Constrained Data-Age with Job-Level Dependencies: How to Reconcile Tight Bounds and Overheads

Tobias Klaus*, Matthias Becker†, Wolfgang Schröder-Preikschat*, Peter Ulbrich‡
*Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
{klaus, wosch}@cs.fau.de
†KTH Royal Institute of Technology, Sweden
mabecker@kth.se
‡Technische Universität Dortmund, Germany
peter.ulbrich@tu-dortmund.de

Abstract—**Many industrial real-time systems rely on the implicit register communication paradigm to minimize overheads and ease distributed development. Here, tasks follow a simple input-processing-output scheme, and data is passed without synchronization by the last-is-best semantics. In these systems, the age of data is the primary real-time objective, which is defined by data-flow chains that span from the system's inputs to outputs. Consequently, a real-time analysis aims to provide guarantees on worst-case data age. In general, there are two main approaches: (1) Task-level scheduling such that inter-task communication is arranged at the beginning and end of a task's execution interval, which guarantees a deterministic yet highly pessimistic data age. (2) Job-level dependencies (JLD) that are added at critical points in the schedule to link specific job instances of tasks of a multi-rate data-flow chain, which provides tighter upper bounds on data ages. However, the drawback is that JLDs induce substantial synchronization overheads, impact the overall schedulability, and are much more challenging to implement.**

In this paper, we address the trade-off between tight data-age guarantees, synchronization overheads, and schedulability in multi-core settings. Our proposed solution is to combine the potential of job-level optimization with the determinism and low overheads of static, task-level approaches. Therefore, we present a novel execution model to efficiently map data-age constrained tasksets with job-level dependencies on event-triggered systems by automated system analysis and transformation. Experimental results of an extensive real-world case study substantiate that our approach can further tighten data-age bounds, reduce overheads, and ease schedulability.

I. Introduction

Practical real-time system development is often characterized by non-technical constraints that prevent the use of the best available concepts and techniques. Therefore an example and the starting point of this paper is the so-called concurrent engineering paradigm [1], which is widely used in the automotive industry. In essence, the numerous functions of a car are developed by different vendors and eventually integrated into a whole system by the manufacturer. The economic benefit of this approach is that development can take place largely independently. However, this loose coupling typically is continued in the system design with numerous (periodic) tasks that follow the input-processing-output scheme. For simplicity, data exchange between the software components is done via implicit register communication (last-is-best semantics).

However, many application domains require strong timing guarantees on the propagation of data through a chain of tasks, i. e., *data chains* [2]–[5]. In particular, automatic control, as the predominant application scenario, is highly sensitive to timing variations [6] due to its connection to the physical world. For example, the age of sensor values has a detrimental impact on the achievable control performance. In a broader context, the fundamental problem is that precedence constraints and dependencies inherent to the application are sacrificed for simplicity and not explicitly implemented. As a result, established scheduling and timing analysis methods either cannot be applied, or they are subject to a high degree of pessimism. For example, the response time of the output task has no bearing on the age of the input data that was used to compute the setpoint in the first place. The decisive factor in systems with implicit register communication is instead the data-propagation delay, which only loosely correlates with traditional scheduling parameters (e. g., period, deadline) of individual tasks. In every step of the processing chain, it must ultimately be assumed that the data of the predecessor has just reached its maximum age when read. This effect is especially pronounced in multi-rate multi-stage chains that cause complex over- and under-sampling situations.

Accordingly, there is a large body of research (cf. Sec. II) that refers to the data age as a real-time objective. These works aim, analog to worst-case response time analysis, to provide guarantees on worst-case age of data. In general, there are three main approaches: (1) An intuitive solution is to align the communication of tasks such that adverse overlapping is impossible. A well-known representative of this type is the Logical Execution Time (LET) paradigm [7]–[9], which fixes data exchange according to the deadlines and WCET of tasks. As a result, the average case corresponds to the worst case, which guarantees a deterministic yet highly pessimistic data age. (2) Another possibility is to reintroduce precedence constraints [10]–[12] at critical points to link specific job instances of a data chain. These, often called job-level dependencies (JLDs), serialize the communication whenever there is a possibility that predecessors could overtake their successors according to traditional scheduling criteria; otherwise, the tasks remain independent. Accordingly, JLDs can guarantee the latest data and thus provide tighter upper bounds on data ages. The drawback, however, is that they require significant changes to the implementation of the system

and induce substantial synchronization overheads, which in turn thwarts the simplicity of implicit register communication. Anticipating our evaluation illustrates the underlying issue: Given the example four-task chain in Figure 1, the maximum data age is 60 ($T_1 = 12$, $\tau_{4,3}$ uses data from $\tau_{1,1}$). Adding JLDs to the chain allows for tightening the bound to 36. However, this improvement is accompanied by synchronization overheads that, even worse, substantially impact overapproximations of static WCET analysis. One reason is the generally poor analyzability of system calls [13], [14], in particular, in generic operating systems (e. g., FreeRTOS, Linux) with reconfiguration capabilities [15], [16]. For example, Schuster et al. observed overapproximation of up to 178 percent in their WCET analysis of FreeRTOS's system calls. For the previous example, we found an increase in overall utilization by 2.27 base points, measured on a Cortex-M4 @ 80 Mhz using FreeRTOS, by mere two semaphores necessary to implement the above example. We show the magnitude of necessary synchronization that can be found in real-world scenarios in Section VI. (3) Such overheads can be avoided by implicit synchronization and the adjustment of release times and deadlines of individual task instants. There are implementations of this schema for time-triggered multi-core systems [12] as well as for event-triggered single-core settings [10], [17]. Both do not apply to our system model of data chains on a shared priority-driven multi-core executive.

A. Problem Statement

This paper's research objective is the specific requirements of multilevel, potentially crossing data chains with hard maximum data age constraints. Our primary concern is the poor temporal analyzability of such chains and the resulting pessimistic bounds. The fundamental challenge is in the development paradigm of such systems, which aims at decoupling of tasks (i. e., application components) and minimizing development and runtime costs and, therefore, typically refrains from explicit task coordination—which may also be hard to generalize for complicated product lines. Existing approaches so far *compromise on data-age predictability, runtime overheads, and overall schedulability* or require recurring manual intervention. We believe that this is due to a lack of analysis and exploitation of data chains' specific properties, a shortcoming that is especially pronounced for crossing chains and multi-core settings.

In our work, we focus on practical aspects, with the primary goal to develop techniques that can be implemented in tools that are useful for developers. Our proposed solution is to combine the potential of job-level optimization with the determinism and low overheads of static, task-level approaches. Therefore, we perform an extended system analysis and unroll all job instances to dedicated tasks. Thereof, we infer task subsets that (a) contain all independent tasks and (b) tasks that are part of a JLD. On the one hand, this enables us to adjust the temporal parameters of specific job instances individually. On the other hand, we provide transformation to tailor dependent tasksets *to tighten data-age bounds further and improve runtime overheads while preserving schedulability*. For example, our tool-based approach can automatically merge tasks on an implementation level to eliminate unnecessary synchronization and preemption overheads. In particular, our approach is, to the best of our knowledge, the first to explicitly consider crossing chains: job instances that are part of multiple parallel chains.

B. Contribution and Outline

This paper makes the following four contributions: (1) A novel execution model for data-age constrained systems that eliminates the need for explicit synchronization. (2) Methods and algorithms that facilitate inference of tight data-age bounds. (3) Toolchain to automatically transform a given task system and its implementation to our model. (4) Extensive case-study based on real-world automotive benchmarks and the WATERS'17 industrial challenge.

The remainder is organized as follows. Section II discusses related work. In Section III, we detail our system model and present relevant background information. Our task model and approach to the efficient implementation of data chains on event-triggered multi-core systems are presented in Section IV. Section VI provides our extensive case-study and experimental results. Finally, we conclude our work in Section VII.

II. Related Work

Data-propagation delay constraints are central to many industrial domains, such as automotive [7], [18] or avionics [5]. Their analysis is the subject of various research works.

The timing analysis of data propagation delays in periodic multi-rate systems with communication via shared registers has been studied by Feiertag et al. [3]. Besides the timing analysis, the authors also highlight the different end-to-end delay semantics that exist (data age, reaction delay, etc.). This timing analysis has further been implemented in different industrial tools used in the automotive domain [19], [20]. In [11], [21], Becker et al. study the timing analysis of these systems at various abstraction levels, where only limited information of the final implementation may be available. Data propagation delays in the multi-periodic synchronous model are studied by Forget et al. in [5]. Data propagation delays on multi-core platforms and under fixed-priority preemptive scheduling are studied in [22]–[25]. Sporadic Data Chains on distributed systems are subject of [26]. The effect of the Logical Execution Time (LET) paradigm on the data propagation delay has been considered in [7]–[9]. Martinez et al. [27] investigate different communication praradigms found in automotive systems, implicit, explicit, and LET communication, and present a timing analysis for mixed preemptive scheduling as it is the case in AUTOSAR.

In addition to research on the temporal analysis of data chains, there has been an effort to modify system parameters such that data propagation delay constraints are met. Schlatow et al. [28] focus on the optimization for multi-core platforms, where task priorities, offsets, as well as the mapping of tasks to processor cores is selected by leveraging a Mixed Integer Linear Programming formulation of the problem. Davare et al. [29] optimize task periods to affect the resulting data

propagation delays in distributed automotive systems. The optimization of the priorities assigned to tasks and messages, as well as the mapping of tasks to cores and signals to messages is addressed in [30]. Verucchi et al. [31] consider multi-rate DAG tasks and present a transformation to single rate DAG that optimize schedulability, as well as age and reaction delays. The work of Becker et al. [11] utilizes the timing analysis of data propagation delays at higher abstraction levels to define job-level dependencies, a partial order of jobs such that data propagation delays are met. This allows maintaining selected design parameters, such as task periods, while only the execution order between specific jobs is relevant to meet all data propagation delay constraints. In [32], Constraint Programming is used to create a time triggered schedule for a clustered many-core platform that also considers JLDs.

