal is to develop a scheduling tool prototype that implements this approach and conducts a performance evaluation using the prototype.

Based on the study and review of the current state of the art and research on the topic and the qualitative analysis of the system components, a model of the system is deduced to offer a complete variable set appropriate for optimization. After the modelling, the optimization algorithm is implemented. Based on this algorithm, the offline scheduling approach is implemented and subsequently tested in an experimental setup.

5 Technical Approach

The proposed procedure overall is not complicated and it consists of five parts, which can be seen in Figure 7. The first step is to model the application so it becomes compatible for solving with the presented approach and techniques. The modeling of the system and all of its parts are in detail described in Section 6 where the modeling of all the features of the approach, such as the phased execution task model and job-level dependencies, is explained. After the system has been modeled, the core of the tool reads the model from an input XML file and tries to solve the constraint satisfaction problem (CSP) and produce the final schedule. This component is called the *Scheduler* and it is realized by several executable files which contain the necessary code needed for the translation of the raw model into a CP model which is suitable for solving by the *IBM ILOG CP Optimizer*. Besides that, after the communication with the *CP Optimizer* is finished and the optimized schedule is retrieved, the program before the end of its execution saves the generated schedule in the form of an *XML file*. Furthermore, the generated files are analyzed in a manner to measure the success rate and efficiency of the method.

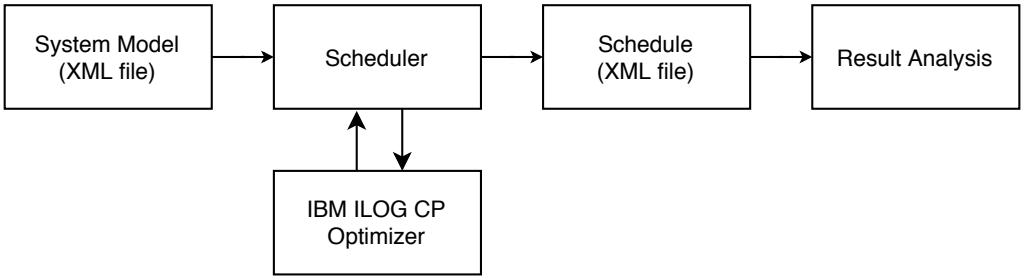

Figure 7: Summary of the processes that the proposed approach consists of.

5.1 Scheduler

The *Scheduler* represents a set of executable files which perform certain actions with the purpose of satisfying and employing all assumptions, features, and approaches used in the presented scheduling method. The program is written in the Python programming language using the *PyCharm* Integrated Development Environment (IDE). Also in the program, as the optimization backend engine, *IBM ILOG CP Optimizer* is employed directly as a Python library called `docplex`. This way, the whole method is contained within a single Python project and the only requirement is that *CPLEX Studio* (in this concrete case version 12.9.0) is pre-installed and accessible from the Python environment.

After the application of interest has been modeled, the model, in the form of an XML file containing descriptions of all the application components, is read by the executable program. The program takes charge of all other processes which are needed to produce a feasible schedule. First of all, the initial application model is transformed into a constraint programming optimization (CPO) model which is attempted to be solved by the *docplex* solver following the guidelines given in [30]. In the end, when the optimizer returns a feasible schedule (or finds that there is no feasible solution), the program outputs the newly obtained schedule as an XML file which is appropriate for further analysis.

5.2 IBM ILOG CP Optimizer

IBM ILOG CP Optimizeer is a system used for more than 20 years in solving scheduling problems based on constraint programming. Since the specialized mathematical algorithms that are usually employed cannot be straightforwardly implemented to solve certain optimization problems, especially ones that are hard to be linearized, *IBM ILOG CP Optimizer* provides a robust optimizer which can handle these problems and find the satisfying solutions and in most cases exact to the ones returned by previously mentioned algorithms. This way, the process of solving constraint programming problems is greatly simplified and by utilizing the *IBM ILOG CP Optimizer* it can be divided into three straightforward steps: *describing*, *modeling* and *solving* the problem. During

these steps, the optimizer allows and provides simple use of many features important for solving a constraint programming problem, such as easy declaration of decision variables, declaration of interval variables, definition of an objective function, definition of timing constraints, precedence constraints and other. *IBM ILOG CP Optimizer* can be utilized and used together with many programming languages and IDEs associated with them, and one of them is Python which is used in the frame of this work.

5.3 Testing and Evaluation

After the presented approaches and methods have been implemented, it is very important to test and analyze the efficiency of the whole apparatus. In that purpose, many synthetic tests are produced and evaluated for, so that the different performance and efficiency indicators can be tested, such as the scheduling success rate, computational time and obtained data-propagation delays. Also, it is important to show how the introduced job-level dependencies impact the schedule generation time. Besides that, the implemented method is compared for a different number of task chains present in the system and for a varying percentages of utilization of the system's processing resources. Based on the obtained results some major conclusions are drawn. A detailed description of the analysis, together with how the testing and evaluation is conducted, is provided in Section 8.

6 Technical Description

This section contains a description and explanation of all technicalities needed to understand the approach to the problem and the proposed processes and solutions. In the first part, the theoretical representation of the analyzed systems is provided and it contains all important information about how each part of the system is modeled. Further, the second part of the section covers necessary instructions and aspects in the form of guidelines which allow the presented processes to be conducted, tested and replicated.

6.1 System Model

Systems analyzed in this work consist of an embedded software application executing on a hardware platform. Both parts contribute evenly to the functionality and performance of the system and thus, need to be clearly defined for further analysis. In this subsection, both platform and application models are presented and described, together with all the relevant variables and parameters.

6.1.1 Platform Model

The application is assumed to execute on a dual-core platform, where each core has a private cache and no shared caches in between. To access the on-chip memory, both cores are connected to a shared system bus. Task execution can take place simultaneously on both cores, while memory accesses are omitted only to one core at a time, due to the shared bus. Furthermore, it is assumed that there is no contention on the bus caused by other modules on the chip, such as *Direct Memory Access* (DMA) and *Input/Output* (I/O) controllers. Operation of the cores is synchronous with the system clock which also enforces a mutual synchronization between the cores.

The described platform model is based, with some major simplifications, on NXP Semiconductors' MPC5777C MCU family for Automotive and Industrial Engine Management [31]. The employed parameters for the platform model are presented in Table 1.

Table 1: Overview of the employed parameters for the platform model.

Parameter	**Value**
Number of cores:	2
Architecture:	32-bit
Clock frequency (f_{CLK}):	300 MHz
Byte R/W per clock cycle (v_{BPC}):	4 $\frac{bytes}{cycle}$

6.1.2 Application Model

An application is defined as a collection of tasks, task chains and job-level dependencies defined over specific task pairs. The application model is mainly based on standard automotive applications. The following paragraphs, introduce and elucidate the individual components of an application.

Task Model: The task set is comprised of periodic non-preemptive tasks that are time-triggered. Event-triggered tasks are not taken into consideration. Each task is statically mapped to a fixed core and no inter-core migrations are possible. Therefore, a task τ_i is described by the tuple $\tau_i = \{T_i,\ C_i,\ p_i\}$, where T_i is the activation period, C_i the worst-case execution time (WCET) of the task and p_i the core it is allocated to. The task deadline D_i, as an important parameter of a task (see Section 2.4), is not explicitly defined and contained in the tuple. In this case, an implicit definition of the deadline is assumed which indicates that each task's deadline is equal to its period, ($D_i = T_i$). That means that the absolute deadline of each job of a task is the absolute release time of the next job. It is important to note, that due to non-preemptive scheduling, once a task starts executing, it must finish before any other task can run.

All tasks are grouped into a task set Γ. When the task set contains multiple tasks with different periods, the execution of the application starts to repeat only after the *least common multiple* of all the involved periods, which is also called the *hyperperiod* of the task set. Hence, the application's schedule is generated for the duration of one hyperperiod, after which it is repeated for each new hyperperiod. The hyperperiod of the task set is obtained by:

$$\mathrm{HP}(\Gamma) = \mathrm{LCM}(T_i \mid \tau_i \in \Gamma) \tag{1}$$

Further, tasks communicate by *register communication*, where a register is represented by a global variable which is updated by a sending task and read by a receiving task with no signaling between tasks. This implies that the receiving task assumes temporal validity of the read register value. With the aim of controlling the bus contention and increasing predictability of the communication delays, each task's execution is divided into three phases: two memory-access phases (*Read* and *Write*) and an *Execution* phase as shown in Figure 8. All the required input data is read during the *Read* phase and stored into local variables. After that, the *Execution* phase performs the needed operations on the inputs without any need to access the bus. Finally, the output values are written into the memory during the *Write* phase. An example of how the implementation of these phases affects the bus schedule can be seen in Figure 9. This execution model is often used in the automotive industry, e.g., it is the base of the implicit communication model in the AUTOSAR platform [32].

For each task job, all three of these phases are performed in a defined sequence (*Read-Execute-Write*). Therefore, each task's total execution time C_i, is the sum of the durations of each individual phase: $C_i = C_{i,R} + C_{i,E} + C_{i,W}$, where $C_{i,R}$, $C_{i,E}$ and $C_{i,W}$ represent the duration of the *Read*, *Execution* and *Write* phases, respectively.

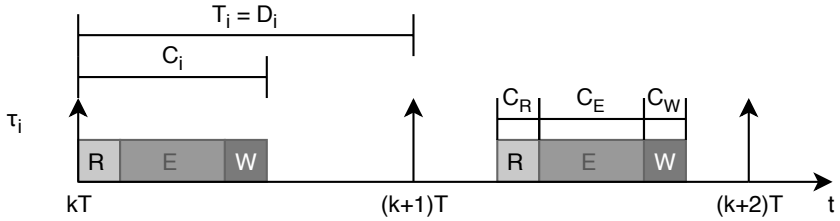

Figure 8: Two jobs of task τ_i consisting of three phases: *Read*, *Execution* and *Write*.

