ABSTRACT

Safety and control functions of modern automotive systems are implemented as chains of periodic data producer/consumer tasks run at different rates. To simplify the development of such systems, automotive standards relax synchronization requirements between tasks, allowing a task to run even if its dependent tasks are inactive. This gives birth to more complex timing constraints such as data age, which specifies the maximum duration of time that input data of a task chain is still allowed to affect the output of the chain.

We develop a technique to compute lower and upper bounds on the data age of multi-rate task chains that execute upon a heterogeneous computing platform using a job-level fixed-priority scheduling policy. To the best of our knowledge, we are the first to consider uncertainties in the timing parameters (namely, both the release jitter and execution-time variation) of the tasks. Such an assumption makes the problem more challenging as it increases the number of possible schedules that the system may encounter during its lifespan. We represent these uncertainties by timing intervals and devise an analysis that uses those intervals to explore possible dependencies between jobs. We incorporated various pruning rules to make the analysis much faster and far less pessimistic than the state of the art. Our evaluations on an industrial case study as well as synthetic task sets show that our analysis reduces the overestimation of the data age by 36% on average (and up to 42%) in comparison to the state of the art when the number of tasks varies from 10 to 50.

1 INTRODUCTION

In modern automotive systems, a large number of safety and control functionalities are implemented in the form of data-dependent periodic tasks that interact with their environment at different rates [3, 4, 21]. These systems have a large number of data types that are being updated or used by tasks (or runnables) across sub-systems designed by various parties.

To allow independent development, enhance compatibility with existing or legacy software, simplify distribution on large heterogeneous computing platforms, and satisfy reliability requirements when a subsystem fails, automotive engineering standards relax the synchronization between data-dependent tasks [1, 8]. The relaxation is technically achieved by allowing tasks to read the most recent data from shared buffer spaces (e.g., global variables) and write their output to these buffers independently of other tasks [10, 27]. Examples of this semantic can be found in the implicit communication model of AUTOSAR [1] and the publisher-subscriber model in robot operating system (ROS) [34].

However, this relaxed data synchronization model comes at a cost. In addition to traditional deadline constraints, it imposes new types of end-to-end latency constraints related to the propagation of data through sequences of tasks that together fulfill a particular system function. Such sequences of tasks are called task chains or cause-effect chains [1, 2, 25]. One of the key classes of end-to-end latency requirements is data age (DA) constraints. A DA constraint specifies the maximum amount of time that input data of a task chain is still allowed to have an effect on the output of this chain [6, 15, 27]. In other words, it specifies that the input data used throughout a chain of tasks should not be ‘too old’. Otherwise, it would not accurately represent the current state of the environment.

The goal of our work is to derive relatively tight lower and upper bounds on the data age of data-dependent tasks. The problem is challenging because (i) not all input data is propagated through a task chain as consumers and producers have different rates, and (ii) data age depends on the order of data-related events (e.g., read and write), which is influenced by the execution-time variations and release jitter of the tasks. In order to obtain such bounds, all possible schedules that can be observed for a task set must be considered.

Fig. 1 shows a subset of schedules of a task chain with three data-dependent tasks, $\tau_1$, $\tau_2$, and $\tau_3$, generated by a non-preemptive fixed-priority scheduling policy (in which $\tau_1$ has the highest and $\tau_3$ the lowest priority) deployed on one processing element. These tasks release jobs periodically (starting from time 0). As it can be seen, the execution-time variations of each task could impact the start time of the other tasks, resulting in several possible schedules.

Considering that all jobs run until their worst-case execution time, the first response to the data produced by the first job of $\tau_1$ is generated at the latest by time 4, whereas the first response to the
third job of $\tau_1$ which releases at time 6 is available at the latest by time 12. Note that input data that is not propagated through a task chain does not impact the output of a task chain. This is the case for the data of the second and fourth jobs of $\tau_2$ as they are not processed by task $\tau_2$. For the schedules shown in this example, the data age is 7 because the longest time that a job of $\tau_1$ can affect the output (a job of $\tau_2$) is the time between the release of the first job of $\tau_1$ and the completion time of the second job of $\tau_3$.

To support the growing number (and computation demands) of tasks and to satisfy the data-age requirements, the automotive industry is moving toward adopting heterogeneous systems-on-chip platforms. However, to provide isolation between the tasks and follow standards, each task is assigned statically to a processing environment [21]. Such a design choice in turn adds to the complexity of deriving bounds of data age as it allows data producer and data consumer tasks to run concurrently.