Chetto et al. [17] show how precedence constraints between individual non-repeating jobs on a single core can be considered by modification of a task's timing parameters, i. e., its release time and the deadline. This encoding is optimal in the class of dynamic-priority single-core schedulers. I.e., if there exists a dynamic priority scheduling algorithm that respects the task's deadlines, as well as the precedence constraints, the method finds an assignment. Since only task parameters are modified, an EDF schedulability test can be used. A fixed-priority scheduling policy for periodic tasksets with precedence constraints is proposed in [33]. This approach does not rely on semaphore synchronization.

Pagetti et al. [10] consider precedence constraints between jobs as part of their multi-task implementation of multi-periodic synchronous programs on single-core systems. In the first step of their approach, the PRELUDE specification is transformed into a *dependent* taskset where precedence constraints are inserted when required (if this is done for tasks of different periods, as for job-level dependencies, not all jobs are affected). This initial dependent taskset is then further transformed into an *independent* taskset. If the final system is scheduled by Earliest Deadline First (EDF), deadlines of jobs that are subject to precedence constraints are adjusted such that the specified execution order is maintained [17]. Systems scheduled by the Deadline Monotonic (DM) scheduling algorithm are scheduled using the methods of [33].

In [12], we studied the integration of job-level dependencies into a compiler-based tool (RTSC) [34], [35] targeting *time-triggered* real-time systems. This work considers the effect of different offline allocation and scheduling algorithms to generate static schedules for systems with data-age constraints.

Instead of guaranteeing that a consumer instance starts its execution only after the producer instance has finished, Sofronis et al. [36] propose the Dynamic Buffering Protocol (DBP) that utilizes buffers to store multiple values such that the correct communication sequence can be achieved.

In contrast, this paper contributes property-preserving transformation rules of the initial taskset to not only adhere to prescribed JLDs but also to optimizes the resulting OS overheads, while targeting a multi-core system dynamically scheduled by partitioned EDF.

III. Background and System Model

In this section, we describe the application model, the application's timing constraints, and the concept of JLDs that are used to describe job-orderings during execution.

1) Platform and Application Model: An application Γ consists of a set of n tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each of the tasks can be described by the tuple $(C_i, \overline{C}_i, D_i, T_i, \phi_i)$, where C_i describes the task's Worst-Case Execution Time (WCET), $\overline{C}_i$ describes the Best-Case Execution Time (BCET), D_i denotes its deadline relative to its release time, T_i describes the tasks period, and ϕ_i its offset. Initially, all tasks in this work are subject to implicit deadlines, i.e., $T_i = D_i$, and have no offset, i.e., $\phi_i = 0$. The application is dynamically scheduled by an Earliest Deadline First (EDF) scheduler. The kth job of a task τ_i is described by $\tau_{i,k}$, has an absolute release time of $r_{i,k} = k \cdot T_i + \phi_i$, and an absolute deadline $d_{i,k} = r_{i,k} + D_i$.

Additionally, a set of data chains $\mathcal{Z}$ is specified that pose semantic relations between tasks. A data chain $\zeta \in \mathcal{Z}$ is a Directed Acyclic Graph (DAG), with vertexes $\mathcal{V}$ and edges $\mathcal{E}$. A vertex in ζ is represented by a task $\tau_i \in \Gamma$. An edge between two nodes denotes communication between the two tasks. While a data chain can have forks and joins, it must have a single entry and exit node. This structure allows the decomposition of a data chain into several sequential chains that are subject to the same timing constraints.

Different tasks communicate using register communication. In register communication, a sending task writes the value to a global register, and a receiving task reads the value from the global register. This type of communication has the benefit that no signaling between sending and receiving tasks is necessary. However, if the sending and receiving tasks execute at different periods, over- and under-sampling effects occur, i. e., values are overwritten before they are read, or the same value is read more than once. To increase determinism, each task uses the implicit communication paradigm. In the implicit communication paradigm, access to communication variables is performed at the beginning and end of the execution, while the task operates on a local copy during the execution. This form of communication has the benefit that a task has a consistent view of all communication variables during one job, even if the value in the communication register has changed.

2) Data-Propagation Delay Constraints: In addition to the tasks local deadlines, the application is subject to data-propagation delay constraints, i. e., the time for data to propagate through a chain of tasks is constrained.

Several types of data-propagation delay constraints can be specified [3]; in this work, the main focus is on the data-age constraint. Data-age constraints are commonly found in control applications, as the age of data influences the control quality. The data age is defined as the time between the initial first reading of an input value by the first task in the data chain, and the last time this data affects the output of the last task in the data chain. Other significant metrics for evaluating the achievable control quality are mean data age and jitter, whose optimization we also address with our approach: generally, the smaller and steadier, the better [37],

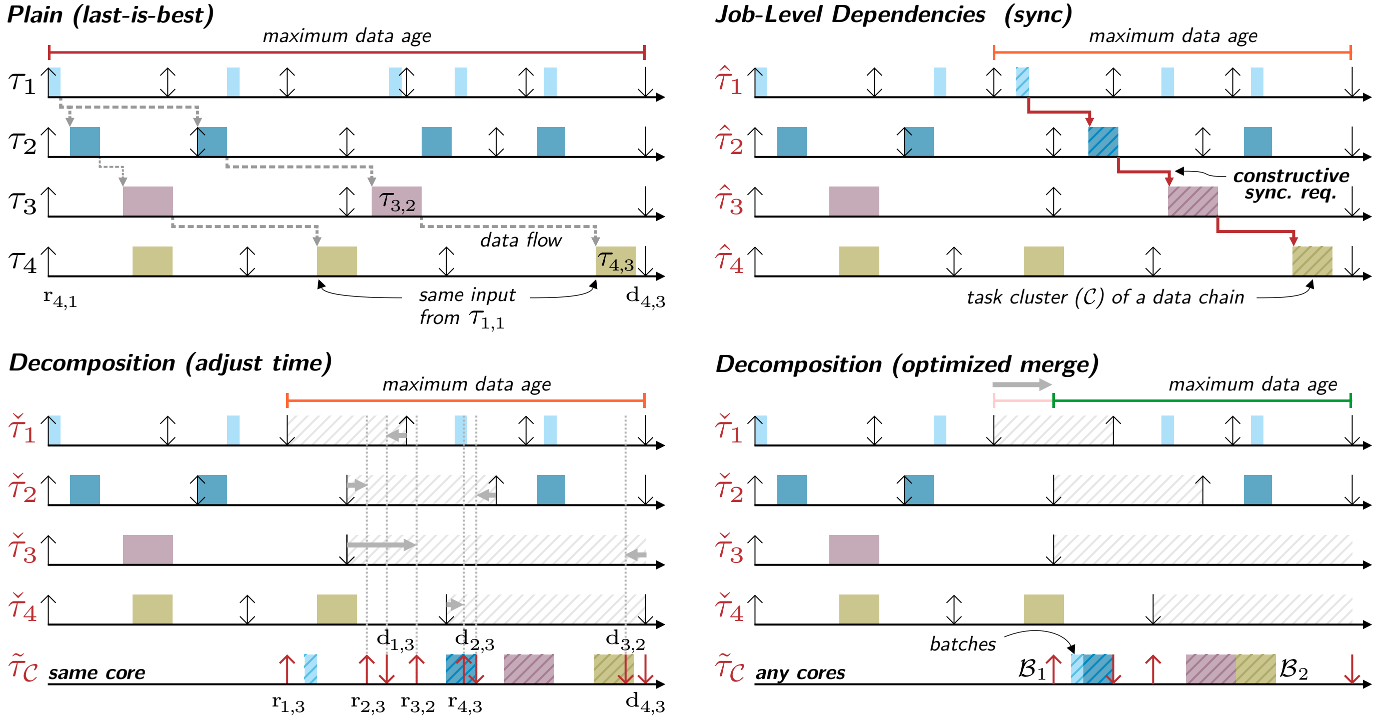


Figure 1. Illustration of our approach's variants and their effects on scheduling and resulting (maximum) data ages for a given data chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$. Each task τ_i spawns task instants $\tau_{i,j}$ (i.e., taskset $\hat{\tau}_i$) that are executed on a shared multi-core platform (i.e., only deadline adherence guaranteed). *Plain* is the baseline without precedence constraints between tasks. *JDL* enforces constraints by costly constructive synchronization. We can transform this to implicit synchronization again, by *Decomposition* of the tasksets into subsets $\tilde{\tau}_i$ and $\check{\tau}_i$ that hold relevant (i. e., cluster tasks) and non-relevant task instants, respectively.

[38]. We want to emphasize that over- and undersampling are an inherent property of our system class. Control stability and safety assessment are a regular part of application development and based on adherence with the specified constraints—our approach neither changes nor influences this methodology.

3) Job-Level Dependencies and Their Generation: JLDs are used to specify a partial ordering between tasks' jobs. A JLD $\tau_i \xrightarrow{(k,l)} \tau_j$ describes that the kth job of τ_i must finish before the lth job of τ_j can start its execution. As a JLD describes a relation between jobs of τ_i and τ_j, the ordering is applied to these jobs in each hyperperiod $\mathrm{LCM}\{T_i, T_j\}$. The set $\mathcal{J}$ contains all JLDs that are specified for the taskset.

Job-level dependencies are typically not defined by the system designer. In [11], a method to generate JLDs for a taskset with specified data-age constraints is proposed. This approach operates solely on the taskset information and is therefore agnostic of the applied scheduling algorithm and platform. If the method of [11] succeeds in determining a set of JLDs, it guarantees that all specified data-age constraints are met as long as each task meets its timing constraints, and the partial ordering of jobs defined by the JLDs is maintained. Hence, the set of JLDs $\mathcal{J}$ that is input to the proposed approach is determined by [11].