Job Model: Since periodic tasks are considered, each task τ_i implicates multiple jobs whose number depends on the task period T_i and the hyperperiod of the whole task set. Thus, the j-th job of task τ_i is represented by the tuple $\tau_{i,j} = \{r_{i,j},\ s_{i,j},\ e_{i,j},\ d_{i,j}\}$, where $r_{i,j}$ is the job's absolute release time, $s_{i,j}$ and $e_{i,j}$ are the absolute start and end time, respectively, of the job execution and $d_{i,j}$ is the job's absolute deadline. In accordance with this description, it must hold:

$$r_{i,j} = jT_i \tag{2}$$
$$d_{i,j} = (j+1)T_i = r_{i,j} + T_i \tag{3}$$
$$r_{i,j} \leq s_{i,j} \leq d_{i,j} - C_i \tag{4}$$
$$r_{i,j} + C_i \leq e_{i,j} \leq d_{i,j} \tag{5}$$
$$C_i \leq e_{i,j} - s_{i,j} \leq T_i \tag{6}$$

It is easily understood that the duration of the job execution, $e_{i,j} - s_{i,j}$, can vary due to the possible contention on the shared bus, but it must be at least the WCET of the task C_i, and yet not greater than the duration of the task's period T_i. Clearly, the main objective of this particular scheduling problem is to obtain the values of the variables $s_{i,j}$ and $e_{i,j}$ for each individual job.

Task Chain Model: A task chain is an arranged sequence of tasks through which the certain data propagates. The observed data usually has specified timing constraints that are commonly given as the maximum age that data is allowed to reach. Therefore, a task chain ζ can be described

as a tuple consisting of two parameters $\zeta = \{\lambda,\ \eta\}$. The parameter λ represents a sequence of tasks that constitute the specific chain and the order of the tasks in λ is the actual order of the tasks in the chain. The limitation is that only one appearance of a task is allowed per particular chain, thus the occurrence of cyclic chains is prohibited. On the other hand, it is allowed that one task can be part of multiple chains. The parameter η represents the maximum allowed data age of the chain.

To understand it easier, an example of a task-chain is provided in Figure 9. The figure depicts a task chain, $\zeta_1 = \{\lambda_1,\ \eta_1\}$, where λ_1 consists of four tasks: $\lambda_1 = \{\tau_1,\ \tau_4,\ \tau_2,\ \tau_5\}$ and η_1 is the pre-determined maximum allowed age of data. These four tasks are executed on three cores, where tasks τ_1 and τ_2 are mapped to Core 1, tasks τ_3 and τ_4 are executed on Core 2 and τ_5 on Core 3. It can be observed how the data propagates through the chain and the total age of data is marked as an end-to-end delay. Also, it can be noticed that the obtained end-to-end delay is less than the allowed data age which usually has to be satisfied so that the data can be considered valid and usable.

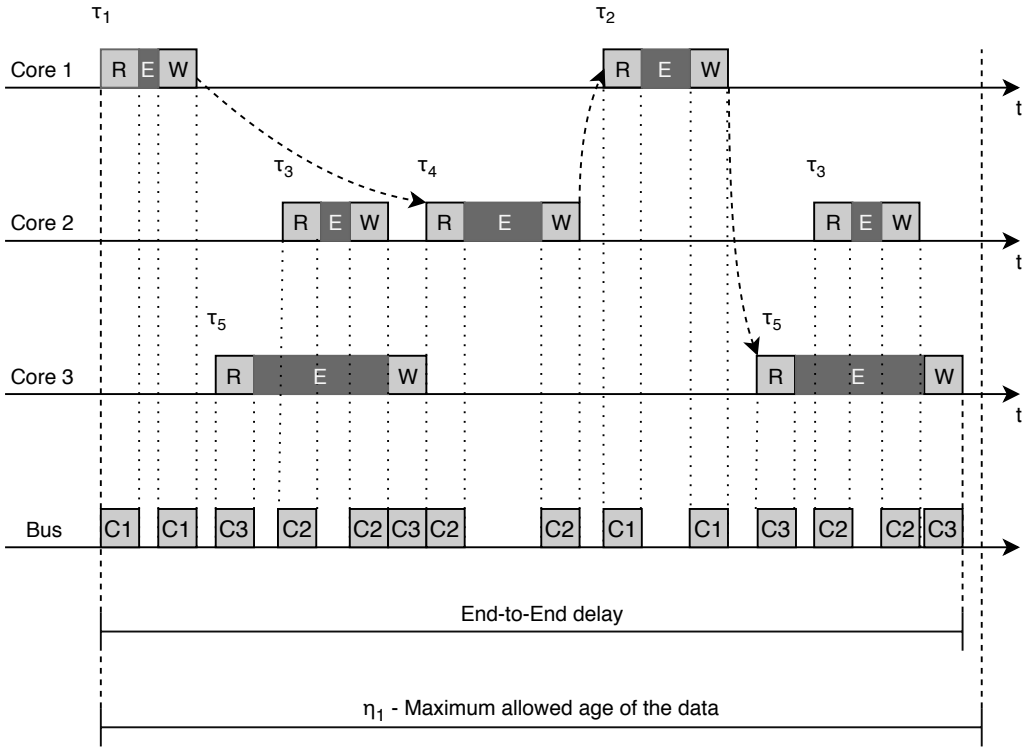

Figure 9: Example of a task chain.

Job-Level Dependencies: A job-level dependency is modeled as a precedence relation between two jobs of different tasks: $\Phi : \tau_i \xrightarrow{(j,l)} \tau_k$, where τ_i and τ_k represent two different tasks and $\tau_{i,j}$ and $\tau_{k,l}$ represent corresponding instances of the given tasks. With this definition of dependencies, it is assured that instance j of task i has to finish its execution before instance l of task k can start its execution. Therefore, by introducing multiple dependencies between tasks in a specific chain, the process of satisfying the end-to-end timing requirements demanded in the task chains can be facilitated and enforced by design.

6.2 Offline Schedule Generation

The procedure to generate an offline schedule using the implemented approach contains several steps that need to be carried out. Therefore, this section provides the needed information and describes the operations that must be performed so the fully valid and executable schedule can be created.

6.2.1 Task Set Creation

As it is described in Section 6.1.2, the model of task τ_i consists of three main parameters: the *period* of the task T_i, the worst-case execution time C_i and the core p_i the task is mapped to. Therefore, to successfully create a task set that is defined by the application, it is needed to create a certain and reasonable number of tasks described by previously mentioned parameters. During the modeling of the tasks, several things need to be kept in mind in order to produce a valid and compatible task set. Since the scheduling approach employed in this work is based on global scheduling, for that purpose it is of crucial importance to limit the hyperperiod of the task set. The length of the hyperperiod directly affects the number of task instances that are generated, and thus increases the time needed to schedule all the instances. That means that the created tasks need to have reasonable period values which are compatible so that a valid hyperperiod can be produced. One of the ways to achieve that is by defining the values of the periods so that they are multiples of each other.

Regarding the generation of execution times, an obvious relation between periods and execution times is that $C_i < T_i$, since implicit deadlines equal to the task periods, are considered. Besides that, according to the task model, each task's execution is divided into three phases (*Read*, *Execution*, *Write*), meaning that the individual execution times for each of these phases need to be defined. Therefore the relation $C_i < T_i$ becomes $C_{i,R} + C_{i,E} + C_{i,W} < T_i$, where can be straightforwardly concluded that if the difference between the left-hand side and right-hand side is smaller, the finding of a feasible schedule will be harder. The durations of the memory phases are not required to be directly specified, but they are defined by the number of bytes that are being read-/written during these phases. On the other hand, the duration of the execution phase requires to be directly assigned while taking into account the previously mentioned relation between execution times and periods. Finally, the tasks need to be assigned to the cores by taking into account the total utilization of each core.

6.2.2 Hyperperiod Calculation

Since the focus of this work is set on periodic tasks and their executions, for successful scheduling of all the instances of the tasks, $\tau_{i,j}$, it is needed to calculate the hyperperiod of the task set. The hyperperiod is defined as *the least common multiple of all periods T_i of tasks τ_i in the task-set* Γ and represents the amount of time inside which all instances can be arranged. Of course that is only implied if the calculation of the hyperperiod has been correctly conducted since the number of task jobs directly depends on the length of the hyperperiod. With the possibility to determine an arrangement of all task jobs on a single whole hyperperiod, the periodicity of the whole system is achieved. That way, it is known in advance, before run-time, how the system will behave since the hyperperiod represents its execution unit that will be repeated innumerable times.

For the calculation of a hyperperiod, Algorithm 1, based on the Formula 1, is implemented. In the provided listing, the calculation process is presented as a pseudo code where all needed steps are contained. As it can be seen, in the beginning it is needed to check if the task-set consists of two or more tasks so that the calculation of the hyperperiod can be valid. If that is not the case, it means that a task set contains only one task and the period of that task is returned. Otherwise, the hyperperiod calculation proceeds by calculating the least common multiple of the first two tasks in the task-set which represents the initial value. Afterward, the hyperperiod is calculated in an iterative process where the new value of the hyperperiod becomes the least common multiple of the old hyperperiod value and the period of the next task in the task-set. That process continues until all the tasks in the task set are included in the calculation. That ensures that the final value of the hyperperiod is divisible by all task periods which is of crucial importance for the successful scheduling of the generated task jobs.

Algorithm 1 Hyperperiod Calculation.

```
1: procedure CalculateHP(Γ)
2:     if length of Γ < 2 then return T_0
3:     HP ← LCM(T_0, T_1)
4:     for each τ_i in Γ do
5:         HP ← LCM(HP, T_i)
6:     return HP
```

In the algorithm used for the calculation of the hyperperiod, the function LCM($x,\ y$) is employed. The implementation of this function is provided in Algorithm 2. The principle used in the calculation is based on the fact that the least common multiple of two numbers x and y can be defined as the quotient of their product and their greatest common divisor (GCD):

$$\mathrm{LCM}(x,\ y) = \frac{xy}{\mathrm{GCD}(x,\ y)} \tag{7}$$

What remains is to implement the calculation of the GCD and for that purpose the Euclidean algorithm is implemented.

