Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains

Matthias Becker*, Dakshina Dasari†, Saad Mubeen*, Moris Behnam*, Thomas Nolte*
*MRTC / Mälardalen University, Sweden
{matthias.becker, saad.mubeen, moris.behnam, thomas.nolte}@mdh.se
† Research and Technology Centre, Robert Bosch, India
dakshina.dasari@in.bosch.com

***Abstract*—Today's automotive embedded systems comprise a multitude of functionalities, many with complex timing requirements. Besides task specific timing requirements, such applications often have timing requirements for the propagation of data through a chain of tasks. An important metric for control applications is the data age, which is addressed in this paper. The analysis of such systems is non-trivial because tasks involved in the data propagation may execute at different periods, which leads to over and undersampling within one chain. This paper presents a novel method to compute worst- and best-case end-to-end latencies for such systems. A second contribution synthesizes job-level dependencies for such task sets in a way that data paths which exceed the age constraint are eliminated. An extensive evaluation is performed on synthetic task sets and the applicability to industrial applications is demonstrated in a case study.**

I. Introduction

Today's automotive embedded systems are highly distributed in nature, and they are realized via multiple Electronic Control Units (ECU), sensors, actuators and communication buses. Physical parameters are sensed by the sensor nodes, passed on to computational nodes which process these parameters and finally sent to the actuator nodes (which may be part of a closed loop). An example is the cruise control application which senses engine speed, vehicle speed and the brake switch, sets and maintains vehicle speed through changing conditions (control) by adjusting the throttle position (actuation). Similar examples can be found in other industries such as avionics, and automation control. To be deemed correct, these systems have to not only execute the functions in a logically correct order to arrive at a desired reaction to the stimulus, but also must do so within predefined time bounds. Effect chains or event chains are a commonly used term to describe this sequence of steps performed along the control path to fulfill a certain functionality. Every effect chain has a timing requirement to satisfy either the fundamental physics of the underlying system or a user's performance demand. This timing requirement mandates that the end-to-end latency right from sampling to the actuation is within a designated range.

One of the challenges in designing such systems in order to meet timing constraints arises due to the presence of effect chains with multi-rate tasks in which the constituent tasks along the chain are activated with different periodicities (or rates). This can lead to consequences of either oversampling (the producer task generates data faster than the consumer task can consume it) and undersampling (the producer task generates data slower than the consumer task can consume it). As a result, data generated by the producer may be overwritten before a successor task has the chance to read them or the same value can be read by multiple instances of a successor task [1]. The complexity of analysis is further enhanced by the fact that certain segments of tasks may be part of multiple chains. As a consequence, the process of system integration, ensuring that all local timing requirements (i.e. the tasks deadlines) and all end-to-end latency requirements are met, becomes a non trivial problem. In this work, we target such applications consisting of multi-rate effect chains with advanced timing requirements.

It is important to note that while effect chains define a logical order of precedence among tasks, they *do not enforce constraints on task instances (or jobs)*, implying that in the presence of multi-rate tasks, a given task instance, say a consumer task may legally read data from *any* instance of its predecessor in the effect chain. Such unconstrained dependencies can result in missing the end-to-end latency requirements of the effect chain. We solve this problem by analyzing such errant effect chains and *enforce dependencies between selected task instances* in such a way that all end-to-end latencies are met while the additionally introduced constraints for the system are minimized. This is done during the early design phase of the system where no concrete knowledge about the platform or the scheduler is available. Hence the approach is independent of the concrete scheduling algorithm.

For an effect chain described by tasks with aforementioned job-level dependencies, [2] showed that fixed priority scheduling mechanisms can be applied without synchronization mechanisms and [3] showed how to schedule such systems using dynamic scheduling algorithms. Similarly [4], [5] define a time-triggered schedule for many-core platforms. However, to the best of our knowledge, no formal method for the analysis of effect chains to identify job-level dependencies has been defined and currently it is left to the discretion of the system designer. But manually identifying these dependencies is extremely challenging for systems with a large number of tasks and interconnected cause-effect chains, as found in today's automotive systems [6], and therefore research to address this problem is warranted.

Towards addressing this problem, the main contributions of this work are:

- A novel method is presented to calculate the possible data propagation paths within a system with possible job-level dependencies, independent of the concrete scheduling algorithm. These paths are then used to compute minimum

and maximum end-to-end latencies of the effect chains.

- A heuristic solution is presented to augment an application model with job-level dependencies in order to meet the specified end-to-end timing requirements of all effect chains specified for the system.
- The approach is evaluated based on synthetic experiments as well as an industrial case study where the solution is compared to a state-of-practice tool.

The rest of this paper is organized as follows. Section II discusses the relevant related work, followed by the discussion of background and the system model in Section III. Section IV presents concepts to decide reachability among jobs. Section V introduces an algorithm to compute minimum and maximum latencies in the system. The synthesis of job-level dependencies is introduced in Section VI. Section VII experimentally evaluates the approach and Section VIII concludes the paper.

II. Related Work

In [7] Forget et al. describe PRELUDE, an architecture language intended for the design of multi-rate dependent control systems. PRELUDE is built on the principles of synchronous languages such as LUSTRE [8] but extends them with rate-transition operations to cater for the needs of multi-rate real-time systems. Several works address systems described by PRELUDE. In [2] Forget et al. show how such systems can be scheduled by fixed priority scheduling policies without the need for additional synchronization mechanisms. Similarly, in [3] Pagetti et al. show how such a system can be scheduled by dynamic priority scheduling schemes such as Deadline Monotonic (DM) or Earliest Deadline First (EDF). Puffitsch et al. [5] describe an end-to-end framework targeting a many-core platform. A heuristic partitions tasks on the individual cores, taking communication between tasks into account. Later in [4], Puffitsch et al. propose a time-triggered framework built on a general many-core model, where constraint programming is used to generate the time-triggered schedule. These works show how to successfully execute complex multi-rate dependent task sets on various hardware architectures using different scheduling policies. However, these works assume that a system designer is responsible to define the rate-transition operations and hence the job-level dependencies. While this is viable in many use-cases, large systems with interleaved dependencies might impose challenges for the system designer exacerbating the selection of rate-transition operations in such a way that all timing requirements are fulfilled.

One of the most complex ECU in an automotive system is the Engine Management System (EMS) where the functionality is spread over up to 2000 modules (atomic SW components) [6]. Several end-to-end requirements are defined in standards such as EAST-ADL [9] and AUTOSAR [10] to guarantee correct behavior of such systems. In [1], Feiertag et al. propose a framework to calculate end-to-end latencies in automotive systems, where the implicit communication model of AUTOSAR is considered. In [11], Mubeen et al. integrate the end-to-end path delay analysis with the Rubus-ICE which is a commercial tool suite for model- and component-based software development of vehicular embedded systems. To the best of our knowledge *no existing method or tool directly supports the end-to-end latency calculations for the chains with specified job-level dependencies*.

Several works focus on the synthesis of tasks, where atomic SW components are mapped to tasks of the operating system. Panic et al. [12] describe an algorithm to parallelize legacy systems on a multi-core architecture. Similarly, Faragardi [13] maps the atomic SW components of an automotive application to tasks of a multi-core system using evolutionary algorithms. Both works consider only end-to-end latencies between tasks of same period. Davare et al. [14] allow for multi-rate end-to-end latencies in their applications but task periods are subject to optimization in order to meet the system requirements.

Schliecker and Ernst [15] apply a recursive approach to determine end-to-end path latencies in heterogeneous multiprocessor systems. Their approach considers pipelined and transient effects, leading to tight end-to-end path latencies. In contrast to our work they calculate maximum and minimum latencies for event chains whereas we are interested in maximum and minimum latencies for a concrete sequence of jobs which in turn is used to synthesize the job-level dependencies.

III. Background and Motivation

This section provides the required background information of the system model and introduces the end-to-end timing requirements found in industrial applications.