IV. Approach

This section presents our approach by a step-by-step evolutionary process of transformation variants, which we each implemented and evaluated in our toolchain. By a simple example, we first put our work into the context of existing approaches, which serve us as a baseline. Based on this, we introduce our execution model and the transformation of JLDs into it, which is a decomposition of tasksets into relevant (determining) and non-relevant task instants. Subsequently, we detail our transformation variants, ranging from adjusting the timing parameters to the optimized merging of relevant tasks. Finally, we turn to data-age bounds and the temporal analysis.

A. Running Example

Figure 1 introduces a running example of the four-task data chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$. Plain (top left) shows a scheduling trace that may be the result of a larger multi-core system, where no measures are taken to affect the chain's data age (last-is-best semantics). In the absence of task coordination, data flows can arrange unfavorably: the same input value from $\tau_{1,1}$ is, for example, determining for two successive output instances $\tau_{4,2}$ and $\tau_{4,3}$. Consequently, the bound on the maximum data-age is far the worst. The data age can positively be affected by the addition of job-level dependencies (top right): $\tau_1 \xrightarrow{(3,3)} \tau_2$, $\tau_2 \xrightarrow{(1,1)} \tau_3$, and $\tau_3 \xrightarrow{(2,3)} \tau_4$. This coordination results in the execution order shown by the red arrows between the relevant task instances. Note that JLDs are defined for the hyperperiod of the constraint tasks, hence $\tau_2 \xrightarrow{(1,1)} \tau_3$ defines the ordering $\tau_{2,1} \rightarrow \tau_{3,1}$ (not shown) *and* $\tau_{2,3} \rightarrow \tau_{3,2}$. These two extremes represent the starting point and baseline for our transformations, which we detail in the following.

B. Transformation of Job-Level Dependencies

Of a task, JLDs only affect specific jobs (instances). This requires the task's conditional coordination, which is complex

to implement and analyze (i. e., bloats WCET) or demands an operating system (OS) mechanism that supports such conditional task dependencies. To avoid these problems we apply the following transformations: Tasks that are not constraint by a JLD in $\mathcal{J}$ are directly included in Γ', without modifying their parameters. For tasks that are constraint by JLDs, we apply a transformation that creates individual tasks for each of the task's jobs, that are then added to Γ'. This is done for all jobs of the original task that are released within the hyperperiod of all tasks that are part of a JLD, which we denote by T^{JLD}. After this transformation, a job $\tau_{i,k}$ of an original task τ_i is represented by the task τ_{i^k}. τ_{i^k} has the same WCET and relative deadline as the original task, but its period is set to T^{JLD} and its offset is set to the release time of $\tau_{i,k}$. With this, the execution window of τ_{i^k} and $\tau_{i,k}$ are identical and merely the representation in the task model changes. For clarity, we denote the set of all transformed tasks that result from the same task τ_i in Γ by $\hat{\tau}_i$.

Besides converting the tasks, an analogues transformation is needed for the JLDs. We denote the transformed set of JLDs by $\mathcal{J}'$. Thus, JLDs are transformed so that the execution of Γ' is constraint by $\mathcal{J}'$ in the same as Γ is constraint by $\mathcal{J}$. Since all transformed tasks have the same period, T^{JLD}, a JLD between any of the new tasks results in a simple one-to-one precedence constraint in $\mathcal{J}'$. With decomposed JLDs, for each transformed task $\hat{\tau}_i$, we can define the subset of tasks that is not constrained by any JLD, $\check{\tau}_i$.

Although this transformation increases the number of tasks, the scheduler's queue length remains unchanged as each newly generated task is only released at its implicit predecessor's deadline. However, we can leverage the added tasks for a fine-grained adaptation of specific task parameters such that JLDs are met.

C. Adjusted Timing Properties

To enforce dependencies without explicit synchronization on a single-core, we now adjust release times and deadlines of dependent tasks, as described by Chetto et al. [17] and [10], [33]. The modification of the task deadline is based on the observation that the urgency of a task is determined not only by its own deadline but also by the deadline of its successor tasks: (1) For each task τ_i that is part of a dependency but has no successor, we set $d_i^* = d_i$. (2) Then we select a dependent task τ_j, whose deadline has not yet been adjusted, but whose successors' deadlines have already been adjusted. If no such task exists, all deadlines have been adjusted. (3) We assign $d_j^* = \min(d_j, \min_{\forall \tau_h | \tau_j \rightarrow \tau_h}(d_h^* - C_h))$ and return to (2).

In [17], it is further observed that the earliest start time of a task is determined by its release but also by the deadline of predecessor tasks. Task offset ϕ are therefore adapted vice-versa: (1) For each task τ_i that is the beginning of a dependency chain and thus has no predecessors, we assign $\phi_i^* = \phi_i$. (2) We select a dependent task τ_j whose offset has not yet been adjusted, but whose successors' have. If no such task exists, all offsets were already adjusted. (3) We assign $\phi_j^* = \max(\phi_j, \max_{\forall \tau_h | \tau_h \rightarrow \tau_j}(\phi_h^* - C_h))$ and return to task selection (2).

The new relative deadlines are then computed as $D_i^* = d_i^* - \phi_i^*$. In case that all tasks of a dependency chain are allocated to the same processor, these timing properties can not only ensure enforcement of dependencies without OS mechanisms but can also be used for schedulability tests for dependent tasks. Allocating all dependent tasks to the same core may seem quite restrictive. However, since these tasks are both sequential and do exchange messages, a joint allocation is desirable and not a significant restriction since they cannot execute in parallel. For a multi-core allocation of a chain, this adjustment would reduce actual contention on synchronization mechanisms but does not avoid their usage.

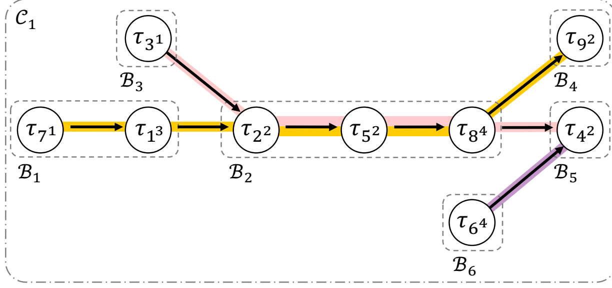
The running example of Fig. 1 (bottom left) depicts the decomposed taskset, where all tasks that are constraint by JLDs are depicted on the same level. The decomposed JLDs thus are $\tau_{1^3} \rightarrow \tau_{2^3}$, $\tau_{2^3} \rightarrow \tau_{3^2}$, and $\tau_{3^2} \rightarrow \tau_{4^3}$. After the adjustment of the dependent tasks' deadlines and offsets, the red marked release and deadlines are maintained, which under EDF results in a schedule that maintains the specified JLDs.

D. Statically Tailored Tasks

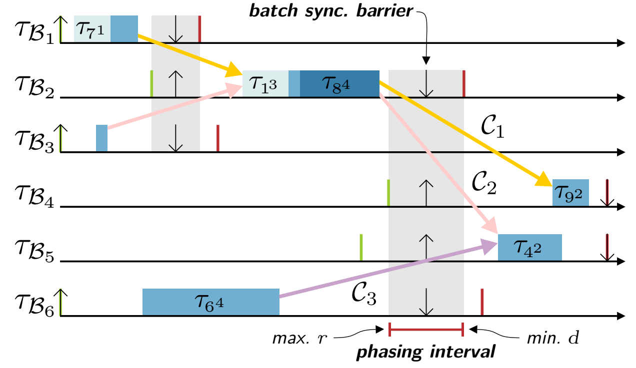
To sum up the steps until now, we have transformed the jobs of the original taskset over the hyperperiod that decomposed the initial JLDs into 1:1 dependencies between tasks. New release times and deadlines have been computed such that the ordering that is required by the dependencies is maintained. As long as all dependency connected tasks are allocated to the same processors, this new taskset is guaranteed to enforce all dependencies [10], [17].

To alleviate further steps, we define *clusters* of tasks $\mathcal{C}$ that are directly connected by dependencies. Such a cluster can then further be decomposed into so-called *batches* $\mathcal{B}$ of tasks that form a sequential chain. Thus, they are connected by exactly one dependency each. That is, τ_i and τ_j are added to the same batch $\mathcal{B}_l$ if the following condition holds $\exists \tau_i \rightarrow \tau_j \in \mathcal{J}' \wedge \nexists \tau_i \rightarrow \tau_p \in \mathcal{J}' \wedge \nexists \tau_p \rightarrow \tau_j \in \mathcal{J}' \wedge \tau_p \notin \{\tau_i, \tau_j\}$. This is demonstrated with an example in Fig. 2. The shown tasks all connect via dependencies; thus, they are all part of the cluster $\mathcal{C}_1$. Six batches can be formed that contain sequential task chains, $\mathcal{B}_1 = \{\tau_{7^1}, \tau_{1^3}\}$, $\mathcal{B}_2 = \{\tau_{2^2}, \tau_{5^2}, \tau_{8^4}\}$, $\mathcal{B}_3 = \{\tau_{3^1}\}$, $\mathcal{B}_4 = \{\tau_{9^2}\}$, $\mathcal{B}_5 = \{\tau_{4^2}\}$, and $\mathcal{B}_6 = \{\tau_{6^4}\}$. Additionally, the figure shows the data flow between tasks as underlays of the dependencies: tasks in a single batch can belong to different data chains (e. g., $\mathcal{B}_2$).