Algorithm 2 Calculation of LCM.

```
1: procedure GCD(x, y)
2:     while y > 0 do
3:         t ← y
4:         y ← x mod y
5:         x ← t
6:         return y
7:
8: procedure LCM(x, y)
9:     return |xy|/GCD(x, y)
```

6.2.3 Generation of Task Jobs

For the successful creation of a global schedule with all the given tasks included, it is required to construct all instances of all the tasks in the given task set. As it is described in Section 6.1.2, a task job $\tau_{i,j}$ is represented by four parameters: the release time of the job $r_{i,j}$, its start time $s_{i,j}$, its end time $e_{i,j}$ and its deadline $d_{i,j}$. The start times of the jobs represent the values that are targeted by the scheduling procedure and these values are found during the process. The other two parameters (release time and deadline) need to be assigned for each job of all the tasks so the mentioned scheduling procedure can be conducted. This can be done by implementing Algorithm 3 provided below.

Algorithm 3 Generation of all task jobs within one hyperperiod.

```
1: procedure GenerateAllJobs(Γ, HP)
2:     for each τ_i in Γ do
3:         #jobs_i ← HP/T_i
4:         for j = 0, j < #jobs_i do
5:             r_{i,j} ← j · T_i
6:             d_{i,j} ← (j + 1) · T_i
7:             j ← j + 1
```

The presented algorithm shows how the jobs of all the tasks in the system can be generated. It is done in a way that for each task in the task set the number of instances is determined. After that, it is possible to create all the instances by implementing a simple *for* loop inside which the

release time and the deadline of each instance are calculated and assigned. The release time is calculated as a multiple of the period of the task, starting from zero, while the deadline is also calculated as a multiple of the period of the task, but starting from one, since the deadline of each task job is de facto the release time of succeeding job.

6.2.4 Specification of Task Chains

By revisiting Section 6.1.2, it can be recognized that probably the most important parameters, regarding the purpose of this work, are the ones that define the maximum allowed data ages for each chain η_i involved in the application. The reason for that lies in the fact that the successful creation of a schedule does not only depend on satisfying all the task deadlines, but it also hugely depends on ensuring that all the end-to-end delays satisfy the data age constraints present in the system. Therefore, a schedule that does not provide the satisfaction of all data age constraints is recognized as a non-valid schedule.

The process of satisfying the data age constraints hugely depends on how the constraints are set. Therefore it is of crucial importance that during the process, reasonable values for maximum data ages are defined. In those terms, it is adequate to define the data age delay of a certain chain η_i that is larger than the hyperperiod of the tasks contained in that chain. Additional problems to the process of satisfying the constraints are caused by the number of the tasks in the chain. The more tasks are involved, the arranging of all the task instances becomes more complicated since the number of timing constraints significantly increases. Also, it is important to try to minimize the number of chains that a task is part of. Since it is allowed that one task can be contained in multiple chains, that brings new complications because the constraints become more complex in order to satisfy end-to-end delays of all the implicated chains. Lastly, for the handling of task chains, it is important to strive to have balanced task chains regarding the periods of the involved tasks. It means that the satisfaction of the constraints will be eased if the tasks in the chain share the same period, because then the problems, known as over-sampling and under-sampling, are avoided.

To ensure that the data age constraints are fulfilled, the calculation of all end-to-end delays in the final schedule is required. The calculation of a delay is conducted in a way that at first, the last task of the chain is analyzed. From each instance of the last task, the closest instance of a neighboring task in the chain is found. Then the same process repeats for the penultimate task and continues in the same manner until the first task of the chain is reached. In the end, the maximum end-to-end delay is determined and that value must satisfy the data age constraint set for the observed chain.

6.2.5 Generation of Job-Level Dependencies

The definition of the job-level dependencies over the task chains, according to the work provided by Becker et al. [10], is of major importance for the satisfaction of the system's end-to-end requirements. Of course, the importance of job-level dependencies cannot be separated from the importance of defining reasonable values for data age delays since the job-level dependencies only ensure that certain task instances precede instances of other tasks. Therefore, to create a correct arrangement of the tasks and generate a valid schedule, the mindful definition of all the requirements is required. In the end, if it is possible to successfully generate the job-level dependencies, it means that all the data age constraints can be met, but does not ensure the satisfaction of other timing requirements which need to be considered. The satisfaction of end-to-end delays is usually eased in systems with higher number of processors since more cores provide more processing resources. Especially, that is the case in situations where the tasks contained in a chain are legitimately allocated to the cores. Since the migration of the tasks from one core to the other is not allowed in this work, the problem may arise if the task-to-core mapping is not carried out in a correct way. In an ideal situation, when each task has its own core to execute on, the satisfaction of end-to-end requirements is trivial.

6.2.6 Constraint Programming Formulation

In order to use a constraint programming based approach for the schedule generation, a set of variables and constraints must be defined. This demands a complete translation of the application model into the CP domain, where each model variable is translated into a suitable CP variable and each relation between the variables is translated into a CP constraint. Hence, this section covers the procedure with all its comprising steps that is required in order to produce an accurate representation of the model in the CP domain.

Decision Variables: As for any optimization problem, a set of decision variables must be defined. The goal of this optimization is to find a valid schedule for each task's jobs so that no timing constraints or deadlines are violated. Hence, the decision variables in the this CP formulation, are the conditional interval variables that are generated for each job in the model.

Each job $\tau_{i,j}$ is represented by a conditional-time interval given with the tuple $\{s,\ e,\ l\}$, where s and e are the *start* and *end* times, respectively, and l the length (duration) of the interval. For each interval variable, to be a true representation of a job, the following must be satisfied:

$$\begin{aligned} s &\in [r_{i,j},\ d_{i,j} - C_i] \\ e &\in [r_{i,j} + C_i,\ d_{i,j}] \\ l &\in [C_i,\ T_i] \end{aligned}$$

Namely, the interval cannot start before the jobs release time $r_{i,j}$ and it must end latest at its deadline $d_{i,j}$. All jobs that are generated within one hyperperiod of the task set $\mathrm{HP}(\Gamma)$ are grouped into one global job set denoted by J. Since there are two cores, on which no two job intervals are allowed to overlap, it is needed to group the job intervals assigned to each core into two disjunctive sets. Therefore, each job interval is added to a set J_p according to:

$$J_p = \{\tau_{i,j} \mid p_i = p\}, \quad p \in \{0,\ 1\},$$

where J_p is the set of all jobs executing on core p and $J = J_0 \cup J_1$.

Constraints: The core of any CP problem is the formulation of a clear set of constraints. To implement the scheduling approach analyzed in this work, three types of constraints must be defined. These include basic scheduling constraints, global memory access constraints due to the shared bus and also the job-level dependency constraints which are needed in order to enforce correct end-to-end timings.

The basic timing requirement for all tasks in the system is to finish execution before their deadline. Each job that is executing on one of the cores is modeled as a time-interval belonging to one of the sets J_p. To produce a feasible schedule with regard to only the task deadlines and physical constraints of each core, two basic conditions must be met. The first condition is to bound each job interval so that each job starts after its release time and finishes latest at its deadline. To achieve this, the minimum and maximum start and end times of each interval must be constrained with:

$$\begin{aligned} \mathrm{start}_{\min}(\tau_{i,j}) &= r_{i,j}, \\ \mathrm{start}_{\max}(\tau_{i,j}) &= d_{i,j} - C_i, \\ \mathrm{end}_{\min}(\tau_{i,j}) &= r_{i,j} + C_i, \\ \mathrm{end}_{\max}(\tau_{i,j}) &= d_{i,j}, \\ \mathrm{length}_{\min}(\tau_{i,j}) &= C_i, \\ \mathrm{length}_{\max}(\tau_{i,j}) &= T_i. \end{aligned}$$

Further, each core allows only one job to execute at a single point in time, meaning that the intervals of the jobs assigned to a single core cannot overlap. Therefore, for each core an additional constraint is defined in order to fulfill this requirement:

$$\mathrm{noOverlap}(J_p), \quad \forall p \in \{0, 1\}.$$

The described constraints will secure a correct relative ordering of the jobs on each core, but they are not sufficient to produce a feasible schedule since the jobs' memory accesses through the shared bus are not taken into account. Bus access is exclusive between all tasks and thus a constraint must be defined which ensures that there is only one memory access at a time. Two jobs can only conflict each other if their intervals of *existence* (i.e. the time between the release and the deadline of a job) overlap. For two jobs, $\tau_{i,j}$ and $\tau_{k,l}$, to overlap the following condition must be true:

$$r_{i,j} < d_{k,l} \ \wedge \ r_{k,l} < d_{i,j}.$$

If the jobs are overlapping, then they must start with a relative offset of at least $C_{i,R}$ or $C_{k,R}$ time units, respectively, so that the jobs' *Read* phases don't interfere with each other. This can be defined with the following constraint:

$$\begin{gathered}\text{startBeforeStart}\big(\tau_{i,j},\ \tau_{k,l},\ C_{i,R}\big) \ \vee \ \text{startBeforeStart}\big(\tau_{k,l},\ \tau_{i,j},\ C_{k,R}\big), \\ \forall \tau_{i,j}, \tau_{k,l} \in J, \quad \tau_{k,l} \neq \tau_{i,j}, \quad r_{i,j} < d_{k,l} \ \wedge \ r_{k,l} < d_{i,j}.\end{gathered}$$

Similarly, the jobs must end with a relative offset of at least $C_{i,W}$ or $C_{k,W}$ time units, respectively, so that the jobs' *Write* phases don't overlap:

$$\begin{gathered}\text{endBeforeEnd}\big(\tau_{i,j},\ \tau_{k,l},\ C_{k,W}\big) \ \vee \ \text{endBeforeEnd}\big(\tau_{k,l},\ \tau_{i,j},\ C_{i,W}\big), \\ \forall \tau_{i,j}, \tau_{k,l} \in J, \quad \tau_{k,l} \neq \tau_{i,j}, \quad r_{i,j} < d_{k,l} \ \wedge \ r_{k,l} < d_{i,j}.\end{gathered}$$

Lastly, the jobs' *Read* and *Write* phases cannot interfere and thus one additional constraint must be specified regarding the access to the shared memory. To omit the overlapping of the *Read* phase of job $\tau_{i,j}$ and the *Write* phase of job $\tau_{k,l}$, the following constraint is applied:

$$\begin{gathered}\text{startBeforeEnd}\big(\tau_{i,j},\ \tau_{k,l},\ C_{i,R}{+}C_{k,W}\big) \ \vee \ \text{endBeforeStart}\big(\tau_{k,l},\ \tau_{i,j},\ 0\big), \\ \forall \tau_{i,j}, \tau_{k,l} \in J, \quad \tau_{k,l} \neq \tau_{i,j}, \quad r_{i,j} < d_{k,l} \ \wedge \ r_{k,l} < d_{i,j}.\end{gathered}$$