A. System Model

We base our application model on standard automotive applications which typically comprise a set of periodic preemptive tasks. A task can be either time-triggered or event-triggered and in this work we focus on applications that are comprised of time-triggered tasks, which are periodically triggered by a system clock. We describe a task τ_i by the tuple $\{T_i, C_i\}$, where T_i describes the fixed activation interval (called the period or rate of the task), and C_i the Worst-Case Execution Time (WCET). Without loss of generality, all tasks are released simultaneously, hence there is no offset. We further assume implicit deadlines, i.e. $D_i = T_i$, and the j^{th} job of τ_i is denoted by $\tau_{i,j}$. All tasks of the application are part of the set Γ and the hyperperiod HP of the task set is the least common multiple of all task periods $\in \Gamma$. Communication among tasks is realized via shared registers; a sending task writes a value to the communication register and a receiving task reads the current register value. Such a scenario is shown in Fig 1. We further assume that each task operates according to the *read-execute-write* semantic, wherein the task reads in all the required inputs into local copies *before* execution, executes by operating on these local inputs and finally writes the output *after* execution. One industrial example for this execution model is the implicit communication model of AUTOSAR [16].

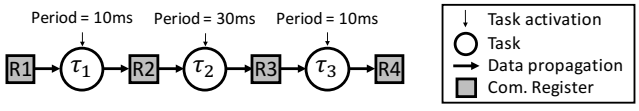


Fig. 1: Register communication among communicating tasks.

The hardware platform is assumed to be either a single- or multi-core processor with shared memory to allow for the introduced communication model.

B. End-to-End Timing Requirements

As discussed earlier, in control systems, often it is not only important that a task finishes execution before a given deadline but also that input data is propagated through a chain of tasks within a given time interval. In AUTOSAR, this chain is called a *Cause-Effect Chain* [10] and hereafter this phrase is used to denote a chain of directed dependent tasks.

We model a cause-effect chain using a Directed Acyclic Graph (DAG) $\zeta = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V}$ is the set of nodes, representing tasks, and $\mathcal{E}$ is the set of edges, representing the data propagation across the nodes. We define $\text{next}(\tau_i, \zeta)$, a function that returns the successor of the task τ_i in the cause-effect chain ζ. Note, such a chain can have junctions and joints but the source node and sink node must be the same for all possible data paths of the chain [10]. However, this does not imply that the latency for a certain input value of the first segment in the chain until the last segment is the same for all possible paths (i.e., the latencies must only fulfill the requirements of being within the specified minimum and maximum latency). If a cause-effect chain contains junctions we linearize it by constructing a chain for both possible paths. Hence all chains considered in the remainder of the paper are linear, acyclic and do not consists of junctions. Note that a single task can further be part of several cause-effect chains. We define Π as the set of all cause-effect chains.

Timing constraints for these chains are specified by two delay semantics, *data age* and *data reaction* constraints. A detailed definition of both can be found in [1]. While the same principles can be used to target the other latency types specified for chains, in this work we focus on *maximum data age* which is described in the following.

Data age constraints are commonly found in control systems, where the data age can directly influence the quality of the control. With a data age constraint, it is important to know for how long input data affects an output, i.e., the time is seen from the occurrence of a response. Therefore, a (max) age constraint of "k" time units for a cause-effect chain mandates that for an occurrence of a response event, the corresponding input data is not older than "k" time units.

This is shown in Fig. 2, where data is propagated through a chain of tasks. Note that the maximum data age can be measured starting from the first instance of τ_1, which reads the input value at $t = 0$. The data then propagates to the first instance of τ_2 and further to τ_3. Note that τ_3 is released more frequently than τ_2 and thus the first three instances of τ_3 read the same input data. Consequently the data age is measured until the end of the instance of τ_3 which consumes the data last (i.e. at $t = 6$). One can also observe that, due to the different execution rates, the values produced by the second and third instance of τ_1 are overwritten before an instance of τ_2 has the chance to read them.

1) Sampling: Since tasks of a chain may possibly have different periods there can exist three different activation patterns between two consecutive tasks of a chain, given τ_i precedes τ_j. Oversampling, where $T_i < T_j$, undersampling, where $T_i > T_j$, and same rate, where $T_i = T_j$.

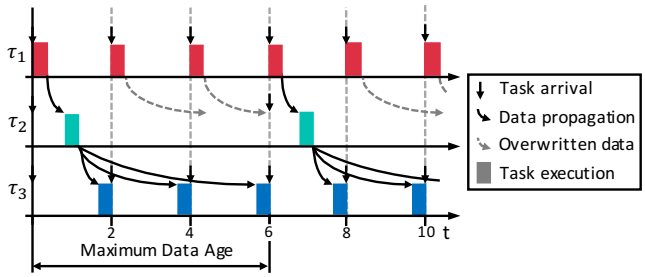


Fig. 2: Data propagation paths and maximum data age in a real-time system.

The least complex scenario, when it comes to the analysis of end-to-end latencies in such systems, is achieved when all tasks of a chain operate at the same rate. In industrial applications such single- or homogeneous-rate chains constitute a majority of the cause-effect chains which are augmented with timing constraints. E.g. in [6] it is reported that 70% of the cause-effect chains specified in a modern Engine Management System (EMS) fall into this category, while the remaining 30% are comprised of tasks of either 2 or 3 different periods in one cause-effect chain.

C. Computation of Possible Data Propagation Paths

Consider a cause-effect chain ζ of tasks $\{\tau_i, \tau_{i+1} \dots \tau_{\text{length}(\zeta)}\}$ with corresponding periods $\{T_i, T_{i+1} \dots T_{\text{length}(\zeta)}\}$. A certain data value can be propagated through multiple data paths until it reaches the end of a cause-effect chain. For a given job of the chain with parameters C_i and T_i, its output data can be available to its successor in the chain for at most $(2 \cdot T_i) - C_i$ time units. This is the case when the two consecutive jobs of τ_i execute as early and as late as possible in their respective execution windows. A successor task with period T_{i+1} in the chain can then read the value of this job and propagate its output to its successors. This principle is then repeated for all segments (a producer and a consumer task) of the cause-effect chain. Then the total number of data propagation paths is given by:

$$\#\,\text{paths} = \prod_{i \in [1, \text{length}(\zeta)-1]} \left(\left\lceil \frac{(2 \cdot T_i) - C_i}{T_{i+1}} \right\rceil + 1 \right)$$

The calculation for each segment needs to be appended with 1 to cover the case where an interval is partially overlapping at input and output of the chain. This value then needs to be multiplied with the number of starting instances of the chain $\frac{\text{LCM}(\zeta)}{T_{root}}$.

D. Introducing Job Level Dependencies

In this work, we synthesize job-level dependencies for a cause-effect chain in a periodic task set. This is done to ensure the end-to-end timing requirements of the system are met while the job-level dependencies are enforced.

A job-level dependency is specified across successive tasks in a chain and specifies which job of a task needs to finish its execution before a job of the successor task can start. Such job-level dependencies are similar to the rate transition operator in PRELUDE [7].

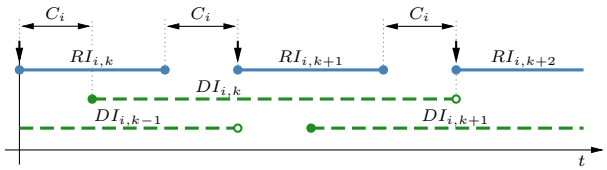


Fig. 3: Read and data intervals of consecutive jobs of τ_i.

Definition 1. *We define a job level dependency* $\tau_i \xrightarrow{(k,l)} \tau_j$*, meaning there is a dependency between the* k^{th} *job of* τ_i *and the* l^{th} *job of* τ_j*. This also implies that the dependency between the two jobs applies for the duration of the hyperperiod of the two jobs only, e.g.* $LCM(\tau_i, \tau_j)$*. For single-rate systems, the job indexes* k *and* l *are the same for both tasks which is equivalent to task level dependencies.*

IV. Deciding Reachability between Jobs

In this section, we introduce basic concepts to decide if two jobs may propagate data between them. This is required to compute the data paths and latencies within a cause-effect chain. A general concept of read and data interval is introduced and gradually extended to capture the characteristics of multi-segment chains and the presence of job-level dependencies.

