Data Propagation Delay Constraints in Multi-Rate Systems – Deadlines vs. Job-Level Dependencies

Tobias Klaus*, Florian Franzmann*, Matthias Becker‡, Peter Ulbrich*
*Friedrich-Alexander University Erlangen-Nürnberg (FAU), Distributed Systems and Operating Systems
Email: {klaus, franzmann, ulbrich}@cs.fau.de

‡KTH Royal Institute of Technology, Electronics and Embedded Systems, School of ICT
Email: mabecker@kth.se

ABSTRACT

Many industrial areas are faced with a continuous increase in system complexity, while systems need to satisfy stringent timing requirements, which are traditionally based on the tasks' local deadlines. However, correct functionality is subject to high-level timing requirements on data propagation through a set of semantically related tasks. Since distributed concurrent engineering is often used to deal with the complexity of such systems, violations of data propagation delay constraints are only visible at late development stages, where changes in system design become increasingly expensive.

In this paper, we leverage job-level dependencies (JLDs) that can be specified at early development stages to guarantee data propagation delay constraints. Therefore, we present an approach that extends the *Real-Time Systems Compiler* to enforce the JLDs in actual multicore schedules. This strategy enables us to perform extensive evaluations of the effectiveness of JLDs in combination with contemporary allocation and scheduling algorithms, where we observed schedulability improvements of up to 42%. Additionally, we identified the effect of the number of available cores on the data age.

ACM Reference Format:
Tobias Klaus*, Florian Franzmann*, Matthias Becker‡, Peter Ulbrich*. 2018. Data Propagation Delay Constraints in Multi-Rate Systems – Deadlines vs. Job-Level Dependencies. In *26th International Conference on Real-Time Networks and Systems (RTNS '18), October 10–12, 2018, Chasseneuil-du-Poitou, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3273905.3273923

1 INTRODUCTION

The majority of embedded applications is subject to strict timing constraints. Here, not only the correctness of the computed result is of importance but also their availability at the correct time. The main focus typically lies on the local deadline of individual periodic tasks that are scheduled by an operating system. However, many application domains require further timing guarantees on the propagation of data through a chain of tasks (so-called cause-effect chains) [12, 15, 24, 41]. Challenges arise, as the different tasks are often independently triggered, possibly at different periods, which leads to complex over- and under-sampling situations that make their timing analysis cumbersome.

In the automotive industry, the complexity and number of software functions that are integrated with a modern car are steadily increasing. As the software development process is driven by the distributed concurrent engineering paradigm [34], different functionality is developed by different vendors and integrated into the system at a later stage by the original equipment manufacturer (OEM). This isolation means that during the software development process detailed information about the hardware platform or other software applications that will share the same platform are unknown. While timing analysis methods are available at the implementation level [12, 19, 28] (where all functionality is integrated, and complete system information is available) vendors cannot directly verify data propagation delay constraints during the development process as information that is required by these timing analysis engines is not available. Hence, if these timing constraints are violated this is typically detected only at the implementation level, late in the development process. These violations can increase design costs significantly, as the cost of design changes massively increases with each development level [26, 39]. One approach to circumvent this challenge is proposed in [7, 8], where methods are introduced that allow for translating the timing constraints on the data propagation into precedence constraints of a selected task's jobs, expressed as *job-level dependencies* (JLDs). This transformation is agnostic of the concrete hardware platform and only requires knowledge about the tasks that are involved in a particular cause-effect chain. Though the theoretic approach of JLDs is sound and the associated guarantees can be trusted the question remains how they affect scheduling at the end of the development cycle and how close their estimate is to the "real" maximal data-ages of specific schedules. Here the Real-Time Systems Compiler (RTSC) [37] comes into play as it bridges the gap between high-level system analysis performed in [7, 8] and concrete schedules aimed at a specific real-time operating systems (RTOSes) and hardware platform. This process is done by automatic application of contemporary allocation and scheduling algorithms without further human interference. This work presents a study of the interplay between different allocation and scheduling algorithms on a real implementation and the generated JLDs to meet data propagation delay constraints.

Contributions: In this work, we investigate the challenges of integrating the data propagation delay constraints into practical implementations by the RTSC [37]. With the RTSC, different task-allocation and scheduling methods can be applied to generate static schedules. In general, scheduling methods do not consider data propagation delay constraints in their decision process. These

This work is partially supported by the German Research Foundation (DFG) under grants no. SCHR 603/9-2, the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR89, Project C1) and the Bavarian Ministry of State for Economics under grant no. 0704/883 25 (EU EFRE funds).

RTNS '18, October 10–12, 2018, Chasseneuil-du-Poitou, France

ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273923

delays not only depend on the tasks that are involved in the cause-effect chain, but also on the actual execution order of the individual task's jobs which results from the applied scheduling algorithm, as well as on the allocation of tasks to cores [12]. Consequently, different allocation and scheduling algorithms can yield various data propagation delays and that a scheduling algorithm that performs well under consideration of task-local deadlines may experience degraded performance when additional data propagation delay constraints are imposed on the system.

JLDs can be generated agnostic of the underlying hardware platform and scheduling algorithm [7, 8]. However, due to this abstract system knowledge, generated JLD sets that, in theory, always result in data propagation delays smaller than the constraints, might not be schedulable on a concrete platform. In this case, the applied scheduling algorithm does not find a valid schedule under consideration of the JLDs, or the number of available processing cores is not sufficient.

An extension of the RTSC is presented that considers JLDs as additional scheduling constraints. For this configuration, system configurations can be generated that utilize a *time-triggered* backend based on the Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (Litmus^{rt}).

Extensive evaluations are performed that compare generated static schedules (based on several heuristics, as well as optimal algorithms) with and without the extension for JLDs. We show that traditional allocation and scheduling algorithms do not influence the resulting data propagation delay constraints and that augmenting the task-set with JLDs increases the system schedulability (task-local deadlines and data propagation delay constraints) by up to 42%.

Outline: The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3 the relevant background information is presented. An overview of our approach is presented in Section 4 and the investigated allocation and scheduling algorithms are discussed in Section 5, followed by the implementation of timing analysis and schedulability test in Section 6. Section 7 presents our evaluation results, and conclusions are drawn in Section 8.

2 RELATED WORK

Scheduling and timing analysis of periodic multi-rate applications is essential in many industrial domains, such as automotive [18] or avionics [13].

Several works address the timing analysis of data propagation delays. Feiertag et al. [12] present calculations for maximum data propagation delays in real-time systems under register communication. They further identify different data propagation delay semantics and highlight their respective importance for system engineers. This analysis is subsequently implemented in several automotive tools [20, 29]. While this work focuses on the implementation level, Becker et al. [5, 7] present a framework to compute data propagation delays at various levels of timing information, and Forget et al. [13] study the formal verification of data propagation delays in multi-periodic synchronous models. Frise et al. [17] present a timing analysis approach for data propagation delays in automotive multi-core platforms under different communication models that are based on constraint modeling.

Mubeen et al. [27] focus on the selection of tasks period in order to meet data propagation delay constraints. Schlatow et al. [38] assign priorities, offsets and processor mapping to tasks such that data propagation delay constraints are met on a multicore platform. The Logical Execution Time (LET) model [21] is further considered to realize deterministic data propagation delay in automotive systems as it decouples the data propagation delay from the tasks execution [10, 18].

Alternatively to influencing data propagation on the implementation level, Becker et al. [7, 8] analyze all possible data propagation paths in a system and then generate an ordering of selected task's jobs such that data propagation delay constraints are met. JLD constraints are considered in [14] for fixed-priority scheduled systems, and in [30] for dynamic priority scheduled systems. Both works target single processor systems. Time-triggered schedules subject to such precedence constraints are further investigated for many-core platforms in [9, 33, 35].

The work presented in this paper differs from related work in that it extends the design flow of an existing compiler-based tool, the RTSC [37], to consider JLD constraints that are generated by the methods described in [7] such that the applications' data propagation delay constraints are met. Integration into the RTSC leverages the already existing flexibility of this platform, such as support for a large number of available scheduling and allocation algorithms and executing resulting schedules based on Litmus^{rt} [11] and other platforms. A systematic evaluation of a large number of applications that are subject to data propagation delay constraints is performed using various combinations of allocation and scheduling algorithms, targeting a multicore platform. Evaluations focus on metrics that are important from a theoretical as well as practical perspective.