Related work. Feiertag et al. [15] are among the first to provide a formal definition for different end-to-end latency semantics (reaction latency, data age, etc.). Additionally, they introduce a method to calculate data age for a given schedule, however, their method does not account for a collection of schedules resulted from the timing variations in the tasks’ release and execution time.

Some works in the literature, try to improve end-to-end latencies by scheduling algorithms [10, 11], or enforcing control-flow dependencies by adjusting effective parameters such as periods, priorities, and offsets to minimize end-to-end latency [12, 23, 39]. Pagetti et al. [33] introduce a framework using PRELUDE language [16], which is a synchronous language based on the principles of LUSTRE [20] for multi-rate real-time systems, to (i) guarantee the synchronization of multi-rate tasks and (ii) provide deterministic dependency between jobs. However, we focus on deriving bounds of data age for systems where it is not possible or advisable to change the scheduling policy or the task execution semantic (e.g., due to requirements enforced by standards).

Various types of end-to-end latency analyses have been addressed in the literature. For example, reaction latency, i.e., the longest delay between an input and the first output of a task chain that responds to that input has been analyzed by [2, 13, 19, 24, 25, 28, 35]. Reaction latency is different from data age in that reaction latency takes into account the time it takes for the next new output to appear, in addition to the data age. As a result, it can only provide an upper bound on data age [13]. In our experiments, we will show that this upper bound is indeed pessimistic (loose).

Becker et al. [6] introduce a schedule-agnostic data age analysis, where apart from the fact that the tasks must be schedulable (respect their deadline), no assumption about the scheduling policy being used is taken into consideration when obtaining data age. Because the analysis does not account the scheduling policy and the response time of the tasks, it is inherently pessimistic, as we will demonstrate in our experiments. Later, Becker et al. [7, 8] extended their previous work in [6] and proposed a method to compute the lower and upper bounds of data age subject to the availability of certain levels of knowledge about the system, such as knowing the response times or the exact schedule. Nevertheless, when the exact schedule is unknown, this work only can exploit the worst-case response time of jobs in order to calculate a more accurate data age and since it assumes the start time intervals of jobs are unknown, the calculated data age could be pessimistic.

Davare et al. [12] provide an upper bound on end-to-end latency by summing up the worst-case response times plus the periods of every task in the task chain. This provides an upper bound on reaction latency or data age. However, Dürre et al. [13] later on showed that this bound is very pessimistic. They also proposed an analysis to derive an upper bound of reaction latency and data age for sporadic tasks in distributed systems. However, when applied to periodic tasks, this work results in pessimistic bounds for the reaction latency and data age. Günzel et al. [19] propose an analysis to derive an upper bound of reaction latency and data age for periodic or sporadic task chains in globally asynchronous distributed systems. This paper supposes that tasks always execute with their worst-case execution time. This allows the paper to simulate the schedule of a few hyperperiods to obtain the bounds on data age. This method, however, would not work when tasks have release jitter or execution-time variations.

This paper. We provide a method to compute lower and upper bounds on the data age of multi-rate task chains that follow the read-execute-write semantics with an implicit communication model as suggested in the AUTOSAR standard [1, 27]. We consider a general system setup, where a set of periodic tasks is scheduled by a non-preemptive job-level fixed-priority (JLFP) scheduling policy, a general class of scheduling policies that includes fixed priority (FP) and the earliest-deadline first (EDF), deployed on a heterogeneous computing platform, where each task is statically assigned to a processing element. We are the first to consider the impact of timing uncertainties on task parameters, in particular, the release jitter (as well as execution-time variation).

The uncertainty in timing parameters of the tasks results in having many possible schedules. We combine these schedules and represent their effect on each job of a task set (within an observation window) using two uncertainty intervals; one for the start time and one for the completion time of the job. We use the schedule-abstraction graph analysis of Nasri et al. [29, 36, 37] to obtain tight bounds on the start and completion intervals of the jobs. Our analysis utilizes these bounds to obtain the possible set of data producers for every data consumer in a task chain until we find the earliest and latest possible data producers of the entire chain. To make this process more efficient, we propose pruning rules to ignore data dependencies that have no impact on data age.

Our results on an industrial case study [22, 40] and on synthetic task sets show that our method is able to reduce the pessimism of
the existing data-age analyses by 30% to 40% on average over all experiments, considering the impact of system utilization, number of tasks, number of task chains, and number of processing elements.

2 SYSTEM MODEL AND DEFINITIONS

We consider standard automotive applications [22, 26] which typically include a set of periodic tasks (or runnables) executing on a heterogeneous processing platform with m processing elements (PE) named P = {φ1, ..., φm}. Each task is statically assigned to a single PE on the platform and thus is not allowed to migrate. A PE can refer to a processor core or an accelerator that is embedded on a chip (such as NVIDIA TX2 or NVIDIA AGX Xavier). We assume that tasks are scheduled according to a job-level fixed-priority (JLFP) scheduling policy. JLFP includes commonly used policies such as the fixed-priority and EDF scheduling policies.