A. Read- and Data-Interval

As described before, a task may execute at any time within its execution window and this exact time of execution is not known a priori at design time when scheduling decisions are not yet taken. Hence all possible cases must be considered. Fig. 3 depicts the time interval during which a task may read its input data (depicted by the solid blue line) as well as the interval for which the output data might be available to its successor task (the dashed green line). The *read interval* is defined as the interval in which a task can possibly read its input data in order to complete its execution before the deadline. The *data interval* is defined as the interval for which the output data of a task can be available to the successor task in the chain. It stretches up-to the latest time at which the output data of a given task instance is possibly available before the next instance overwrites it.

In order to simplify the explanation we define a set of notations for a job $\tau_{i,j}$ of a task τ_i, where $j \geq 1$:

$$\begin{aligned} R_{min}(\tau_{i,j}) &= (j-1) \cdot T_i \\ R_{max}(\tau_{i,j}) &= R_{min}(\tau_{i,j+1}) - C_i \\ D_{min}(\tau_{i,j}) &= R_{min}(\tau_{i,j}) + C_i \\ D_{max}(\tau_{i,j}) &= R_{max}(\tau_{i,j+1}) + C_i \end{aligned}$$

Where $R_{min}(\tau_{i,j})$ and $R_{max}(\tau_{i,j})$ describe the earliest and latest point in time a job $\tau_{i,j}$ can read data (i.e. the job starts at release time or the job finishes with its deadline). Similarly $D_{min}(\tau_{i,j})$ and $D_{max}(\tau_{i,j})$ describe the earliest and latest point in time the output data of a job $\tau_{i,j}$ can be available to a successor. The earliest point in time output data can be available is C_i time units after the release of the job. This output data can be available as late as the latest finishing time of the next job (within its deadline). Note that $D_{min}(\tau_{i,j+1})$ is smaller than $D_{max}(\tau_{i,j})$. Hence, if a successor reads the data in the overlapping window the data can be from either instance of τ_i, depending on the specific schedule.

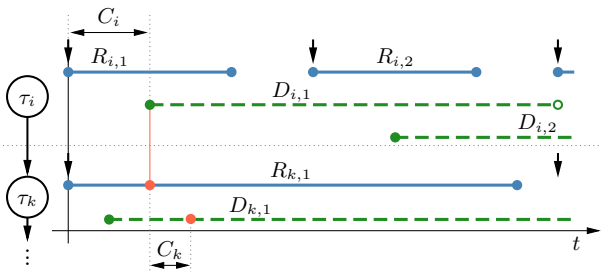


Fig. 4: $\tau_{i,1}$ affects the window $R_{k,1}$ and $D_{k,1}$ if the data of $\tau_{i,1}$ is consumed by $\tau_{k,1}$.

The read interval $RI_{i,j}$ of $\tau_{i,j}$ is defined by $[R_{min}(\tau_{i,j}), R_{max}(\tau_{i,j})]$ and the data interval $DI_{i,j}$ by $[D_{min}(\tau_{i,j}), D_{max}(\tau_{i,j}))$.

B. Data Propagation in the Cause-Effect Chain

In order to compute the minimum and maximum end-to-end latencies within the system, it is important to know for each pair of consecutive tasks of a cause-effect chain (such a pair can be referred to as a segment of the cause-effect chain), which job of a successor task might read data produced by a specific job of its predecessor task. This section introduces relevant concepts to define when data can be propagated between two instances of a chain segment.

Consider the cause-effect chain segment $\tau_i \rightarrow \tau_k$ with a producer job $\tau_{i,j}$ and a consumer job $\tau_{k,l}$. For $\tau_{k,l}$ to be a possible consumer of the data produced by $\tau_{i,j}$, the following needs to be true:

$$R_{max}(\tau_{k,l}) \geq D_{min}(\tau_{i,j}) \wedge R_{min}(\tau_{k,l}) < D_{max}(\tau_{i,j}) \quad (1)$$

We define a function $\text{Follows}(\tau_{i,j}, \tau_{k,l})$, that returns *true* if, as per the above condition (1), the read interval of $\tau_{k,l}$ intersects with the data interval of $\tau_{i,j}$, making $\tau_{k,l}$ a possible successor of $\tau_{i,j}$.

1) Dependency between Two Tasks: Let's assume that the segment $\tau_i \rightarrow \tau_k$ is part of a cause-effect chain. As shown before, all possible successors of a job $\tau_{i,j}$ can be described by the set:

$$\mathcal{P}_{\tau_{i,j} \rightarrow \tau_k} = \{\tau_{k,l} | \, \text{Follows}(\tau_{i,j}, \tau_{k,l}) = true\} \quad \tau_k \in \Gamma, \; l \in \mathbb{N},$$

The first instance l of τ_k which might be a successor of $\tau_{i,j}$ can be computed via the following equation:

$$l = \left\lceil \frac{D_{min}(\tau_{i,j})}{T_k} \right\rceil \quad (2)$$

Starting from this instance, we iterate through all successors of $\tau_{i,j}$ until $\text{Follows}(\tau_{i,j}, \tau_{k,l+m})$ returns $false$ (where $m \in \mathbb{N}$).

2) Dependency between Multiple Tasks: The previous computation is applicable only to a chain containing two tasks. We now describe the method to compute the dependency between a chain of multiple tasks.

Let's assume the task τ_i precedes τ_k in a cause-effect chain. As the example in Fig. 4 depicts, the output window of $\tau_{k,1}$ may start before the output window of $\tau_{i,1}$ does. Hence, if $\tau_{i,1}$ precedes $\tau_{k,1}$, the output window $D_{k,1}$ cannot directly be used to determine possible predecessors of $\tau_{k,1}$.

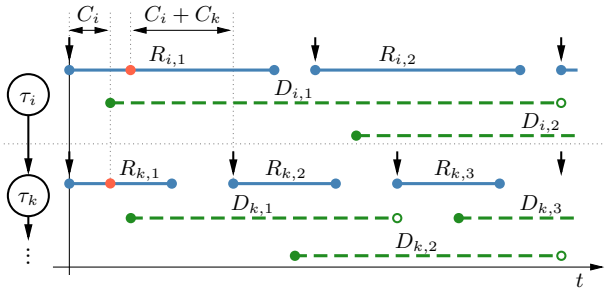


Fig. 5: Required modifications to the read interval of job $\tau_{i,1}$ and job $\tau_{k,i}$, if they are dependent through a precedence constraint $\tau_i \xrightarrow{(1,1)} \tau_k$.

We can overcome this limitation by introducing $D'_{min}(\tau_{k,l}, \tau_{i,j})$, which describes the first possible data output of $\tau_{k,l}$ in the case that $\tau_{i,j}$ is its predecessor. This can be computed as the maximum of $D_{min}(\tau_{k,l})$ and the first appearance of the predecessor data augmented with the computation time of the successor $D_{min}(\tau_{i,j}) + C_k$:

$$D'_{min}(\tau_{k,l}, \tau_{i,j}) = \max(D_{min}(\tau_{i,j}) + C_k, D_{min}(\tau_{k,l}))$$

Note, that for jobs without any predecessor (for example, the job of the first task of a cause-effect chain), this function returns $D_{min}(\tau_{k,l})$.

If a chain contains several tasks, we now need to substitute $D_{min}(\tau_{i,j})$ in Equations 1 and 2 with $D'_{min}(\tau_{i,j}, \tau_{a,b})$, where $\tau_{a,b}$ is selected as predecessor of $\tau_{i,j}$. This constitutes the forward reachability criterion described in [1], i.e. a reading task can not start its execution before a writing task finishes.

Note, the adjustment to the data interval must be local to each branch in the data propagation tree since this modification reflects the property of forward reachability of a *specific task instance* selected for the current branch.

C. Including Job-level Dependencies

While cause-effect chains do impose a logical ordering of tasks, they *do not* enforce any explicit dependencies between task instances. Since the intention of this paper is to augment the task constraints by job-level dependencies in order to meet the end-to-end latency requirements, such job-level dependencies must be considered in the calculations.