If the above constraints are defined as described, a schedule can be produced that is feasible in regard to the basic task timing and shared memory access restrictions. On the other hand, the end-to-end timing constraints are in that case not considered. To produce a complete schedule where the fulfillment of each end-to-end requirement is guaranteed, the generated job-level dependencies need to be translated into precedence constraints and as such added to the CP formulation. Hence, for each job-level dependency $\Phi : \tau_i \xrightarrow{(j,l)} \tau_k$ that has been generated in the model, a set of precedence constraints is defined. The constraints are defined over one hyperperiod of the job-level dependency, i.e. $\text{HP}(\Phi) = \text{LCM}(T_i,\ T_j)$, and repeated for each consecutive hyperperiod to account for the repeating jobs in the whole task-set hyperperiod $\text{HP}(\Gamma)$. The constraints that are defined for each dependency are:

$$\begin{gathered}\text{endBeforeStart}\big(\tau_{i,y},\ \tau_{k,z}\big), \quad y = j + (x-1)\frac{\text{HP}(\Phi)}{T_i}, \quad z = l + (x-1)\frac{\text{HP}(\Phi)}{T_k}, \\ \forall \Phi : \tau_i \xrightarrow{(j,l)} \tau_k, \quad \forall x \in \left[1,\ \frac{\text{HP}(\Gamma)}{\text{HP}(\Phi)}\right].\end{gathered}$$

By including these constraints, the produced schedule is guaranteed to fulfill any end-to-end requirements that are imposed on the application through the defined task chains and age constraints that are associated with them.

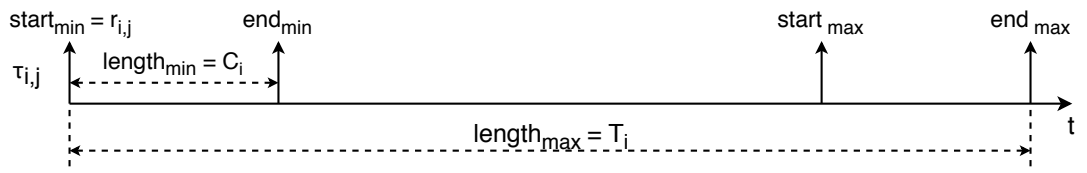

Figure 10: Definition of the interval of an instance.

6.2.7 Solving the Constraint Satisfaction Problem

After all decision variables and constraints are correctly formulated, the *constraint satisfactory problem* is attempted to be solved by the CP solver. During this process, the solver explores all the possible paths that exist in the variable space until a feasible solution which satisfies all the defined constraints is found. The duration of this procedure time highly depends on the number of jobs and job-level dependencies that are specified in the model. Each job introduces a new interval variable, and thus extends the variable space. Furthermore, with each additional job or dependency, several constraints are introduced, making the evaluation of the criteria more complex and time-consuming. To reduce the solving times for each model, this method does not incorporate any minimization or maximization criteria, meaning that the first solution that is found to satisfy all constraints, is accepted as the optimal one. However, in some future works, the method could be extended with a cost function to minimize the end-to-end delays.

After the solver has found a viable schedule, each job in the initial application model inherits the *start* and *end* times of their interval variable counterpart in the solution of the CSP. Therein, the schedule is complete and can be further processed or put into action on an actual execution platform.

7 Limitations

Throughout the process of the method implementation in this thesis, several design choices had to be made. Each design choice may have introduced several direct or indirect limitations into the applicability and performance of the proposed method. Therefore, it is important to address these choices, along with their possible implications in the final outcome of the method implementation.

The systems considered in this work are narrowed down to dual-core processors with a shared system bus and caches that are private to each core. Hence, only the system bus is assumed to be a source of possible inter-core interferences, which restricts the applicability of the presented approach to abstract systems and renders it inadequate for real industrial systems. Furthermore, the approach assumes a phased model for the task execution in order to efficiently control tasks' accesses to the shared memory. Apart from the memory access requests, no other sources of bus congestion are considered. This represents a serious limitation, since many microcontroller systems contain various on-chip modules, such as *Direct Memory Access* (DMA) and *Input/Output* (I/O) controllers, which also share the system bus and thus can be the source of further congestion. However, the proposed method rather serves as a proof of concept and with some additional generalization, it can serve as a good base for further development. For instance, although the approach was designed for a specific subset of multi-core systems, it is possible to adapt it to systems with higher core counts by adjusting a set of parameters. This however, may open the question of how the approach, in its current state, scales in terms of schedulability and computational time with regard to the number of cores in the analyzed system. But again, this can be a possible direction for further research.

There are several issues that can appear and endanger the scheduling process. An important one is that the proposed method assumes that task-to-core allocation is static and done beforehand, meaning that the user has to manually map each task to a core for execution. This, obviously, introduces the risk of non-optimal or unbalanced core utilizations which can impact the success rate of finding a feasible schedule. This can be aided by utilizing different optimization techniques or heuristics. The underlying optimization problem is as complex as the *bin-packing* problem which is an NP-hard problem [5]. The proposed approach, however, does not implement any mapping utility and thus expects from the user to provide it *a priori*.

Additionally, it is assumed that each task's code is pre-fetched and available in each core's local memory, so that the only data that is being read and/or written during the memory phases of each task are the labels needed for the inter-task communication. This may not always be the case, as the total footprint of the tasks allocated to a particular core may exceed its available local memory. In those cases, it is more appropriate to fetch a single task's code during run-time, prior to its execution. This reduces the amount of occupied local memory and also allows for dynamic task-to-core allocation, but prolongs the duration of the task's memory access intervals. For the sake of simplicity, this thesis omitted any considerations towards this idea.

8 Evaluation

In the interest of evaluating the developed scheduling approach as thoroughly as possible, a set of test cases must be designed. Performance aspects, such as scheduling success rate and solving complexity, are of main interest for this thesis and are investigated through an experimental setup. The evaluation of the implemented scheduling method is carried out in three steps whose detailed descriptions are provided below. Hence, this section presents how the experiment is designed and conducted, but it also contains the analysis of the results produced by the evaluation and it presents some basic conclusions that can be drawn from them.

8.1 Design of a Synthetic Test Case Set

The offline schedules that are produced within this thesis are mainly aimed at real-world automotive embedded applications. Currently, there is a deficiency in publicly available application examples mainly due to intellectual property (IP) protection. Although exact applications are not provided, the implementations of many of them are usually based on common patterns and practices. Thereby, many research group resort to generating test models which are random, but still adhere to those patterns. By analyzing some common practices in the automotive industry, Kramer et al. were able to extract some general information about real-world application models based on which they proposed a method which supports the generation of realistic, but IP free benchmarks. In the associated paper [33], they define a set of application characteristics which are sufficiently abstract to not reveal IP, but nevertheless allow creating realistic application benchmarks from it. The test cases that are utilized in this thesis are generated based on the characteristics and the method presented in the paper.

To produce test cases that realistically reflect real-world applications, clear guidelines must be set. The procedure of generating a random, yet realistic application model, is depicted as a flow chart in Figure 11.

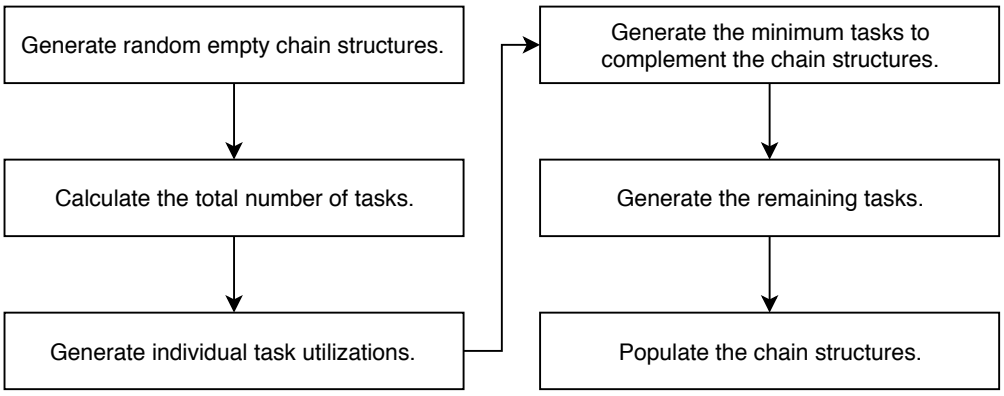

Figure 11: Overview of the model generation procedure.

The core problem of generating a valid test model is related to setting up a random task set to match a random set of chains. In order to deal with this problem, the chains are generated first. Namely, for each chain an empty structure is produced with a random number of activation patterns. The probability distribution of the number of involved activation patterns per each chain is given in Table 2.

Table 2: Distribution of the number of involved activation patterns per each chain.

Involved activation patterns	**Probability**
1	70%
2	20%
3	10%

Each activation pattern consists of two to five tasks which share the same period. Consequently, the length of a random task chain can vary from two to fifteen tasks. The periods of the patterns are randomly selected from the set $T \in \{1 \text{ ms}, 2 \text{ ms}, 5 \text{ ms}, 10 \text{ ms}, 20 \text{ ms}, 50 \text{ ms}, 100 \text{ ms}, 200 \text{ ms}\}$, which is a subset of the task periods commonly found in automotive applications [33]. The distribution of the number of tasks per pattern is presented in Table 3, while the distribution of the pattern periods is given in Table 4. An example chain, where seven tasks are grouped into three activation patterns, is depicted in Figure 12.

Table 3: Distribution of the number of tasks per each activation pattern.

Number of tasks	**Probability**
2	30%
3	40%
4	20%
5	10%

Table 4: Distribution of the task periods.