Task model. We consider a system that contains a set \( \Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\} \) of n data-dependent periodic tasks (or runnables), where each task follows the read–execute–write semantics specified by AUTOSAR [1] 1. A task \( \tau_j \in \Gamma \) is a tuple \((m_j, [c_{\text{min}}^j, c_{\text{max}}^j], T_j, D_j, X_j)\), where \( m_j \) denotes the maximum release jitter, \( T_j \) is the period, \( D_j \) is the relative deadline (where \( D_j \leq T_j \)), and \( X_j \in P \) is the PE on which \( \tau_j \) execute. The release jitter could be the result of interrupt latencies or timer inaccuracies [29]. Interval \([c_{\text{min}}^j, c_{\text{max}}^j] \) models the range of possible execution times of task \( \tau_j \), where \( c_{\text{min}}^j \) and \( c_{\text{max}}^j \) represent the best-case (BCET) and worst-case (WCET) execution times, respectively, when it is executed on the processing element \( X_j \). The execution-time variations can be due to cache behavior, input data variations, or diversity of program paths. The timing parameters can be discrete or continuous. For a set of periodic tasks that arrive at the time \( t = 0 \) with a constrained-deadline and without release offset, the hyperperiod is the shortest time interval after which the arrival pattern of the jobs repeat. The hyperperiod is the least common multiple of the periods (i.e., \( H = \text{lcm}(T_1, T_2, \ldots, T_n) \) [9].

We assume a run-to-completion semantics for task execution. There will be a short discussion on how our analysis is applicable to preemptive task sets in Sec. 3.5.

Job model. Each task \( \tau_j \in \Gamma \) releases jobs periodically with period \( T_j \) from the starting moment of the system, which we assume to be at time \( t = 0 \). The \( k^{th} (1 \leq k) \) job of a task \( \tau_j \) is released non-deterministically within the interval \([r_{\text{min}}^j, r_{\text{max}}^j] \), where \( r_{\text{min}}^j = (k - 1) \cdot T_j \) is the earliest release time (a.k.a. arrival time [5]) and \( r_{\text{max}}^j = r_{\text{min}}^j + \sigma_j \) is the latest release time of the job. Following Audsley’s model, the absolute deadline of the job is at \((k - 1) \cdot T_j + D_j \).

We form a finite job set \( \mathcal{J} = \{J_1, J_2, \ldots, J_n\} \) consisting of all the jobs released by the task set \( \Gamma \) during the interval \([0, OW) \), where \( z = |\mathcal{J}| \) and \( OW \) is an observation window which is an upper bound on two values: (i) the shortest interval of time after which the release pattern of the tasks repeats and all shared data buffers are filled with a data generated by a data-producer task (i.e., 2H, where \( H \) is the hyperperiod of the task set), and (ii) the worst-case value of the data age among all task chains. We will discuss this further in Sec. 2.2 after we formally define data age.

A job \( J_i \in \mathcal{J} \) is a tuple \((r_{\text{min}}^i, r_{\text{max}}^i), [c_{\text{min}}^i, c_{\text{max}}^i], d_i, p_i, X_i)\), where \( r_{\text{min}}^i \) and \( r_{\text{max}}^i \) are the earliest and latest release times, \( d_i \) is the absolute deadline, \( X_i \) is the processing element, and \( p_i \) is the priority of the job (assigned by the JLFP scheduling policy). We assume a smaller value of \( p_i \) represents a higher priority, and ties are broken arbitrarily but consistently. Other job parameters (i.e., \([c_{\text{min}}^i, c_{\text{max}}^i], X_i\) are inherited from its task. Function Task(J_i) maps jobs to their corresponding tasks, i.e., Task(J_i) = \( \tau_j \) implies that \( J_i \) is a job of task \( \tau_j \) and function PE(J_i) maps jobs to their assigned processing element \( X_i)\).

We use angle brackets to denote sequences and curly brackets to denote sets. For set \( X \), we let \( X^\ast \) denote the positive Kleene closure, i.e., the set of all non-empty sequences of elements of \( X \).

Data model. We assume that the tasks follow the read–execute–write semantics according to the implicit communication model of AUTOSAR [1, 17], namely, a task reads all its required inputs as soon as it starts and stores a local copy of them into its memory. The task writes its outputs back to shared buffers when finishing its execution. The inter-task data transfer is done via global shared buffer. The time required to safely access these shared buffers have been accounted for in the tasks’ WCET. Note that the implicit communication model does not enforce a synchronization between data consumers and producers.