3 BACKGROUND

This section provides the required background information, with the system model and the data age constraint that are the main focus of the paper. We further describe the different parts of the Real-Time Systems Compiler and its transformation mechanisms.

3.1 System Model

This section first describes the basic application model and the data propagation delay constraints that are typically found in automotive systems. In order to transform these timing constraints on the data propagation into direct scheduling constraints, JLDs are used.

3.1.1 Application Model. One application is described by the task set Γ, where Γ contains n periodically activated tasks. A task is described by the tuple $\{C_i, T_i\}$. C_i describes the task's worst case execution time (WCET), and T_i describes the its activation period. Each task has an implicit deadline $D_i = T_i$. The hyperperiod of the task set is described by the least common multiple of all task periods, lcm(Γ). The j^{th} job of τ_i is depicted as τ_i^j.

A cause-effect chain ζ is represented by a directed acyclic graph (DAG), and contains a set of vertices $\mathcal{V}$ and a set of directed edges $\mathcal{E}$. Each vertex represents a task $\tau \in \Gamma$, and each edge constitutes a communication between the two tasks. Such a chain can have forks and joins, but the initial task and the final task must be the same for all paths of the chain [2]. As timing properties of each possible data path are of interest a chain can be decomposed into several sequential chains, in the remainder of the paper, we only focus on sequential chains.

Communication between different tasks is realized via *register communication*. This communication form utilizes shared variables for communication, where a sender task writes to the shared variable and a reader task reads from it. As there is no signaling between tasks, the tasks can execute independently of each other. To further increase the determinism in communication, the tasks execute based on the *read-execute-write* semantics. In this execution model, a task creates local copies of all input variables at the beginning of its execution. During the execution phase, only those local copies

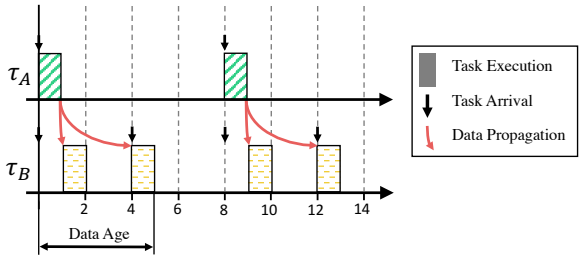


Figure 1: Example of data age in a cause-effect chain containing two tasks.

are accessed. Finally, the output variables are written at the end of the task's execution. This communication model is in line with the *implicit communication* paradigm of AUTOSAR, which is the most-used communication paradigm in automotive systems [24].

3.1.2 Data Propagation Delay Constraints. In order to render the correct functionality of the system, data propagation delay constraints can be specified on a cause-effect chain ζ. This is for example the case in control applications, where sensor data may be sampled by one task, while a second task executes the control algorithm, and finally, a third task triggers the actuator with the updated data. As these tasks may be activated at different periods, over- and under-sampling may occur. Because of this, the same input value may affect the output of the chain multiple times.

Several data propagation delay constraints can be specified [12]. In this work, we focus on the *Maximum Data Age* constraint, as this constraint type is most important for control applications. Our approach focuses on the integration of job-level dependencies (JLDs) in the scheduled system in order to meet data propagation delay constraints. Thus, the approach is applicable to other data propagation delay metrics if the JLDs are selected targeting the respective constraint type, as shown in [8]. The data age is a metric that describes the relative age of data, from sampling by the first task in the chain until the last corresponding output is produced by the last task of the chain. Fig. 1 shows an example of the data age in a system of two tasks. τ_A and τ_B. The two tasks are activated at different periods. τ_A has an activation period of $T_A = 4$ time units and τ_B has an activation period of $T_B = 2$ time units. Hence, over-sampling is observed as τ_B reads its input values more frequently than τ_A produces new values. This can be seen, as the first and second job of τ_B both consume the *same* value that has been produced by the first job of τ_A. In this example, the maximum data age spans from the start of execution of the first job of τ_A until this value has its last effect on the output, when the second job of τ_B terminates.

3.1.3 Job-Level Dependencies. To ensure that all specified data propagation delay constraints are met, JLDs are specified on the task set Γ. A JLD constrains the execution order of specific jobs of two tasks. In this way, possible data propagation between these tasks can be influenced. For example, consider two tasks τ_A and τ_B that are adjacent in a cause-effect chain ζ, where T_B is $2 \cdot T_A$. The analysis of all possible data propagation paths of ζ [7] shows that there exists a data propagating path in which data propagates between the job τ_A^1 and the job τ_B^1. For this path, the maximum possible data age exceeds the specified data age constraint. By specifying a precedence constraint between the task job $\tau_{A,}^2$ and τ_B^1 it is guaranteed that τ_B^1 never reads the data that is produced by τ_A^1, as τ_A^2 overrides the data before τ_B^1 is executed. Consequently, the data propagation path that violates the specified data age constraint is avoided as long as the precedence constraint between the two jobs is met.

A JLD is defined as $\tau_i \xrightarrow{(k,l)} \tau_j$, where τ_i is the sender task, and τ_j is the receiver task. The indices k and l relate to the specific task jobs that are constrained, i. e., the JLD specifies that the job τ_i^k must have finished executing before the job τ_j^l starts. Note that a JLD is always specified concerning the hyperperiod of the two tasks $\text{lcm}(\tau_i, \tau_j)$ and repeats itself over the complete hyperperiod of the task set. If two tasks have the same period, both tasks only execute one job during their hyperperiod $\text{lcm}(\tau_i, \tau_j)$. Thus, k and l are 1.

In [7], a heuristic method is shown that generates a set of job-level dependencies such that all specified data propagation delays are met. This has the advantage that, as long as all specified job-level dependencies are satisfied, the data age constraints are met as well without explicitly considering the data propagation delay constraints during scheduling. In this work, the MECHAniSer[1] tool [6] is used to generate the JLDs based on the methods of [7].

3.2 The Real-Time Systems Compiler

The Real-Time Systems Compiler (RTSC) [16, 37] is a flexible generic real-time systems transformation tool based on the LLVM. It operates directly on the source code of soft, firm or hard real-time applications enriched by a real-time task database and extracts their fine-granular OS and architecture agnostic *intermediate representation* called Atomic Basic Block (ABB) graphs that captures all relevant timing and structural characteristics. Since ABB graphs neither depend on a particular real-time paradigm nor a specific OS, they serve as the basis for arbitrary real-time-invariant-preserving transformations. The ultimate goal is to decouple functional and real-time development of a real-time application and thus to be able to quickly deploy the same application on different hardware, OS, and real-time paradigms and thus quickly assess their impact on the application's performance. Since in this paper we evaluate the impact of different allocation and scheduling algorithms, as well as different multi-core configurations on the worst case data age of event chains, the subsequent presentation of RTSC internals, will focus on generating multi-core time-triggered systems from event-triggered input systems.

3.2.1 Atomic Basic Blocks. The ABB graph representation [36] of real-time systems was inspired by the basic-block intermediate representation found in compilers. While basic blocks begin and end with instructions that are the target or source of a branch in the function-local CFG, ABBs are started and terminated by a branch in the global control flow of the real-time system. Such instructions are called *ABB terminations* and consist of system calls such as triggering tasks, setting and waiting for event flags, mutual exclusion and sending data from task to task. Depending on the ABB termination's semantics the ABBs are connected by appropriate ABB dependencies which then describe the cross-function and cross-task relationships in the real-time system. Consequently an ABB consists of one or more basic blocks of a function. Each ABB has a unique entry basic block, which is the only basic block in the ABB's control flow that may have predecessors in the control flow of the function that are not part of the ABB. Each ABB has at most one exit basic block. Since the semantics of system calls are traced by ABB dependencies the operating system calls can be removed which allows the system to be represented in an OS and hardware agnostic fashion. Each ABB can be executed on its own without further interference with other parts of the system, as long as its dependencies are fulfilled ABBs. This atomicity makes them ideal fine-granular scheduling entities for the RTSC.