The next logical step is to reduce scheduling overheads and improve the average case latency of data passing through batches. Scheduling overheads due to context switches can effectively be decreased by reducing the number of tasks through task merging [39]. Thus, tasks that are part of a batch $\mathcal{B}_i$ are merged and scheduled sequentially in a dedicated task $\tau_{\mathcal{B}_i}$. For a given cluster $\mathcal{C}_k$, all tasks $\tau_{\mathcal{B}_i}$ that are the result of merging batches $\mathcal{B}_i \in \mathcal{C}_k$ are part of the same set $\tilde{\tau}_k$. Note that our toolchain can also split tasks [40] according to their



(a) Dependency cluster $\mathcal{C}_1$ and the resulting batches $\mathcal{B}_1$ to $\mathcal{B}_6$. Colored underlays represent the three different original data chains.



(b) Resulting multi-core schedule that takes the different crossing data chains into account.

Figure 2. Task cluster that contains different batches and its merged multi-core execution.

runtime, which can be leveraged in a subsequent step to regain utilization if necessary.

The task parameters of $\tau_{\mathcal{B}_i}$ are determined by the different tasks of the batch. The period $T_{\mathcal{B}_i}$ is equal to the period of the tasks in $\mathcal{B}_i$. Since all such tasks are subject to task transformation (see Sec. IV-B), their periods are identical.

Two different approaches are proposed to determine the exact composition of the merged tasks, as well as their parameters $\phi_{\mathcal{B}_i}$ and $D_{\mathcal{B}_i}$. The first approach maintains all properties of the original taskset, while the second approach has the potential to further increase the system's schedulability by intentionally weakening the original taskset properties.

1) Property-Preserving Merge: In the property-preserving merge, offset and deadline of $\tau_{\mathcal{B}_i}$ are selected such that $\tau_{\mathcal{B}_i}$ can only execute during the interval in which all tasks of $\mathcal{B}_i$ can execute. That means that all tasks τ_{i^k} are not to be released before the release time of their origin job τ_{i_k} and their absolute deadlines D_{i^k} shouldn't be put after the original D_{i_k}. As it is not guaranteed that all tasks of the batch $\mathcal{B}_i$ actually overlap in their execution, the property-preserving merging of the batch can result in several individual tasks.

To determine where to split up the batch, we use the following constructive approach. The initial task of the batch sequence is selected as $\tau_{\mathcal{B}_i}$. The next task in the sequence, let's call it $\tau_{\mathcal{B}_i}^{next}$, is selected as a candidate to be merged with $\tau_{\mathcal{B}_i}$. The tasks are merged if they overlap in their execution for at least $C_{\mathcal{B}_i} + C_{\mathcal{B}_i}^{next}$ time units:

$$min(d_{\mathcal{B}_i}, d_{\mathcal{B}_i}^{next}) \geq (max(r_{\mathcal{B}_i}, r_{\mathcal{B}_i}^{next}) + C_{\mathcal{B}_i} + C_{\mathcal{B}_i}^{next}) \quad (1)$$

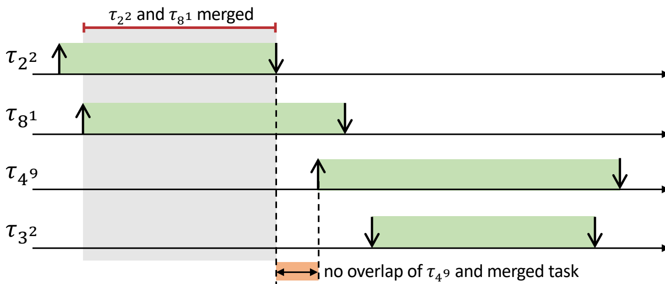


Figure 3. Example to illustrate the property-preserving merge.

If they are merged, the WCET of $\tau_{\mathcal{B}_i}^{next}$ is added to $C_{\mathcal{B}_i}$, the deadline is set to $d_{\mathcal{B}_i} = \min(d_{\mathcal{B}_i}, d_{\mathcal{B}_i}^{next})$, and the release time to $r_{\mathcal{B}_i} = \max(r_{\mathcal{B}_i}, r_{\mathcal{B}_i}^{next})$. This procedure is repeated with the merged task $\tau_{\mathcal{B}_i}$ and successor $\tau_{\mathcal{B}_i}^{next}$ in the batch sequence, as long as the intersection of their execution windows is at least large enough to allow both tasks to execute (see Equation 1). If this is not the case, a new batch task $\tau_{\mathcal{B}_i'}$ is introduced, and the scheme continued. This is illustrated by Fig. 3. All tasks are part of the same batch and are executed in the order $\tau_{2^2} \rightarrow \tau_{8^1} \rightarrow \tau_{4^9} \rightarrow \tau_{3^2}$, i.e., from top to bottom in the figure. The proposed merging strategy starts with task τ_{2^2}, and, as $d_{2^2} \geq r_{8^1}$, the task is merged with τ_{8^1}. $\tau_{\mathcal{B}_i}$ receives the release time of τ_{8_1} and the deadline of τ_{2^2}. As τ_{4^9} does not overlap in its execution with the previously merged tasks $\tau_{\mathcal{B}_i}$, a new task is started, and τ_{4^9} is merged with τ_{3^2}. Note that the resulting tasks are subject to precedence constraints. However, as their intervals between release time and deadline do not overlap, the precedence order is implicitly maintained.

In Fig. 1 (bottom right), this results in two batch tasks that do not overlap in their execution interval and thus maintain the required ordering. τ_{1^3} is merged with τ_{2^3}, inheriting the deadline of τ_{1^3} and the release time of τ_{2^3}. As this merged task's deadline is before the release of τ_{3^2}, τ_{3^2} is merged with τ_{4^3} as a dedicated merge task.

2) Schedulability-Enhancing Merge: The schedulability-enhancing merge is based on the supposition that a transformed task τ_{i^k}, that is part of a batch $\mathcal{B}_i$ can execute before its intended release or after its intended deadline, if:

1) τ_{i^k} is not the source or sink of any data chain $\zeta \in \mathcal{Z}$.
2) There is no data chain $\zeta \in \mathcal{Z}$ of which τ_{i^k} is part of, which also includes tasks that are not part of $\mathcal{B}_i$.
3) Other tasks of $\mathcal{B}_i$ that are part of the same data chains $\zeta \in \mathcal{Z}$ as τ_{i^k} always appear in the same order.

This is the case as tasks within a batch must be scheduled sequentially. Their makespan's union is equal to or laxer than the typically overlapping execution windows of batch tasks. Therefore, when merging computation-only batch tasks, we can assign the earliest release time and the latest deadline of all tasks – thereby maximizing the execution window of the

merged task. This ensures maximum laxity of the new task and generally improves schedulability. Thus, a task τ_{i^k} can violate its prescribed release and deadline if it is communicating *only* with other tasks of the same batch. Which would, therefore, lead to an execution with the same input data and provide the same output data to the same succeeding tasks as if it would adhere to its original release time and deadline. Hence, the same functionality is maintained. This can be observed at the example of τ_{5^2} in Fig. 2, which is part of two data chains whose respective predecessor and successor tasks are in the same batch $\mathcal{B}_2$.

Considering the above criteria, an original batch $\mathcal{B}_i$ is divided such that intermediate tasks do not communicate with tasks outside the batch. For each resulting batch $\mathcal{B}_i^*$, the offset is set as $\phi_{\mathcal{B}_i^*} = \phi_{first(\mathcal{B}_i)}$, and the deadline as $d_{\mathcal{B}_i^*} = d_{last(\mathcal{B}_i)}$. Where $first(\mathcal{B}_i)$ returns the first task of $\mathcal{B}_i$ and $last(\mathcal{B}_i)$ the last task of $\mathcal{B}_i$, respectively. Again, the relative deadline $D_{\mathcal{B}_i^*}$ is computed as $d_{\mathcal{B}_i^*} - \phi_{\mathcal{B}_i^*}$. The WCET $C_{\mathcal{B}_i^*}$ is equal to the sequential execution of all tasks of the batch $\mathcal{B}^*$.

E. Multi-core Considerations

The adjustment of tasks offsets and deadlines of Sec. IV-C guarantees that dependencies are met as long as dependent tasks are executed on the same processor. This, for example, means that the cluster $\mathcal{C}_1$ of Fig. 2 is required to execute on the same processor. However, several batches could potentially be executed in parallel on a multi-core platform. To leverage multi-core allocation of batches, either OS mechanisms are required to enforce dependencies between tasks on different cores, or additional modification of the dependent tasks is required such that a preceding task must finish its execution before a succeeding task can start its execution. To reduce explicit synchronization overheads, *batch synchronization barriers* are introduced over which the dependent task parameters are selected in a way that multi-core execution is possible.

If there exists a dependency between different batches of a cluster, the respective tasks overlap in their execution window [11]. As this overlap gives rise to potential wrong execution on multi-core processors, logical barriers are introduced such that execution overlap between the dependent batches of a cluster are avoided. For each task that is a successor task, a batch synchronization $\mathcal{S}_i = \{\mathcal{S}_i^{pred}, \mathcal{S}_i^{suc}\}$ is created, that contains a set of predecessor and successor tasks that need to be synchronized. For each set, all predecessor tasks of the initial task are added. Sets that have shared predecessor tasks are merged together. For the example in Fig. 2, this leads to 3 sets. $\mathcal{S}_1 = \{\{\mathcal{B}_1, \mathcal{B}_3\}, \{\mathcal{B}_2\}\}$, $\mathcal{S}_2 = \{\{\mathcal{B}_2\}, \{\mathcal{B}_4\}\}$, and $\mathcal{S}_3 = \{\{\mathcal{B}_2, \mathcal{B}_6\}, \{\mathcal{B}_5\}\}$. As $\mathcal{B}_2$ is in $\mathcal{S}_2^{pred}$ and $\mathcal{S}_3^{pred}$, the two are merged in $\mathcal{S}_2$. Finally, this leaves two sets: $\mathcal{S}_1 = \{\{\mathcal{B}_1, \mathcal{B}_3\}, \{\mathcal{B}_2\}\}$ and $\mathcal{S}_2 = \{\{\mathcal{B}_2, \mathcal{B}_6\}, \{\mathcal{B}_4, \mathcal{B}_5\}\}$.