Data propagation model. We assume the application to be modeled as a data propagation graph (DPG). This is a directed acyclic task graph (DAG) in which vertices represent tasks and edges represent their data dependencies. Namely, an edge from task \( \tau \) to task \( \tau' \) implies that some of the outputs of \( \tau \) are used by \( \tau' \). In this case, \( \tau \) is called a data producer for \( \tau' \) and \( \tau' \) is called a data consumer of \( \tau \).

---

1In this work, we assume that each task consists of one elementary software module or runnable. Thus, our task model is consistent with the implicit model of AUTOSAR [1] as well as the task model introducted in the RTSS'21 industrial challenge [27].
We use \( \text{Pred}(\tau) \) to refer to the set of data producers for task \( \tau \), also called the predecessors of \( \tau \).

Fig. 2 shows an example of a DPG in an autonomous driving application (from the WATERS’2019 challenge [22]). This DPG shows an advanced driver-assistance system (ADAS), covering the entire computation path, from sensor input to steering command. In this example, different sensor systems (LiDAR, cameras, and GPS) collect data according to their update rate. These data are then propagated to tasks that periodically perform operations such as localization, detection, and prediction, thereby computing the trajectory of the vehicle. The data-propagation paths in the DPG that perform a certain system functionality are called task chains.

**Definition 2.1 (Task chain (a.k.a. cause-effect chain)).** A sequence of data-dependent tasks \( \alpha \subseteq \Gamma^* \) is a task chain only if for every two consecutive tasks \( \tau \) in \( \alpha \), \( \tau \in \text{Pred}(\tau') \).

We denote the set of all given task chains in a DPG with \( \mathcal{A} \). For every \( \alpha \in \mathcal{A} \), we let \( |\alpha| \) denote the number of tasks in \( \alpha \). For brevity, we let \( \alpha_k \) (1 \( \leq k \leq |\alpha| \)) denote the \( k \)th task of the task chain \( \alpha \). The example in Fig. 2 contains four task chains defined by the user: \( \mathcal{A} = \{ (\tau_0, \tau_4, \tau_5), (\tau_1, \tau_3, \tau_7, \tau_8), (\tau_2, \tau_3, \tau_7, \tau_8), (\tau_0, \tau_4, \tau_5, \tau_8) \} \) (taken from [40]).

Further, we consider that \( \text{Pred}(\tau, \alpha) \) returns the immediate predecessor of \( \tau \) in chain \( \alpha \) and is defined as follows:

\[
\text{Pred}(\tau, \alpha) = \{ \tau' \mid \exists n, 1 \leq n \leq |\alpha|, \tau' = \alpha_n \land \tau' \in \text{Pred}(\tau) \}. \tag{1}
\]

\( \text{Eq. (1)} \) returns an empty set when \( \tau \) is the first task of the chain.

**Schedule.** A schedule \( S \) is an assignment of given jobs \( J \) to time intervals. More formally, a schedule \( S = \{ (J_1, [s_1, f_1]), \ldots, (J_e, [s_e, f_e]) \} \) is a set of tuples where each tuple determines the start time \( s_i \) and finish time \( f_i \) of job \( J_i \) [9]. In a feasible schedule, every job starts no earlier than its earliest release time and completes no later than its deadline [9].

**Bounds on the start and finish times of the jobs.** Our analysis to derive data age for each task chain requires knowledge about all possible start times and finish times of a job for all feasible schedules. We model the set of these start and finish times each by an interval, denoted by \( \text{ST}(J) = [\text{EST}(J), \text{LST}(J)] \) and \( \text{FT}(J) = [\text{EFT}(J), \text{LFT}(J)] \), respectively, where \( \text{EST}(J) \) and \( \text{LST}(J) \) denote the earliest and latest start times and \( \text{EFT}(J) \) and \( \text{LFT}(J) \) denote the earliest and latest finish times of a job \( J \in J \), respectively.

Obtaining the earliest and latest start times and finish times of a job is challenging when tasks are subject to release jitter and execution-time variations because these timing variations result in having more than one possible schedule (ordering between the execution of the jobs). However, exploiting the recent class of reachability-based response-time analysis techniques, known generally as the schedule-abstraction graphs (SAG) [29–31, 36, 37], it is possible to calculate a tight bound for these values for any possible schedule a system may encounter over the course of its lifetime.

### 2.1 Data Age for a Single Schedule

As the tasks release jobs periodically, task chains can be instantiated multiple times resulting in so-called chain instances. These are formally defined as follows.