Task period	**Probability**
1 ms	4%
2 ms	3%
5 ms	3%
10 ms	30%
20 ms	30%
50 ms	4%
100 ms	25%
200 ms	2%

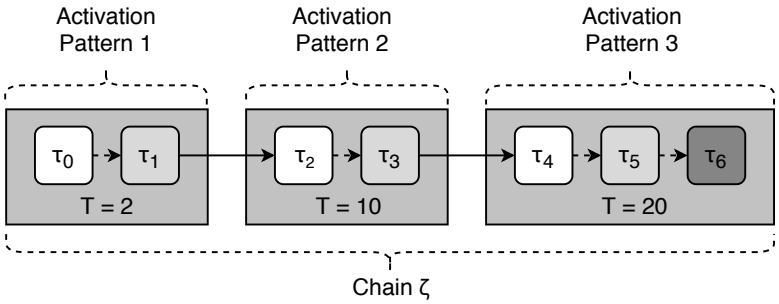

Figure 12: A possible chain structure: seven tasks distributed among three activation patterns with defined periods.

Another very important aspect when defining task chains is the relation of the sampling rates of the comprising tasks, as communication based on periodic sampling and shared variables can result in large end-to-end latencies. Therefore, communicating task periods within a chain must be carefully selected. The allowed period pairs are presented in Table 5, where each row and column refer to the period of the sending and receiving pattern, respectively, and allowed communications are marked with a dark cell at the intersection. For instance, an activation pattern with period $T_2 = 5\ ms$ is not allowed to receive data from an activation pattern with period $T_1 = 2\ ms$, while on the other hand, an activation pattern with period $T_3 = 10\ ms$ is allowed to.

Table 5: Allowed communication pairs among activation patterns. Each row and column refer to the period of the sending and receiving activation pattern, respectively.

Period	**1 ms**	**2 ms**	**5 ms**	**10 ms**	**20 ms**	**50 ms**	**100 ms**	**200 ms**
1 ms								
2 ms								
5 ms								
10 ms								
20 ms								
50 ms								
100 ms								
200 ms								

Lastly, to complete the chain structures, the age constraints of each individual chain must be specified. In real applications, the value of these constraints depends highly on the type and the character of the data that is being propagated through the chain, therefore, the selection of appropriate values for these constraints becomes a non-trivial task. Within this thesis, these values are chosen based on intuition and are randomly generated according to the formula:

$$\eta = \mathrm{HP}(\lambda) \cdot \mathrm{random}(1.5,\ 2), \quad \forall \zeta = (\lambda,\ \eta), \tag{8}$$

where $\mathrm{HP}(\lambda) = \mathrm{LCM}(T_i \mid T_i \in \lambda)$ is the hyperperiod of the chain and the function $\mathrm{random}(a,\ b)$ returns a random real number $c \in [a,\ b]$ with a uniform distribution. Thus, the maximum data age of each generated chain can lie between 1.5 and 2 of its hyperperiod. This bounds allow for the specification of artificial constraints, whose individual strictnesses can vary in a realistic range.

After the chain structures are completed, the total number of tasks that need to be generated is chosen. This value is also selected through a random process with some intuitive reasoning. The adoption of this value follows the formula:

$$n = \min\big(6,\ \lceil m \cdot \mathrm{random}(1.5,\ 2) \rceil\big), \tag{9}$$

where m is the number of tasks needed to fully populate all chain structures without repeating. By applying this formula, it is guaranteed that the task set will contain both independent tasks and tasks that are connected through some chains. In case that the model contains no chains, then $m = 0$, and the minimum number of six tasks is chosen.

Further, for the selected number of tasks, individual task utilizations are generated. As the assumed execution platform model contains two cores, the total CPU utilization can vary from 0 to 200%. Additionally, the implemented model generation process assumes that each core is utilized to the same extent ($U_0 = U_1$) and thereby distributes the tasks evenly among the cores during the task-to-core mapping, meaning that each core is assigned $\frac{n}{2}$ of the tasks. For each core a set of $\frac{n}{2}$ individual task utilizations is generated, based on the *UUniFast* algorithm [34].

When creating the task set, the m tasks are generated first, since they need to ensure that there is at least one combination of tasks to populate the chain structures. The periods of the tasks must comply with the underlying chain structures. Only after these tasks are successfully created, the task set is filled with the other $n - m$ independent tasks. The periods of the independent tasks are randomly generated according to Table 4. To generate the WCET of a task, an unused utilization is randomly selected from the two task utilization sets and multiplied by the period of the task. Lastly, for each task a number of *Read* and *Write* bytes is defined, in order to generate the memory access times. Each task can read and write a random number of bytes (the byte size distribution is shown in Table 6), respectively, and thus its WCET includes the two associated memory accesses, whose duration is calculated as:

$$C_R = \frac{B_R}{f_{CLK} \cdot v_{BPS}}, \tag{10}$$

$$C_W = \frac{B_W}{f_{CLK} \cdot v_{BPS}}, \tag{11}$$

where B_R is the number of *Read* bytes, B_W the number of *Write* bytes, f_{CLK} the clock frequency and v_{BPS} the number of bytes that can be accessed for the duration of a single clock cycle.

Table 6: Distribution of R/W label sizes.

Label size	**Probability**
1 byte	35%
2 bytes	49%
4 bytes	13%
5-8 bytes	0.8%
9-16 bytes	1.3%
17-32 bytes	0.5%
33-64 bytes	0.2%
65-128 bytes	0.2%

The last step in the model generation procedure, is to populate the empty chain structures with random tasks from the task set which match the defined periods of the activation patterns. The tasks are selected randomly from the whole task set, and not from the first m tasks, with the intention of enabling some tasks to be part of multiple chains and thus additionally increasing the complexity of the generated model.

By following the presented procedure, a high degree of randomness is introduced into the model generation, nevertheless, the variability of the random factors is realistically bounded. Hence, the produced models are very good abstractions of many real-world applications.

Furthermore, to better understand the results of the evaluation, it is of utter importance, to observe what are the characteristics of the generated models in regard to the number of chains and the total utilization of the system. Figure 13 shows how the number of tasks generated per model

changes depending on the given system utilization and the given number of chains, while Figure 14 depicts a similar dependency, but for the number of jobs in each model. As it can be seen, both the average task number and the average job number stay relatively constant regardless of the increase in utilization, due to the *UUniFast* algorithm which divides the total available system utilization among a fixed number of tasks. Furthermore, since the size of the task set is directly proportional to the number of chains, as the chain number increases, the number of tasks (and jobs) increases with a fixed step per chain.

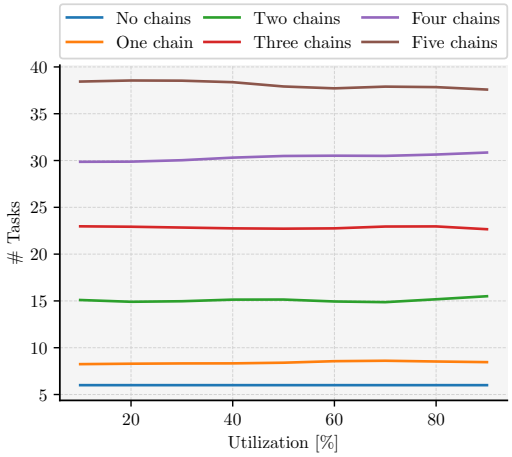

Figure 13: Average number of tasks per model with respect to the total system utilization.

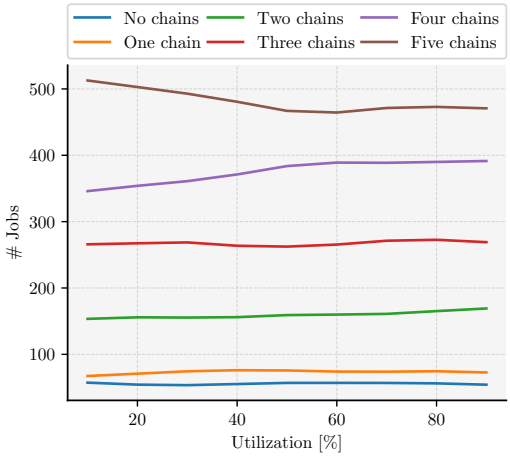

Figure 14: Average number of jobs per model with respect to the total system utilization.

8.2 End-to-end Schedulability Rate

The second step of the conducted evaluation is in regard to the scheduling success rate. Schedulability rate, in this case, refers to the percentage value of the models that the method can successfully schedule with regard to not only the task deadline constraints, but also in regard to all eventual end-to-end deadlines. The schedulability status of a single model is binary, meaning that it is either schedulable if a valid schedule can be produced for it or non-schedulable if not. Also, it is important to note, that by utilizing the generated job-level dependencies as constraints, any produced schedule is guaranteed to fulfill all end-to-end timing deadlines, as any invalid latency path is eliminated by design.

The method was tested on a synthetic test case set where each test case has a variable degree of utilization of the system's processing resources and a variable number of involved task chains. The system utilization is allowed to vary from 20 to 180%, with an additional condition that the utilization is evenly split among the two cores, consequently meaning that the utilization of each core can synchronously vary from 10 to 90%. Thus, it can be tested how the method scales in terms of increasing density of the underlying application requirements. Also, to test how the performance of the method reacts to an increase in the amount of end-to-end timing constraints, the number of involved task chains is varied from zero to five chains. Since the solve times of each model are varying, it is mandatory to limit the maximum calculation time that can be allocated to each model. With this in mind, the maximum solving time for each model is specified as $t_{max} = 1\ min$. It is important to note, that this value has a great impact on the outcome of the experiment, since each model for which a schedule can not be found within the defined time limitation is deemed non-schedulable during the result analysis. Hence, increasing the maximum solve time would provide more realistic results. However, it would also increase the overall time needed for the experiment. Due to a very limited time frame that was available for the experiment, it was decided to proceed with the maximum solve time defined above.

