Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science,
Embedded Systems Specialization - 30.0 credits

MINIMIZING INTER-CORE DATA-PROPAGATION DELAYS IN PARTITIONED MULTI-CORE REAL-TIME SYSTEMS USING SCHEDULING HEURISTICS

Emil Åberg
eag19002@student.mdh.se

Examiner: Mohammad Ashjaei
Mälardalen University, Västerås, Sweden

Supervisors: Saad Mubeen
Mälardalen University, Västerås, Sweden
Matthias Becker
KTH Royal Institute of Technology, Stockholm, Sweden

09/06/2021

Abstract

In the field of embedded systems, computers embedded into machines ranging from microwave-ovens to assembly lines impact the physical world. They do so under tight real-time constraints with ever-increasing demand for computing power and performance. Development of higher speed processors have been hampered by diminishing returns on power consumption as clock frequency is further increased. For this reason, today, embedded processor development is instead moving toward further concurrency with multi-core processors being considered more and more every day. With parallelism comes challenges, such as interference caused by shared resources. Contention between processor cores, such as shared memory, result in inter-core interference which is potentially unpredictable and unbounded. The focus of this thesis is placed on minimizing inter-core interference while meeting local task timing requirements by utilizing scheduling heuristics. A scheduling heuristic is designed and a prototype scheduler which implements this algorithm is developed. The scheduler is evaluated on randomly generated test cases, where its ability to keep inter-core data-propagation delays low across different core counts and utilization values was evaluated. The algorithm is also compared with constraint programming in a real world industrial test case. The obtained results show that the algorithm can produce schedules with low inter-core delays in a very short time, although not being able to optimize them fully compared to constraint programming.

Contents

List of Tables

1. Introduction

In the field of Embedded Systems, single-core microprocessors have been the dominant architecture for many years. Traditionally, improving computational capacity was done mainly by increasing the clock frequency. However, there are diminishing returns on power consumption when further increasing the frequency. Equation 1 explains the approximate relationship for power consumed in a processor [1].

$$\text{power} = \text{capacitance} \times \text{voltage}^2 \times \text{frequency} \tag{1}$$

As can be seen, frequency is only proportional to power consumption. However, higher frequencies also require higher voltage, meaning the frequency to power ratio will increase. Power efficiency is of particular importance in embedded processors since they are often battery driven. Dissipated heat worsens reliability, which is also of particular concern, since many embedded systems are safety critical. For these reasons, processor development today is instead moving towards further parallelism [1]. Multi-core architectures are being considered more and more every day. They are already widely used in, for instance, automotive applications [2], [3].

The transition from single-core to multi-core systems introduces a set of challenges, notably interference caused by shared resources. One such instance occurs when multiple cores are competing for shared memory, which introduces additional data-propagation delay [4]. This thesis will focus on developing new approaches based on heuristics to create time-triggered schedules for each core (i.e. offline partitioned scheduling).

1.1. Problem Formulation

One way to keep multi-core systems *timing predictable* is to use TDMA (time-division multiple access) bus arbitration, which is a schedule where each core is assigned pre-determined time slots for use of the bus. However, this approach can result in large inter-core data-propagation delays.

A system is said to be *timing predictable* for a given system model and a set of assumptions if it is possible to show, prove or demonstrate that timing requirements specified on the system will be satisfied when the system is executed [5].

Hasanagić and Vidović [4], [6] have proposed using constraint programming to produce feasible schedules optimized for the minimization of inter-core data-propagation delays . In their work, a task model is used where each task follows the Read-Execute-Write semantic [7], where the execute phases of the tasks can run in parallel, but only one task at a time may access shared memory during the read and write phases.

The goal of this thesis is to develop a technique where schedules can be produced according to the same model and restrictions used in [4], [6], but using heuristics rather than constraint programming, which allows schedules to be produced in far less time. Constraint programming also may not be scalable for larger systems, while solutions based on heuristics can be highly scalable. A proof of concept for the technique was developed as a software tool, where heuristics will produce schedules for a given task set. To limit the scope of this thesis, the following assumptions have been made:

- Scheduling that is non-preemptive for each task phase (read, execute, write) will be used.
- No shared cache. Only architectures where each core has its own dedicated cache will be considered.
- Only periodic tasks with implicit deadlines will be considered.
- Only homogeneous multi-core architectures will be considered.

The research goal of this thesis is to find ways to optimize schedules with respect to inter-core data-propagation delays for embedded multi-core architectures using heuristics. Based on this goal, the following research questions have been formulated:

RQ1: How can heuristics-based partitioned scheduling be used to minimize inter-core data-propagation delays in multi-core architectures while keeping schedules feasible in terms of meeting

task deadlines?
RQ2: How does heuristics-based partitioned scheduling compare with constraint programming with respect to age delay and inter-core data-propagation delay optimization in multi-core systems?

1.2. Targeted Architecture

For this thesis, the following assumptions are made regarding the targeted architecture: it is a multi-core architecture where cores have access to shared memory via a crossbar switch, each core has their own private, single level, instruction and data caches of equal size. For the purpose of exploring the scalability of the scheduling heuristics in regard to CPU count as well as to extend the applicability to a wide selection of real-world architectures, both CPU count and cache memory size will be of considered as variables for this thesis. Figure 1 shows the architecture under consideration, where N indicate the number of cores and m indicate the cache size.

One commercially available platform that fits this description is the MPC5777C microcontroller [8]. It is a widely used architecture in the automotive and industrial domains. It is featured in applications ranging from Engine Control Units (ECUs) to DC motor control and aerospace engines. It has two CPU cores, each with its private 16Kb data and instruction caches. Communication between the cores and the memory and peripherals is facilitated by a crossbar switch.

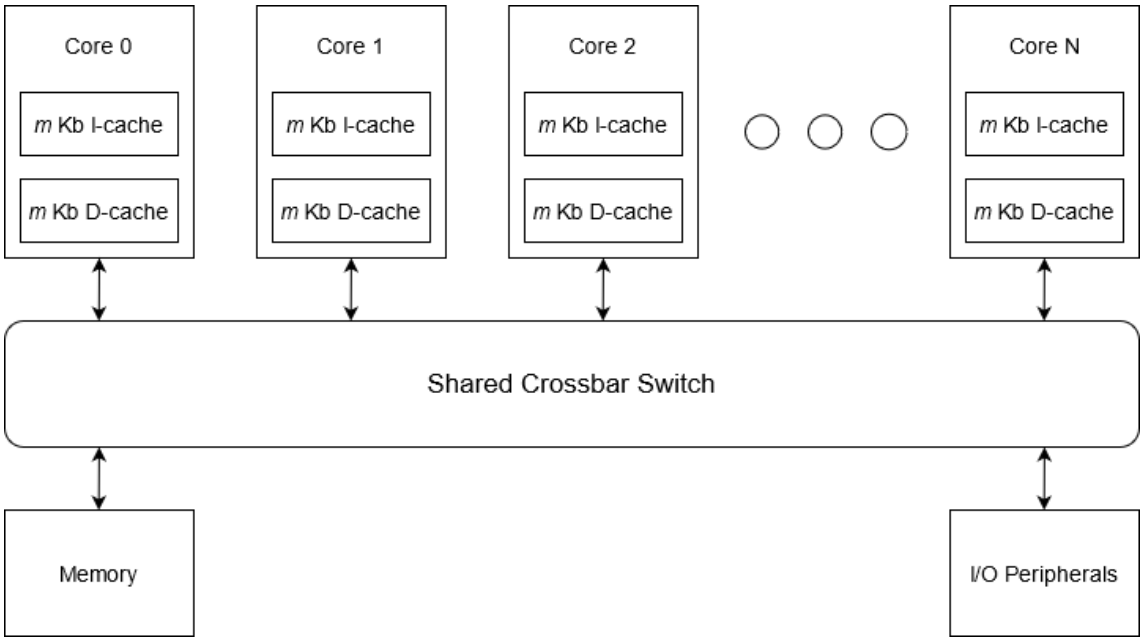

Figure 1: Diagram of the targeted architecture.

2. Background

2.1. Embedded Systems

An embedded system is a computer that's embedded into another system such as a car, microwave, robot, assembly line or airplane. The software within an embedded system has some sort of direct impact in the physical world. Unlike general-purpose computers, embedded systems are designed for a specific purpose [9].

They can be seen as a five-stage pipeline that form a closed feedback loop with the environment it acts upon. Figure 2 illustrates this concept. The first stage of the pipeline are the sensors, that measure the physical quantities to be controlled. The second is an analog-to-digital converter that converts the analog signals of the sensors to digital signals. The third stage is the processor, that uses the sensor data as input for signal processing and a control algorithm. The processor output is then fed to the next stage, a digital to analog converter, whose output in turn is fed to the actuators, completing the loop. The sensors may for example be rotary encoders, measuring the speed of an engine, or an air-pressure sensor, that is used to calculate the altitude of an airplane; the actuator could for instance be a DC-motor.