**Definition 2.2 (Chain Instance).** A partial order of jobs \( \beta \) that appear in a schedule \( S \) is a chain instance iff it satisfies the following conditions:

(i) There exists a task chain \( \alpha \) that has the same task sequence as \( \beta \). More formally:

\[
\exists \alpha \in \mathcal{A}, \ (|\alpha| = |\beta|) \land (\forall k, 1 \leq k \leq |\alpha|, \text{Task}(\beta_k) = \alpha_k), \tag{2}
\]

where the function \( \beta_k \) returns the \( k \)th job in the chain instance \( \beta \).

(ii) For every two consecutive jobs \( J_i \) and \( J_p \) in the job chain \( \beta \) from the schedule \( S \), we must have:

\[
\beta J_k \in J, \ (\text{Task}(J_k) = \text{Task}(J_i)) \land (f_i < f_k < s_p) \tag{3}
\]

We call the first job in the chain instance (i.e. \( \beta_1 \)) the source job and the last job in the chain instance (i.e. \( \beta|_\mathcal{J} \)) sink job of \( \beta \). We also define function \( \text{Inst}(\alpha) \) which returns the set of all chain instances of task chain \( \alpha \) in schedule \( S \).

**Delay.** Before we introduce data age, we define a delay function \( \Delta(J_p, J_c) \) for two arbitrary jobs \( J_p \) and \( J_c \), s.t. \( f_c > f_p \). It returns the difference between the finish-time of \( J_p \) and the release time of \( J_p \) that finishes after \( J_c \) in a schedule \( S \). Namely,

\[
\Delta(J_p, J_c) = f_c - f_p \tag{4}
\]

**Data age.** Data age determines for how long an input data impacts outputs of a task chain [15]. More formally, the data age of a task chain \( \alpha \) is defined as the longest delay between any source job and the latest sink job that is related to that source job via one of the chain instances in the schedule \( S \), namely,

\[
\text{DA}(\alpha) = \max \left\{ \Delta \left( \beta_1, \beta|_\mathcal{J} \right) \mid \beta \in \text{Inst}(\alpha) \right\}. \tag{5}
\]

### 2.2 A Safe Bound on the Observation Window

Our data-age analysis is based on finding the lower and upper bounds of data age for all chain instances that are found in any schedule generated by a set of jobs with timing uncertainties. Therefore, it is crucial for us to ensure that the largest and smallest values of data age will be present (somewhere) among the set of schedules generated for the jobs we consider in our observation window. We form the observation window according to the following criteria: (i) for any shared buffer accessible by a data-consumer task, there must be at least one new data in the buffer produced by a data-producer task (otherwise, we may not see the longest chain instance), (ii) after all shared buffers have been updated at least once by a data-producer task, the OW must be large enough to include all possible patterns in the arrival times of the tasks (namely, it must be larger than one hyperperiod since a hyperperiod is a tight bound on the interval of time after which the arrival pattern of the tasks repeats [9]), (iii) the largest and smallest values of data age should be observable in at least one of the schedules that are built for the jobs in the observation window, and (iv) the OW must be an integer multiple of \( H \), otherwise the schedule-abstraction graph will not give us safe bounds on the WCRT of jobs at the end of OW.

To satisfy (i), we must ensure that each data-producer task is executed at least once, hence, \( OW \geq T_{\text{max}} \), where \( T_{\text{max}} = \max(T_i \mid 1 \leq i \leq n) \). A simple upper bound on \( T_{\text{max}} \) is \( H \). To satisfy (ii), the observation window must be as large as \( H + T_{\text{max}} \) or simply \( 2H \) (to respect (iv)). Finally, to satisfy (iii), we use a safe
upper bound of data age (called the worst-case data age (WCL)) defined by Becker et al. [6] for a task chain $\alpha$, as follows:

$$WCL(\alpha) = \sum_{\tau \in \alpha} 2 \times T_{\tau}.$$  \hspace{1cm} (6)

Since $WCL(\alpha)$ might be larger than $H$, we take the maximum between $2H$ and the largest $WCL$ among all task chains:

$$OW = \max \left\{ 2H, \max_{\alpha \in \mathcal{A}} \left( \frac{WCL(\alpha)}{H} \right) \times H \right\}. \hspace{1cm} (7)$$

3 DATA-AGE ANALYSIS

Our analysis has two phases: a preparation phase (shown in Fig. 3) and an analysis phase. The preparation phase starts by obtaining the observation window according to Eq. (7). We then form the job set $\mathcal{J}$ from the jobs of the periodic tasks that could be released within the observation window (i.e., any job which earliest release time is no larger than $OW$). We extract the start- and finish-time intervals of each job in $\mathcal{J}$ utilizing schedule-abstraction graph [29, 36, 37] since it provides tight bounds on these intervals when tasks are statically assigned to processing elements. This tightness implies that for any job $J \in \mathcal{J}$ and for any time instant $t \in [EST(J), LST(J)]$, there exists a schedule in which $J$ starts at time $t$, and similarly, for any $t' \in [EFT(J), LFT(J)]$, there exists a schedule in which $J$ finishes at time $t'$.