Furthermore, for each combination of the utilization points and chain counts, 200 test models are created in accordance to the method presented in Subsection 8.1. For each test model, job-

level dependencies are generated and a schedule is attempted to be found. The results of this process are written to a log file. After all 200 sample are evaluated, the success rate of the scheduling is determined for the current utilization point and chain count, by dividing the number of successfully found schedules with the total number of samples. Figure 15 shows how the success rate changes with an increase in system utilization. Each curve is generated for an individual number of chains. As shown in the graph, the schedulability ratio monotonically decreases with the increase in utilization. With each additional chain in the system, the schedulability drops further, and as the utilization approaches its upper limit, all curves converge to zero. The reason the schedulability decreases, lies in the fact that with the higher utilization degrees, the scheduler has less available CPU time to schedule certain activities. With the increase in utilization, due to the way how the models are generated, the number of tasks stays relatively constant, but their WCET increases and thus renders the scheduling attempt more prone to failure. Also, each additional chain imposes additional end-to-end constraints with potentially some tasks being part of several chains. This produces very complex sets of precedence constraints among the jobs, which have a drastic impact on the schedulability ratio.

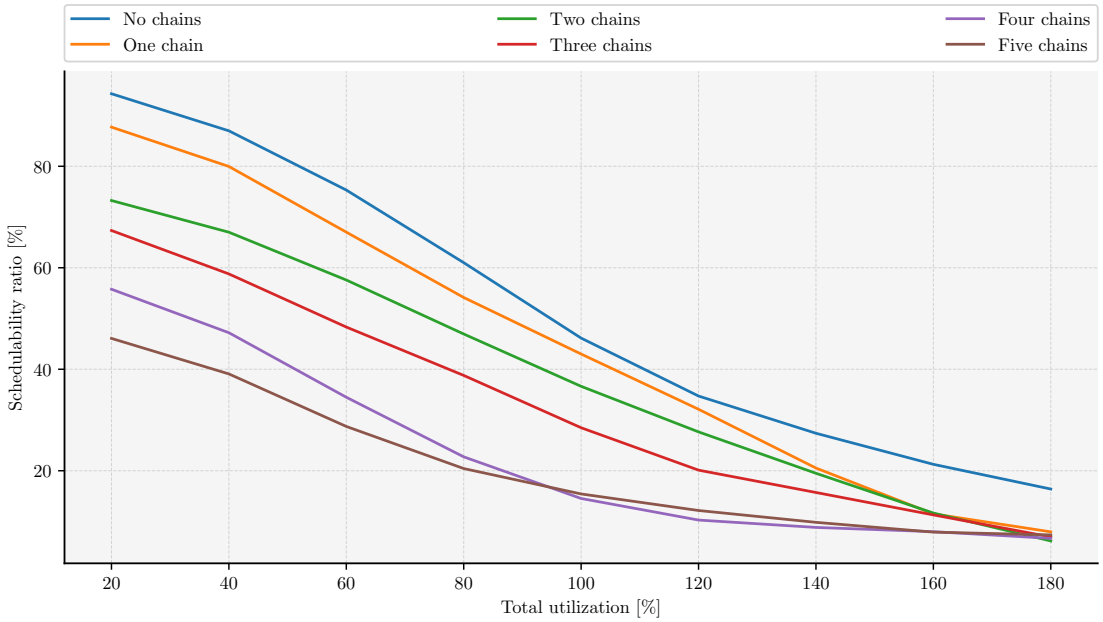

Figure 15: Success rate of the scheduling method with regard to the change in system utilization and number of involved task chains.

Although a decrease in schedulability is expected to occur, it is however, very interesting that it starts to decrease significantly early on, even at lower utilization percentages. One plausible reason for this effect can be the fact, that the task-to-core mapping procedure is conducted randomly, instead of allocating the tasks to each core in an optimal manner. A more controlled approach to task allocation could potentially illuminate the reason behind the observable discrepancy in performance.

8.3 Computational Complexity and Solving Time

Another very important aspect of the performance analysis of the proposed method is the time required to find an optimal schedule for a model. The total time consists of two components, that is, the time needed to preprocess the initial model and the time that the optimization engine requires to find a feasible schedule. The preprocessing of the model includes the generation of job-level dependencies to enforce end-to-end timing constraints and the translation of the model into the CP domain, in a form that is appropriate for the CP solver.

In order to measure the temporal characteristics of the implemented method, again 200 test samples are created, for each utilization point and for each individual number of involved chains. The results obtained for each of the individual 200 models are averaged into mean values with the aim of producing a better representation of the outcomes. The experiment was conducted using an *Intel* i7 8550U CPU with four cores at 1.8 GHz and 16 GB of RAM.

Before any timing results are presented, it is beneficial to understand to which extent a varying system utilization and the number of task chains impact the number of generated job-level dependencies and subsequently, the number of constraints introduced into the CSP. Figure 16 shows that the average number of generated job-level dependencies follows a linear increase trend with respect to the increasing number of specified chains. As it can be seen, for each specified chain, on average, three additional job-level dependencies are created. Thus, naturally, it is to expect that the solving complexity is increasing due to the additional constraints that are introduced into the CPO model by each new job-level dependency. This can be confirmed by observing Figure 17, from where it can be seen that the average number of constraints in the CSP increases exponentially per specified chain. This number represents the total number of constraints that are present in the CSP, which includes the basic scheduling constraints, memory access constraints and job-level dependency constraints. In both figures, it is notable that system utilization does not have a significant impact on the number of dependencies and constraints.

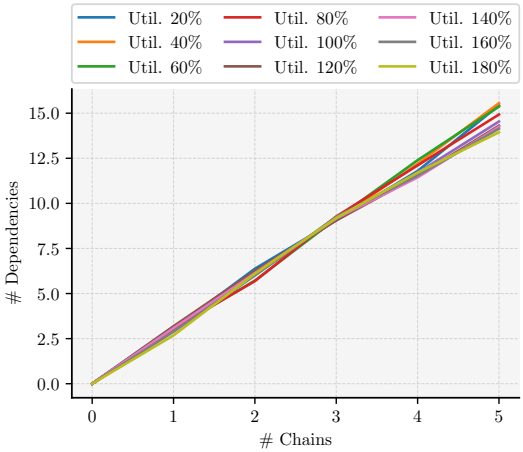

Figure 16: Average number of job-level dependencies generated per model with respect to the number of specified chains.

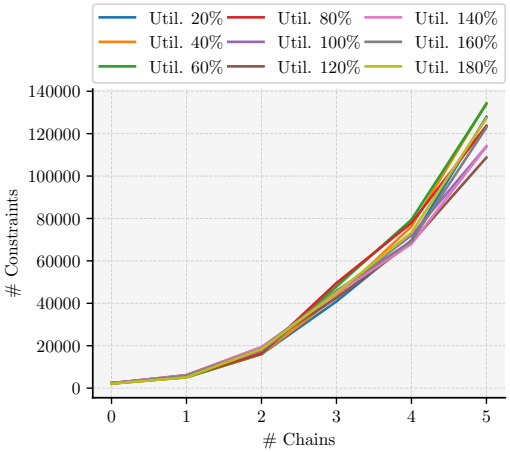

Figure 17: Average number of constraints per model with respect to the number of specified chains.

Furthermore, Figure 18 depicts the average time needed to generate the job-level dependencies in relation to the total system utilization, while the same value, but with respect to the number of chains, is presented in Figure 19. In the first graph, the value fluctuates irregularly, which can be attributed to the random factor of the model generation process. However, the second graph shows a clear indication that the time needed for the dependency generation has the overall tendency to exponentially increase with each additional task chain. This is in line with the previous observations, since each chain intensifies the complexity of the model, both in terms of the number of tasks/jobs that are generated and the number of end-to-end timing constraints that are introduced.

The average solving time of the CSP in presented in figures 20 and 21, with respect to the system utilization and number of chains, respectively. As shown, the solve time start to follow the increasing system utilization in an almost linear manner, up to a point where it starts to level out to a constant value. By comparing the schedulability graph in Figure 15 and the graph presented in Figure 20, it can be noted that the solving time starts to deflect from the initial linear form approximately at the same point ($U \approx 80\%$) at which the schedulability ratio starts to decrease significantly. The lower solve times at the lower system utilizations can be explained by the fact that there exist more feasible solutions in the search space of the CSP, due to the lower WCETs of the involved tasks which are less interfering with each other, making it easier to arrange the jobs

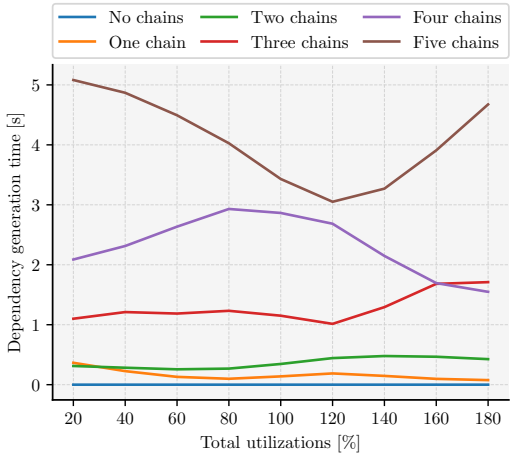

Figure 18: Average job-level dependency generation time per model with respect to the total system utilization.

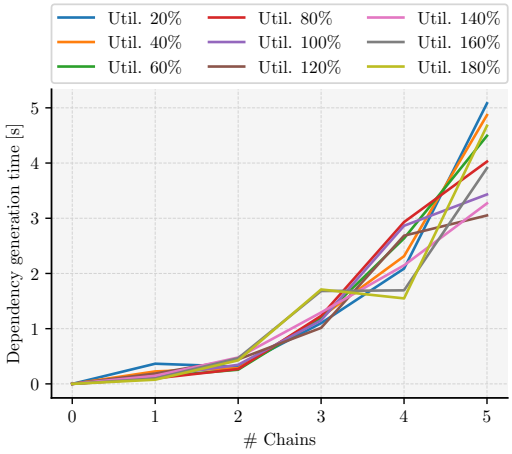

Figure 19: Average job-level dependency generation time per model with respect to the number of specified chains.

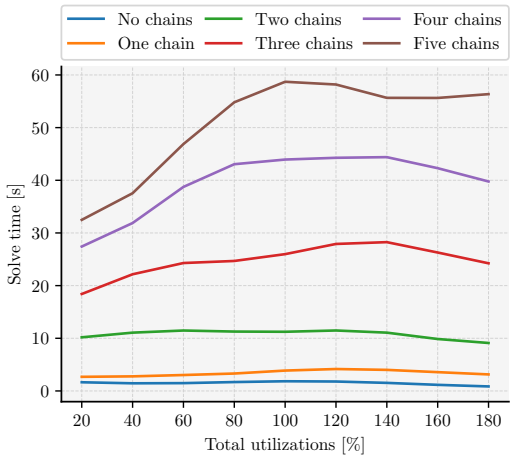

Figure 20: Average CSP solving time per model with respect to the total system utilization.