[1]The tool is freely available at www.mechaniser.com

3.2.2 Real-Time Task Database. Although ABB graphs already capture all internal, structural properties of the real-time system, these graphs do not yet have a connection to the environment. This connection is established by the *real-time task database*. This database contains *events*, which can either be periodic or non-periodic, and activate a task. Tasks can be attributed with a soft, firm or hard *relative deadline d*. Additionally, periodic events carry a period and jitter, while non-periodic events only have a minimal interarrival time. Tasks are composed of a *root subtask* and zero or more additional subtasks. These subtasks are connected by directed and undirected dependencies and become ready for execution as soon as their parent task's event has occurred, and all of their dependencies are satisfied. The instantiation of a subtask's ABBs are called *jobs* and are created as soon as the subtask becomes ready. Subtasks are decomposed into ABBs by the RTSC, a process which will be described in detail in the next section.

3.2.3 Real-Time Systems Processing. This subsection will present the steps performed by the RTSC to map a source real-time system to the target system. Likes other compilers, the RTSC is composed of source-system-architecture specific front ends, a middle end, and target-system-specific backends.

Front End. The RTSC's front end is responsible for converting the real-time system to the intermediate representation of ABB graphs. First, the identifiers of the subtasks stored in the real-time task database are associated with the respective handler functions in the real-time applications. Next, local ABB graphs are created for individual functions. ABB terminations are identified and basic blocks that would contain one or more terminations in the middle are split. Terminations are found by identifying all system calls in the function, which makes clearly defined system call semantics and knowledge of the called function at the call site mandatory. The resulting ABBs are connected by implicit dependencies, tracing the CFG gleaned from the relationship of the basic blocks, resulting in *local ABB graphs*. These graphs are connected to a *global ABB graph* by identifying compatible ABB terminations, for example, ones that establish a producer-consumer relationship and refer to the same system object. After cleaning all system calls the resulting ABB graphs are entirely OS independent and allow for arbitrary transformations w. r. t. the structure of the real-time system.

Middle End. The goal of the middle end is to prepare the real-time system for the code-generation step in the back end. To this end, an allocation of ABBs to processors and a schedule table for each processor is calculated. Once the RTSC enters the middle end, all transformations take place in the context of the target system. This is important since properties like the WCET of individual ABBs, which is necessary for scheduling and allocation, can only be determined for the target architecture, and not in a generic fashion.

The first step in the middle end is to calculate the hyperperiod of the real-time system as the least common multiple of the periods of all events. To fill the hyperperiod, ABBs and connecting ABB dependencies are cloned accordingly, which is necessary since in scheduling and allocation ABBs serve as jobs, and each job can only be scheduled precisely once per hyperperiod. Next, the global ABB graph is linearized. This prevents control-flow-graph structures like separate branches that can never be executed within the same hyperperiod from being scheduled without need. Dependencies are moved out of loops and branches and logical guards are inserted that retain their semantic. After that mutually exclusive branches are merged into one ABB, creating a *linearized ABB graph*. This is needed to facilitate the WCET analysis of each ABB, which is done by the external tools aiT from Absint[2] or platin [22], depending on the target architecture. Additionally WCET annotations can be used to skip this computational expensive task in the case that WCETs are already known.

Now all information necessary for allocating and scheduling the ABB graph is available. The RTSC offers multiple heuristics and optimal approaches for solving the allocation and scheduling problem and generating a time-triggered schedule for each of the available processing nodes. Since the impact of allocation and scheduling algorithms on the data age of event chains is the subject of this paper we will go into greater detail on this topic in Section 5 and assume for now that we found a feasible assignment and schedule.

To generate an executable real-time system, the RTSC's ABB graphs still have to be post-processed. Not every ABB can be executed directly since not every ABB constitutes the beginning of a function. Wrapping *every* ABBs in functions, comes with a runtime cost as additional code is inserted and function state has to be transferred. Therefore during scheduling measures are taken that make it unlikely that functions are scheduled in an interleaving manner. In post-processing time intervals that have been assigned to individual ABBs are merged wherever this is advantageous. This way, whenever two neighboring ABBs are connected by control flow already present in the basic blocks, a combined busy interval is created that contains both ABBs, effectively removing one entry from the schedule table. In some cases, however, despite all the steps the RTSC takes to avoid this kind of situation, an ABB that is not a function entry ends up at the start of an interval. Since the timer interrupt handler has to enter this ABB by executing a function call, a function wrapper for the interval is generated that takes the necessary state for continuing the control flow as a parameter. Likewise, whenever an interval ends with an ABB that is not a function exit, the RTSC generates code that stores the necessary state for continuing execution. The result of all performed processing steps allows the RTSC's backends to generate executable code.

Back End. In the backend, the RTSC generates configuration files and an application scaffolding for the real-time application. The configuration files contain the schedule tables. In a final step before generating the executable code for the target system all remaining annotations are removed, and, after that, assembly code is generated. Besides *OSEKTime* the RTSC's backend is capable of generating time-triggered systems aiming Litmus^{rt} which was the target platform for the evaluation performed in this paper.

4 APPROACH

In order to satisfy data propagation delay constraints in real-time systems, specifying JLDs is one proposed method that augments a traditional task model such that data propagation delay constraints are met [6–8]. This approach has the benefit of being agnostic of the underlying hardware platform or scheduling algorithm. If specified JLDs are met by any scheduler on any platform, the specified data propagation delay constraints are implicitly met as well. Though this is beneficial if such design decisions are not yet decided, the question arises what happens to data propagation delays on the actual hardware and real-time operating system (RTOS).

To incorporate the generated JLDs into a complete development chain, the RTSC is chosen as a shortcut from high-level system analysis to the evaluation of concrete systems design. As the generated JLDs, the ABB graph itself is agnostic of a concrete hardware platform and scheduling algorithm [16, 37] but *automatically* applies

[2] https://www.absint.com/ait/

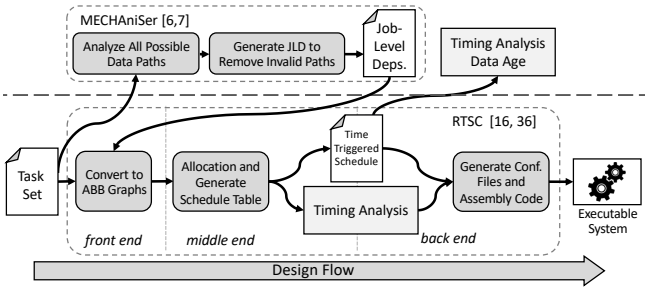


Figure 2: Overview of the design flow.

the complete development chain including allocation and scheduling algorithms as well as adaption to hardware platforms to the real-time application in question. Thus, the combination of JLDs and RTSC has the potential to address a broad spectrum of hardware platforms and scheduling strategies and thus analyze and compare them for the same systems.

In order to assess the effect of different allocation and scheduling algorithms on systems that are subject to data propagation delay constraints, the design flow of the RTSC is augmented to incorporate the JLDs that are generated. This is done using the MECHAniSer tool [6] which implements the approach of [7] to generate JLDs. Fig. 2 shows the traditional workflow of the RTSC (below the dashed line) and the workflow that integrates the JLDs as an additional input to the RTSC. In order to obtain the actual data age values of the generated systems, an additional timing analysis engine is implemented as part of the RTSC (a more detailed discussion follows in Sec. 6.1).

From Fig. 2, it can also be seen that this approach does not change the original workflow of the RTSC if no JLDs are generated, and all RTSC benefits (such as system generation and timing analysis of the generated systems) are retained.

5 ALLOCATION AND SCHEDULING

To put the Multi-rate Effect CHains ANaliSER (MECHAniSer)'s guarantees and the effect of JLDs to the test with real schedules a variety of combinations of allocation and scheduling algorithm has been evaluated.

5.1 Allocation

Allocation by Peng and Shin's optimal algorithm (PS). For optimal (if an allocation and a schedule exists that is valid and feasible, it will be found) allocation of ABBs to the available processing cores the RTSC uses a specialized branch and bound algorithm based on the one due to Peng et al. [32]. Branch and bound creates an initial solution from which refined solutions are derived successively, potentially exploring the complete solution space. This property ensures the algorithm's optimality but also means that its worst-case runtime is exceptionally high. However, on average, branch and bound finds an acceptable solution quite quickly.