We first illustrate the motivation and the main idea of our proposed method with an example (in Sec. 3.1), then we provide the steps of calculating the lower and upper bounds of data age for each task chain in detail (Sections 3.2 to 3.4).

3.1 Motivation and Basic Idea

Consider the task set shown in Fig. 4, scheduled by a fixed-priority non-preemptive scheduling policy on one processing element, where the priority order of the tasks is as follows: $t_1 < t_2 < t_3 < t_4$. This task set is subject to timing uncertainties (due to execution time variations of tasks $t_2$ and $t_3$). These uncertainties result in having several possible schedules (shown in Fig. 4(a) and (b)). In Fig. 4(a), all tasks are executed with their worst-case execution time. In this case, two chain instances $(J_6, J_8, J_{10})$ and $(J_5, J_6, J_{11})$ emerge in the schedule where the chain instance $(J_6, J_8, J_{11})$ determines the data age which is equal to 12. However, if task $t_1$ executes with its best-case execution time (as shown in Fig. 4(b)), then there would be two chain instances: $(J_1, J_2, J_{10})$ and $(J_5, J_6, J_{11})$. The chain instance $(J_5, J_6, J_{11})$ determines the data age which is equal to 19. This example shows that a task set may have several execution scenarios and schedules. Each of these schedules may result in a different set of chain instances and hence observe a different data age.

As we discussed before, in order to find chain instances in the schedule, we need to know the start and finish times of each job. However, as shown by the previous example, one task set can have several schedules resulting in the different start and finish times for each job. Our solution to deal with multiple schedules is to abstract the start time and finish time of a job in all possible schedules that can happen for that job with two intervals (one for all possible start times and one for all possible finish times). Fig. 4(c) shows these intervals for the given task set (derived from the SAG technique [37]).

Using these start-time and finish-time intervals, we can find potential data dependency between jobs by identifying jobs of the data producer task that possibly finish before a data consumer job without being overwritten by another job (see Sec. 3.2). By repeating the process of finding the data dependency between jobs of every two consecutive tasks in a task chain, we can form all possible chain instances in the schedules and determine the source jobs in

Figure 3: Overview of proposed data age analysis.

Figure 4: Two different scheduling scenarios of a task set $\Gamma = \{t_1, t_2, t_3, t_4\}$ with no release jitter. The execution time range for $t_1$ and $t_3$ is $[1, 1]$, for $t_2$ is $[6, 8]$, and for $t_4$ is $[1, 10]$. There is one task chain $\alpha = \{t_1, t_2, t_3\}$. (a) An execution scenario where all tasks execute with their worst-case execution time. (b) An execution scenario where task $t_1$ executes with its best-case execution time. (c) The abstraction of all possible schedules with start and finish time intervals for each job.
each chain instance. These chain instances allow us to identify all source jobs for each sink job, and hence, derive a safe bound for the data age (Sec. 3.3). Furthermore, since intermediate chain instances have no impact on the lower and upper bounds of the data age, the intermediate possible data producer jobs in each iteration that form these ineffective chain instances can be discarded (Sec. 3.4).

3.2 Data Dependency Between Jobs

When a data-consumer job has an uncertain start time, it may have multiple jobs that can produce data for it. Let $J’ \in J$ be any job of a data-consumer task $\tau’ = Task(J’)$ that is part of a task chain $\alpha$. We identify the potential data-producer jobs for $J’$ by ruling out jobs that can never produce data for $J’$ using the following facts:

**FAct 1.** The shared buffer between two consecutive tasks in the task chain keeps only the most recent data written by a job of the data-producer task.

According to Fact 1, a data-producer job that finishes just before the earliest start time of $J’$, has overwritten the previous data in the shared buffer. We denote this job by $P_{first}(J’, \alpha)$. None of the jobs of the data-producer task with a release time earlier than $P_{first}(J’, \alpha)$ can produce data that is used by $J’$. For example, in Fig. 5(a), $J_1$ cannot be a data producer for the job $J_6$ under any scenario because job $J_2$ will certainly overwrite the data in the shared buffer after $J_1$. It is important to note that in this example, $J_3$ is not the first potential data producer for $J_6$ since it is possible that $J_6$ starts its execution before $J_3$ completes. In that case, $J_6$ will read the data that is produced by $J_2$ and not $J_3$.