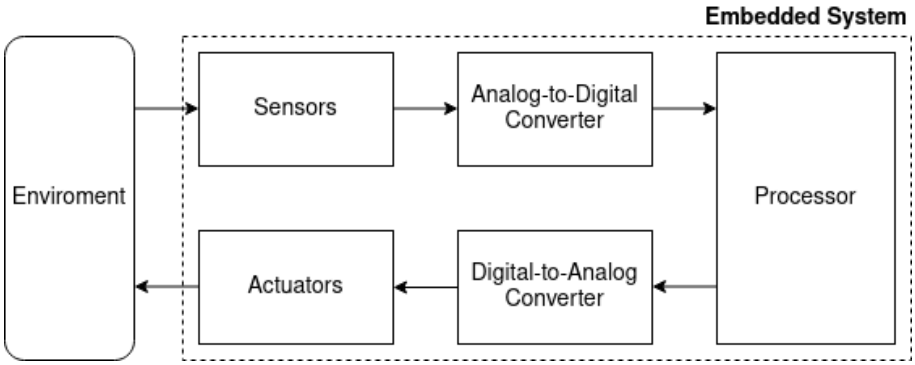

Figure 2: Diagram of embedded systems modeled as a five-stage pipeline. Embedded systems form a closed feedback loop with the physical environment.

The computing component in an embedded system is typically a micro-controller, which is a whole computer system on a single chip, specifically designed for embedded systems. It contains a processor, memory, programmable I/O and usually digital-to-analog and analog-to-digital converters along with other communication peripherals. Alternatives exist, such as digital signal processors (DSPs), which are processors optimized for signal processing applications, and Field Programmable Gate Arrays (FPGAs), which contain a matrix of configurable logic gates that can be programmed to implement virtually any digital circuit.

Embedded software are often subject to strict timing constraints, as tasks are required to finish execution within certain time frames. Tasks that require such timing constraints are called real-time tasks. Embedded systems with such time constraints are called real-time systems.

2.2. Task Model

In regard to timing and core allocation, a task τ_i can be expressed by the following parameters:

- T_i - *Period:* The time between release of two consecutive task instances.
- D_i - *Deadline:* The time after release in which a task has to finish its execution.
- C_i - *Execution Time:* The worst case estimate of time the processor has to keep the task in its context in order to finish.
- p_i - *Core:* The core which the task is assigned to.

The execution time C_i can in turn be divided in a read phase C_R, an execute phase C_E, and a write phase C_W. In other words, for a task τ_i the execution time can be expressed as:

$$C_i = C_{Ri} + C_{Ei} + C_{Wi} \tag{2}$$

For each task instance, these phases will occur in that order. The reason for dividing up the execution phases is that tasks may execute in parallel, but they can't read or write to memory simultaneously. It also allows tasks to have a consistent view of the system. Because of that, this type of communication is often used in embedded systems, and will be one of the constraints of the scheduler. In accordance to the system model assumptions, deadlines are implicit, meaning they are equal to the periods of their respective tasks, meaning $T_i = D_i$. Figure 3 shows the timing parameters of a task and how they appear during the course of two instances.

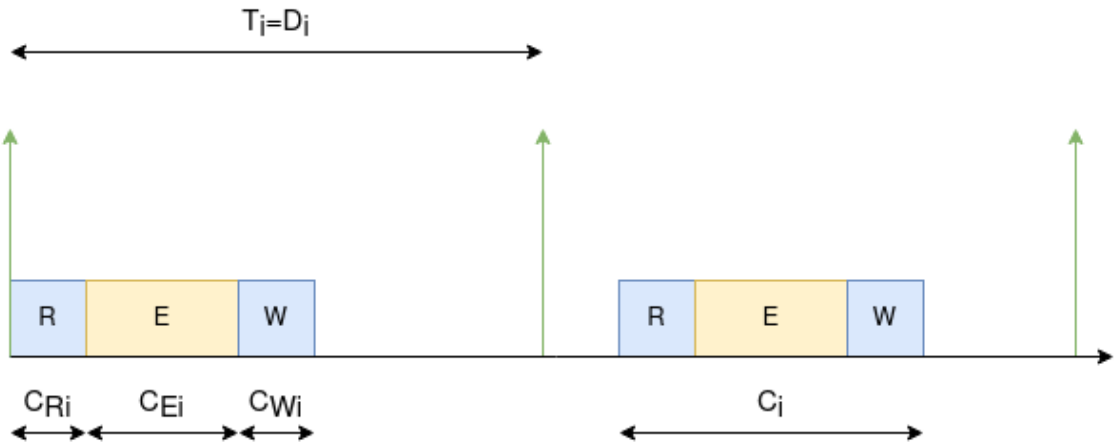

Figure 3: Two instances of task τ_i with its timing parameters displayed.

2.3. Task-to-Core Allocation

In single-core systems, the job of scheduling consists solely of deciding when each tasks should run. In multi-core system, in addition to when, the scheduler needs to decide on which core each task should be allocated to. Scheduling paradigms for multi-core systems can be divided into two types: *Partitioned Scheduling*, where tasks are assigned to cores during design-time, and *Global Scheduling*, where tasks are assigned to cores during run-time. In Partitioned Scheduling, each core has its own individual scheduler and ready queue, whilst in Global Scheduling, a single scheduler schedules and assigns tasks to cores from a single ready queue. A key difference is that in Global Scheduling the same task may run on different cores. Task migration, much like task preemption, may improve schedulability but is also a source of overhead and timing unpredictability [10]. As stated in the problem formulation, timing predictability is prioritized in this thesis, therefore the scope of this thesis is limited to Partitioned Scheduling.

2.4. Task Chains

In this thesis, sequences of tasks are modeled as task chains. Task chains are sets of periodic tasks where each subsequent task in the chain is dependant on the output of its predecessor [11]. In embedded systems, the first task in the chain typically reads data from one of the sensors, while the task at the end of the chain transmits data to an actuator. The tasks in-between perform different types of data processing. The output of each task is stored in a memory location, i.e. register, that is then read by the subsequent task. A task chain whose tasks have different periods is called a multi-rate chain. Figure 4 shows an example of a task chain consisting of three tasks (τ_1, τ_2, τ_3) and two intermediate registers (r_{12}, r_{23}). In this example, the tasks have different periods and is therefore a multi-rate chain. It is important to note that the same task may be part of multiple task chains.

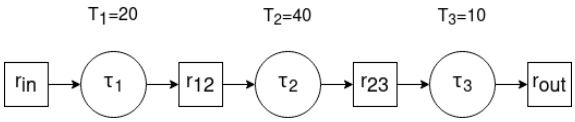

Figure 4: An example of a multi-rate task chain. Task chains are typical in embedded software, where data propagates from a sensor, through a series of tasks, and then outputted to an actuator.

2.5. End-to-End Path-Delays

The time it takes for data to propagate from the beginning to the end of a task chain is called the end-to-end delay. For a task-chain with uniform periods, this is dependant on the sum of the WCETs (Worst Case Execution Times) as well as the delay in activation between each task. However, in multi-rate chains, due to the different periods, data may also be overwritten before it is passed further down the chain. Likewise, a task may activate before its input data has been updated, which causes it to operate on "old data" and produce an output that is based on the same input data as the previous instance. [11].

Figure 5 illustrates a situation, where task C produces four outputs based on the same input data before task B updates it. Also, the output of every second instance of task A is overwritten before it gets to propagate forward trough the chain, resulting "dead end" paths. First and last output refers to the first and last output of the last task in the chain, that operate on the same input data, in regards to the hyperperiod. First and last input refers to instances of the first task, among "non dead end" instances, that have been subject to, and not subjected to, overwrite respectively.

The first input to first output (first-to-first) and last input to last output (last-to-last) are also called Reaction Delay and Age Delay, respectively. They are of particular interest in the automotive domain; in applications such as body electronics, the reaction delay represents the worst case delay from button press to actuator action while the age delay represents the maximum age of data, which is of importance in control systems [11]–[15].

In addition deadlines of individual tasks, timing constraints can be associated with these types of delay. For this thesis however, feasibility is determined only by local task deadlines; a schedule is considered feasible if all tasks meet their deadlines.

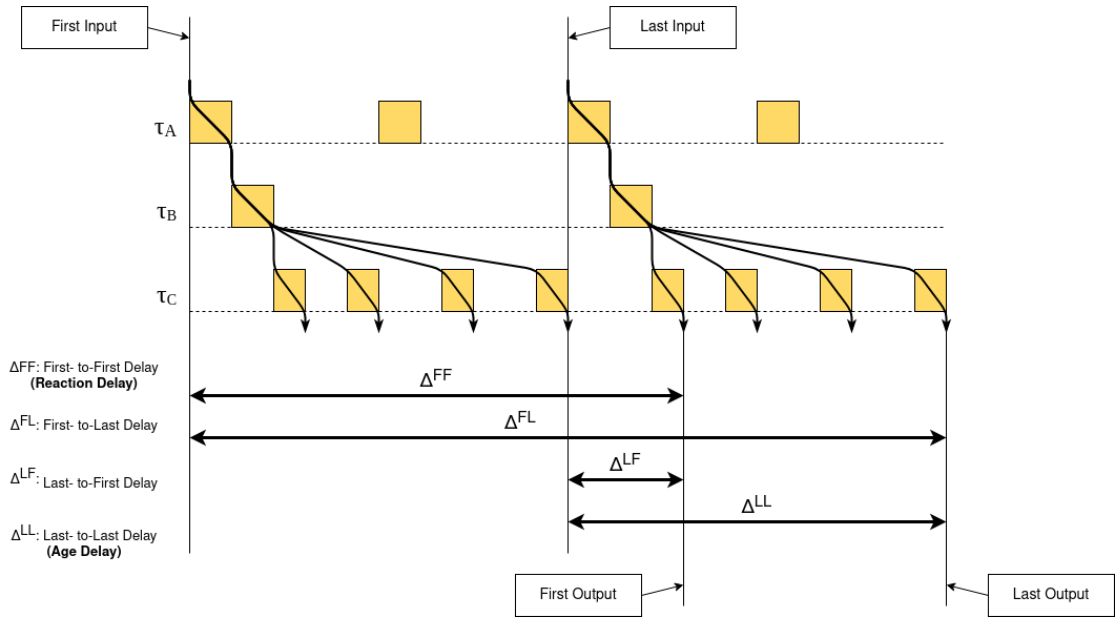

Figure 5: End-to-end delays.