Each batch synchronization translates to one logical barrier, which can be placed at any point in the so-called *phasing interval*:

$$[\max_{\forall \tau_{\mathcal{B}_i} \in \mathcal{S}^{suc}}(r_{\mathcal{B}_i}), \min_{\forall \tau_{\mathcal{B}_i} \in \mathcal{S}^{pred}}(d_{\mathcal{B}_i})] \quad (2)$$

A logical barrier of a batch synchronisation $\mathcal{S}_i$ therefore requires that all batch tasks in $\mathcal{S}_i^{pred}$ finish latest at the barrier, and all batch tasks in $\mathcal{S}_i^{suc}$ start earliest at the barrier. The phasing intervals are highlighted in the example of Fig. 2b. Deadlines of tasks in $\mathcal{S}^{pred}$ and release times of tasks in $\mathcal{S}^{suc}$ can be set to the same value within the phasing interval, ensuring execution of all predecessors before successor release.

There are many potential optimization criteria for choosing a common release and deadline within the phasing interval. In this paper, we set values to be in the middle of the interval.

F. Timing Analysis and Allocation

Up to this point, the original taskset Γ and the set of JLDs $\mathcal{J}$ have been transformed into almost independent tasks Γ' that can be executed by a partitioned EDF, if overlapping tasks of a cluster are allocated to the same core [10], [41]. If batch synchronization barriers are used, all precedence constraints are implicitly coded by the tasks' timing parameters. Thus all allocation and scheduling algorithms that are capable of handling asynchronous periodic tasksets can be used to execute Γ. The same holds true for schedulability tests.

For the remainder of this paper, we use worst-fit heuristic allocation to allocate jobs to cores and schedule them using partitioned EDF. The worst-fit heuristic is chosen for task allocation as it has shown the best performance for task sets with JLDs [12]. For tasksets, without batch synchronization barriers, we employ a slightly adapted version of the Worst-Fit algorithm that ensures that all batches of a cluster are allocated to the same core. Prior to execution, we test each core with the processor demand criterion for schedulability.

G. Tight Data-Age Delay Analysis

Besides the schedulability of individual tasks, meeting data-age constraints is crucial. The baseline analysis of [11] considers only execution times and periods to determine the job's read and data interval. While this provides data-age bounds agnostic of the concrete platform or scheduling algorithm, it also leads to pessimistic bounds. The method proposed in this paper transforms the initial taskset Γ such that tasks constrained by JLDs are represented by individual tasks, and such transformed tasks may further be part of a batch and, therefore, subject to merging. As these transformations restrict the tasks' execution freedom, they can lead to less pessimistic data age bounds [21].

Therefore we present a data-age delay analysis that takes the concrete modifications of the taskset into account, providing tighter data age bounds than the baseline of [11]. As it is also the basis for our analysis, we refer the interested reader to [11] for a detailed description of the baseline analysis.

1) Read and Data Interval: The data-age analysis is performed on the original taskset Γ, where a job $\tau_{i,j}$ is assigned a *read interval* $RI_{i,j}$, and a *data interval* $DI_{i,j}$. The read interval spans the time during which the job may read its input data, where $R_{min}(\tau_{i,j})$ and $R_{max}(\tau_{i,j})$ define the earliest and latest time this can happen, i.e. $RI_{i,j} = [R_{min}(\tau_{i,j}), R_{max}(\tau_{i,j})]$. Similarly, the data interval starts at the earliest time the output data of the job can be available, and ends at the time the successor job $\tau_{i,j+1}$ must have overwritten the data, that is $DI_{i,j} = [D_{min}(\tau_{i,j}), D_{max}(\tau_{i,j})]$.

If there exists an execution where a job $\tau_{k,l}$ can consume the output data of a job $\tau_{i,j}$, then $RI_{k,l} \cap DI_{i,j} \neq \emptyset$. Hence, there must exist a point in time where the data that is produced by $\tau_{i,j}$ may be read by $\tau_{k,l}$. Let $\texttt{follows}(\tau_{i,j}, \tau_{k,l})$ be a function that returns `true` if this is the case and no JLD is specified that prohibits this execution sequence, and `false` otherwise.

Eq. (3) and (4) present the boundaries of the read interval, dependent if a job of a original task $\tau_i \in \Gamma$ is transformed into separate tasks, is part of a batch, or has been unchanged.

$$R_{min}(\tau_{i,j}) = \begin{cases} r_{i^j} & \text{if } \tau_i \in \mathcal{D} \wedge \tau_{i^j} \notin \mathcal{B}_k \\ r_{\mathcal{B}_k} & \text{if } \tau_i \in \mathcal{D} \wedge \tau_{i^j} \in \mathcal{B}_k \\ r_{i,j} & \text{otherwise} \end{cases} \quad (3)$$

$$R_{max}(\tau_{i,j}) = \begin{cases} d_{i^j} - C_i & \text{if } \tau_i \in \mathcal{D} \wedge \tau_{i^j} \notin \mathcal{B}_k \\ d_{\mathcal{B}_k} - C_i & \text{if } \tau_i \in \mathcal{D} \wedge \tau_{i^j} \notin \mathcal{B}_k \\ d_{i,j} - C_i & \text{otherwise} \end{cases} \quad (4)$$

The boundaries of the data interval are dependent on $R_{min}(\tau_{i,j})$. The data produced by a job may be available at its output $\overline{C}_i$ time units after the earliest read time. On the other hand, the latest time the data produced by a job must be overwritten is when the next job of the same task must have finished, i. e., its deadline. Expressed by (3) and (4) this leads to:

$$D_{min}(\tau_{i,j}) = R_{min}(\tau_{i,j}) + \overline{C}_i \quad (5)$$

$$D_{max}(\tau_{i,j}) = R_{max}(\tau_{i,j+1}) + C_i \quad (6)$$

2) Determining Maximum Data Age: The analysis of a data chain ζ examines each *initial job* of the chain. Initial jobs are all jobs of the first task in ζ that are released in the hyperperiod of Γ. The analysis first computes the maximum data age that is possible starting at each initial job.

Starting from an initial job, a depth-first approach is used along the tasks of ζ to determine the data propagation path along the tasks' jobs that leads to the maximum data age. If $\tau_{i,j}$ is the initial job, and τ_k is the second task in ζ, then the data propagation path is extended to the job of τ_k that has the largest release time among all jobs where `follows()` returns `true`. This process continues until a job of the last task in ζ is reached, or if `follows()` does not return `true` for any of the jobs. In this case the last segment of the data propagation path is removed and the next earlier job is examined. If a job of the last task of the data chain is reached, the maximum data age of the data propagation tree is determined by Eq. 7. If the complete data propagation tree is evaluated without reaching a job of the last task in ζ, the data produced by the initial job is overwritten before it reaches the end of the data chain, no maximum data age exists starting from the investigated initial job. Due to the last-is-best communication between tasks, it is however guaranteed that there exists a data propagation path for at least one of the initial jobs of ζ.

The maximum data age that results of a specific initial job $\tau_{i,j}$ can then be computed by $A(\tau_{i,j})$, between $\tau_{i,j}$ and the job $\tau_{o,p}$ of the last task in the chain:

$$A(\tau_{i,j}) = R_{max}(\tau_{o,p}) + C_o - R_{min}(\tau_{i,j}) \quad (7)$$

The maximum data age of a data chain ζ can then be computed as the maximum value $A(\tau_{i,j})$ for all initial jobs of ζ.

H. Post-Allocation Data-Age Optimization

To minimize the average and worst-case data age, we chose to tighten the deadlines of chain tasks. Generally, smaller deadlines decrease the reachability of an initial job as it reduces R_{max} and D_{max} of its predecessor job. Additionally, decreasing a output task's deadline lowers $A(\tau_{i,j})$ directly.

However, we are not aware of an algorithm that constructively minimizes deadlines for multiple tasks at once while preserving the system's schedulability, as the algorithm *minD* [42] does for a single task. However, by applying minD sequentially on each I/O task, one would implicitly favor the first task the algorithm is applied to. This would not decrease deadlines for all tasks evenly, as we desire. Therefore we use the following iterative approach to equally tightening deadlines without decreasing the system's overall schedulability. To decrease the potential effect on schedulability, we do not tighten the deadlines of all chain-related tasks but only of I/O tasks, i.e., beginning and end of a chain. This is beneficial since the deadline of the end task is part of Eq. (7) The deadlines of all selected tasks are then set to $D_{i^k}^{new} = C_{i^k}^{*} + p \cdot (D_{i^k} - C_{i^k})$. p is initally set to 0 and iteratively increased until the taskset is schedulable according to the processor demand schedulability test. Assuming that the original taskset was schedulable with initial release times and deadlines, this algorithm stops at least at the original laxity $D_{i^k} - C_{i^k}$. If the taskset is schedulable for $p = 0$, all tasks have their relative deadlines D_{i^k} set to their WCET C_{i^k}. Since the processor demand schedulability test is only meaningful for allocated systems, this optimization must take place after the allocation of jobs to computing cores.

V. IMPLEMENTATION

Our concepts are only half the story; they do not solve the concurrent engineering paradigm's dilemma of loose coupling, extensive legacy code, and product line complexity. Therefore, the other half is tooling for their use in existing systems.

We implemented our approach within the RTSC [34], a static analysis tool based on the LLVM-framework, and made it available online[1]. Figure 4 illustrates its workflow: as inputs, it takes an abstract system model specified in AMALTHEA, the C source code of the application tasks, and the JLDs (as XML), which we infer from the abstract system model by the approach presented by Becker et al. [11], [43].