If a job-level dependency is defined between tasks of a cause-effect chain, the possible data propagation gets affected. 1) The read intervals of the affected jobs need to be adjusted to incorporate these constraints. 2) Job-level dependencies additionally introduce logical boundaries for the possible data propagation. Both cases are addressed separately, as described below.

1) Adjusting Read Intervals: Intuitively, for two jobs constrained by $\tau_i \xrightarrow{(j,l)} \tau_k$, $\tau_{k,l}$ must start at least C_i time units after the release of $\tau_{i,j}$ and similarly $\tau_{i,j}$ must finish at least C_k time units before the latest execution of $\tau_{k,l}$. This is shown in Fig. 5, where job $\tau_{i,1}$ must precede job $\tau_{k,1}$ (i.e. R_{min} of $\tau_{k,1}$ is delayed and R_{max} of $\tau_{i,1}$ is earlier).

If a task set Γ has a number of job-level dependencies in the set Ψ, the calculations for the modified read-intervals of jobs constraint by $\tau_i \xrightarrow{(j,l)} \tau_k$ must consider all other dependencies

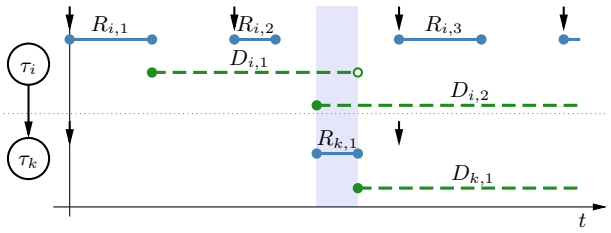


Fig. 6: Read and data interval of τ_i and τ_k, if they are dependent through a precedence constraint $\tau_i \xrightarrow{(2,1)} \tau_k$.

which share the same job. Hence, for a job-level dependency $\tau_i \xrightarrow{(j,l)} \tau_k \in \Psi$, we define a set $\mathcal{P}$ and a set $\mathcal{S}$:

$$\mathcal{S} = \{\tau_{f,a} | \exists \tau_i \xrightarrow{(j,a)} \tau_f \in \Psi, \forall \tau_f \in \Gamma, a \in [1, \frac{HP}{T_i}]\}$$

$$\mathcal{P} = \{\tau_{f,a} | \exists \tau_f \xrightarrow{(a,l)} \tau_k \in \Psi, \forall \tau_f \in \Gamma, a \in [1, \frac{HP}{T_k}]\}$$

The set $\mathcal{S}$ contains all jobs part of a job-level dependency $\in \Psi$, which have the job $\tau_{i,j}$ as predecessor (i.e. on the left side of the job-level dependency). Similarly, the set $\mathcal{P}$ contains all jobs of a job-level dependency $\in \Psi$ which have the job $\tau_{k,l}$ as successor (i.e. on the right side of the job-level dependency).

In the following, we show with an example, how to adapt the read interval, modifying R_{min} for $\tau_{k,l}$ and R_{max} for $\tau_{i,j}$, considering the first occurrence of the affected jobs:

$$R'_{min}(k,l) = \max(\max_{\forall \tau_{f,a} \in \mathcal{P}}(R_{min}(f,a) + C_f), R_{min}(k,l))$$

$$R'_{max}(i,j) = \min(\min_{\forall \tau_{f,a} \in \mathcal{S}}(R_{max}(f,a) - C_i), R_{max}(i,j))$$

Note that changing $R'_{max}(i,j)$ will impact $D_{max}(i,j-1)$ as well. Not all jobs of τ_i and τ_k are impacted by the job-level dependencies. A job-level dependency defines dependencies in the duration restricted to the hyperperiod of the two tasks. Hence, only every $\frac{LCM(\tau_i,\tau_k)}{T_i}$ job of τ_i, starting from job $\tau_{i,j}$, and every $\frac{LCM(\tau_i,\tau_k)}{T_k}$ of τ_k, starting from job $\tau_{k,l}$, is affected.

The modifications described here are global for a task set and therefore modifications can be done in a preprocessing step before the individual data propagation paths are computed.

2) Incorporating Logical Boundaries: The presence of job-level dependencies introduces additional constraints for the selection of possible successor jobs. Assume a job-level dependency which enforces that a job $\tau_{i,j}$ must execute before $\tau_{k,l}$. Then this dependency introduces a logical boundary for the data propagation by which an earlier instance of τ_i, lets say $\tau_{i,j-1}$, *cannot* propagate its output data to $\tau_{k,l}$, even though the function $\text{Follows}(\tau_{i,j-1}, \tau_{k,l})$ may indicate otherwise. If the data propagation would occur, the job-level dependency constraint is violated. Fig. 6 shows a scenario where $\tau_{k,1}$ may receive data from two jobs of τ_i, visualized by the shaded background during the read interval of $\tau_{k,1}$. However, since a job-level dependency is specified as $\tau_i \xrightarrow{(2,1)} \tau_k$, the data propagation from $\tau_{i,1}$ to $\tau_{k,1}$ is infeasible or not allowed. Hence, $\text{Follows}(\tau_{i,j-1}, \tau_{k,l})$ must be extended to incorporate these logical boundaries.

In order to check if an edge from a job $\tau_{i,j}$ to a job $\tau_{k,l}$ is feasable and does not violate a job-level dependency,

it is important to verify that there is no job $\tau_{i,a},\ (a > j)$, which must execute before $\tau_{k,l}$ in order to satisfy a job-level dependency.

For each job-level dependency $\tau_i \xrightarrow{(x,y)} \tau_k \in \Psi$ it must be checked if such a case exists. The steps to verify and act on instance $\tau_{k,l}$ are listed here.

1) Compute δ_k, the number of jobs that τ_k may execute in the hyperperiod of the job level dependency, i.e. $LCM(\tau_i, \tau_k)$.
2) Compute the offset of the instance l. The modulo operation $l \mod \delta_k$ then yields the offset (in jobs) of the job instance from the start of a hyperperiod of the job-level dependency. A special case exists if the modulo function returns 0, then the offset of the job is δ_k.
3) If the computed offset equals the offset y of the considered job level dependency, then the instance $\tau_{k,l}$ *is affected* by the job level dependency.
4) Then the instance number of the job of τ_i which must precede $\tau_{k,l}$ can be computed by $(\left\lceil \frac{j}{\delta_i} \right\rceil \cdot \delta_i) + x$. If this is larger than j, $\tau_{k,l}$ must succeed a later instance of $\tau_{i,j}$ and hence $\tau_{i,j}$ can have *no edge* in the data propagation tree to $\tau_{k,l}$.

V. Constructing the Data Propagation Tree

In this section, we present the algorithm to construct the data propagation tree. A data propagation tree consists of a hierarchy of levels, wherein each level corresponds to a task in the cause-effect chain. In this tree, the job of the first task in a cause-effect chain is designated as the root. At any level in the tree, each job then has an edge to all its possible successor jobs belonging to the next task in the chain. All data paths from the root node to the different leaf nodes represent the possible paths through which the data can be propagated in the cause-effect chain. This takes the individual execution of jobs, as well as specified job-level dependencies into account, since the concepts described in Section IV are applied.

A prerequisite to the algorithm is the job-set $\mathcal{J}$ which includes jobs of all tasks involved in the cause-effect chain. We can bound the number of jobs in $\mathcal{J}$ by including only those which are released within the worst-case end-to-end latency of the chain, starting from the root job of the data propagation tree. The worst-case end-to-end latency of a cause-effect chain, consisting of a set of tasks, can be computed by cumulatively summing up each tasks' period and response times [14]. This scenario results if every task in the chain executes *as early* as possible but the input data is written *just after* the task started, implying this instance misses the input data. Additionally, the next job of the task is scheduled *as late* as possible in its execution window, i.e., it executes upto its Worst-Case Response Time (WCRT). After execution, the output data is written, leading to the worst case data propagation time. Hence the effective latency between the input and the availability of the *corresponding output* is one entire period for the first instance plus the WCRT of the second instance. Since it is not known how tasks are scheduled, the WCRT for the task instance can be safely assumed to be equal to its period. Hence, a pessimistic value for the worst-case end-to-end latency (WCL) can be computed for a given cause-effect chain ζ: $WCL = \sum_{\tau_i \in \zeta} 2 \cdot T_i$.