2.6. Inter-Core Data-Propagation Delays

The objective of this thesis is to minimize *inter-core data-propagation delays* using heuristic-based scheduling. Data-propagation delays refer to the delay between two communicating tasks in a task chain. Formally it is defined as the amount of time between the end of a producer job and the start of the consumer job which reads the output. In the case that the producer and consumer are allocated to different cores, this is referred as *inter-core* data-propagation delay. Figure 16 illustrates this concept, where two subsequent tasks in a task chain execute on different cores.

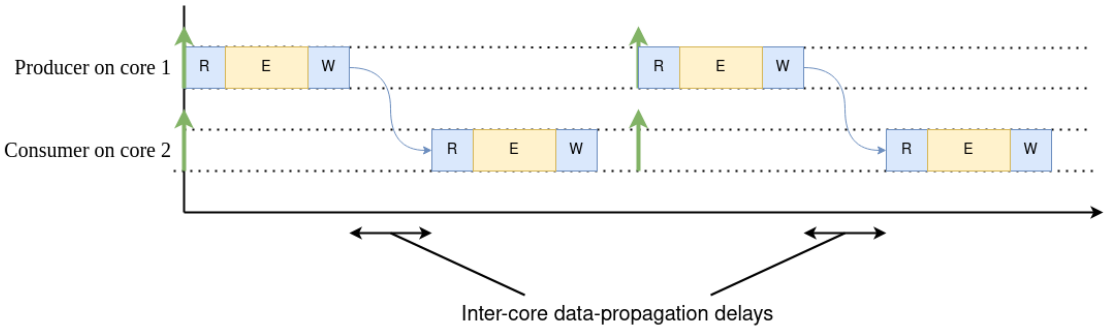

Figure 6: Inter-core data-propagation delays.

3. Related Work

This thesis focuses on techniques for scheduling multi-core systems in a scalable way while at the same time keeping the system timing predictable. A published survey provides an overview of the scientific literature on timing-verification techniques for multi-core real-time systems [16]. It covers the latest published research up to the end of 2018. It states that research in timing-verification techniques for multi-core real-time systems can be classified into four categories, namely:

- **Full Integration:** Research in this category considers the behavior when all information about the task under study as well as parallel tasks that are running on the same hardware are taken into account. There are two dominant approaches in this category: analyses based on abstract processor state and WCET analysis based on model checking. Model checking is a validation technique where the system is modeled mathematically and the requirements are formulated as logical notation, which the mathematical model is then tested against.
- **Temporal Isolation:** Research in this category considers different techniques to achieve temporal isolation. Temporal isolation is achieved whenever the capacity of a set of tasks to fulfill their timing constraints does not depend on the temporal properties of other, unrelated tasks. This can be achieved via software means such as phased execution or hardware means such as TDMA arbitration.
- **Integrating Interference Effects into Schedulability Analysis:** Research in this category aims to integrate the effects of the effects of the interference due to shared resources and co-runners into the schedulability analysis.
- **Mapping and Scheduling:** Research in this category considers mappings and scheduling techniques for producing feasible schedules, both in terms of timing- and memory requirements. Such techniques include constraint programming, heuristic solutions, genetic algorithms, integer linear programming (ILP) and bin-packing. The research done in this thesis falls into this category.

One of the foremost conclusions in the paper is that 2020 and onward, real time-systems will involve multiple, concurrent, complex applications executing on hardware platforms consisting of multiple cores, perhaps interconnected in small clusters that may themselves be connected to other clusters. This highlights the importance of scalable scheduling approaches in multi-core systems.

3.1. Constraint Programming

One of the available mapping and scheduling techniques is constraint programming. In constraint programming users declare a set of variables and state the constraints that define feasibility. Users may also specify a property, defined as a function of these variables, called the objective function which is to be minimized or maximized. Formally, a given problem is expressed as:

- A set of variables: $\mathcal{X} = \{x_1, ..., x_n\}$
- The domain of each variable: $\mathcal{D} = \{\mathcal{D}_1, ..., \mathcal{D}_n\}$
- The constraints of each variable: $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_n\}$
- The objective function: $f(\mathcal{X})$

These inputs are fed to a constraint solver, which is an algorithm implementation that outputs a solution. There are many publicly available constraint solvers, such as IBM ILOG CP, Gecode and CHIP [17].

Hasangić and Vidović [6] proposed a solution to the problem of interference due to shared resources, based on constraint programming, where inter-core data-propagation delay was chosen as the property to be minimized. For their proof of concept, Hasangić and Vidović developed a scheduler utilizing the IBM ILOG CP optimizer. Due to the fact that there is an absence of real-world benchmarks due to IP-protection, their scheduler was tested on a set of randomly generated test cases which sought to emulate the timing characteristics of real world automotive embedded applications. Their method proved successful in producing optimized schedules in terms of minimizing inter-core data-propagation delays.

3.2. Memory Centric Heuristic

Many-core architectures are already widely used in automotive applications [2], [3]. To meet the demand of power-efficient solutions in the automotive industry, Becker et al. [7] proposed a contention-free framework based on using a Massively Parallel CPU Array (MPPA) architecture.

An MPPA is a many-core platform which can contain hundreds of cores working in parallel. The cores are organized into clusters connected by a shared network. It excels in power efficiency compared to the other architecture types (CPU, GPU, FPGA) [18].

In order to eliminate shared resource contention Becker et al. [7] proposed a heuristic scheduling algorithm called Memory Centric Heuristic. It is based on the idea that the crucial resource to be scheduled is the shared network and shared memory, rather than the cores which are plentifully available in their targeted architecture. Like in this thesis, they make use of the read-execute-write semantic to privatize access to shared resources. As it is presented in [7], the Memory Centric Heuristic assumes global scheduling. By omitting the job-to-core assignment of the Memory Centric Heuristic so that it is adaptable for partitioned scheduling problems, a good starting point for the design of the heuristic to be designed in this thesis is provided.

4. Research Method

4.1. System Development

The research method used in this thesis will be the System Development method as presented by Nunamaker and Chen in [19], which combines system development as done by the engineer with empirical research methodology as done by the scientist. It is a suitable method in research where the goal is to create new system to solve a particular problem, rather than study existing alternatives. The method is based on developing answering the research questions by developing a prototype system which will then be subject to study or experiment. The method can be described in five phases:

1. **Construct a conceptual framework.**
   During this phase the research questions should be justified. By studying relevant disciplines, having a good understanding of the state of the art and brainstorming, it should be made sure that the research problem is new and meaningful in the field. These efforts also serve the purpose of understanding the system that is to be built.
2. **Develop a system architecture.**
   An architecture design should be developed and software components should be defined. This serves as the high-level plan for the system building process.
3. **Analyze and design the system.**
   In this phase the blueprint of the system is developed. For software, this means that data structures, databases and knowledge-bases are determined.
4. **Build the system.**
   This is the phase where the system is implemented. The purpose of implantation lies not only in providing an object for evaluation and proof of concept, but also to provide experience about the building process itself.
5. **Observe and evaluate the system.**
   Once the system is built it can be subject for observation and evaluation. This can be done either by case study, field study or experiment. The results will be interpreted based on the foundation laid out in the conceptual framework.

It is important to note that the phases do not strictly go in that order. Whenever a need for revision appears, researchers can go back to an earlier stage, for instance if errors in the blueprint don't become apparent until halfway trough the building process. Figure 7 shows a diagram of the system development method. The arrows indicate that the process starts from the left and continues right and that it is possible to revert when necessary.

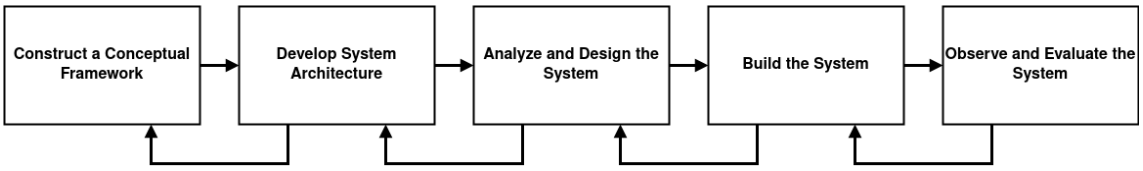

Figure 7: Diagram of the Systems Development method.