In Peng et al.'s algorithm two kinds of solutions exist: Incomplete ones in which only some jobs have been assigned to processors, and complete ones in which all jobs have been assigned. Incomplete solutions have to be refined further while complete ones are candidates for a feasible assignment.

Peng et al.'s algorithm calculates refined solutions from an incomplete one by assigning the next unassigned job to each processor in turn, i. e., for a real-time system that is to be mapped to a four-processor machine, from each incomplete solution four refined solutions are derived. For each incomplete solution a lower bound of the cost is calculated by generating a local schedule for each processor and estimating a *regular measure*[3] of its cost from the completion time, deadline and release time of each job. The lower bound of the cost of a solution is calculated as the maximum of the cost of all its jobs. The cost of an incomplete solution is only a lower bound of the cost since the cost of unassigned jobs is taken into account under optimistic assumptions, and a job's predecessor's release time is used as a lower bound for its completion time if the predecessor runs on a different processor than the successor. Even for complete solutions, the resulting schedule may, therefore, be invalid since successors may start before their predecessor's result is available.

An upper bound of the cost is calculated for complete solutions. If an incomplete solution has a lower bound of the cost that is higher than the current best complete solution's upper bound, it is discarded immediately. This is justified since the cost of a refined solution can only be the same or higher than its parent's cost. Furthermore, discarding solutions means that large parts of the solution space do not have to be explored since all children of a discarded solution are eliminated from the search space as well. Peng et al.'s algorithm terminates once a complete solution has been found and no incomplete solution remains that can result in better cost than the current best complete solution. The result of Peng et al.'s algorithm minimizes the maximum cost of the assignment, and, given an appropriate measure of cost, the probability of missing a deadline at runtime, even if timing parameters like the estimated WCET are violated [31]. A regular cost often used for real-time systems and thus also in the RTSC is the Normalized Task Response Time (NRT). Thus Allocation by Peng and Shin (PS) allocation minimizes maximal NRT of the allocation. Due to its general approach, additional optimization subject can be specified. The default implementation, for example, is also parametrized to minimize the usage of cores and thus leaves cores unused if not necessary for schedulability. In contrast, the modified PS/maxCore is parametrized to use as many cores as possible.

Heuristic Approach. In addition to this complex and resource-consuming near-optimal solution, several well-known heuristic algorithms for allocation have been implemented. Each of these heuristics computes the utilization u_{ABB} as the fraction of its WCET C_{ABB} and its relative deadline d_{ABB} ($u_{ABB} = \frac{C_{ABB}}{d_{ABB}}$) of each ABB. An ABB *fits* on a processing node if the sum of the already allocated ABBs on this node adding that of the current node is less than one ($\sum u_{alloc} + u_{ABB_{curr}} \leqq 1$). The FirstFit algorithm always starts at the first processing node and places the current ABB at the first core that has enough capacity left [23]. A slight variation of this approach is the NextFit algorithm [23]. Here the core that the last ABB has been allocated to is saved and serves as a starting point for the allocation of the next ABB, which is again placed on the first suitable core. Instead of allocation to the first fitting processor, the BestFit and WorstFit algorithms iterate over each node for every ABB and compare the resulting load on each node. The BestFit algorithm then allocates the ABB to the core that would have the highest resulting computational load i. e., the core's slack fits *best* to U_{ABB}. In contrast, WorstFit locates the current ABB to the core with the lowest resulting load and thus distributes the computational load evenly to the available cores. A more naïve heuristic allocation is RoundRobin. It iterates over all cores and evenly distributes ABBs on cores. As long as the ABB fits, no further qualification is performed.

[3] A measure of the cost of a job is regular if increasing completion time means non-decreasing cost. [4] This is a precondition for the correctness of the branch and bound scheme.

It is worth noting that in contrast to the PS allocation algorithm none of the heuristics consider dependencies and thus also neglect the JLDs.

5.2 Scheduling

After an allocation has been found a feasible and valid schedule for the ABBs on each core has to be determined. Again we compared two different algorithms for scheduling data-age constrained systems.

Earliest-Deadline First Scheduling. The first and most generic scheduling algorithm used in the RTSC is based on the well-known Earliest Deadline First [25] scheduling algorithm and the principle of branch and bound [1]. For each moment in time, the algorithm prioritizes the next ABB that is runnable, and that has the most urgent deadline. The EDF-implementation in the RTSC supports processor-local as well as cross processor dependencies and the generation of preemptive schedules. Cross-processor dependencies are repaired in a way similar to the optimal allocation algorithm. It, therefore, is optimal in the sense that if no dependencies exist, it finds feasible schedules for core allocations whose computational load is below one [1].

minimax Scheduling. The second approach to scheduling uses the cost-optimal schedule calculate during optimal processor allocation [3, 32] as a basis. However in contrast to Peng et al.'s original algorithm the version implemented in the RTSC attempts to repair violated cross-processor dependencies similar to the approach of Abdelzaher et al.: It generates new child solutions from violating complete solutions by shifting the deadline of the predecessor into the past and the release time of the successor into the future where necessary. It ensures that the job that causes the least cost, i. e., has the least NRT finishes last, while at the same time satisfying all directed dependencies. Since this scheduling algorithm is fully integrated with the optimal branch-and-bound algorithm used for processor allocation, it can currently not be combined with the simple heuristic allocation approaches.

6 TIMING ANALYSIS AND SCHEDULABILITY TEST

This section discusses the integration of a timing analysis engine for the data age within the RTSC and a necessary schedulability test that detects JLD settings that lead to unschedulable systems caused by cyclic dependencies.

6.1 Timing Analysis and Implementation

As the RTSC was extended to account for data propagation delay constraints, the RTSC-internal analysis engines need to be extended as well. In the RTSC the goal is to analyze the data age of these systems in a generated schedule.

As a result of the schedule generation using the RTSC, a schedule table is produced. This table includes the *start* and *completion* time of each ABB, as well as its core-mapping. For a job τ_i^j, the start time is denoted as $s(\tau_i^j)$, and the finish time as $c(\tau_i^j)$. With the specified JLDs, the RTSC guarantees that for jobs of tasks that are constrained by a JLD the earliest start time of the successor job is always larger or equal than the latest finishing time of the predecessor job. This means, due to the properties of the JLD generation, a schedule that satisfies all JLDs also automatically meets all specified data propagation delay constraints [7]. As it is often essential how significant the actual worst-case data propagation delay in a system is, timing analysis of the generated system needs to be performed [5, 12].

To extend the timing-analysis engines of the RTSC to analyze the maximum data age, the cause-effect chain semantics and their timing constraints are integrated with the RTSC. This information is needed when analyzing the schedule table.

A new component within the RTSC is responsible for the timing analysis of a cause-effect chains' data age. The maximum data age of the generated systems is computed by traversing backward from each job τ_i^k of the last task τ_i of a cause-effect chain ζ_l. The implemented algorithm recursively selects the first job of its respective predecessor task (in the cause-effect chain ζ_l) that is finishing before the start time of the current job. Once a job of the first task of the chain is reached (let's say τ_j^p), the data age can be computed as: $c(\tau_i^j) - s(\tau_j^p)$. Note that only the first and last job of such a data path is required to compute the data age [12]. The maximum data age of a cause-effect chain ζ_l is the maximum value of all data age values that are computed for each job of τ_i.

6.2 Cyclic Job-Level Dependencies

As the heuristic algorithm [7] that is used to generate the JLDs sequentially assigns dependencies to the different cause-effect chains of the system, cases have been observed in which cyclic dependencies were generated.

Since tasks can be part of multiple cause-effect chains, it is possible that cycles in the graph of all chains' data paths exist. In such cases, the heuristic can specify circular dependencies in the system. While cycles in the graph of all data propagation paths are allowed, cycles in the specified JLDs are not, as no valid schedule is possible that can fulfill such JLDs.

Testing the JLDs that are generated for the system for cycles represents a *necessary* condition for schedulability. I.e., if a cycle exists, no schedule exists that can satisfy all JLDs. This check for schedulability can be efficiently implemented in order to detect unschedulable systems before the complete system generation with the RTSC is triggered.