According to Fact 2, jobs of a data-producer task that certainly finish after the latest start time of $J’$, cannot produce data for $J’$. Equivalently, all other jobs of a data-producer task whose finish-time interval intersects with the start-time interval of $J’$ are potential data producers for $J’$. We denote these jobs by $P_{rest}(J’, \alpha)$. In the example in Fig. 5(a), $P_{rest}(J’, \alpha)$ includes $J_3$ and $J_4$ but not $J_5$ because it is impossible for $J_5$ to finish before $J_6$ starts. Note that a data producer job produces its data only at its finish time.

**Obtaining $P_{first}(J’, \alpha)$.** To identify $P_{first}(J’, \alpha)$, we consider two cases depending on whether the data producer task for $\tau’$ is assigned to another PE (Case 1), or to the same one (Case 2).

**Case 1.** When the data producer and consumer are not on the same PE, they could execute in parallel without blocking each other. In that case, $P_{first}(J’, \alpha)$ is a data-producer job $J’$ that certainly finishes before the earliest start time of $J’$ (i.e., $LFT(J’) < EST(J’)$) and satisfies the following condition:

$$Task(J’) = Pred(\tau’, \alpha) \land PE(J’) \neq PE(J’’) \land LFT(J’) < EST(J’’) \land J’’ \in J, Task(J’’’) = Pred(\tau’, \alpha) \land LFT(J’’) < LFT(J’’’). \tag{8}$$

In the example in Fig. 5(a), $T_1$ and $T_2$ are assigned to two different PEs, hence, it is possible for $J_3$ to run in parallel with $J_6$. Therefore, $J_6$ may read data from $J_2$ if $J_3$ has not yet finished its execution. That is why $P_{first}(J_6, \alpha) = \{J_2\}$.

**Case 2.** When the data producer and data consumer tasks are assigned to the same PE and run non-preemptively, a data-producer job $J’ \in J$ where $Task(J’) = Pred(\tau’, \alpha)$ could potentially ‘block’ $\tau’$ and hence forces $\tau’$ to run after its completion even though there might be an intersection between the finish-time interval of $J’$ and the start-time interval of $J’$. In this case, $P_{first}(J’, \alpha)$ is a data-producer job $J’$ that certainly starts before $J’$ (namely, $LST(J’) < EST(J’)$) and there is no other job of the data-producer task that certainly starts after $J’$ and before $J’$, namely, $J’$ must satisfy the following condition:

$$Task(J’) = Pred(\tau’, \alpha) \land PE(J’) = PE(J’’) \land LST(J’) < EST(J’’) \land \exists J’’ \in J, Task(J’’’ = Pred(\tau’, \alpha) \land LST(J’) < LST(J’’’) < EST(J’’). \tag{9}$$

Consider the example in Fig. 5(b), where $T_1$ and $T_2$ are assigned to the same PE. Since $J_3$ is the last data-producer job that certainly starts before $J_6$ and occupies the PE until it completes (recall that jobs execute non-preemptively), $J_3$ certainly rewrites the shared buffer, and consequently, the data produced by the previous jobs such as $J_2$ will be overwritten by $J_3$. Therefore, in this case, $P_{first}(J_6, \alpha) = \{J_3\}$. Combining the two cases, we have:

$$P_{first}(J’, \alpha) = \{J’ \mid J’ \in J \land J’’’ satisfies (8) or (9)\}. \tag{10}$$

**Obtaining $P_{rest}(J’, \alpha)$.** Next, we specify other potential data-producer jobs that start after $P_{first}(J’, \alpha)$ and could possibly finish before the latest start time of $J’$, denoted by $P_{rest}(J’, \alpha)$:

$$P_{rest}(J’, \alpha) = \{J’ \mid J’ \in J \land Task(J’) = Pred(\tau’, \alpha) \land EST(P_{rest}(J’, \alpha)) < EST(J’’) \land (FT(J’’) \land ST(J’)) \neq \emptyset\}. \tag{11}$$

In summary, the set of potential data-producer jobs for the job $J’$ are obtained as follow:

$$AP(J’, \alpha) = P_{first}(J’, \alpha) \cup P_{rest}(J’, \alpha). \tag{12}$$

**Lemma 3.1.** There is no job $J’ \in J \setminus AP(J’, \alpha)$ and Task($J’$) = $Pred(J’, \alpha)$ that can produce a data for $J’$ in any feasible schedule.
Data propagation when the length of task chain \( J \) (from Eq. (8) and Eq. (9)) or (ii) finishes after the latest start time of \( J \).