4.2. Application of the Research Method

In this section the instantiation of the System Development research method is explained. Here, the application of each phase on this thesis is described in order.

4.2.1 Construct a Conceptual Framework

In this step, the knowledge in the contained in the related works of this report was acquired. It was decided that the Memory Centric Heuristic [7] could be used as a starting point of the algorithm as it operates under similar assumptions with the exception of it being designed for global scheduling.

4.2.2 Develop System Architecture

Figure 1 shows the software component diagram of the scheduler that was developed. The parser component reads the input, which is in the form of a text file. This text file lists the set of tasks to be scheduled, with parameters as described in section 2.2.. Also included in the text file is the set of task chains that are part of the scheduling problem.

The *algorithm core* is the main part of the program. It does the bulk of the job of scheduling the task set. It's operation is described in section 5.1.. However, whenever a job is selected to be scheduled among the available ready jobs this is done by a separate function that implements a set of three rules which are described sections 5.2., 5.3., 5.4. respectively. These rules are designed to prioritize jobs in such a way as to minimize inter-core data-propagation delay. The job selection module is represented as the *getNextJob* function in algorithm 1.

The scheduler also calculates all the inter-core data-propagation delays of the produced schedule. Both the schedule and the delays are then finally outputted to a text file by a 'writer' module.

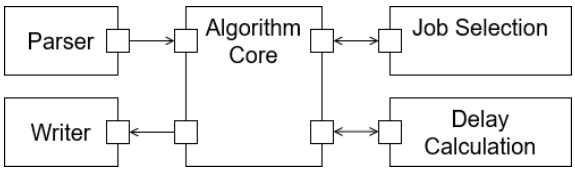

Figure 8: Diagram of the software components of the scheduler.

4.2.3 Analyze and Design the System

It was decided that linked lists are the data structures to be used in the algorithm. Two linked lists were used, one for the ready queue and one for the job queue in algorithm 1. As the algorithm executes, the next job to be accessed in the ready queue is typically the one with the highest priority, and the next job to be accessed in the job queue is typically the one with the lowest release time. Linked lists are dynamic and it allow the ready queue to be sorted in order of priority and the job queue to be sorted in order of release time. This allows for quick access to the jobs in those queues.

4.2.4 Build the System

Once the architecture and data structures were decided, the scheduler was developed in the C programming language. The compartmentalized architecture allowed for components to be tested individually before being assembled together for more robust development.

4.2.5 Observe and Evaluate the System

Once the system was built it was evaluated in two ways: an experiment where the scheduler was tested on thousands of synthetically generated test cases, and a test where it was evaluated on an industrial use case. The purpose of the experiment was to evaluate the capability of the scheduler to reduce inter-core data-propagation delays across different core counts and utilization values. Te purpose of the test on the industrial use case was to compare the schedulers' performance against constraint programming. The evaluation procedure and its results are described in detail in section 6..

5. Scheduling Algorithm

An integral part of the System Development Research Method described in section 4 is the development of a prototype to solve the problem at hand. In this thesis a scheduling tool was developed, written in the C programming language, that implements a heuristic that is designed to minimize inter-core delay while keeping solving time low. This heuristic will from here on be called the Delay Minimization Heuristic. This section is dedicated to explaining the design of this heuristic as well as the reasoning behind the design choices.

The high-level design of the algorithm is that by default, jobs are scheduled and prioritized based on deadline (earliest deadline first); by adding additional rules the algorithm will be readjusted to minimize inter-core delays. The core of the algorithm is described in section 5.1. The rules responsible for reducing inter-core delays are defined in sections 5.2-5.4. Finally, section 5.5 will explain how those rules are combined when the scheduler chooses between jobs to be scheduled.

5.1. Algorithm Core

By modifying the Memory Centric Heuristic in [7] for partitioned scheduling and refactoring, a basic algorithm, Algorithm 1, was produced. This algorithm can be used to solve multi-core partitioned scheduling problems, where read, execute and write jobs are scheduled individually. This algorithm makes up the core of the Delay Minimization Heuristic. The algorithm is then adjusted for optimizing inter-core delays by modifying the 'getNextJob' function in line 7.

The input $\mathcal{T}$ is the set of tasks that must be completed during one hyperperiod, where the algorithm finishes. S contains the schedule of each core, whenever a job is added to S it is assumed to be scheduled in the individual schedule of its allocated core. The current time is kept track of in the variable 'ctime'. Two queues are used to keep track of unscheduled jobs, Q_{job} and Q_{ready}. Q_{job} contains jobs that are not yet ready, meaning the release time of all jobs in Q_{job} is greater than the current time. The queue Q_{ready} contains jobs that are ready, but have not been scheduled by the algorithm yet. The algorithm will incrementally schedule jobs as it progresses from the start of the hyperperiod until both queues are empty. For a schedulable task set where all jobs meet their deadlines, both queues will become empty at or before the algorithm reaches the end of the hyperperiod. Q_{job} is initially populated with all read jobs. j_{curr} will always contain the currently executing read or write job except for when there is no job in execution, at which point it will be $\emptyset$. Following initialization, starting at line 6, the algorithm will repeat the following steps until both Q_{job} and Q_{ready} are empty:

1. *Transfer Ready Jobs:* At line 6, the algorithm will check if any of the jobs in Q_{job} should be moved to Q_{ready} before proceeding to schedule those.

2. *Choose next job to schedule:* At line 7, the algorithm chooses the highest priority eligible job. A core whose last job was an execute job is only eligible for a write job. A core which is currently executing is not eligible for any job. As previously stated, by default, job priority is decided by deadline, where jobs with shorter deadline take precedence over jobs with higher deadline while the additional rules described in section 5.2-5.4 modify the priorities for inter-core delay optimization. If there is no eligible job to be scheduled, the algorithm will move the current time forward until the release of next job.

3. *Schedule a read or write job:* When a job is chosen at the previous step, it will immediately be scheduled to start at the current time. This job, j_{curr}, will either be a read or write job, therefore the current time is moved forward until its completion. All read or write jobs are scheduled at line 9. If this was a read job, its corresponding execute job will immediately be scheduled afterwards and its write job will be added to Q_{job}. The release time of the write job will be equal to the finishing time of the execute job. Upon completion of write jobs, the algorithm will check if it finished after its deadline, in which case the algorithm will terminate, returning the status "UNSCHEDULABLE".

Algorithm 1: GenerateSchedule($\mathcal{T}$)

```
1  S ← ∅ ;                                                    // empty schedule
2  ctime ← 0 ;                                                // current time
3  Q_job ← generateReadJobs(T);
4  Q_ready ← ∅;
5  while (Q_job ≠ ∅ or Q_ready ≠ ∅) do
6      Q_ready.add({j_i ∈ Q_job | rel_i = ctime});
7      j_curr ← getNextJob(Q_ready);
8      if (j_curr ≠ ∅) then
9          S.add(j_curr);
10         ctime ← ctime+C_curr;
11         if (j_curr is a read job) then
12             j_exec ← generateExecuteJob(j_run);
13             j_write ← generateWriteJob(j_run);
14             S.add(j_exec);
15             Q_job.add(j_write, rel = ctime+C_exec);
16         if (j_curr is a write job) then
17             if (j_curr finished after its deadline) then
18                 return UNSCHEDULABLE;
19     else
20         ctime ← nextRelease(Q_job);
21 return S;
```

Algorithm 1: The core of the Delay Minimization Heuristic. The input $\mathcal{T}$ denotes the set of tasks to be scheduled.

5.2. Precedence Constraints

When producer and consumer jobs are released simultaneously, such as at the beginning of the hyperperiod, it is possible that producer jobs finish after the start of their respective consumer jobs, effectively missing each other. When this happens, the output of the producer job will not be read until the next instance of the consumer causing large data-propagation delay; inter-core delay in the case that the producer-consumer pair are assigned to different cores. Such a case is illustrated in figure 9 which shows the schedule containing two task chains: (τ_2, τ_1, τ_3) and (τ_4, τ_1). In this case, the first instance of τ_4 finishes after the start of the first instance of τ_1, the output of τ_4 is not read until the second instance of τ_1 starts. There is large delay between the two jobs.

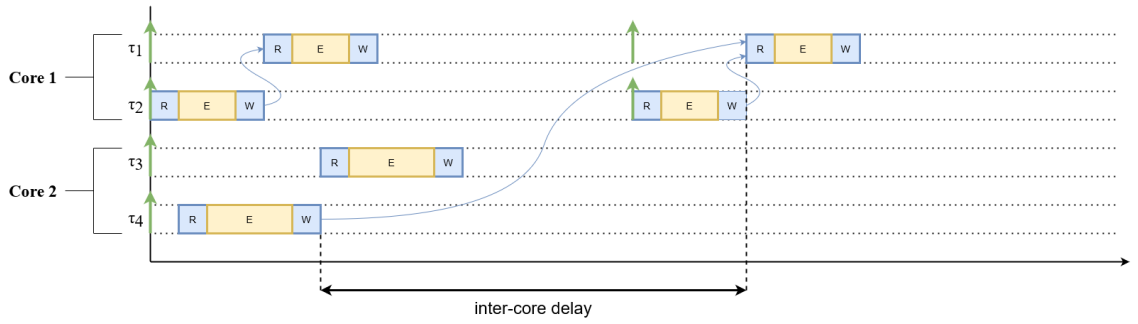

Figure 9: Four tasks running on two cores which are scheduled without precedence constraints. The first instance of τ_1 does not get to read the output from the first instance of τ_4, resulting in large inter-core delay as this output is not read until the second instance of τ_1.