7 EVALUATION

In this section the evaluation results are presented. First, the system generation process is discussed, and the basic properties of the process are shown. We then evaluate a large number of randomly generated systems using the proposed approach. Here different schedulability criteria are of interest, as well as the resulting response time and data age measures. Finally, the effect of additional cores on the resulting data age is evaluated.

7.1 System Generator

For the experiments, random systems are generated based on characteristics of automotive applications that are reported in [24]. Each system contains randomly generated task sets. Task periods are selected out of the set {1, 2, 5, 10, 20, 50, 100, 200, 1000} ms, and WCETs are selected out of the range $[50, 150]$ μs.

Each generated cause-effect chain can have 1–3 different involved periods, where 20% or the chains contain only tasks of the same period, 40% contain tasks of two different periods, and 40% of the cause-effect chains contain tasks of three different periods. For each period value selected for a chain, 2–5 tasks are involved with a probability of 30%, 40%, 20%, or 10% respectively. The data age constraint for a chain is generated by multiplying a random factor with the chains hyperperiod (i.e., the hyperperiod of all tasks that are part of the chain), in the range $[1.8, 2.5]$. Note that tasks of the

same period always appear sequentially in the cause-effect chain and the same task can be part of multiple cause-effect chains [24]. If the same subset of tasks is part of multiple cause-effect chains, these tasks always appear in the same ordering in both chains.

The system generation is performed based on the following steps:

- Generate the blueprint for each cause-effect chain. This includes the number of activation patterns, and the number of tasks as well as the assigned period for each activation pattern.
- Generate random tasks such that each chain can be filled. I.e., for each activation period, generate the number of tasks that are maximally assigned to any of the cause-effect chain blueprints.
- Generate random tasks until the task set utilization is reached.
- For each cause-effect chain, randomly pick tasks from the task set that have the activation period which is defined by the respective activation patterns of the cause-effect chain.
- Finally, assign a data age factor is selected with uniform distribution in the range of [ageMin, ageMax]. This factor is then multiplied with the hyperperiod of the cause-effect chain to set the data age constraint.

Generating the systems in this manner allows controlling the cause-effect chain characteristics in a precise way.

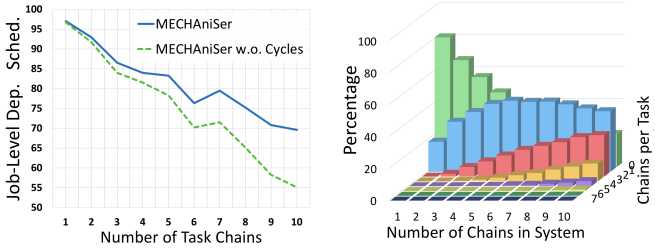
For each system the JLDs are generated using the methods of [6, 7].

7.1.1 Success-Rate JLD Generation and Cyclic Dependencies. In this section, the JLD schedulability (i.e., the algorithm of [7] finds a valid setting for JLDs) is compared in a version with and without a check for cyclic dependencies. The generated systems include task sets of an average utilization of 1.9. 300 random systems generated with a varying number of chains, while all other system parameters are kept constant.

This means, the more cause-effect chains are part of the system, the more interleaved the system gets. I.e., tasks can be part of more than one cause-effect chain. For this experiment, the distribution of how many chains a task is a part of is shown in Fig. 3b in respect to the number of chains in the system.

Fig. 3a shows the relation of systems where the heuristic of [7] reports a JLD configuration, compared against the systems that do not experience cyclic dependencies (as discussed in Sec. 6.2). It can be seen that the systems that are subject to cyclic dependencies are increasing with the number of cause-effect chains in the system. The more entangled the system is, the more likely it becomes that no valid setting for JLDs is found.

As the focus of this work is to analyze the influence of JLDs on the system properties, in the remainder, only systems where dependencies are generated successfully are further considered, as the primary objective is to evaluate the performance of systems with and withoutJLDs in different target systems.



(a) Impact of cyclic dependencies on schedulability.

(b) Average number of tasks that are part of varying numbers of cause-effect chains.

Figure 3: Success rate in generating JLDs, and the distribution of tasks to chains in the experiment.

7.1.2 Generated Task-Sets for the Main Evaluation. To compare the MECHAniSer's analysis with concrete schedules and examine the effect of the heuristic JLD creation we generated 433 systems using the described system generation process. The systems are generated with utilization between 0.6 and 2.0 and an average of 1.2. Each generated system has between 59 and 1000 jobs with an average of 458 and comprises $1-3$ event chains. For all these systems corresponding C code as well as the real-time database needed for the RTSC have been generated. The C code mostly consists of dummy loops that retain the timing properties of the task and enforces data propagation and if necessary JLDs. Then all systems have been analyzed by the RTSC for maximum data age with all available combinations of allocation and scheduling as explained in Section 5 for up to four cores each with and without JLDs. This sums up to a total of 27712 RTSC runs and leads to about 570 CPU-hours on our experiment cluster[4].

7.2 Task-local Deadlines and Maximal Response Times

To compare the schedulability ratio[5], with and without generated JLDs, different allocation and scheduling algorithms have been evaluated on a varying number of available cores. Figure 4a shows the schedulability ratio of systems without JLDs whereas Figure 4b depicts the same with JLDs in effect. In both experiments, the trend looks similar. While fewer than 50% of the systems are schedulable on one core, the schedulability ratio increases to almost 100% for systems with three cores. It can further be seen that the addition of JLDs does not strongly impact the schedulability ratio. The maximum reduction in schedulability (on average over all cores), when adding JLDs can be observed for BestFit with 5%.

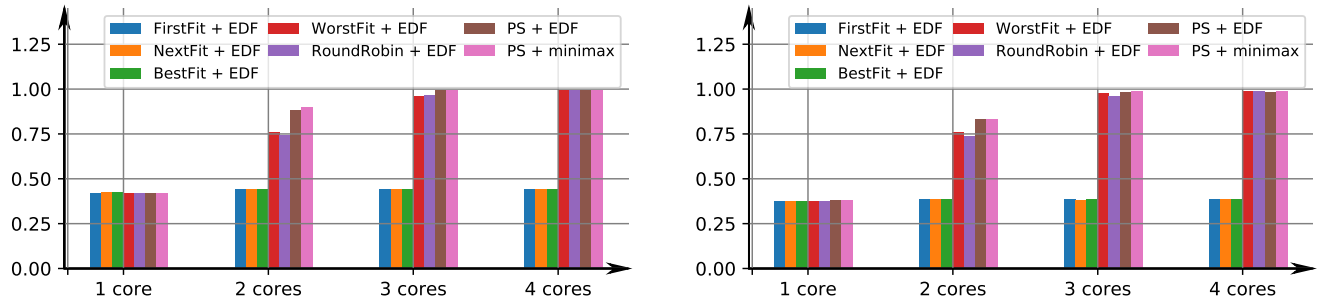
Since the utilization of the systems is below two FirstFit, NextFit and BestFit by design, do not utilize more than two of the available cores and therefore do not improve as more cores become available. In contrast, the WorstFit heuristic which uses as many cores as available is surprisingly competitive to the optimal allocation algorithms when more than two cores are available. Due to the systems' utilization, the most laborious task for allocation algorithms is the case where the additional effort put in the optimal allocation pays of: PS finds for all three scheduler configurations the most schedulable systems.

The same holds true for the maximal normalized response times (max NRT) depicted in Figure 5: Even for one core the optimal scheduling algorithms produce less maximum normalized response times in the mean, but as soon as more possibilities for the allocation exist, the optimal allocation algorithms result in smaller response times, while the difference between EDF and optimal cost-based scheduling is not as strong but exists.

From this experiment, we can see that augmenting the task set with JLDs in order to meet data propagation delay constraints does not strongly impact the schedulability ratio under consideration of task local deadlines only.