These artifacts are imported and transformed to a platform-agnostic internal representation. While temporal properties of tasks stem from the system description, the global control-flow is inferred by static analysis from the source code. Subsequently, tasks with precedence constraints are unrolled

[1] https://www4.cs.fau.de/Research/RTSC/mechaniserdynamic

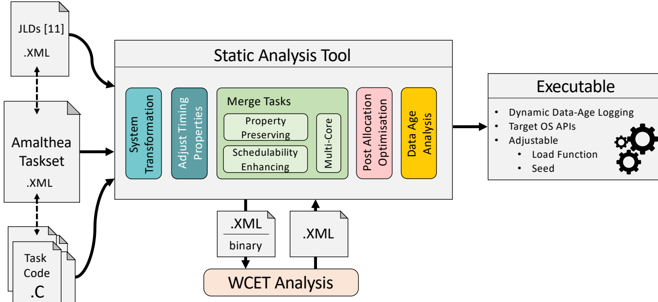


Figure 4. Overview of the developed static analysis tool.

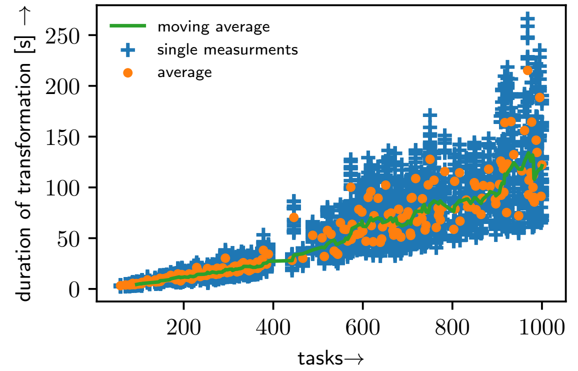


Figure 5. RTSC runtime depending on the number of processed tasks

to the chains' hyperperiod. Then, according to our model (cf. Section IV-B), we add task bodies for each dependent task on the LLVM-IR level (holding the actual implementation). We also verify that all register communication is realized via C11 *atomic_load* and *atomic_store*.

The next step implements the transformations and optimizations from Sections IV-C ff. as individual LLVM passes. By this, we can selectively apply them and, even more important, omit parts of the source system that do not contain data chains (i. e., tasks not to be affected). In the case of modifying passes, LLVM-IR is injected to achieve the desired behavior. For example, simple jumps to concatenate merged jobs or system calls of the target RTOS to enforce remaining dependencies. Because these transformation steps have a potential impact on the WCET of the tasks, we selectively rerun the WCET analysis. Various backends can be coupled with our toolchain, for example, commercial tools like AbsInt's aiT [44]. While sound WCET analysis is appropriate to rule out temporal interference of merged jobs, their spatial isolation is subsumed by the merge. Our toolchain can enforce spatial isolation at option by weaving in code for protection domain switches (i. e., MMU/MPU reconfiguration). While these techniques are available [34], [40], [45], we deem isolation considerations out of the scope of this paper.

Finally, all code required for compiling the target system is generated, including target RTOS system calls. Various target platforms are supported, $\text{Litmus}^{\text{rt}}$ [46] acts as the execution platform for our experiments.

We are aware that unrolling the chains' hyperperiod raises fears about the scalability of our approach. Therefore, we shall detail this aspect before diving into the actual evaluation. Analysis-wise, unrolling affects only the abstract task configuration and therefore scales relatively well. Figure 5 gives the runtime of our tool including all analyses depending on the number of processed tasks: even for a few hundred tasks, the runtime is in the minute range, which we consider feasible for everyday work. Execution-wise, we assume a run-to-completion semantics of jobs; an automated conversion of a process-oriented implementation is possible but outside the scope of this paper. Consequently, we can reuse the existing process stacks. The number of process control blocks will grow depending on the selected target output. In our running example, the number of processes increases from an initial four to 14 for adjust time and decomposition. This number can be reduced again by merging (12 in this case). However, we can also generate processes such that they count activations and execute jobs conditionally. This way, the process count remains constant.

VI. Evaluation

After showing our approach's scalability in the former section, we study its effectiveness by applying it to two previously published real-world case studies: interleaved data chains and Waters Industrial Challenge '17. Additionally, we analyzed its impact on schedulability by a broader set of synthetic benchmark systems.

Table I
Overview of evaluation variants and identifiers.

	ID	Variant	Overhead	Data-Age	Sched'ity
Baseline	**PLN**	Plain (last-is-best)	+	−	+
	JLD	Job-Level Dependencies	−	○	−
	ADJ	Adjust Time Parameters	+	○	−
Our Approach	$\textbf{PLN}_{\textbf{opt}}$	Plain (optimized I/O)	+	○	+
	$\textbf{JLD}_{\textbf{opt}}$	JLD (optimized I/O)	−	+	−
	MRG	Merge	+	○	○
	$\textbf{MRG}_{\textbf{max}}$	Merge (max. laxity)	+	○	+
	$\textbf{MRG}_{\textbf{mc}}$	Merge (multi core)	+	○	+
	$\textbf{MRG}_{\textbf{opt}}$	Merge (optimized I/O)	+	+	○
	$\textbf{MRG}_{\pi}$	Merge (combinations)	+	+	+ / ○

A. Experimental Setup

In the first step, for each of the experimental setups, we generated JLDs, according to [11]. We then applied the transformation variants described in Section IV. These are summarized in Table VI, which gives the identifiers used in the following along a rough evaluation of the properties achieved in each case. The baseline for our experiments is given by PLN, JLD, and ADJ, which represents uncoordinated, explicit, and implicit synchronization as described in IV-C, respectively. Furthermore, we have also applied our optimization of the scheduling parameters to PLN_{opt} and JLD_{opt} for comparison. Finally, the group of merge variants can be divided into individual transformation objectives: MRG is the basic property-preserving merge described in Section IV-D without any specific optimizations and objectives. MRG_{max} aims to

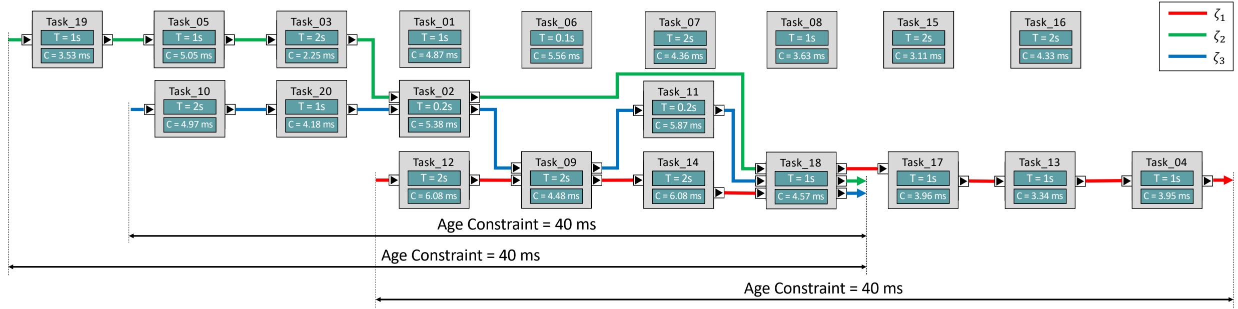


Figure 6. Case Study with 3 data chains that share common tasks in their data path.

maximize laxity of batches and thus to enhance overall schedulability. $\mathrm{MRG}_{\mathrm{mc}}$ employs batch synchronization to allow for multi-core allocation. $\mathrm{MRG}_{\mathrm{opt}}$ strives to improve the timing of relevant input and output jobs. Also, we have evaluated all combinations of the merge variants. $\mathrm{MRG}_{\mathrm{maxopt}}$ is, for example, the combination of $\mathrm{MRG}_{\mathrm{max}}$ and the proposed post-allocation optimization. However, for the sake of simplicity, we only show those combinations in the experiments where they produce different results. In Table VI, MRG_{π} represents the results of the combinations.

If executed, systems are implemented on Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (Litmus$^{\mathrm{rt}}$) and scheduled by partitioned Earliest Deadline First (EDF). For the non-multi-core variants, we adapted the worst-fit allocation algorithm to allocate all members of a cluster to the same core, not to violate the assumptions made in Sec. IV-C. In both case studies, we executed the systems with each variant for 100 hyperperiods and with a varying fraction 0.1, 0.3, 0.5, 0.7, 0.9 of the specified WCETs and logged the actual data ages of each data chain. Where applicable, we normalize data ages to the worst-case data age of their data chain without any precedence constraints. We refer to this as the LET-value, as the obtained bound equals the one of the LET paradigm [47].

B. Case Study: Interleaved Data Chains

The first system we examined as a case study is based on an automotive application [2]. A component view of the system is shown in Figure 6. In this system, the functionality is realized by 20 tasks that are activated at different periods. Three different data chains are specified, where different chains share tasks along their path. After the task transformation, the system consists of 40 independent periodic data chain related tasks with a period of 2 s and differing phases each, as well as the five untouched Tasks *Task_06 – 08*, and *Task_15 – 16*.

1) Additional Load: To study the effect of our approach on the composability of tasksets and the impact of composed systems on data age performance, we added additional load without data propagation constraints to the *base* system. Since an additional task has the most interference with an existing one when being released at the same time, we duplicated the *base* tasks and adjusted their deadlines in the following ways. First, we duplicated the deadlines, resulting in a composed system *1.0* where both systems are initially equally prioritized by an EDF-scheduler. Second, we loosened the deadlines of the additional taskset by multiplying its relative deadlines by 1.1, prioritizing the *base* system, including its data propagation chains. This composed system is referred by *1.1* in the following. Third, we tightened the additional taskset's deadlines by multiplying them with 0.9, resulting in a composed system *0.9* that prioritizes the extra load when scheduled by EDF.