In order to prevent this from happening, the following precedence constraints have been added to the algorithm: *Each consumer job may not start before each of its currently ready or running producer jobs has finished at least once since the previous instance of the consumer job.* The reason for only including producers that are ready or running to these constraints is to avoid deadlock at

cases where a consumer task runs more frequently than its paired producer task. In other words, when there exist more consumer jobs than producer jobs. In such cases, some consumer jobs would otherwise be postponed indefinitely. Figure 10 shows how the task-set in figure 9 will be scheduled when these precedence constraints are in place. Here, the first instance of τ_1 does not start before the first instance of τ_4, eliminating the delay between the two tasks.

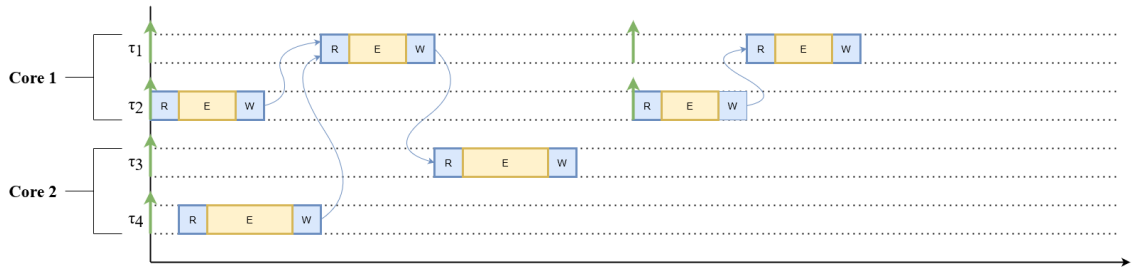

Figure 10: The same task set as in figure 9, scheduled with precedence constraints. There is no inter-core delay.

5.3. Priority Increase

To further improve inter-core delay, an additional rule has been added. Whenever a producer job finishes execution, the priority of the first next job, for all consumer tasks in which it is a producer for, will have increased priority, meaning their execution takes full precedence even if their deadline is longer. In the event that multiple ready jobs have had their priority increased this way then priority among those are determined by deadline. The purpose of this rule is to keep producer-consumer chains clear from the interference of other, independent, tasks.

5.4. Priority for Interconnected Tasks

In order to reduce interference in task chains caused by other task chains, an additional rule is added. Here a value is calculated for each task that represents how much the chains that task is part of share tasks with other chains. This value will from hereafter be called *task connectivity*. The rule is that when jobs otherwise have the same priority, priority will be higher for jobs whose tasks have higher connectivity. The reasoning is that delaying tasks in chains that share tasks with other chains will result in a higher total delay than delaying tasks in task chains that share fewer tasks with other chains. Since this applies only to jobs that otherwise share the same priority, this rule is not an exception to the default of prioritizing by deadline. If prioritizing by deadline is assumed to be what is most likely to result in a schedule where all tasks meet their deadline, then this rule should have no negative impact on the likely-hood that the schedule produced is feasible.

In order to calculate the connectivity value for a task, first a connectivity value for each chain is calculated. The *chain connectivity* of a task chain is the amount of times each of its tasks are shared with other tasks. Formally, the connectivity of a chain is calculated with algorithm 2. Once the chain connectivity is calculated, the task connectivity of each task is equal to the maximum chain connectivity among chains that task is a member of.

Algorithm 2: CalcConnectivity($Chain_a$)

```
1 conn ← 0;
2 for each τ_i ∈ Chain_a do
3 |   for each chain, Chain_i, other than Chain_a do
4 |   |   if τ_i ∈ Chain_i then
5 |   |   |   conn ← conn +1;
6 return conn;
```

Algorithm 2: Calculation of chain connectivity.

5.5. Combining the Rules

In order to keep track of the priority of jobs as they are defined, in the rules mentioned so far, jobs in the ready queue are kept in an ordered list where they are sorted by priority. The resultant prioritization from combining the rules is shown in figure 11. The figure shows a case when it contains 8 jobs, represented by the boxes, demonstrating how they are ordered. All consumer jobs that has their priority increased upon completion of their paired producer, in accordance with the priority increase rule defined in section 5.3, have priority over all other jobs.

Whenever a producer job finishes, the next consumer job, for each task it is a producer for, will be moved up the queue order to be placed among the other priority increased jobs if it isn't already located there.

Secondarily, priority is ordered by deadline. In both section of the queue, the one dedicated to increased-priority tasks as well as the one dedicated to non-increased priority tasks, the jobs are ordered by deadline.

Thirdly, jobs that otherwise share the same priority are ordered by connectivity. Finally, the function 'getNextJob' in Algorithm 1 is defined to return the highest priority job in the ready queue among jobs that:

- Pass the priority constraints described in section 5.2.
- Whose core isn't currently executing another job.
- If they are read jobs, aren't preceded by execute jobs.

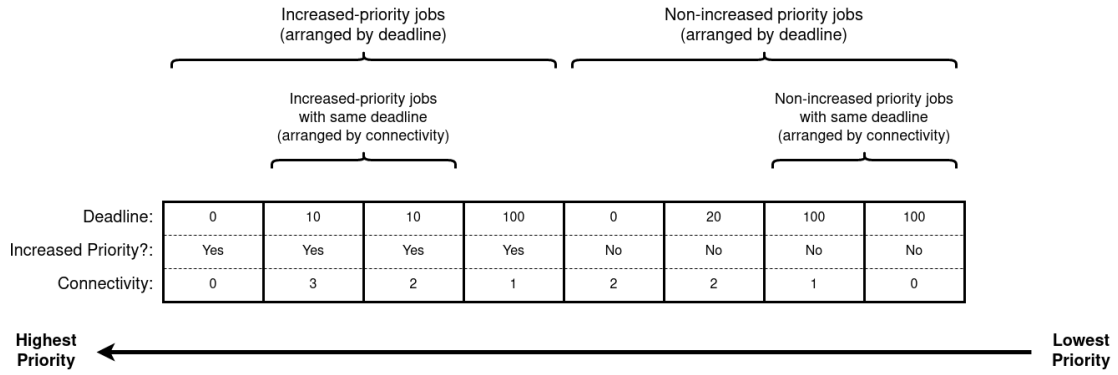

Figure 11: The ready queue of the Delay Minimization Heuristic, in this case containing 8 jobs, demonstrating how they are ordered.

6. Evaluation and Results

This section is concerned with the evaluation of the Delay Minimization Heuristic. The heuristic is evaluated in two types of tests. In the first test, randomly generated test cases were created to evaluate the scheduler's performance over varying utilization and core count. In the second test, the Delay Minimization Heuristic was subject to an automotive industrial use case, where its performance is compared to the performance of the constraint programming implementation in [6] in regards to age delay, inter-core data-propagation delay and solving time.

6.1. Experiment on Synthetic Test Cases

In order to evaluate the performance of the Delay Minimization Heuristic and prove its capacity to reduce inter-core data-propagation delays, an experiment was carried out where the Delay Minimization Heuristic was tested on randomly generated synthetic test cases that conform to automotive applications. Due to IP-protection, there is an absence of real-world benchmarks [20]. Generating test cases synthetically bypasses this issue.

In order to see what effect the rules described in sections 5.2-5.4, actually have in regard to reducing inter-core data-propagation delays, the Delay Minimization was ran on the synthetic test cases with and without those rules. When the algorithm is run without the rules, job priority is determined by deadline only. The experiment has two independent variables: utilization and core count; and one dependent variable: total inter-core data-propagation delay.

6.1.1 Generation of Synthetic Test Cases

Each synthetic test case is composed of 11 tasks which apply the set of periods used in the evaluation of the Memory Centric Heuristic in [7], whose experiments were designed to conform to automotive applications. The periods are listed in 1. This constitutes a set of 100 jobs over a hyperperiod of 1000ms.

Task	Period
τ_1	100ms
τ_2	1000ms
τ_3	1000ms
τ_4	1000ms
τ_5	1000ms
τ_6	1000ms
τ_7	50ms
τ_8	200ms
τ_9	200ms
τ_{10}	200ms
τ_{11}	20ms

Table 1: Task periods of the synthetic test cases.

Tasks are assigned to cores randomly under the constraint that tasks are distributed as evenly as possible among cores in terms of task count. Each test case contain three task chains consisting of four, three, and two tasks respectively. The tasks chosen for the task chains are decided at random under the constraint that each subsequent task in a chain must be allocated on a different core than the previous task. The purpose of this constraint is to keep the number of core-to-core transmissions constant in all test cases.