[4]In the remainder, selected evaluation results are shown. The complete set of obtained figures is available at www4.cs.fau.de/Research/RTSC/experiments/mechaniserstatic

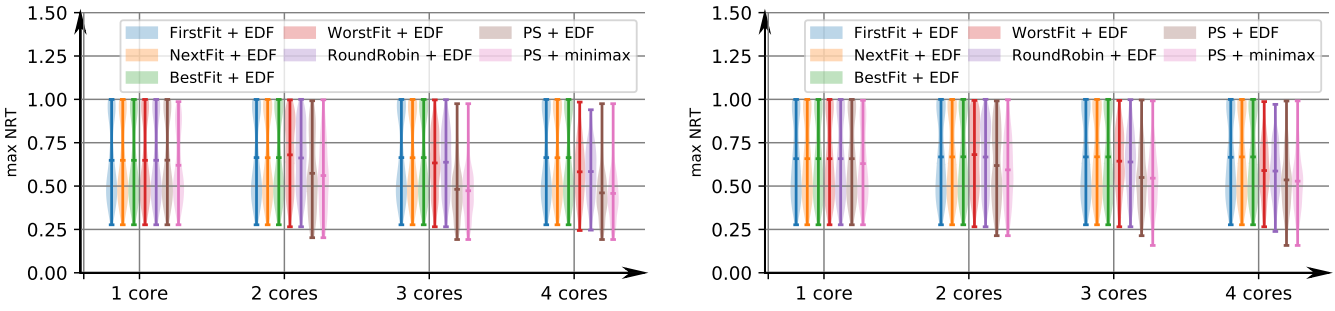
[5]Here, schedulability ratio refers to the traditional schedulability ratio of task sets that meet all task-local deadlines compared to the ones that do not.



(a) Task-local schedulability without JLDs

(b) Task-local schedulability with JLDs

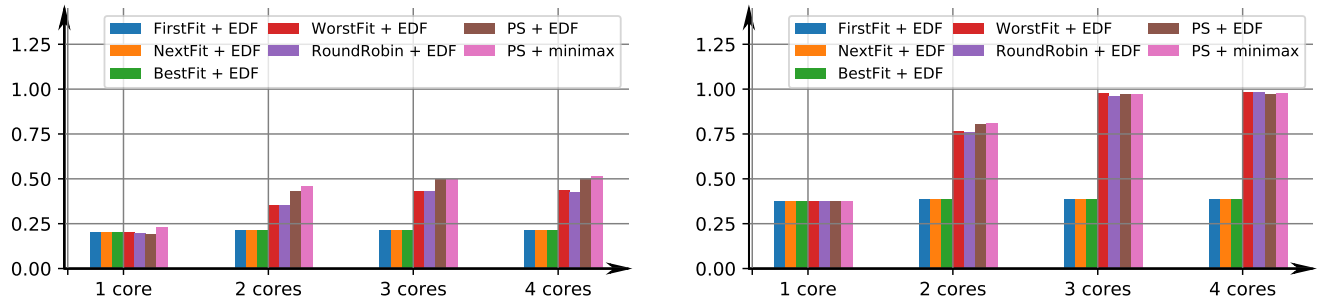
Figure 4: Impact of JLDs on task-local schedulability



(a) Max normalized response time without JLDs

(b) Max normalized response time with JLDs

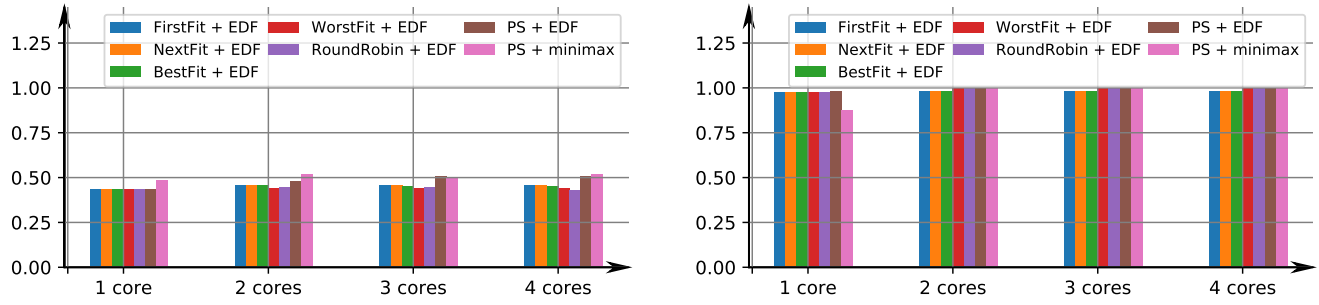
Figure 5: Impact of JLDs on max normalized response time



(a) Combined task-local and data-age schedulability without JLDs

(b) Combined task-local and data-age schedulability with JLDs

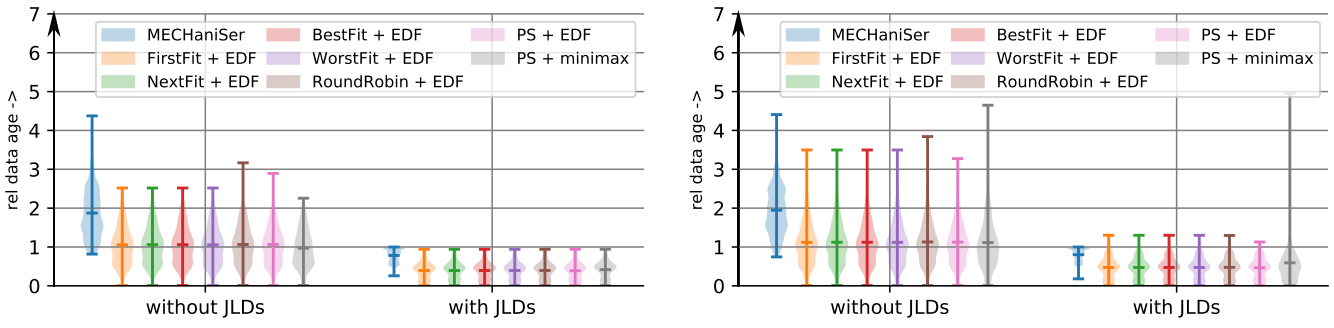
Figure 6: Impact of JLDs on combined task-local and data-age schedulability



(a) Data-age schedulability without JLDs

(b) Data-age schedulability with JLDs

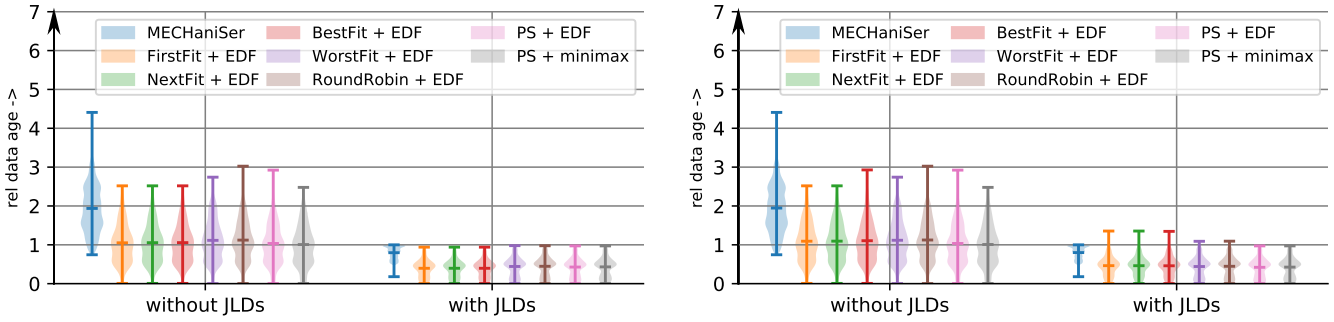
Figure 7: Impact of JLDs on data-age schedulability



(a) Considering only systems that meet all task local deadlines

(b) Considering all systems

Figure 8: Comparison of data ages as guaranteed by the MECHAniSer and the data ages achieved by the various allocation and scheduling algorithms for one core



(a) Considering only systems that meet all task local deadlines

(b) Considering all systems

Figure 9: Comparison of data ages as guaranteed by the MECHAniSer and the data ages achieved by the various allocation and scheduling algorithms for four cores

7.3 Data-Age as Schedulability Criterion

For many industrial domains [13, 18] it is equally important to meet task-local deadlines as well as all specified data propagation delay constraints in order to deem a task set schedulable. While the previous section only focused on task-local deadlines as schedulability criterion, this section takes data propagation delay constraints into account.