2) Data Age Results: Figure 7 depicts maximum and mean data ages observed during the execution of the case study as well as the system's average upper bound grouped by the additional load and applied transformation. To calculate single statistical values over *all* chains, we normalized all measured data age values to the chain's maximum data age, without JLDs. Please note that both, the maximum data-age results, as well as the maximum observed data-age results are average values over all systems. Therefore, the maximum observed values can be higher than the analytical results in this visualization. Considering the *maximum* data ages, one can see the positive effects of adding JLDs as well as the positive impact of our approach. Additionally, all theoretical upper bounds retain stable for all load configurations. While all variants besides $\mathrm{PLN}_{\mathrm{all}}$ generate comparable maximum and average data ages, the simple MRG approach produces slightly bigger maximum and average data ages. Interestingly, although

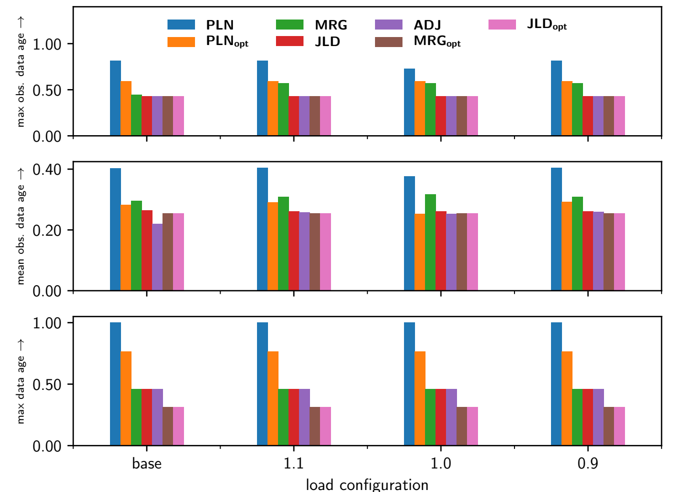


Figure 7. Measurements of our approach's variants for the three data propagation chains in varying load scenarios: Data ages are normalized to their chains upper data-age bound according to the *LET* paradigm.

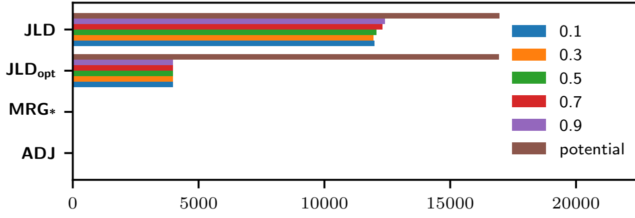


Figure 8. Count of actual blocking synchronization for each optimization variant during actual execution. Interleave chain system on four cores.

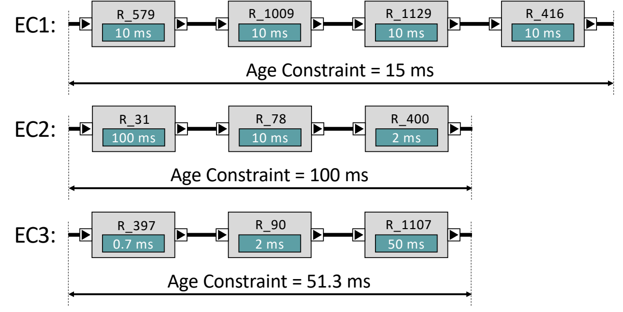


Figure 9. Waters Industrial Challenge '17: data propagation chains.

all optimized variants (ALL_{opt}) exhibit better theoretical data ages, they performed comparable to JLD. This indicates that our optimization is rather beneficial for the analysis, and a better fit for the actual execution, than for the performance. Observed data-ages are also stable for all load configurations while the simple overhead reduced variants MRG and ADJ seem to be beneficial for *base* and *1.0* when it comes to average data ages.

3) Overhead reduction: For Figure 8, we logged all semaphore synchronizations during execution and if they lead to a blocked job. The higher the load fraction, i.e., run-time utilization, the more often blocking occurs in the JLD systems. However, since our optimized-overhead variants ADJ$_*$ and MRG$_*$ are synchronized implicitly, no blocking or explicit synchronization occurs and can, therefore, be neglected for WCET and schedulability analysis.

C. Case Study: Waters Industrial Challenge '17

To demonstrate the scalability of our approach, we applied it to the Waters Industrial Challenge '17, available for download [18]. This system comprises three independent data chains (see Fig. 9) that include nine periodic and one sporadic runnables. Additionally, the system consists of 46 sporadic and 1194 periodic nonpartisan runnables and initially consists of 21 container tasks. To show the effectiveness of our merging algorithm for tasks developed by the concurrent-engineering paradigm we broke up the container task and interpreted each runnable as a task on its own. To fit our system model, we transformed the sporadic into periodic tasks with their minimal intermediate-arrival time as a period. We executed the system for 100 *chain hyperperiods* to measure data-ages. Although being a rather complex system, it only features three simple yet diverse data chains, wherefore we discuss them individually in the following. Fig. 9 depicts the structures of chains *EC1*, *EC2*, and *EC3* and Fig. 10 shows all data-age measurements aggregated across the fractions of WCETs. As those three chains do not cross, we omitted the multi-core variants.

EC1 consists of 4 tasks, each released with the same period $10\,\text{ms}$. Its LET data age without dependencies is, therefore, $40\,\text{ms}$. To guarantee the (artificial) data age constraint of $15\,\text{ms}$, JLDs are generated between all tasks, lowering the worst case to $10\,\text{ms}$. Since these JLDs are defined between tasks with the same period, they are actually simple 1:1 dependencies Fig. 10 shows the impact of JLDs and our approach to measured data ages. While the *plain* variant goes up to $32\,\text{ms}$, i. e., nearly to its worst-case, all JLD-affected executions stay well below their worst-case value of $10\,\text{ms}$. As expected, the merge and the *opt* variants show a better average performance than unoptimized JLD execution. Due to its pessimism, *adjtime* produces a range of observed data-ages almost 3 times bigger than *opt*.

Data chain *EC2* comprises 3 tasks with strictly decreasing periods from source "R_31" to sink "R_400" of $100\,\text{ms}$, $10\,\text{ms}$, and $2\,\text{ms}$. This implies a LET-data-age of $212\,\text{ms}$, while the generated JLDs allow to guarantee $100\,\text{ms}$. The strictly decreasing periods, however, lead to more or less equally distributed data ages between close to $0\,\text{s}$ and the maximum JLD data age, since at least every $2\,\text{ms}$ a $2\,\text{ms}$ older data age value is recorded. Therefore, the observed violations of data age constraints for *plain* execution are by far less than for *EC1*. Accordingly, the benefit of JLDs as a whole and our optimization are somewhat limited but visible. However, the additional laxity introduced by MRG_{max} increases the maximum data age significantly. Although our optimization eases this effect for $\text{MRG}_{\text{maxopt}}$, it stays above the other variants that incorporate JLDs.

Contrarily the data chain *EC3* features strictly increasing periods from $T_{R_5397} = 700\,\mu\text{s}$ to $T_{R_1107} = 50\,\text{ms}$. This pattern implies that for all unoptimized worst-case scenarios T_{R_1107} dominates the worst-case analysis. That means that even with generated JLDs, the worst-case data age $51.3\,\text{ms}$ is only slightly better than the LET based analysis with $53.3\,\text{ms}$. Since the worst-case, i. e., reading predecessor data at release time and writing it shortly before its *next* release time, for R_1107 never happens during actual execution, the measured data ages are far less than the worst-case values. That's where our optimization comes into play: The theoretical execution window of R_1107 is adjusted to the actual execution window by tightened deadlines that the worst analysis is as low as $8.13\,\text{ms}$ of the original analysis but still sound. Again the additional laxity of the MRG_{max} variant increases the upper bound significantly. However with $6.1\,\text{ms}$ the $\text{MRG}_{\text{maxopt}}$ has the lowest bound, reaching $11\,\%$ of the original value.

D. Schedulability

To assess the impact of our approach on schedulability, we applied it to a representative benchmark [12]. Its 433 systems are based on the characteristics of automotive applications reported in [2]. Each one comprises of randomly generated

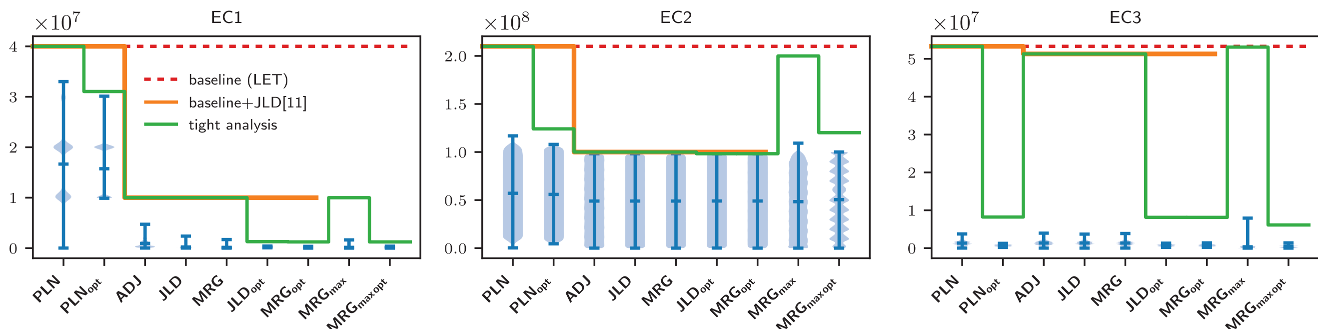


Figure 10. Analyzed and observed data age depending on optimization for each chain. Given lines mark sound upper bounds per analysis technique.

tasksets with a utilization between 0.6 and 2.0, leading to an average of 1.2. Occurring task periods are 1, 2, 5, 10, 20, 50, 100, 200, 1000 ms, and WCET ranges from [50,150] µs. The systems consist of 59 to 1000 tasks each and contain 1 to 3 data chains. Figure 11 depicts the systems' schedulability tested by the processor demand criterion after having partitioned the systems by worst-fit allocation to *two* cores. Schedulability of JLD serves as the lower baseline as it is a worst-case schedulability analysis for a dependent taskset, including additional WCET used for explicit synchronization. PLN however, is our upper baseline: No dependent task, no additional synchronizations overheads and thus the best schedulability, at the cost of worst data-age guarantees.