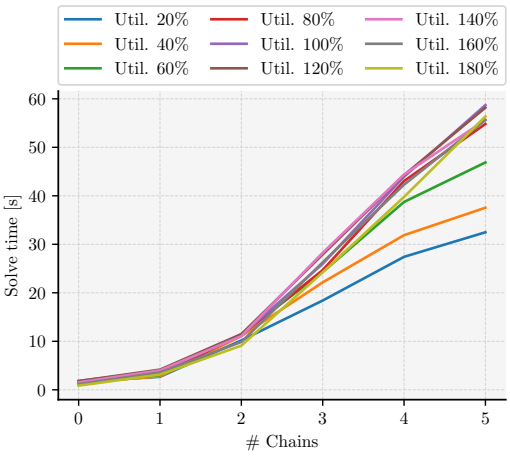

Figure 21: Average CSP solving time per model with respect to the number of specified chains.

associated with them. Since the CSP problem does not have have cost function specified, the first feasible schedule that is found is accepted as optimal, and thus the solve time stays low. However, as the utilization increases, so do the task WCETs, and the set of feasible solutions narrows down, increasing the search time. The solve time increases to the point where the maximum allowed time limit is reached ($t_{max} = 1\ min$). This can be observed in Figure 20, where the curve representing the average solve time of the models that contain five task chains, reaches the maximum solve time at around 100% system utilization.

8.4 Discussion

This subsection contains discussions on the subjects covered in this thesis and the results obtained during the evaluation phase. Specific observations and conclusions are presented with regard to the outcome of the conducted experiment, together with an analysis of the factors that could potentially threaten its validity.

8.4.1 Scheduling Performance

In Section 8.2, it is shown that the success rate of finding a feasible schedule depends both on the total system utilization of the task set and the number of chains specified among the tasks. Considering the way how the experiment is designed, the number of task chains proved to have a greater impact on the outcome. With each additional task chain, the schedulability ratio drops for a significant amount. This is attributable to many factors. Firstly, an increasing number of chains implies a increasing number of tasks in the task set of the designed model. Each task implicated multiple jobs of which each translates into an additional CP interval variable with its associated memory access and scheduling constraints. Additionally, each chain introduces a new end-to-end timing constraint which must be fulfilled by all the latency paths of the chain. Lastly, but probably most importantly, how the models are designed, it is possible that several tasks become part of multiple chains. The probability and degree of this *task entanglement* increase together with the number of chains and thus it imposes very complex constraints upon the job intervals, which become extremely hard to meet. This thesis does not investigate this relation in full detail, but based on the outcome, some overall conclusions can be pointed out. In order to better understand the actual extent of this adverse effect, further research in this direction is of interest.

The task set's system utilization has a slightly weaker, but still weighty effect on the success rate. As shown in Figure 15, with the growing utilization it becomes more and more difficult to schedule the task set. This is due to the increasing WCETs which makes it more difficult for the scheduler to sufficiently allocate the processor time required for each task job. However, as it is pointed out earlier, the success rate starts to drop early on. Although, multiple reasons could be suspected, the most presumable one is the non-optimal task-to-core mapping. In the experiment, the tasks are randomly allocated to each core, with the only criterion of having both cores equally loaded.

One aspect that was not addressed by this thesis, which can be an additional metric on the performance of the approach, is how the method scales with regard to the size of the data labels being read/written. In the employed experiment design, the sizes of the labels are randomly selected from a distribution table. By incrementing the amount of data that each task can read or write, the associated memory access phases become longer and so the bus utilization increases, making it more difficult to produce a schedule where there are no overlapping accesses to the system bus. Although such investigations would allow for a better evaluation of the introduced phased execution model, they would increase the overall time needed for the experiment, which for the duration of this thesis, was the most critical resource.

8.4.2 Temporal Performance

This method is aimed at applications whose dimensions and scale can vary and due to that, the amount of time needed to determine a schedule for such systems can become very large. Therefore, the temporal characteristics of the method must be analyzed accordingly.

Based on the timing results presented in Section 8.3, some important observations can be stated regarding the generation of the job-level dependencies and the solving of the constraint satisfaction problem. The time needed to perform both steps depends on a multitude of factors, which among others, include the number of tasks in the model, the number of jobs, the number of chains, system utilization and the number of job-level dependencies. It is very difficult to determine to which extent each individual parameter affects the final solve or generation time. The design of the experiment specifies only two independent variables: the number of chains and the percentage of system utilization. Therefore, the timing performance is analyzed with respect to these two values. It is shown that both generation and solve time are strongly affected by the number of chains. This, again, can be explained by the fact that all the other relevant variables are increasing with a higher chain count, such as the number of tasks, jobs or job-level dependencies. Each of them contribute to the total number of variables and constraints that are added to the CSP and thus increase the time needed to find a viable solution.

On the other side, system utilization proved to have relatively no impact on the timing of the generation of the dependencies. However, when solving time is considered, a linear relation at lower utilizations is evident, while for higher utilizations the maximum solving time is reached and the initial linearity is violated. As it is mentioned before, it is very difficult to determine why

utilization affects the solve time in this manner, but a possible cause can be pointed out. At lower utilizations, there exist more solutions to the CSP as the scheduler has more available processor time on disposal. Since the solution density is increased, the time needed to find of the solutions is lower.

8.4.3 Threats to Validity

The most challenging aspect of this thesis was the vast amount of time that has to be allocated to the evaluation phase. The initial design of the approach was centered on different scheduling techniques, such as partitioned scheduling with a TDMA arbitration policy for the system bus, but due to difficulties regarding the implementation, it had to be revised to its current form. Therefore, in the limited time frame provided for the thesis, very little time was left to allocate for the evaluation. Some of the produced results are observably affected by this limitation. During the evaluation, for each test sample a maximum solve time of $t_{max} = 1\ min$ was allocated, which for more complex models is definitely not sufficient. Each model whose schedule cannot be found within the allowed solving time is deemed non-schedulable within the analysis, and this degrades the quality of the results. For more realistic results, a larger time limit must be set. Additionally, the number of test model per each utilization point and chain count was set to 200. Increasing this parameter would ensure that more accurate mean values are produced. However, the obtained results provide a foundation for some basic conclusions.

Lastly, in order to provide a large amount of test cases, a synthetic set of test cases was generated. Each test case was created randomly adhering to a set of design rules presented in Section 8.1. The rules are defined in order to enforce a better representation of real-world applications among the generated models. Although the test cases are very detailed abstractions of applications commonly found in the industry, the randomness factor still plays a huge role in their generation. This can cause very complex models to be created, which otherwise, could never occur in a real application. Nevertheless, with some additional real-world use-cases, the industrial applicability of the procedure could be verified.

9 Conclusion

The work presented in this thesis describes the design, implementation and evaluation of an approach to scheduling of multi-core systems where the main goal is to optimize inter-core data propagation delays. For that purpose different techniques are investigated, and the assistance of several is employed, such as the implementation of job-level dependencies to enforce end-to-end predictability, the utilization of a phased execution model to provide temporal isolation of the tasks and the assistance of constraint programming to guarantee that the produced schedule fulfills all constraints and requirements. Initially, the system model is created and subsequently formulated as a constraint satisfaction problem. In the end, a fully defined offline schedule is produced to enforce all specified task and end-to-end deadlines.

The main focus of the work is set on inter-core propagation delays which are observed as end-to-end delays representing the maximum allowed latency of task chains. The applied method guarantees that each produced schedule, by design, satisfies all data age constraints that are defined in the system. The approach, at the very beginning of the scheduling process, eliminates all non-valid propagation paths and if in the continuation of the process a feasible schedule is found, it guarantees that all end-to-end delays are less than the defined data age constraints. That is implied by the main characteristic of utilizing job-level dependencies, since all data age constraints are certainly met if a successful generation of job-level dependencies is possible. Each job-level dependencies can straightforwardly translates to the constraint programming formulation, because each dependency actually represents precedence constraints between two individual task instances.

Another very important aspect of the developed approach is the implementation of a phased model for task execution, which facilitates the temporal isolation of the tasks on each core. The core idea of this model is to divide each task's execution into three phases, two memory phases and one non-memory phase, meaning that all the data needed for a task's execution is read/written only during the memory phases. This way, the congestion on the shared bus can be easily controlled, since all memory accesses of each task are localized at the start and at the end of the task instance (Figure 9).

To successfully find a solution of the multi-core scheduling problem, the application model is translated into the constraint programming domain, as a constraint satisfaction problem, which can be attempted to be solved by a constraint programming solver, in this thesis, the IBM ILOG CP solver.

To evaluate the performance of the developed apparatus, a vast number of synthetic test cases is generated and tested upon, so the relevant metrics, such as scheduling success and computational time, can be determined. As it is expected, the schedulability ratio decreases with the increase of the system utilization and also with the number of the task chains. Also, to show how the system scales in regard to the complexity of the model and the number of constraints, a timing analysis has been conducted. It is shown that the solve time exponentially increases with respect to the number of task chains.

However, it is important to mention that the results are obtained on a synthetic test case set, where the test cases are created randomly. The presented work is not tested with any real-world applications, mainly due to a delayed implementation phase of the thesis, but it is believed that the proposed method can find limited applicability in such use cases.

Several disadvantages are recognized and suggested to be resolved in order to improve and extend the method's applicability (Section 7). The one that can be highlighted as the most important one, refers to the congestion on the shared bus which is assumed to be caused solely by the tasks accessing the shared memory from each core. In accordance with this assumption, all other possible contention sources, mainly other on-chip modules, are neglected.

Alongside, the presented limitations of the implemented method, several techniques are suggested as potential improvements for future works, such as allowance of preemption and inter-core task migration. The effects of these techniques on the current work and the advantages that they can potentially introduce are described in Section 10.