We first consider systems that meet all task-local deadlines *and* where all data-age constraints are considered for the schedulability criterion (Figure 6). After, all systems that meet their data-age constraints regardless of the task-local deadlines are seen as schedulable (Figure 7). This is important for systems where a bounded number of task-local deadlines can be missed without affecting the performance (for example control applications [40]).

7.3.1 Task-local Deadline and Data Age Schedulability. As the MECHAniSer guarantees that systems that impose JLDs *and* meet all task-local deadlines also meet all data-age constraints, the task-local schedulability with JLDs in Figure 4b is identical to the combined schedulability in Figure 6b. In comparison to the schedulability of systems where no JLDs are considered (Fig. 6a) the addition of JLDs has a maximum improvement (average over all cores) of 42% for RoundRobin.

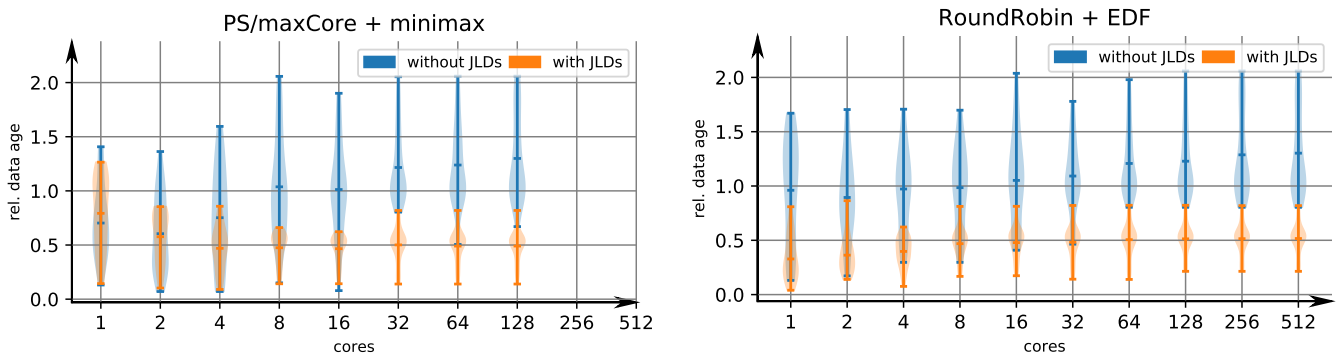
Looking at Figure 6a, where no JLDs are considered, there is only a slight increase in combined schedulability by increasing the number of cores for systems that are RoundRobin, WorstFit or PS-allocated. For one and two cores the combined schedulable systems are approximately half of the task-local schedulable systems but do not increase with three or four cores.

7.3.2 Data Age Schedulability. To further identify the source of this observation we look at the schedulability ratio of systems where *only* data propagation delay constraints are of importance and task local deadlines may be violated. In Figure 7a it can be seen that the percentage of systems that meet their data age constraints is not affected by the selected allocation and scheduling algorithm. It can also be seen that the number of available cores also does not affect the schedulability.

In contrast Figure 7b shows that introducing JLDs boosts meeting data-age constraints even for systems that do not meet all task-local deadlines, since even with one core most data-age constraints are met (some systems are overloaded when only one core is available). For two and more cores RoundRobin, WorstFit or PS-allocated systems reach 100% data-age schedulability and even the heuristics that never accomplish more than 50% task-local schedulability reach more than 95% data-age schedulability with JLDs.

Thus, it can be observed that traditional allocation and scheduling algorithms do not have a direct effect on meeting data propagation delay constraints. The addition of JLDs as scheduling constraints can improve the total system schedulability while its impact on task-local schedulability is minimal (as shown in Section 7.2).

7.3.3 Resulting Maximum Data Age. To put this effect further into perspective Figure 8 and Figure 9 depict the effect of adding JLDs to the systems' cause-effect chains' maximum data age relative to the event-chains data-age requirements (i.e., a value larger than 1 indicates a violated data age constraint). While Figure 8 considers systems that have been allocated and scheduled to one core Figure 9 depicts the same systems and event-chains allocated to four cores. The (a) subfigures comprise only schedules that meet *all* task-local deadlines and therefore comply with the assumptions that the



(a) optimal PS allocation that tries to allocate as many cores as possible combined with cost based scheduling

(b) RoundRobin heuristic allocation combined with EDF scheduling

Figure 10: impact of using up to 512 cores on data age

MECHAniSer uses to compute its data-age guarantees while the (b) subfigures consist of all 433 systems. In addition to the resulting data age values based on allocation and scheduling algorithm, here also upper bounds given by MECHAniSer are shown.

Again it can be observed that adding more cores but no JLDs does not have a substantial effect and the mean maximum data-age. On the other hand, enforcing JLDs as suggested by the MECHAniSer drastically improves the timeliness of event-chains, even for systems that do not meet their assumptions. However, especially PS + minimax generate massive outliers, and no guarantees can be given without meeting all task-local deadlines as these results are only statistical. It can be seen that the upper bounds that are provided by the MECHAniSer also no longer hold for systems where task-local deadlines can be missed, as the underlying assumptions for the timing analysis have been violated. Additionally, one can note that allocation + scheduling does not affect the distribution of the cause-effect chains' maximum relative data-age.

7.4 Do many cores improve data age?

As platforms with an increasing number of cores become available, this experiment investigates the effect of the number of cores on the data age properties.

Since optimal allocation of hundreds of jobs for many cores is resource consuming and the solution space grows exponentially with the numbers of cores three of the 433 systems were selected. The selected systems comprise 99, 305, 361 jobs respectively and have utilizations of 1.2, 1.6, 1.8 and 1, 2, 2 event-chains. This gives a total of 43 data age values since in this experiment we do not only analyze the maximum data-ages of each chain as before but consider all data-propagation paths within the hyperperiod. Again we used all combinations available to allocate and schedule these systems but chose to depict those allocations that use as many cores as possible. Therefore, Figure 10a shows PS maxCore modifications and Figure 10b the RoundRobin heuristics. Please note that PS/maxCore could only be executed for up to 128 cores due to the resource requirements of the algorithm (more than 500 GB of RAM for allocations with more than 128 cores).

Both graphs show that instead of improving i. e., lowering data age values, adding additional cores had mostly the contrary effect. The addition of the second core improves the data-ages for the optimal allocations since the second core is needed to meet all task-local deadlines. For the systems without JLDs, the average data age is at its minimum value when the system becomes schedulable at two cores. After that, the addition of cores increases the average data age. This is the case for the optimal allocation as well as for the RoundRobin allocation.

Similar effects can be observed for systems where JLD are considered. However, the maximum data age values do not increase over the data age constraint (except for the optimal allocation on one core, which is a result of an unschedulable system). In Figure 10a systems with JLDs still decrease data ages up to four cores but with more than four cores all allocation algorithms cease to improve on data age and start to increase data ages.

For both settings, once the system becomes schedulable, with the addition of cores, it becomes likely that two tasks that are consecutive in a cause-effect chain do no longer execute in sequence. This holds true for all other graphs not included in the paper due to space limitations. By adding more cores, the parallelization of previously sequentially executed communicating tasks becomes more likely which increases the likelihood that a task has to wait for an additional hyperperiod to consume the value of its predecessor task. The JLDs do not prevent this increase in data age but guarantee that the increase is bounded by the specified constraint. Thus, in system design with data propagation delay constraints, over-provisioning the system negatively affects the data age.

8 CONCLUSION AND OUTLOOK

In this paper, we extended the MECHAniSer and the RTSC to work closely together and thus bring data-age analysis closer to real systems and real executions. This collaboration enabled us to gain detailed insights into the behavior of contemporary allocation and scheduling algorithms in respect to data age constraints. Moreover, we could examine how additional computational resources influence the data-ages of concrete systems.