Utilization is divided evenly among all cores, where the utilization of each individual task is randomly generated using the Randfixedsum algorithm [21] which can efficiently generate a set of random numbers with uniform probability distribution which sum to a constant value. The algorithm is publicly available through an open source MATLAB implementation. Task execution time is derived from period and utilization using equation 3.

$$C_i = U_i \cdot T_i \tag{3}$$

The execution time of the read and write phases are both set as 5% of the total execution time, leaving 90% for the execution phase.

$$C_{Ri} = C_i \cdot 0.05 \tag{4}$$

$$C_{Wi} = C_i \cdot 0.05 \tag{5}$$

$$C_E = C_i \cdot 0.9 \tag{6}$$

6.1.2 Results of the Synthetic Test Cases

The plots in figures 12, 13, 14 show mean, max and minimum inter-core data-propagation delay respectively. The delay has been calculated at three different utilization values: 0.5, 1.0, 1.5, and core counts ranging from 2 to 8. The same test-cases have been tested both with the full algorithm as described in the entirety of section 5 and without the rules described in sections 5.2-5.4, meaning priority is scheduled merely based on deadline. In order to generate the plots, the scheduler has evaluated 20000 synthetic test cases for each data point. Of those test cases, only the ones that the scheduler returned feasible schedules for are used in making the plots. Figure 15 show the percentage of schedules for which the scheduler could schedule feasibly; for each utilization value, with and without the rules.

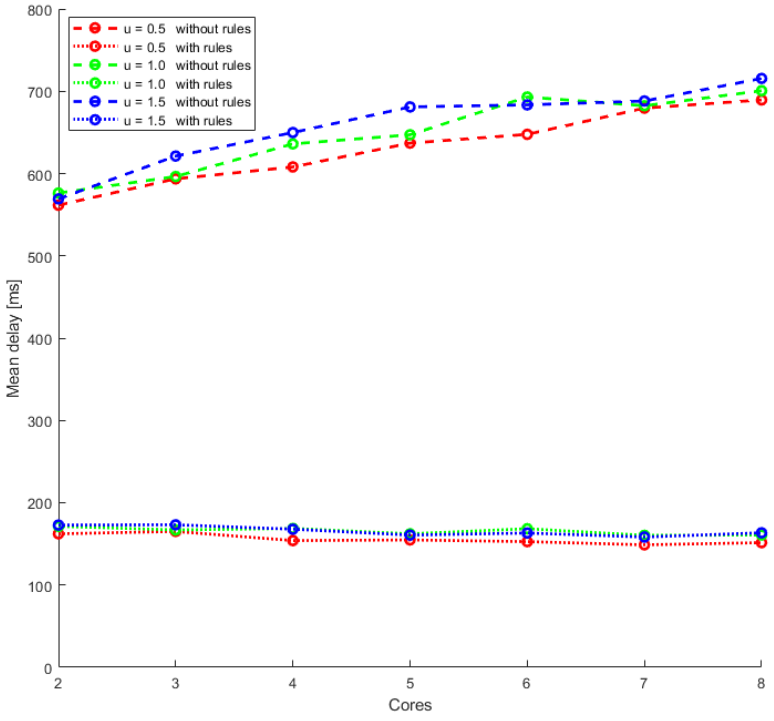

Figure 12: Mean inter-core data-propagation delay of the synthetic test cases.

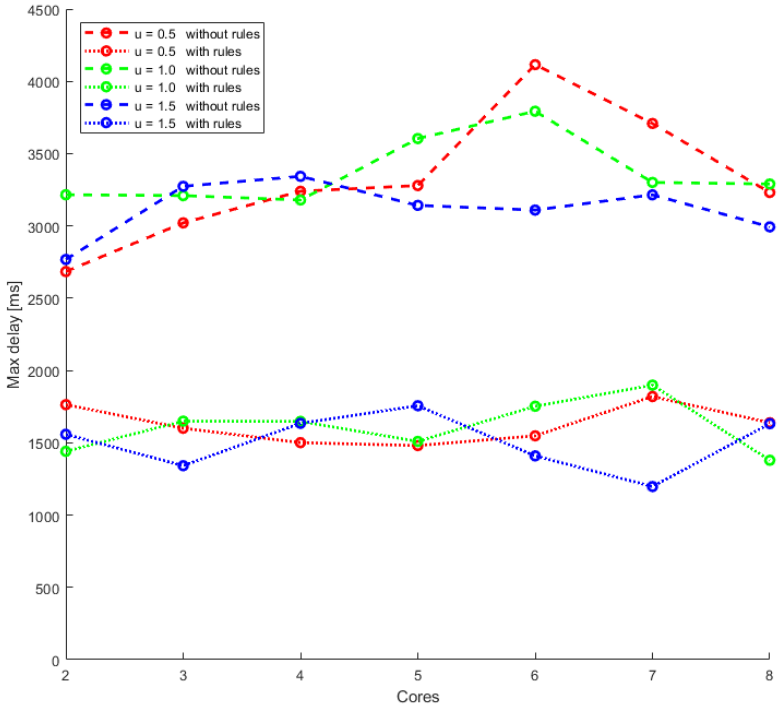

Figure 13: Maximum inter-core data-propagation delay of the synthetic test cases.

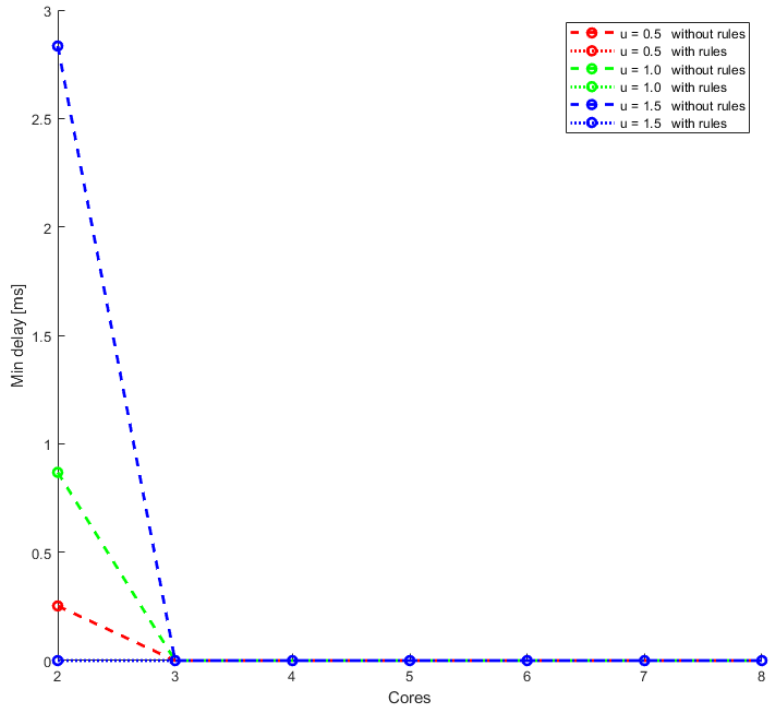

Figure 14: Minimum inter-core data-propagation delay of the synthetic test cases.

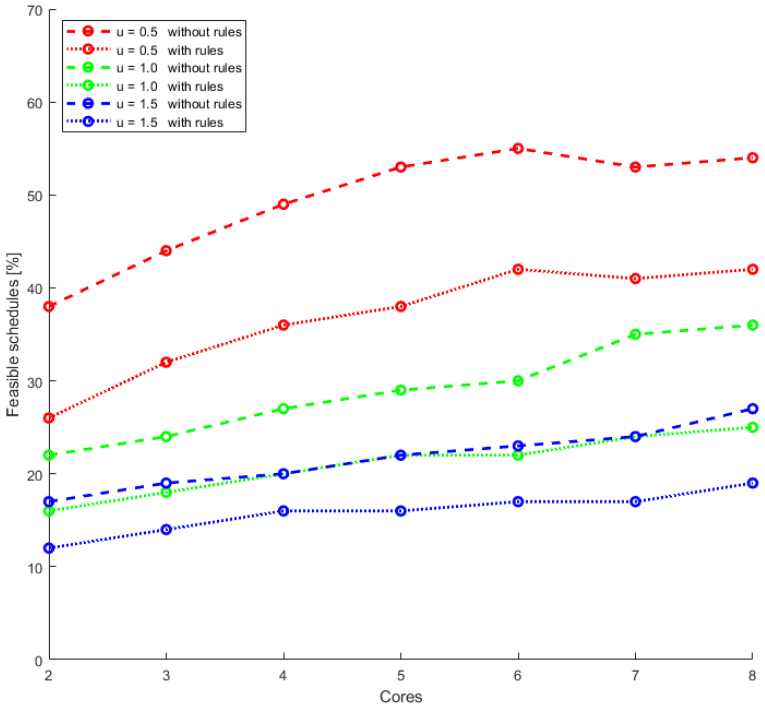

Figure 15: Percentage of feasible schedules of the synthetic test cases.