We construct a data propagation tree by a recursive approach described in Algorithm 1. The algorithm takes several input parameters: the cause-effect chain ζ, the current node P of the data propagation tree, the job $\tau_{i,j}$ associated with the node P, and the job set $\mathcal{J}$.

The algorithm starts by finding τ_k, the successor of τ_i in the chain (line 2). If τ_k is the last task in the chain, the $\mathrm{next}()$ function returns $\emptyset$. In this case, a leaf node is reached and the algorithm returns (line 3). In all other cases, the algorithm examines all jobs in $\mathcal{J}$ belonging to τ_k. If these jobs are consumers of the data produced by $\tau_{i,j}$, (line 6) D'_{max} is updated, a new tree node is created corresponding to each consumer job, and appended to the current branch of the data propagation tree (line 7-9). The tree is further populated with a recursive invocation using the task $\tau_{k,l}$ as input parameter (line 10) and converges when all possible paths have been explored and added to the tree.

Algorithm 1: buildPropTree($\zeta, P, \tau_{i,j}, \mathcal{J}$)

```
1  begin
2      τ_k ← next(τ_i, ζ);
3      if τ_k = ∅ then
           // Found a leaf node
4          return
5      for ∀τ_{k,l} ∈ J do
6          if Follows(τ_{i,j}, τ_{k,l}) then
               // Add a new node to the branch
7              update D'_max locally;
8              n ← createChild(τ_{k,l});
9              appendNode(P, n);
10             buildPropTree(ζ, n, τ_{k,l}, J);
```

1) Pruning Unnecessary Branches: The main objective of the data propagation tree is often to obtain only minimum and maximum latencies of a cause-effect chain. Therefore, pruning branches which cannot lead to minimum or maximum latencies can significantly speed up the computation. At any level, among all possible jobs (i.e. the sibling nodes in the tree), only two jobs need to be considered for further computation.

Let us define a set $\mathcal{G}$ that contains all jobs of task τ_k that are possible successors of a job $\tau_{i,j}$ (which can be found by the recursive algorithm described earlier). If the set $\mathcal{G}$ contains multiple successor jobs, only two jobs need to be considered for the calculation of minimum and maximum latency. For the minimum latency, the job $\tau_{k,l}$ needs to be considered, where $l = \min_{\forall \tau_{k,w} \in \mathcal{G}}(w)$ or simply put, l has the minimum job index in the set $\mathcal{G}$. Likewise, for the maximum latency the job $\tau_{k,m}$ needs to be considered, where $m = \max_{\forall \tau_{k,w} \in \mathcal{G}}(w)$ or simply put, m has the maximum job index in the set $\mathcal{G}$.

Lemma 1. *If all jobs $\in \mathcal{G}$ are possible successors of $\tau_{i,j}$, then the (minimum and maximum) latency between $\tau_{i,j}$ and a job $\tau_{k,l} \in \mathcal{G}$ monotonically increases with l.*

Proof: The proof is based on the premise that the reference job $\tau_{i,j}$ is fixed and the release time of two consecutive successor jobs in $\mathcal{G}$ is separated by the interval T_k. Specifically, a job $\tau_{k,l}$ is released a time $T_k \cdot (l-1)$ and two jobs of the same task can never co-exist at the same time (as per the task

model). This also implies that the execution window of every next successor job in $\mathcal{G}$ also monotonically increases by T_k time units, effectively leading to an increase in the latency. ■

A. Generating all Possible Data Propagation Trees

The algorithm presented earlier to construct the data propagation tree only explores one job of the first task found in the cause-effect chain. In this paragraph, we describe how to compute all possible data propagation trees of one cause-effect chain which is needed to determine the minimum and maximum latencies of the chain.

Algorithm 2 describes how to construct the set of possible data propagation trees $\mathcal{T}$. The algorithm requires the cause-effect chain under analysis ζ, the task set Γ, and the set of job-level dependencies Ψ as input parameters. During the initialization phase (line 2-3), the hyperperiod of the task set and all jobs of tasks in Γ are generated for the required interval. Further, the first task in the chain ζ is defined on line 4 and the set of all data dependency trees is generated on line 5. Note that this set is initially empty.

The algorithm selects all jobs of τ_i which are scheduled within HP and generates the respective data propagation tree dpt (line 8). This data propagation tree is then added to $\mathcal{T}$ (line 9). Once all data propagation trees are constructed the algorithm returns $\mathcal{T}$ on line 10.

Algorithm 2: generateTrees(ζ, Γ, Ψ)

```
1  begin
2      HP ← LCM(Γ);
3      J ← generateJobs(Γ, Ψ);
4      τ_i ← next(∅, ζ); // Get first task in chain
5      T ← ∅; // empty set of data prop. trees
6      for ∀τ_{i,j} ∈ J, j < HP/T_i do
7          dpt ← createRootNode(τ_{i,j});
8          buildPropTree(ζ, dpt, τ_{i,j}, J);
9          T.add(dpt);
10     return T;
```

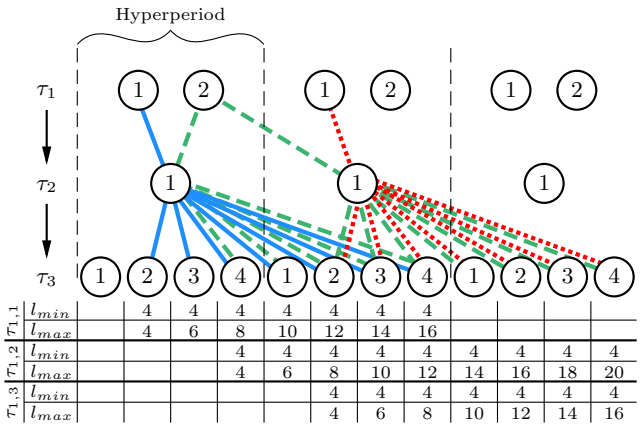
B. Data Age Latencies in the Data Propagation Tree

Once all data propagation trees are computed, it is possible to calculate minimum and maximum possible path latencies in the system. In the given context, they correspond to the data age latencies, since they compute the delay between the instant a input value is sampled first until the output value becomes available. Note that we assume that for each path in the tree, tasks are executed in a way such that the data propagates between the instances within the path from the root node to the respective leaf. All such combinations therefore represent possible data paths in the system but they can not all be observed in the system simultaneously.

Let's define the function $\mathrm{maxLatency}(\tau_{root}, \tau_{leaf}, dpt)$ to return the maximum latency of the path from τ_{root} to τ_{leaf}, where both root and leaf job are part of the data propagation tree dpt. Then $\mathrm{maxLatency}(\tau_{root}, \tau_{leaf}, dpt)$ is computed as:

$$(R_{max}(\tau_{leaf}) + C_{leaf}) - R_{min}(\tau_{root})$$

As presented in the equation above, the maximum latency from τ_{root} to τ_{leaf} is achieved if both jobs are scheduled at the extremes of their intervals, i.e. the root job starts as early as possible and the leaf job finished as late as possible.



$\tau_{1,1}$	l_{min}		4	4	4	4	4	4	4			
	l_{max}		4	6	8	10	12	14	16			
$\tau_{1,2}$	l_{min}				4	4	4	4	4	4	4	4
	l_{max}				4	6	8	10	12	14	16	18	20
$\tau_{1,3}$	l_{min}						4	4	4	4	4	4	4
	l_{max}						4	6	8	10	12	14	16

Fig. 7: Cause-effect chain C and the data propagation trees for the first three instances of τ_1. Minimum and maximum latencies l_{min} and l_{max} for each data propagation tree is shown in the table.