The most important insight is, that traditional allocation and scheduling approaches that focus on task-local deadlines are unsuitable for optimizing the data age of cause-effect chains, as long as these are not modeled correctly by dependencies. Moreover, we have shown that the same holds true when the systems computational capabilities are increased by adding additional cores to the hardware platform. One of the critical observations is that the resulting data age values increase when additional compute cores become available. This is contrary to common conceptions and must be carefully taken into consideration during system design. On the other hand, this shows how important it is to model dependencies from the beginning and if this is not possible how useful tools like the MECHAniSer are.

Future work will focus on runtime experiments of the systems in order to measure the resulting runtime data age values as well as the OS overheads introduced by the JLDs. Also, the extension of the existing optimal PS allocation and the minimax scheduler to directly minimize data propagation delays.

REFERENCES

[1] Tarek F. Abdelzaher and Kang G. Shin. 1999. Combined task and message scheduling in distributed real-time systems. *IEEE TPDS*, 10, 11, 1179–1191. ISSN: 1045-9219. DOI: 10.1109/71.809575.

[2] 2014. AUTOSAR - Spec. of Timing Extensions. AUTOSAR, (2014).

[3] K. R. Baker, E. L. Lawler, J. K. Lenstra and A. H. G. Rinndooy Kan. 1983. Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints. 31, 2, (Apr. 1983), 381–386.

[4] Kenneth R. Baker and Dan Trietsch. 2009. *Principles of Sequencing and Scheduling*. John Wiley & Sons, Inc., Hoboken, New Jersey. ISBN: 978-0-470-39165-5.

[5] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam and Thomas Nolte. 2017. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80, Supplement C, (Oct. 2017). http://www.es.mdh.se/publications/4877-.

[6] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam and Thomas Nolte. 2016. MECHAniSer - a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies. In *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[7] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam and Thomas Nolte. 2016. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proceedings of the 22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 159–169.

[8] Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam and Thomas Nolte. 2017. A generic framework facilitating early analysis of data propagation delays in multi-rate systems. In *Proceedings of the 23th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 1–11.

[9] Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam and Thomas Nolte. 2018. Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints. In *23rd Asia and South Pacific Design Automation Conference*. (Jan. 2018).

[10] Alessandro Biondi and Marco Di Natale. 2018. Achieving predictable multicore execution of automotive applications using the let paradigm, (Apr. 2018).

[11] Björn B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis. The University of North Carolina at Chapel Hill. http://www.cs.unc.edu/~bbb/diss/.

[12] Nico Feiertag, Kai Richter, Johan Norlander and Jan Jonsson. 2008. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Proceedings of the 1st International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*.

[13] J. Forget, F. Boniol and C. Pagetti. 2017. Verifying end-to-end real-time constraints on multi-periodic models. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. (Sept. 2017), 1–8.

[14] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens and Claire Pagetti. 2010. Scheduling Dependent Periodic Tasks Without Synchronization Mechanisms. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Stockholm, Sweden, 301–310.

[15] Julien Forget, Frédéric Boniol and Claire Pagetti. 2017. Verifying end-to-end real-time constraints on multi-periodic models. In *Proceedings of the 22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*.

[16] Florian Franzmann, Tobias Klaus, Peter Ulbrich, Patrick Deinhardt, Benjamin Steffes, Fabian Scheler and Wolfgang Schröder-Preikschat. [n. d.] From intent to effect: tool-based generation of time-triggered real-time systems on multi-core processors. In 134–141. DOI: 10.1109/ISORC.2016.27.

[17] Max Friese, Thorsten Ehlers and Dirk Nowotka. 2018. Estimating latencies of task sequences in multi-core automotive ecus, (June 2018).

[18] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler and Falk Wurst. 2017. Communication Centric Design in Complex Automotive Embedded Systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS)* (Leibniz International Proceedings in Informatics (LIPIcs)). Vol. 76, 10:1–10:20. ISBN: 978-3-95977-037-8.

[19] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter and R. Ernst. 2005. System level performance analysis - the symta/s approach. *IEE Proceedings - Computers and Digital Techniques*, 152, 2, 148–166.

[20] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter and R. Ernst. 2005. System level performance analysis - the symta/s approach. *Computers and Digital Techniques*, 152, 2, (Mar. 2005), 148–166. ISSN: 1350-2387. DOI: 10.1049/ip-cdt:20045088.

[21] Thomas A. Henzinger, Benjamin Horowitz and Christoph Meyer Kirsch. 2001. Giotto: a time-triggered language for embedded programming. In *Embedded Software*. Thomas A. Henzinger and Christoph M. Kirsch, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 166–184.

[22] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch and Peter P Puschner. 2015. The platin tool kit-the t-crest approach for compiler and wcet integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7*.

[23] David Stifler Johnson. 1973. *Near-Optimal Bin Packing Algorithms*. PhD thesis. Massachusetts Institute of Technology, (June 1973).

[24] Simon Kramer, Dirk Ziegenbein and Arne Hamann. 2015. Real world automotive benchmarks for free. In *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[25] C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20, 1, 46–61. ISSN: 0004-5411.

[26] C.C. Michael, Ken van Wyk and Will. Radosevich. 2005. Risk-based and functional security testing. last visited 26.04.2017. (2005). https://www.us-cert.gov/bsi/articles/best-practices/security-testing/risk-based-and-functional-security-testing.

[27] S. Mubeen, M. Sjödin, T. Nolte, J. Lundbäck, M. Gålnander and K. L. Lundbäck. 2015. End-to-end timing analysis of black-box models in legacy vehicular distributed embedded systems. In *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*. (Aug. 2015), 149–158. DOI: 10.1109/RTCSA.2015.24.

[28] Saad Mubeen, Jukka Mäki-Turja and Mikael Sjödin. 2013. 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*, 10, 1.

[29] Saad Mubeen, Jukka Mäki-Turja and Mikael Sjödin. 2013. 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*, 10, 1, (Jan. 2013), 453–482.

[30] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla and David Lesens. 2011. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21, 3, 307–338.

[31] Dar-Tzen Peng and Kang G. Shin. 1993. A new performance measure for scheduling independent real-time tasks. *J. Parallel Dist. Comput.*, 19, 1, 11–26. DOI: 10.1006/jpdc.1993.1086.

[32] Dar-Tzen Peng, Kang G. Shin and Tarek F. Abdelzaher. 1997. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE TOSE*, 23, 12, 745–758. ISSN: 0098-5589. DOI: 10.1109/32.637388.

[33] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat and Benoit Triquet. 2016. Mapping Hard Real-Time Applications on Many-Core Processors. In *24th International Conference on Real-Time Networks ans Systems (RTNS'16)*. (Oct. 2016).

[34] Biren Prasad. 1996. *Concurrent engineering fundamentals*. Vol. 1. Prentice Hall Englewood Cliffs, NJ.

[35] Wolfgang Puffitsch, Eric Noulard and Claire Pagetti. 2015. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51, 5, (Sept. 2015), 526–565.

[36] Fabian Scheler and Wolfgang Schröder-Preikschat. 2006. Synthesising real-time systems from atomic basic blocks. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS'06)* (Work-in-Progress Session). (Apr. 2006), 49–52.

[37] Fabian Scheler and Wolfgang Schröder-Preikschat. 2010. The RTSC: leveraging the migration from event-triggered to time-triggered systems. In *13th IEEE Int. Symp. on (ISORC '10)* (Carmona, Spain). IEEE, Washington, DC, USA, (May 2010), 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[38] J. Schlatow, M. Möstl, S. Tabuschat, T. Ishigooka and R. Ernst. 2018. Data-age analysis and optimisation for cause-effect chains in automotive control systems. In *13th International Symposium on Industrial Embedded Systems (SIES)*.

[39] 2002. The economic impacts of inadequate infrastructure for software testing (planning report 02-3). Gaithersburg, MD: National Institute of Standards and Technology. NIST, (2002).

[40] S. Tobuschat, R. Ernst, A. Hamann and D. Ziegenbein. 2016. System-level timing feasibility test for cyber-physical automotive systems. In *11th IEEE Symposium on Industrial Embedded Systems (SIES)*. (May 2016), 1–10.

[41] Rémy Wyss, Frédéric Boniol, Claire Pagetti and Julien Forget. 2013. End-to-end latency computation in a multi-periodic design. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 1682–1687. ISBN: 978-1-4503-1656-9.