6.2. Industrial Use Case

Hasangić and Vidović [6] developed a scheduling tool that applies Constraint Programming, designed to optimize inter-core data-propagation delays. This scheduling tool is referred to as *the Optimization Engine* in [6] and will from hereafter be referred to the same in this report. They compared the Optimization Engine with the scheduler of Rubus-ICE, which is an integrated environment for model-driven embedded software development, developed by Arcticus Systems [22], [23]. Currently, Rubus-ICE does not offer full support for multi-core systems. They can be modeled as well, but during scheduling, each core is modeled separately which means that inter-core data-propagation delays can be large. In other words, Rubus-ICE is naive when it comes to inter-core data-propagation delay. The Optimization Engine and Rubus-ICE were tested on a publicly available model of a real world engine application [24].

In order to provide comparison between the Delay Minimization Heuristic and Constraint programming where both algorithms are evaluated in the exact same conditions, the same use case was applied in this thesis on the Delay Minimization Heuristic. The parameters compared were the inter-core data-propagation delay and age delay for The Delay Minimization Heuristic, the Optimization Engine and Rubus-ICE. In addition, solving times were measured for The Optimization Engine and the Scheduling Heuristic.

6.2.1 Industrial Use Case Setup

The industrial use case consists of 18 tasks with a hyper period of 1000ms constituting a total of 146 jobs and 3 task chains. Table 2 shows the tasks and their parameters of the industrial use case, as it has been applied in [6]. The three task chains are shown in figure 16, where each task is color coded by period. Green corresponds to $T = 50ms$, yellow to $T = 100ms$ and red to $T = 1000ms$. As can be seen, each chain is a multi-rate chain.

Solving time of The Delay Minimization Heuristic and The Optimization Engine were measured by running both algorithms on a PC with an i5-9600K CPU and 16Gb of RAM. The industrial

use case ran multiple times on both algorithms in order to find the average solving time. It ran 20 times on The Optimization Engine and 100000 times on The Delay Minimization Heuristic, due to it's significantly smaller execution time.

Task	C_R [ns]	C_E [ns]	C_w [ns]	C [ns]	T [ms]	p
CylNumObserver	908	573000	25	573933	1000	1
IgnitionSWCSync	958	2461000	195	2462153	1000	2
MassAirFlowSWC	908	86000	28	86936	50	1
ThrottlseSenseSWC	908	169000	55	169963	50	2
APedSensor	908	482000	55	482963	50	1
APedVoterSWC	55	144000	28	144083	100	1
ThrottleCtrl	990	2892000	55	2893045	100	2
ThrottleActuator	1013	2957000	83	2958096	100	1
BaseFuelMass	1148	3188000	28	3189176	100	2
ThrottleChangeSWC	1060	2957000	83	2958096	100	1
TransFuelMassSWC	1148	3188000	28	3189176	100	2
IgnitionSWC	1060	2269000	25	2270085	1000	1
TotalFuelMassSWC	988	677000	28	678016	100	1
OperatingModeSWC	965	19641000	390	19642355	200	2
IdleSpeedCtrl	833	843000	240	844173	200	1
APedSensorDiag	908	118000	0	118908	1000	1
InjBattVoltCorrSWC	28	274000	28	274056	1000	1
InjectionSWC	985	1651000	195	1652180	1000	2

Table 2: Parameters of the industrial use case.

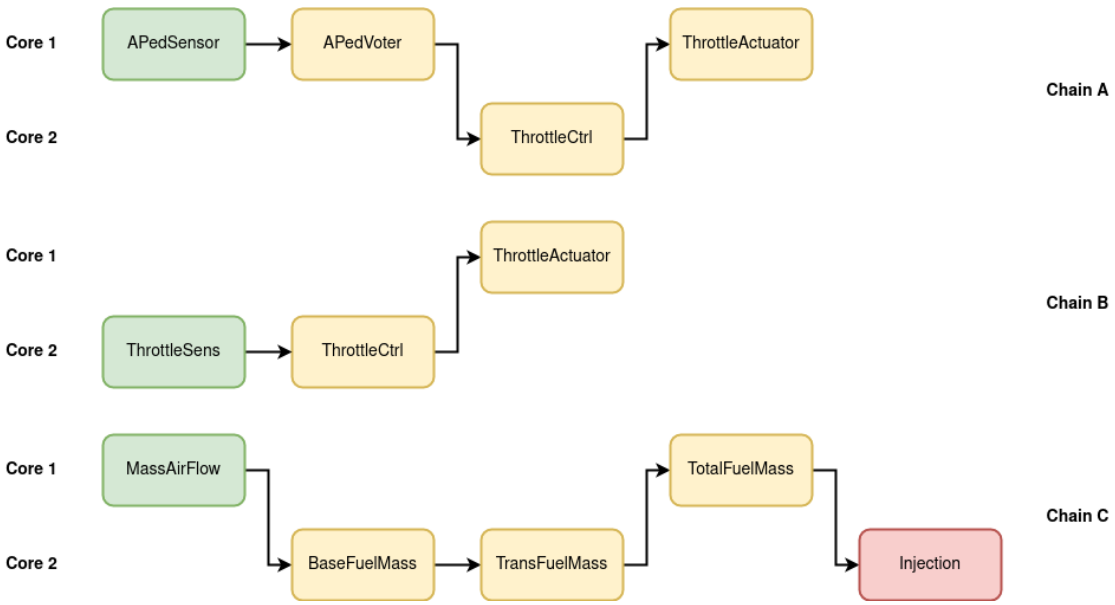

Figure 16: Task chains in the industrial use case.

6.2.2 Results of the Industrial Use Case

Table 3 shows the maximum inter-core data-propagation delay for each inter-core producer-consumer pair when The Optimization Engine, The Delay Minimization Heuristic and Rubus ICE is applied to the industrial use case, table 4 show the maximum age delay for each chain and algorithm, table 5 shows the average solving times. A delay of 0 in table 3 indicates a perfect result for that producer-consumer pair. It means that that for each instance, the start of the consumer is

always equal to the end of the producer. There is never any delay between the two communicating tasks. In this sense, the Optimization Engine has produced a perfect schedule in terms of inter-core data-propagation delay.

Task Chain	(Producer, Consumer)	Optimization Engine [ms]	Delay Minimization Heuristic [ms]	Rubus-ICE [ms]
Chain A	(APedVoterSWC, ThrottleCtrl)	0	19.18	54.86
	(ThrottleCtrl, ThrottleActuator)	0	0.001	91.42
Chain B	(ThrottleCtrl, ThrottleActuator)	0	0.001	91.42
Chain C	(MassAirFlowSWC, BaseFuelMass)	0	21.99	91.42
	(TransFuelMassSWC, TotalFuelMassSWC)	0	0	47.21
	(TotalFuelMassSWC, InjectionSWC)	0	1.760	848.8

Table 3: Maximum inter-core data-propagation delay for each scheduler.

Task Chain	Optimization Engine [ms]	Delay Minimization Heuristic [ms]	Rubus-ICE [ms]
Chain A	106.69	58.744	203.53
Chain B	57.640	58.745	153.53
Chain C	201.65	10.198	153.80

Table 4: Maximum age-delays for each scheduler.

The Optimization Engine	Delay Minimization Heuristic
421.3ms	0.788ms

Table 5: Solving time.

7. Discussion

7.1. Experiment on Synthetic Test Cases

As seen in figure 12, which show the mean delay, there is a very slight increase in delay as utilization is increased, although the results are otherwise similar across all utilization levels. This is likely due to the design of the test cases; the utilization only decides execution time, as defined in equation 6, and nothing else. The number of task chains and, on average, the number of jobs remain the same. This means that the average number of inter-core communications remain the same as well, which should be the parameter which affects inter-core data-propagation delay the most.

Inter-core data-propagation delays are significantly lower when scheduling with the rules than without them. Not only are they lower but they also remain flat when increasing the core count, unlike from when they are not used, at which case the inter-core data-propagation delays are shown to increase with respect to core count.

A possible reason that inter-core data-propagation delays increase with core count when scheduling without the rules is that dividing execution over more cores increases the chance for producer and consumer jobs to miss each other by running concurrently on different cores. Such a case is demonstrated in figure 9. The precedence constraints laid out in section 5.2. guard against this occurring when the rules are applied.

The plots of the maximum delays, 13 show similar behaviour as the mean delays, but are very shaky. This makes sense since each data point only represents a value from one test case, which is the maximum delay. In other words, the maximum delay is subject to more randomness than the mean delay. The minimum delays, shown in figure 14 show zero or near-zero for all data points. This is also expected behaviour, since when thousands of tests are involved in each data point, it is very likely that at least one test returns no delay.

The rules are effective at reducing inter-core data-propagation delays over a wide range of core counts, however, when additional cores are added they are not being put to use in such a way as to bring the inter-core data-propagation delay even further down, even though solutions with minimal or no inter-core data-propagation delay should exist when the number of cores is very high in proportion to the number of tasks, such as when the core count is equal to 8, meaning most cores only have one task assigned to them.

While the rules are effective at reducing delay, they do have negative effect on schedulability, as seen in figure 15. This is expected, since the rules provide exception to the default of scheduling earliest deadline first, which is better for schedulability. When the rules are applied, less test cases can be scheduled feasibly by the scheduler. On average, the number of feasible schedules is reduced by 26.3 percent (not percentage points) when the rules are applied as compared to when they are not.