Similarly, we define $\mathrm{minLatency}(\tau_{root}, \tau_{leaf}, dpt)$ a function which returns the minimum latency of the same path. For the minimum latency, both jobs must be scheduled as close together as possible. This means, the root job must be scheduled as late as possible, C_{root} time units before the read job of the second job in the branch $\mathrm{succ}(\tau_{root})$ can be scheduled, and the leaf job is scheduled as early as possible. Then $\mathrm{minLatency}(\tau_{root}, \tau_{leaf}, dpt)$ is computed as:

$$D_{min}(\tau_{leaf}) - \max(R_{min}(\tau_{root}), R_{min}(\mathrm{succ}(\tau_{root})) - C_{root})$$

Both formulations reflect the age of one data path of the data propagation tree. In order to compute the *global* minimum and maximum latencies of a cause-effect chain denoted by l_{min} and l_{max} respectively, *all* root/leaf combinations of *all* data propagation trees in $\mathcal{T}$ must be considered.

$$l_{min} = \min_{\forall \tau_{root}, \tau_{leaf} \in dpt, \forall dpt \in \mathcal{T}} (\mathrm{minLatency}(\tau_{root}, \tau_{leaf}, dpt))$$

$$l_{max} = \max_{\forall \tau_{root}, \tau_{leaf} \in dpt, \forall dpt \in \mathcal{T}} (\mathrm{maxLatency}(\tau_{root}, \tau_{leaf}, dpt))$$

Note that l_{max} is a tighter estimate for the longest possible end-to-end latency than the previously introduced WCL, since it takes into account the actual possible data propagation between tasks of a chain.

C. Example

We now illustrate a data propagation tree for a small example task set of three tasks τ_1 to τ_3, where $\tau_1 = \{4, 2\}$, $\tau_2 = \{8, 1\}$, and $\tau_3 = \{2, 1\}$. These tasks are part of the cause-effect chain $\zeta = \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. The resulting data propagation tree of the first three instances of τ_1 is shown in Fig. 7. The tree is essentially a graph, where the edges reflect reachability and thus convey the notion of time through the way nodes are connected from top to bottom. The placement of nodes within one hyperperiod, however, does not reflect any time information. It can be seen that the number of paths in the resulting data propagation tree is already large for a cause-effect chain of three tasks. The instances $\tau_{1,1}$ and $\tau_{1,2}$ result in 7 and 12 data propagation paths respectively. Note that these paths consider the reachability requirements introduced in this

section, and hence the number of data propagation paths is much smaller than the worst case of 18 paths which can result from one instance of τ_i, as computed by the formula presented in Section III-C. It is also visible that an instance of the last task in the chain can possibly be connected to several instances of the first task in the chain. This in turn produces a multiplicity of possible minimum and maximum latencies of the chain. It is also worth pointing out that the hyperperiod of the system is 8. This means that the job $\tau_{1,1}$ and $\tau_{1,3}$ are actually identical jobs in consecutive hyperperiods. Similarly this is the case for the jobs of the other two tasks. As expected, $l_{min} = 4$ for all data paths, while l_{max} varies between 4 and 20 for the different paths. This variation is huge and may cause that end to end latencies are violated although precedence constraints are met. This necessitates that job level dependencies must be enforced as described in the next section.

VI. Synthesizing Job-Level Dependencies

In this section, we describe an approach that analyzes tasks in a cause-effect chain and identifies jobs across which precedence constraints must be enforced to ensure that latency constraints of all cause-effect chains can be satisfied, regardless of the scheduling decisions. Specifically, the periodic task set Γ is augmented with job-level dependencies in such a way that the end-to-end latency requirements, specified for the cause-effect chains Π, are met. Hence, all *invalid* paths of the data propagation tree which do not yield an end-to-end latency within the timing requirements must be pruned by inserting job-level dependencies. The problem of adding job-level dependencies is exponential with the number of jobs in one hyerperiod, which justifies the use of a heuristic to arrive at a solution in reasonable time.

A. Pruning Branches of the Data Propagation Tree

As discussed in the previous section, a specific data value can propagate through the cause-effect chain via several possible paths. Furthermore, a node in the tree may propagate its data through valid or invalid timing paths, depending on the concrete scheduling decisions. In this section we discuss how to prune branches of a data propagation tree in such a way that all remaining data paths yield valid end-to-end latencies. The premise is that adding a job level dependency constrains the freedom of scheduling while also avoiding the possibility that the data is propagated to successor jobs which lead to invalid end-to-end latencies.

It is important to remember that a data propagation tree exists for each first instance of a task. Each node of the tree may have an edge to several nodes of the successor task but can *only have one edge* to its predecessor task. A heuristic can be defined to prune invalid branches of the data propagation tree (i.e. branches leading to data paths with a latency larger than the chains timing constraint).

1) Find the first leaf node $\tau_{j,l}$ with the smallest release time which violates the timing requirements, implying it belongs to an invalid path.
2) Find the first parent, say $\tau_{i,k}$ of the identified leaf node $\tau_{j,l}$ that has sibling nodes that lead to paths *with valid and invalid* end-to-end latencies respectively.
3) Enforce a job level dependency between the job $\tau_{i,k+1}$ and $\tau_{j,l}$.

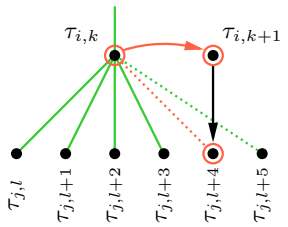


Fig. 8: Adding a job-level dependency to prune invalid data paths.

It is more efficient to prune branches higher up in the tree since entire infeasible subtrees can be eliminated when compared to pruning branches at the lower levels (i.e. at only those nodes which yield only invalid paths).

Lemma 2. *A job-level dependency can be used to prune branches of the data propagation tree.*

Proof: Assume an edge of the data propagation tree goes from $\tau_{i,k}$ to $\tau_{j,l}$. In order to prune this branch, $\tau_{j,l}$ must be prevented from reading data of $\tau_{i,k}$. Specifying a job-level dependency $\tau_i \xrightarrow{(k+1,l)} \tau_j$ forces job $\tau_{i,k+1}$ to execute before job $\tau_{j,l}$ which in turn overwrites the output data of job $\tau_{i,k}$, and thus successfully prunes the branch in the data propagation tree. ■

The indexes of the involved jobs in the job level dependency must be selected based on the hyperperiod of the involved jobs and the relative offset of the jobs to the start of the respective hyperperiod.

This principle is depicted in Fig. 8. Instance $\tau_{j,l+4}$ is the first leaf instance with invalid timing requirements (depicted by the dotted line). From here, the algorithm traces back to job $\tau_{i,k}$. Since $\tau_{i,k}$ has both valid and invalid sibling paths the next instance $\tau_{i,k+1}$ is selected for a job-level dependency with the initial job $\tau_{j,l+4}$. Such a job-level dependency will effectively prune the branches affected by timing violations.

B. Adding Job-level Dependencies for one Cause-Effect Chain

The algorithm to augment the system with job-level dependencies in order to satisfy the constraints of a single cause-effect chain is described in Algorithm 3. The central idea of the algorithm is that job-level dependencies are added to prune the invalid paths as long as they exist in the specific data propagation tree (line 6-9).

The algorithm takes the task set Γ, the cause-effect chain ζ, and the set of already specified job-level dependencies Ψ as input values. The algorithm begins by assigning the root task of ζ to τ_{root}, in line 2. The main body of the function is then executed for each of the initial jobs of the chain, where an initial job is a job of τ_{root} which is released in the first hyperperiod (line 3).

As a first step, a set of jobs $\mathcal{J}$ of the corresponding task set Γ is generated which also factors in the specified job-level dependencies Ψ (line 7) (i.e the read- and data-intervals of the jobs are adjusted as described in Section IV). In order to account for all possible cases, this job set contains jobs up to the worst case latency of the chain, starting from

the last possible root job of the hyperperiod. Then among all the paths generated, the first (minimum) invalid path of the data propagation tree is identified by the function $\mathrm{getMinimumInvalidPath}(\tau_{root,a}, \zeta)$(line 7). where a *minimum invalid path*, is defined as the first path which invalidates the timing requirements.