By taking everything mentioned into account, the presented work provides a detailed explanation and answer to the main research question which asks, *how the inter-core data propagation delays can be optimized in multi-core real-time systems that are scheduled using offline scheduling.* The answer to this question is provided through the sections 5 and 6 where the model of the sys-

tem and all its features is provided. All the processes and methods that are utilized are in detail described, so all the needed information and guidelines for the scheduling procedure are provided. Also, an appropriate evaluation is conducted so the relevant information for answering the sub-questions is provided. Insight into the performance and scalability of the proposed approach is delivered in Section 8.

10 Future work

The devised scheduling approach described in this work, in addition to its success rate and efficiency presented in Section 8, does have much room for improvement. Since the main goal of the work is to optimize inter-core data-propagation delays, there are different techniques that can be introduced to the current apparatus so their impact on the observed delays can be tested. Therefore, in the succeeding subsections, several features are selected and described as potential future work upgrades to the base of this work.

10.1 Preemptive Scheduling

The currently developed framework does not allow preemptions which means that one instance of a task cannot be interrupted by another instance of any other task before it finishes its execution. The allowance of preemptions brings with it additional CP constraints that need to be introduced between instances. Therefore, the initial assumption of no preemptions in the system has been made in order to keep the number of constraints as small as possible while satisfying all timing requirements at the same time. The reason for that is to reduce the computational time needed to find feasible schedules, which cannot be negligible, especially in models with a huge number of tasks. The computational complexity of the developed apparatus has been already intensified by the implementation of the phased execution model which divides every task into three phases and introduces additional precedence constraints between them.

On the other hand, by enabling preemptions, the observed data-propagation delays can be significantly reduced and other timing requirements may be easier to satisfy. Which implies that the implementation of preemptions can have a huge impact on schedulability of the systems and consequently on the success rate and efficiency of the approach. Hence, there is an important decision to make: *Is the potentially improved schedulability more important than the time needed to find a feasible schedule and how to trade off between computational time and success rate?* If the decision is to proceed with success rate and to introduce preemptions, the process of defining constraints needs to be carefully conducted, especially when it comes to the specialized *span* constraints [12][13], since more constraints enhance the complexity the constraint satisfaction problem.

10.2 Dynamic Task-to-core Allocation and Task Migration

The allowance of task migrations can have a huge impact on the scheduling process. At this moment, the developed apparatus requires a static task-to-core allocation to be defined by the user. It means that, on one hand, the whole process of scheduling relies on the user's knowledge of the system and its behavior. This can be avoided by implementing a mechanism which allows for dynamic task-to-core mapping and inter-core task migrations during run-time. Besides the fact that automatic assignment of the cores facilitate the operation of the method, it also has a huge impact on the success rate and performance of the scheduling approach. Although the process of the cores assignment during the scheduling process surely extends the computational time, the induced improvement regarding the success rate is more significant.

The prime advantage of the task migration feature can be seen in cases where the utilizations of the cores are not balanced or when one of the cores has less timing requirements to fulfill. In these cases, migrations from core to core can help and facilitate fulfilling all the deadlines. To show how the task migrations can affect the scheduling process, a simple example can be observed. A simple system with two cores is assumed. The first core has two tasks τ_1 and τ_2 assigned to it, and the second core has only one task τ_3. Also, it can be assumed that during the scheduling process all of these three tasks are activated and their deadlines are set, so the observed moment is where the executions need to be performed. In the first case where the task migrations are not allowed, which is provided in Figure 22, tasks τ_1 and τ_2 cannot finish their execution before the marked deadlines which makes the schedule infeasible and the scheduling process unsuccessful. On the other side, the scheduling process that allows migrations of the tasks can fulfill all the timing demands for this case. The obtained schedule is presented in Figure 23 where it can be seen how the task τ_2 migrated from the first core to the second core which makes all the deadline constraints satisfied.

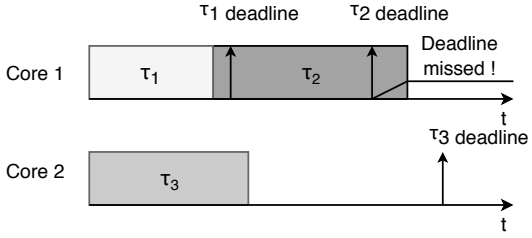

Figure 22: Part of the schedule of an observed task set when task migration is not allowed.

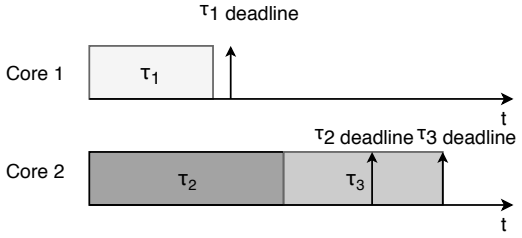

Figure 23: Part of the schedule of an observed task set with allowed task migration.

10.3 Other

Besides the presented techniques above, there are other possible modifications to the developed method that can have a positive affect on its efficiency. For instance, the computational time of the approach hugely depends on the number of the constraints in the system. That is one of the aspects where a notable difference can be made by seeking for the approach that minimizes the number of the constraints. One of the ways to accomplish it is to introduce a TDMA arbitration policy for the access to the shared bus, instead of the implementation of job-level dependencies. This way, the large number of constraints that are introduced by the job-level dependencies would be dismissed. The problem of the approaches that implement a TDMA policy is that the satisfaction of end-to-end deadlines becomes more complicated and demands a special treatment.

Regarding the limitations presented in Section 7, a great majority of them can be avoided in future works. It is possible to take into account all the delays that appear due to the contention on the bus. More precisely, it is possible to produce a schedule that counts in all the delays caused by the modules that require access to the bus.

11 Acknowledgements

We would like to express our very profound gratitude to the supervisor of this thesis, Saad Mubeen, for the continuous guidance and feedback throughout the entire thesis. We also want to extend our gratefulness to Matthias Becker for being a constant source of help and of whose expertise we were able to greatly avail ourselves.

Lastly, but most importantly, we thank our families for providing us with unfailing support and continuous encouragement throughout the years of our studies. This accomplishment would not have been possible without them. *Thank you!*

References

[1] J. F. Nunamaker and M. Chen, "Systems development in information systems research," in *Twenty-Third Annual Hawaii International Conference on System Sciences*, vol. 3, Jan 1990, pp. 631–640 vol.3.

[2] J. A. Stankovic and K. Ramamritham, "What is predictability for real-time systems?" *Real-Time Syst.*, vol. 2, no. 4, pp. 247–254, Oct. 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01995673

[3] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Dec 2007, pp. 49–60.

[4] A. Schranzhofer, J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2010, pp. 215–224.

[5] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "Reducing inter-task interference delay by optimizing bank-to-core mapping," *IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, 2015.

[6] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, vol. 2.

[7] M. Barr, "Embedded Systems Glossary." http://www.netrino.com/Embedded-Systems/Glossary.

[8] J. Martin, *Design of real-time computer systems*, 1967.

[9] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society, 2009.

[10] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug 2016, pp. 159–169.

[11] G. Venter, "Review of optimization techniques," *Encyclopedia of Aerospace Engineering*, 2010.

[12] P. Laborie and J. Rogerie, "Reasoning with conditional time-intervals." in *FLAIRS conference*, 2008, pp. 555–560.

[13] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím, "Reasoning with conditional time-intervals. part ii: An algebraical model for resources," in *Twenty-Second International FLAIRS Conference*, 2009.

[14] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 22:1–22:27, Jan. 2011. [Online]. Available: http://doi.acm.org.ep.bib.mdh.se/10.1145/1880050.1880058

[15] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 741–746.

[16] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 63–72. [Online]. Available: http://doi.acm.org.ep.bib.mdh.se/10.1145/2380356.2380372

[17] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov, "A formal approach to the wcrt analysis of multicore systems with memory contention under phase-structured task sets," *Real-Time Systems*, vol. 50, no. 5, pp. 736–773, Nov 2014. [Online]. Available: https://doi.org/10.1007/s11241-014-9211-y

[18] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, Nov 2011, pp. 1068–1075.

[19] J. Kim, M. Yoon, R. Bradford, and L. Sha, "Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 321–331.

[20] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Syst.*, vol. 48, no. 6, pp. 681–715, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1007/s11241-012-9158-9

[21] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, "Global real-time memory-centric scheduling for multicore systems," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 65, no. 9, 2016.

[22] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-aware multicore wcet analysis through tdma offset bounds," in *2011 23rd Euromicro Conference on Real-Time Systems*, July 2011, pp. 3–12.

[23] ——, "Static analysis of multi-core tdma resource arbitration delays," *Real-Time Systems*, vol. 50, no. 2, pp. 185–229, Mar 2014. [Online]. Available: https://doi.org/10.1007/s11241-013-9189-x

[24] C. Ekelin and J. Jonsson, "Solving embedded system scheduling problems using constraint programming."

[25] C. Ekelin, *An optimization framework for scheduling of embedded real-time systems*. Chalmers University of Technology, 2004.

[26] J.-E. Kim, M.-K. Yoon, S. Im, R. Bradford, and L. Sha, "Optimized scheduling of multi-ima partitions with exclusive region for synchronized real-time multi-core systems," *Department of Computer Science, University of Illinois at Urbana-Champaign*, 2013.

[27] M. F. Tompkins, "Optimization techniques for task allocation and scheduling in distributed multi-agent operations," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[28] W. Puffitsch, E. Noulard, and C. Pagetti, "Off-line mapping of multi-rate dependent task sets to many-core platforms," *Real-Time Syst.*, vol. 51, no. 5, pp. 526–565, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1007/s11241-015-9232-1

[29] M. Becker, S. Mubeen, D. Dasari, M. Behnam, and T. Nolte, "Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2018, pp. 560–567.

[30] "Tutorial: Getting started with Scheduling in CPLEX for Python," https://ibmdecisionoptimization.github.io/tutorials/html/Scheduling_Tutorial.html?, [Online; accessed 14-Apr-2019].

[31] *MPC5777C Microcontroller Data Sheet*, NXP Semiconductors, 8 2018, rev. 13.

[32] *Specification of RTE Software*, AUTOSAR, 10 2018, rel. 4.4.0.

[33] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[34] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11.