7.2. Industrial Use Case

In regard to inter-core data-propagation delays, the Delay Minimization Heuristic managed to significantly reduce inter-core data-propagation delays compared to Rubus-ICE as expected, since Rubus-ICE isn't designed for inter-core data-propagation delay optimization. Altough, the Delay Minimization Heuristic is outperformed by the Optimization Engine which produces an ideal schedule in terms of inter-core data-propagation delay. As is characteristic of a heuristic, it can provide a solution significantly faster than the Optimization Engine. These results, together with the results from the synthetic test cases, highlight the capacity of the Delay Minimization Heuristic to provide schedules with small, but not fully optimized inter-core data-propagation delay in a short amount of time.

The Delay Minimization Heuristic performed surprisingly well in terms of age delay, outperforming both the Optimization Engine and Rubus-ICE, considering that wasn't a property it was intended to minimize. However, it needs to be stated that it can not prove a general capacity of the Delay Minimization Heuristic to outperform the other algorithms as this is data from only a single use case. Although, it does raise the question whether the Delay Minimization Heuristic can be used for this purpose.

8. Conclusions

In this thesis a heuristic algorithm was designed for solving partitioned scheduling problems in homogeneous multi-core systems, assuming the phased execution model. The goal was to optimize for the minimization of inter-core data-propagation delays. In accordance with the System Development research method, a prototype scheduling tool which implements the heuristic was developed in the C programming language, which served as proof of concept as well as subject of experiment. The scheduler was subject to experiment by evaluating it on randomly generated IP-free test cases as well as on an industrial automotive use case where it was compared with a scheduler which implements constraint programming, the Optimization Engine. The delay optimization heuristic was proven to be able to produce schedules with significantly lower inter-core data-propagation delays compared to algorithms that are naive to inter-core data-propagation delays, while having a significantly lower solving time than the Optimization Engine, however it was unable to produce schedules that are ideal in terms of inter-core data-propagation delay, even when such solutions exist.

9. Future Work

9.1. Investigating End-to-End Delays

The Delay Minimization Heuristic performed well in regard to minimizing age-delay on the industrial use case. End-to-end delays such as age delay are an important metric in automotive embedded systems, therefore it would be interesting to see if other types of end-to-end delay, namely reaction delay, are low in schedules generated by the Delay Minimization Heuristic and more importantly, to see if they are low for task-sets in general. End-to-end delays are highly related to inter-task delay, so it would make sense if the algorithm does perform well in terms of end-to-end delays. Answering this would require an experiment where the algorithm is tested in scale on a wide range of different test cases.

9.2. Sporadic Tasks

Tasks that handle safety critical functions in embedded systems such as the airbag suspension in a car is often sporadic. Execution of safety critical tasks such as this need to be guaranteed at design time. A sporadic task is characterized by an execution time, an upper limit on its invoke rate as well as a hard deadline, relative to the moment of its release. Extending the functionality of the scheduler to also be able to guarantee the execution of a set of sporadic tasks in addition to its already existing functionality would widely extend its applicability in the automotive domain.

9.3. Global Scheduling

In this thesis, partitioned scheduling was assumed over global scheduling, as global scheduling is a source of overhead and timing unpredictability. Task-to-core allocation was assumed to be done beforehand. However, the use of global scheduling would enable the algorithm to be adjusted in such a way that the core on each load is balanced. In addition, being able to reallocate tasks to cores provides another way in which inter-task delay can be further reduced.

The necessary modifications to enable the Delay Minimization Heuristic to solve global scheduling problems would mean that the core of the algorithm will likely be more similar to the Memory Centric Heuristic, the algorithm from which it was originally inspired, which is a global scheduling algorithm.

10. Acknowledgments

I would like to give my deepest thanks to the supervisors of this thesis, Saad Mubeen and Matthias Becker. The guidance and feedback they have provided during the course of this thesis is greatly appreciated. I would also like to thank my examiner Mohammad Ashjaei who provided me with important feedback in the midterm meeting. Also, I would like to thank my parents for their support throughout the course of my studies.

Lastly, I would like to extend special thanks to my friends who took the time to proofread my thesis and who have been cheering me on the entire way.

References

[1] S. Jonas, L. John, L. Jakob and S. Patrik, *Embedded Multicore: An Introduction*. 2009, pp. 8–15. [Online]. Available: https://www.nxp.com/files-static/32bit/doc/ref_manual/EMBMCRM.pdf.

[2] P. Gai and M. Violante, 'Automotive embedded software architecture in the multi-core age,' in *2016 21th IEEE European Test Symposium (ETS)*, 2016, pp. 1–8. DOI: 10.1109/ETS.2016.7519309.

[3] L. L. Bello, S. Saponara, S. Mubeen and R. Mariani, 'Recent advances and trends in on-board embedded and networked automotive systems,' *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, Feb. 2019.

[4] L. Hasanagic, T. Vidovic, S. Mubeen, M. Ashjaei and M. Becker, 'Optimizing inter-core data-propagation delays in industrial embedded systems under partitioned scheduling,' in *26th Asia and South Pacific Design Automation Conference*, Jan. 2021. [Online]. Available: http://www.es.mdh.se/publications/5912-.

[5] S. Mubeen, E. Lisova and A. V. Feljan, 'Timing predictability and security in safety-critical industrial cyber-physical systems: A position paper,' *Applied Sciences–Special Issue "Emerging Paradigms and Architectures for Industry 4.0 Applications"*, vol. 10, no. 3125, pp. 1–17, Apr. 2020. [Online]. Available: http://www.es.mdh.se/publications/5808-.

[6] T. Vidović and L. Hasanagić, *Tighter inter-core delays in multi-core embedded systems under partitioned scheduling*, Mälardalen University, 2020. [Online]. Available: http://www.diva-portal.org/smash/get/diva2:1438299/FULLTEXT01.pdf.

[7] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis and T. Nolte, 'Contention-free execution of automotive applications on a clustered many-core platform,' in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 14–24. DOI: 10.1109/ECRTS.2016.14.

[8] *Mpc5777c reference manual*. [Online]. Available: https://www.mouser.com/pdfDocs/NXP_MPC5777C_RM.pdf.

[9] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, 2nd. Springer Publishing Company, Incorporated, 2010, ISBN: 9789400702561.

[10] S. Baruah, M. Bertogna and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Jan. 2015, ISBN: 978-3-319-08695-8. DOI: 10.1007/978-3-319-08696-5.

[11] M. Becker, D. Dasari, S. Mubeen, M. Behnam and T. Nolte, 'Synthesizing job-level dependencies for automotive multi-rate effect chains,' in *The 22th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2016. [Online]. Available: http://www.es.mdh.se/publications/4368-.

[12] M. Becker, S. Mubeen, D. Dasari, M. Behnam and T. Nolte, 'End-to-end timing analysis of cause-effect chains in automotive embedded systems,' *Journal of Systems Architecture*, vol. 80, pp. 104–113, 2017, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.09.004.

[13] ——, 'A generic framework facilitating early analysis of data propagation delays in multi-rate systems,' in *in Proceedings of the 23th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Invited Paper, 2017.

[14] 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 *CRTS Workshop*, Dec. 2008.

[15] S. Mubeen, J. Mäki-Turja and M. Sjödin, 'Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study,' *Computer Science and Information Systems, ISSN: 1820-0214*, vol. 10, no. 1, pp. 453–482, Jan. 2013.

[16] C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer and R. Davis, 'A survey of timing verification techniques for multi-core real-time systems,' *ACM Computing Surveys*, vol. 52, pp. 1–38, Jun. 2019. DOI: 10.1145/3323212.

[17] C. Schulte and M. Carlsson, 'Chapter 14 - finite domain constraint programming systems,' in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, F. Rossi, P. van Beek and T. Walsh, Eds., vol. 2, Elsevier, 2006, pp. 495–526. DOI: https://doi.org/10.1016/S1574-6526(06)80018-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574652606800180.

[18] D. Thomas, L. W. Howes and W. Luk, 'A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation,' in *FPGA '09*, 2009.

[19] J. F. Nunamaker and M. Chen, 'Systems development in information systems research,' in *Twenty-Third Annual Hawaii International Conference on System Sciences*, vol. 3, 1990, 631–640 vol.3. DOI: 10.1109/HICSS.1990.205401.

[20] S. Kramer, D. Ziegenbein and A. Hamann, 'Real world automotive benchmarks for free,' Jul. 2015.

[21] P. Emberson, R. Stafford and R. Davis, 'Techniques for the synthesis of multiprocessor tasksets,' English, in *WATERS workshop at the Euromicro Conference on Real-Time Systems*, 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems ; Conference date: 06-07-2010, Jul. 2010, pp. 6–11.

[22] S. Mubeen, J. Mäki-Turja and M. Sjödin, 'Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study,' *Computer Science and Information Systems*, vol. 10, no. 1, 2013.

[23] S. Mubeen, H. Lawson, J. Lundbäck, M. Gålnander and K. Lundbäck, 'Provisioning of predictable embedded software in the vehicle industry: The rubus approach,' in *4th IEEE/ACM International Workshop on Software Engineering Research and Industrial Practice*, May 2017.

[24] P. Dziurzanski, A. Singh, L. Indrusiak and B. Saballus, 'Benchmarking, system design and case-studies for multi-core based embedded automotive systems,' Jan. 2016.