If such a path exists, a job-level dependency is added by the function $\mathrm{addDependency}(path, \Psi)$ in order to prune the invalid branch (line 9). A job level dependency is added as per the steps described in Section VI-A. It is important to note that function $\mathrm{addDependency}()$ adds a job-level dependency between two tasks only if *no other job-level dependency between them is already specified* in Ψ. In case there is already a dependency between tasks of the two levels, the function adds a dependency one level higher (and closer) to the root of the data propagation tree. If there is already a job-level dependency specified between all tasks of the cause-effect chain the function returns with $false$, indicating that no dependency was added. If this was the case the algorithm returns "*unsuccessful*"[1] (line 12). In all other cases $\mathrm{addDependency}()$ returns "*true*", and the algorithm updates l_{max} (line 10) before continuing in the while loop until all branches of this data dependency tree lead to valid end-to-end latencies, and returns "*success*". Note that for each iteration the job-set $\mathcal{J}$ must be regenerated or updated to reflect the newly added job-level dependency.

It is important to note that the algorithm does not construct the full data propagation tree in order to find minimum invalid paths and the principles described in Section V-1 can be applied to find the total maximum path. Traversing and constructing the data propagation tree backwards from the maximal path until a valid path is found reveals the least invalid path (this is done in a depth-first manner).

Algorithm 3: synthesizeDependencies(Γ, ζ, Ψ)

```
1  begin
2      τ_root = getRootTask(ζ);
3      for ∀τ_{root,a} , a ∈ LCM(ζ)/T_root do
4          l_max = WCL;
5          dep = true;
6          while l_max > deadline ∧ dep == true do
7              J = generateJobs(Γ, Ψ)
               path = getMinimumInvalidPath(τ_{root,a}, ζ);
8              if path exists then
9                  dep = addDependency(path, Ψ);
10             l_max = getMaxLatency(ζ, Γ, Ψ);
11         if l_max > deadline then
12             return unsuccessful;
13     return success;
```

C. Generating Dependencies for the Complete System

As described earlier, a system typically consists of multiple linear cause-effect chains which may share common tasks. In this section the approach presented above, is extended to find job-level dependencies at a system level, allowing *all* cause-effect chains to become schedulable within their end-to-end latency constraints. The task-set Γ, the set of cause-effect chains Π, and the job-level dependencies Ψ are provided as input to the algorithm. Note that at this point there are no job-level dependencies specified.

For deciding the order of analyzing cause-effect chains, the algorithm exploits the observation that longer chains (consisting of higher number of tasks) are prone to latency violations and are better candidates to be selected first for pruning. This is because longer chains lead to a widespread number of data paths within the data propagation tree, including many paths which potentially can be pruned to meet the end-to-end latency requirements. Consequently the algorithm first sorts all cause-effect chains in Π by their chain-length.

The heuristic then utilizes the previously explained Algorithm 3 to generate the job-level dependencies for the individual cause-effect chains specified for the system. If not all invalid branches of a cause-effect chain can be pruned, the algorithm is unsuccessful. However in the case when job-level dependencies could be added such that all cause-effect chains have worst-case latencies less than their timing constraints, the algorithm is deemed successful.

VII. Evaluation

This section presents an evaluation of our proposed heuristic. A set of synthetic data sets is first used to evaluate the general characteristics of the heuristic. This is followed by considering an industrial use case to demonstrate the impact of the proposed approach in the overall design process.

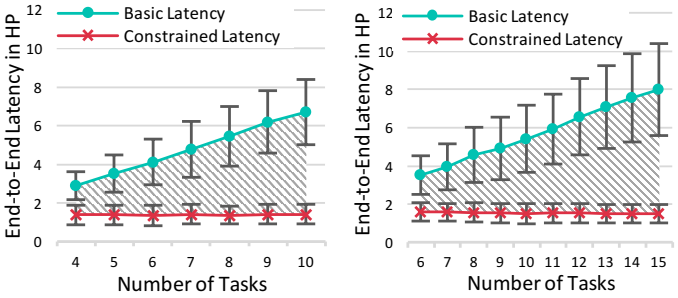
A. Heuristic Characterization with Synthetic Data Sets

1) Experimental Setup: For these experiments, cause-effect chains are randomly generated while conforming with the detailed automotive benchmarks characterization [6]. Task periods are selected with uniform distribution, out of the periods found in automotive applications $[1, 2, 5, 10, 20, 50, 100, 200, 1000]$ms. As per [6], cause-effect chains are generated to include tasks of either 1, 2, or 3 different period wherein tasks of the same period can appear 2 to 5 times, where no cyclic transitions between periods in one chain are allowed. The communication matrix, presented in [6] is referenced to allow communication across tasks with specified periods in a cause-effect chain. The individual task utilizations are generated based on UUniFast [17]. Note that for *each data point* in the graph *500 random cause-effect chains* are examined.

2) Impact on End-to-End Latency: This experiment investigates the capability of our proposed approach to limit the worst-case end-to-end latency using job-level dependencies. For a cause-effect chain, the maximum latency is computed and an age constraint equal to this maximum latency is then imposed on the system. The latency is then gradually reduced, further constraining the system, until the heuristic *cannot* find a valid placement for job-level dependencies such that the constraint is met. This is done for cause-effect chains with 2 different and 3 different periods as reported in the two sets of experiments in Fig. 9. Note that all presented values are normalized in respect to the chains hyperperiod

[1]Note that the result "*unsuccessful*" does not imply that no valid placement of job-level dependencies exists. A system designer can manually improve the generated dependencies and still achieve the desired age constraint.

(HP) in order to allow for comparison of chains with various hyperperiods. For both experiments, different chains of lengths varying between 4 to 10 tasks and 6 to 15 tasks are generated respectively for the experiments with 2 and 3 involved periods.



(a) Chains with 2 involved periods. (b) Chains with 3 involved periods.

Fig. 9: Comparison of the worst case end-to-end latency in unconstrained systems (Basic Latency) to the minimum latency achieved by the proposed heuristic (Constrained Latency) for chains of different length.

For unconstrained systems, i.e. systems *without* job-level dependencies, the experiment shows that worst case end-to-end latencies increase linearly with the number of tasks in a cause-effect chain. On the other hand, the end-to-end latencies for systems with smallest age constraint (i.e. before the proposed heuristic cannot find a solution anymore) are constant. These observations hold for both experiments. It can also be observed that the standard deviation increases with the number of tasks for unconstrained systems while it stays constant for the curve showing the systems with imposed job-level dependencies. For each age constraint between the two curves a valid placement of job-level dependencies could be found by the proposed approach, while not over-restricting the system. Thus, the end-to-end latencies in the gray shaded area become available for system designer without a need for further knowledge of the underlying scheduling policies.

B. Case Study: Air Intake System (AIS)

We demonstrate the applicability of our proposed heuristic on an AIS which is a part of the Engine Management System (EMS) and this case study is adapted from the results presented in [18]. For a car to run smoothly, it needs the proper mixture of air and fuel and this amount of air in the engine is controlled by the AIS. Specifically, a throttle body that is a part of the AIS, controls the amount of air that gets into the engine.

The AIS depicted in Fig.10, calculates the desired throttle position (TO1) depending on the accelerator pedal position (PI1, PI2) and the current throttle position (TI1, TI2). Since this subsystem is part of a x-by-wire technology, both sensors need to be duplicated to reach the required system safety. The system contains 6 individual tasks where 4 different end-to-end latency paths can be identified from each of the input values, PI1, PI2, TI1, TI2 to the output value TO1. Since the age constraint depends on the point in time when the sensors are sampled and not on the point in time when the sensor value changes, the chain PI1 $\rightarrow$ TO1 and PI2 $\rightarrow$ TO1 are equivalent. The same reasoning applies for the two paths starting from the throttle sensors. Hence, it is only required to examine the path from PI1 to TO1, representing cause-effect chain ζ_1, and the path from TI1 to TO1, representing cause-effect chain ζ_2.