While the schedulability of most variants that feature JLDs, including MRG variants are in the middle between our two baseline variants, the scheduling enhanced merge MRG_{max} is consistently better than the others. However, MRG_{max} is outperformed by $\text{MRG}_{\text{maxMCopt}}$, the combination of MRG_{max}, the multicore optimization MRG_{MC} and our data-age optimization $_{\text{opt}}$, which almost reaches the same schedulability as PLN. That means that 50 % more systems with utilization of 1.8 are schedulabel by $\text{MRG}_{\text{maxMCopt}}$ than by JLD. When it comes to data-age performance $\text{MRG}_{\text{MCopt}}$ and MRG_{opt} perform best, while the best schedulability variant $\text{MRG}_{\text{maxMCopt}}$ is slightly worse, but still considerably better than the variants that feature JLDs but no additional optimization. Additionally, it can be seen that the more utilization a system has, the less effective our data-age optimization is. This is because our optimization cannot tighten deadlines of IO task if the system has no more laxity. The largest improvement between JLD and our proposed methods is 20 % for MRG_{opt} at 0.6 utilization.

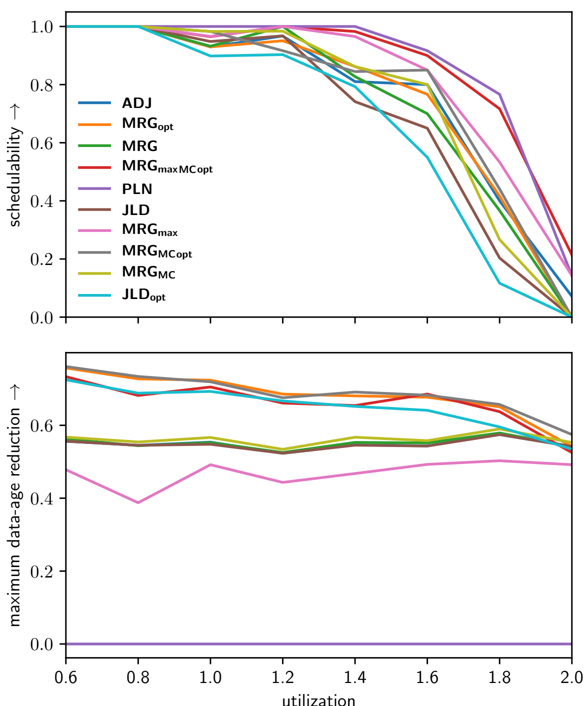


Figure 11. Schedulability and data age reduction of the experiment's system set depending on the integration mechanism used. For data age reduction, we only considered systems with JLDs, as our optimizations are based on JLDs.

VII. Conclusion and Future Work

We presented a novel approach to map data-age constrained tasksets onto multi-core real-time systems efficiently. We addressed the fundamental challenge of data-age bounds, synchronization overheads, and overall schedulability by an automated system transformation and analysis. Our transformation variants improve the scheduling of decisive jobs of data chains that impact data age and are, to the best of our knowledge, the first to explicitly consider crossing chains. Our evaluation proves that compared with traditional JLDs we can substantially reduce synchronization overheads, improve the schedulability up to 50 percent points, and reduce data-age analysis pessimism by up to 20 percent points.

To this end, our approach resembles a time-driven execution in sections by using tight execution windows. Consequently, our approach does not interfere with priority preemptive scheduling methodology in general yet requires deadline/release-time adherence. We are currently working on adapting our approach to fixed-priority preemptive scheduling by leveraging Chetto et al. [17] to map the precedence constraints, respectively.

Acknowledgment

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89 "Invasive Computing" and grant no. SCHR 603/15-2 and SCHR 603/9-2.

References

[1] B. Prasad, *Concurrent Engineering Fundamentals*. Prentice Hall Englewood Cliffs, NJ, 1996, vol. 1.

[2] S. Kramer, D. Ziegenbein, and A. Hamann, "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)*, 2015.

[3] N. Feiertag, K. Richter, J. Norlander, and J. Jonsson, "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)*, 2008.

[4] R. Wyss, F. Boniol, C. Pagetti, and J. Forget, "End-to-end latency computation in a multi-periodic design," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1682–1687.

[5] J. Forget, F. Boniol, and C. Pagetti, "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)*, 2017.

[6] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen, "How does control timing affect performance? Analysis and simulation of timing using jitterbug and truetime," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 16–30, 2003.

[7] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 10:1–10:20.

[8] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2018, pp. 240–250.

[9] J. Martinez, I. Sañudo, and M. Bertogna, "Analytical characterization of end-to-end communication delays with logical execution time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2244–2254, Nov. 2018.

[10] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete event dynamic systems*, vol. 21, no. 3, pp. 307–338, 2011.

[11] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "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*, 2016, pp. 159–169.

[12] T. Klaus, F. Franzmann, M. Becker, and P. Ulbrich, "Data propagation delay constraints in multi-rate systems: Deadlines vs. job-level dependencies," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*. ACM, 2018, pp. 93–103.

[13] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, 2001, pp. 191–198.

[14] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static timing analysis of real-time operating system code," in *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '04)*, ser. Lecture Notes in Computer Science. Springer, 2004, pp. 146–160.

[15] J. Schneider, "Why you can't analyze RTOSs without considering applications and vice versa," in *Proceedings of the 2nd Workshop on Worst-Case Execution Time Analysis (WCET '02)*, 2002, pp. 79–84.

[16] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis," in *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium (RTAS '19)*, I. C. Society, Ed., Montreal, 2019, pp. 318–330.

[17] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.

[18] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," 2017 [Online].

[19] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEEE Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.

[20] 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.

[21] M. Becker, D. Dasari, S. Mubeen, 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.

[22] T. Kloda, A. Bertout, and Y. Sorel, "Latency analysis for data chains of real-time periodic tasks," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, Sep. 2018, pp. 360–367.

[23] M. J. Friese, T. Ehlers, and D. Nowotka, "Estimating latencies of task sequences in multi-core automotive ecus," in *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, Jun. 2018, pp. 1–10.

[24] A. Girault, C. Prévot, S. Quinton, R. Henia, and N. Sordon, "Improving and estimating the precision of bounds on the worst-case latency of task chains," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2578–2589, 2018.

[25] T. Kloda, A. Bertout, and Y. Sorel, "Latency upper bound for data chains of real-time periodic tasks," *Journal of Systems Architecture*, 2020.

[26] M. Dürr, G. V. D. Brüggen, K.-H. Chen, and J.-J. Chen, "End-to-end timing analysis of sporadic cause-effect chains in distributed systems," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, pp. 58:1–58:24, Oct. 2019.

[27] J. Martinez, I. Sañudo, and M. Bertogna, "End-to-end latency characterization of task communication models for automotive systems," *Real-Time Systems*, 2020.

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

[29] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *Proceedings of the 44th Annual Design Automation Conference (DAC)*, 2007, pp. 278–283.

[30] W. Zheng, Q. Zhu, M. D. Natale, and A. S. Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *28th IEEE International Real-Time Systems Symposium (RTSS)*, Dec. 2007, pp. 161–170.

[31] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate dags from multi-rate task sets," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 226–238.

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

[33] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2010, pp. 301–310.

[34] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Proceedings of the 13th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE, May 2010, pp. 34–41.

[35] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors," in *Proceedings of the 19th IEEE International Symposium on OO Real-Time Distributed Computing (ISORC)*, May 2016, pp. 134–141.

[36] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, 2006, pp. 21–33.

[37] B. Wittenmark, J. Nilsson, and M. Törngren, "Timing problems in real-time control systems," in *Proceedings of the American Control Conf.*, vol. 3. New York, NY, USA: IEEE Press, 1995, pp. 2000–2004.

[38] J. Nilsson, B. Bernhardsson, and B. Wittenmark, "Stochastic analysis and control of real-time systems with random time delays," *Automatica*, vol. 34, no. 1, pp. 57–64, 1998.

[39] A. Bertout, J. Forget, and R. Olejnik, "Minimizing a real-time task set through task clustering," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, 2014, pp. 23–31.

[40] T. Klaus, P. Ulbrich, P. Raffeck, B. Frank, L. Wernet, M. Ritter von Onciul, and W. Schröder-Preikschat, "Boosting Job-Level Migration by Static Analysis," in *Proceedings of the 15th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, Jul. 2019, pp. 17–22.

[41] M. Chetto, *Real-time Systems Scheduling 1: Fundamentals*. John Wiley & Sons, 2014, vol. 1.

[42] H. Hoang, G. Buttazzo, M. Jonsson, and S. Karlsson, "Computing the minimum edf feasible deadline in periodic systems," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. IEEE, 2006, pp. 125–134.

[43] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "MECHA-niSer - 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)*, 2016.

[44] "Absint angewandte informatik gmbh. ait: Worst case execution time analyzers. http://www.absint.com/ait," 2020.

[45] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann, "Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems," in *Proceedings of the 14th IEEE International Symposium on OO Real-Time Distributed Computing (ISORC)*. IEEE, Mar. 2011, pp. 163–170.

[46] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011. [Online]. Available: http://www.cs.unc.edu/~bbb/diss/

[47] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *International Workshop on Embedded Software*. Springer, 2001, pp. 166–184.