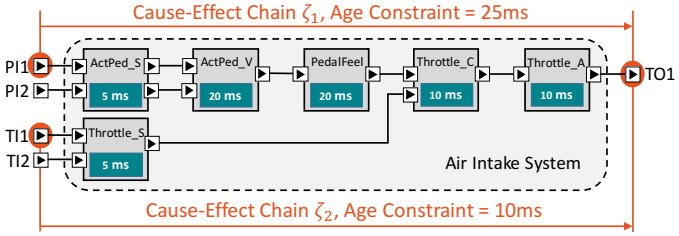


Fig. 10: Component view of the AIS of an Engine Management System.

Table I describes the timing properties of the individual tasks included in the AIS. All tasks have implicit deadlines, i.e. the deadline is equal to the task period. The age constraint for the chain ζ_1 is set to 25 ms and the age constraint for the chain ζ_2 is set to 10 ms.

1) Analysis of Latencies using the Data Propagation Tree: In this section, we examine the possible end-to-end latencies without any knowledge of the applied scheduling decisions. Hence we apply the methods to generate and analyze the data propagation trees for all initial jobs of the chain. The hyperperiod of the cause-effect chain ζ_1 is 20 ms, hence the first four jobs of the task *ActPed_S* must be examined. This leads to a total of 76 possible data propagation paths with a minimum latency of 694 µs and a maximum latency of 75 ms, as calculated by the method in Section V. The complete data propagation tree is shown in Fig. 11a. The second cause-effect chain ζ_2 has a hyperperiod of 10 ms which reduces the number of initial jobs of *Throttle_S* to 2, leading to only 6 data propagation paths as shown in Fig 11c. The calculated minimum latency of this chain is 405 µs, while the maximum latency is at 25 ms. The worst-case latency for both chains exceeds the specified age constraints.

2) The State-of-Practice Solution: In this section we examine the schedule generated by Rubus-ICE [19], a state-of-practice tool commonly found in the automotive industry. The tool may apply several optimization techniques beyond the basic scheduling algorithms and therefore the placement of jobs in the schedule might vary depending on the applied optimization.

For this case-study, the complete schedule of the EMS software presented in [18] was generated with a total of 16 tasks and a system utilization of 41%. As per the tool, the end-to-end latencies for ζ_1 results to 30.273 ms and for ζ_2 to 11.279 ms. The computation is based on the analysis presented in [11]. The end-to-end latencies for both chains exceed the specified age constraint.

3) Generating Job-level Dependencies: In order to fulfill the specified age constraints, the system model is augmented with job-level dependencies. This is done for both chains, using the methods described in this paper. The proposed heuristic

TABLE I: Timing properties of the tasks in the Air Intake System.

	ActPed_S	Throttle_S	ActPed_V	PedalFeel	Throttle_C	Throttle_A
Period	5 ms	5 ms	20 ms	20 ms	10 ms	10 ms
WCET	96 µs	131 µs	186 µs	138 µs	97 µs	177 µs

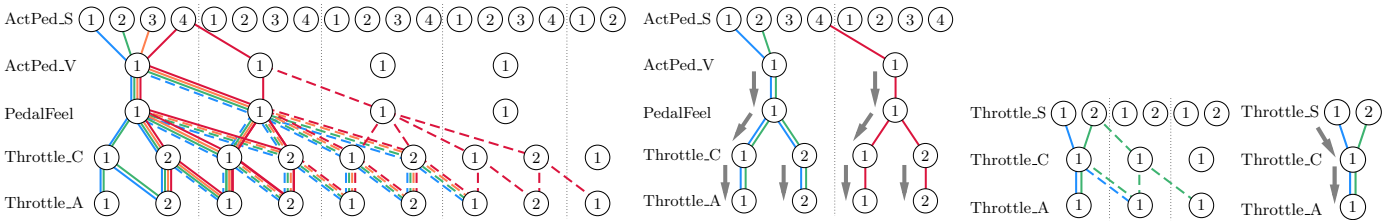


Fig. 11: Data propagation trees for the two cause-effect chains with and without generated job-level dependencies. The cause-effect chain ζ_1 is shown in (a). Data paths leading to latencies larger than the age constraint are shown in dashed lines. (b) shows the same data propagation tree with generated job-level dependencies. Similarly the cause-effect chain ζ_2 is shown in (c) and (d). Generated job-level dependencies are shown with gray arrows.

successfully finds a placement of job-level dependencies to fulfill the requirements. Both data propagation trees with generated job-level dependencies are shown in Fig. 11b and 11d. On applying the dependencies, the worst case end-to-end latency results to $25\,\mathrm{ms}$ and $10\,\mathrm{ms}$ for the cause-effect chains ζ_1 and ζ_2 respectively. Thus the specified age constraints are met while schedulable data paths are not removed when not necessary.

VIII. Conclusion

This paper addresses multi-rate cause-effect chains with specified age-constraints. For such systems, it is not only important that the individual tasks execute within their deadlines but also that the end-to-end latency requirements of cause-effect chains are met. The contribution of this paper is twofold. A novel method of analysis is proposed to identify and calculate the minimum and maximum latencies of cause-effect chains for systems with specified job-level dependencies. As a second contribution, a heuristic method is presented to augment a task set with job-level dependencies such that all specified age constraints are satisfied. Using job-level dependencies, it is possible to restrict the data propagation between consecutive tasks of a cause-effect chain [2], [3], [4], [5] which can lead to smaller maximal end-to-end latencies, when placed correctly. The problem complexity however makes this a challenging task for the system engineer. Synthesizing the job-level dependencies for a given system such that all cause-effect chains meet their timing requirements can thus improve the development process. The proposed methods are evaluated based on synthetic experiments as well as an industrial case study where we compare against a state-of-practice solution. The experiments show that the proposed methods in this paper can greatly improve the development process already during earlier stages of the system design.

In the future we plan to extend our work to incorporate other end-to-end latency constraints defined in the AUTOSAR standard [10].

Acknowledgment

The work presented in this paper is supported by the Swedish Knowledge Foundation (KKS) through the projects PREMISE and DPAC; and the Swedish Foundation for Strategic Research (SSF) through the projects PRESS.

References

[1] 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 *Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008.

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

[3] 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.

[4] W. Puffitsch, E. Noulard, and C. Pagetti, "Mapping a multi-rate synchronous language to a many-core processor," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013, pp. 293–302.

[5] ——, "Off-line mapping of multi-rate dependent task sets to many-core platforms," *Real-Time Systems*, vol. 51, no. 5, pp. 526–565, 2015.

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

[7] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A real-time architecture design language for multi-rate embedded control systems," in *ACM Symposium on Applied Computing*, 2010, pp. 527–534.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[9] *EAST-ADL - Domain Model Specification*, EAST-ADL Association Std. V2.1.12, 2014.

[10] *AUTOSAR - Spec. of Timing Extensions*, AUTOSAR Std. 4.2.2, 2014.

[11] 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.

[12] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, and F. J. Cazorla, "Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2014, pp. 29:1–29:10.

[13] H. R. Faragardi, B. Lisper, K. Sandstrom, and T. Nolte, "A communication-aware solution framework for mapping autosar runnables on multi-core systems," in *IEEE International Conference on Emerging Technology and Factory Automation*, 2014, pp. 1–9.

[14] 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 *44th ACM/IEEE Design Automation Conference*, 2007, pp. 278–283.

[15] S. Schliecker and R. Ernst, "A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems," in *7th International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 433–442.

[16] *AUTOSAR - Specification of RTE*, AUTOSAR Std. 4.2.2, 2014.

[17] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems Journal*, vol. 30, no. 1-2, pp. 129–154, 2005.

[18] P. Frey, "Ulmer Informatik Berichte Nr 2010-03 - Case Study: Engine Control Application," University Ulm, Tech. Rep., 2010.

[19] Arcticus Systems, "Rubus ICE," [Online] https://www.arcticus-systems.com/products/, last visited 16.05.2016.