3.3 Finding Possible Source Jobs for a Sink Job

So far, we have discussed the data dependency between two jobs of two consecutive tasks in a task chain. As the data age is defined as the maximum delay of the source and sink jobs (i.e., jobs of the last task in the task chain) in chain instances, finding potential source jobs for each sink job is essential. In the remainder of this section, we discuss how to find source jobs for a sink job by obtaining the potential data dependency between jobs of every two consecutive tasks in a task chain. This is done in Algorithm 1, which is an iterative procedure that starts from one job of the sink task. In each iteration, Algorithm 1 determines the potential data dependency between jobs of two consecutive tasks in a task chain.

In the example in Fig. 6(a), \( \alpha \) is a task chain \( \langle \tau_1, \tau_2, \tau_3 \rangle \), where the tasks are assigned to different PEs. Fig. 6(a) shows the start and finish time intervals of each job. In order to find possible chain instances, first, we start from jobs of \( \tau_3 \) due to the fact that they can be sink jobs for chain instances. Note that if no schedule exists in which a data-consumer job \( J^c \) of a task at the end of the task chain is a part of a chain instance, \( J^c \) cannot be a sink job.

In the example in Fig. 6(a), the data-consumer task \( \tau_3 \) has only one job \( J_{10} \). Eq. (12) provides the set of potential data-producer jobs for \( J_{10} \), i.e., \( AP(J_{10}, \alpha) = \{ J_6, J_7, J_8 \} \) shown in Fig. 6(b). Among these jobs, \( J_6 \) comes from Eq. (8) and \( J_7 \) and \( J_8 \) are from Eq. (11). We repeat the process to find data-producer jobs for \( J_6, J_7, \) and \( J_8 \). Fig. 6(c) shows the resulting data producers. Now we reach to the source task of the chain, i.e., \( \tau_1 \) that has no data-producer in this chain. The jobs that we found in this step, form the set \( \{ J_1, J_2, J_3, J_4 \} \) of possible source jobs for sink job \( J_{10} \). Also, as shown in Fig. 6(d) by connecting the data-dependent jobs that we found in two iterations, we can form the possible chain instances.

Algorithm 1 iteratively finds all potential data-producer jobs \( O \) that could contribute to the upper or lower bounds of data age for a given sink job which is defined in Algorithm 2 in line 7. This recursive algorithm in each iteration finds all potential data-producer jobs for jobs in \( C \) using Eq. (12) and calls itself again with the calculated set of data-producer jobs and the new task chain whose last task was removed (lines 18,19). The recursive algorithm exits when the length of task chain \( \alpha \) equals two because it shows that the calculated list of data-producer jobs (O) is the list of potential source jobs and we reached the head of the task chain.

Algorithm 2 iterates over all jobs of the sink task in the observation window. For each job, it gets the effective source jobs from Algorithm 1. If the set is not empty, it draws one upper and lower bound on data age. These bounds are kept in \( DA_{min} \) and \( DA_{max} \) variables that are initialized in line 1 and updated in lines 9-10. If the task chain has only one task, the data age bounds are the same as the response-time bounds of the task, which can directly be obtained from the schedule-abstraction graph in line 3.

Lemma 3.2. Algorithm 1 (excluding line 14) returns all potential source jobs of any given job \( J_{10} \) of the sink task in a task chain \( \alpha \).

Proof. Proof trivially follows Lemma 3.1. Let \( C \) be the set of data-consumer jobs in iteration \( k \) of the Algorithm 1. For each job \( J^c \in C \), the algorithm uses Eq. (12) to find all possible data producers in any feasible schedule (proven in Lemma 3.1). Moreover, since the threshold \( L \) that is passed to the Algorithm 1 is equal to the latest start time of job \( J_{10} \) and since a data consumer job (here, the sink
Algorithm 1: Find potential source jobs for a job

Input: Job set of data consumers \( C \), Start time upper bound \( L \), Task chain \( \alpha \).

Output: Potential source jobs \( O \).

1 Function findSources \((C, L, \alpha)\):
2 \( O \leftarrow \{\} \);
3 if \( C = \emptyset \) then
4 \( \text{return } 0 \);
5 foreach job \( J \in C \) do
6 \( \text{Obtain } AP(J, \alpha) \text{ from Eq. (12)} \);
7 \( O \leftarrow O \cup AP(J, \alpha) \);
8 foreach job \( J \in O \) do
9 if \( U \leq EST(J) \) then
10 \( O \leftarrow O \setminus \{J\} \);
11 if \( |O| > 2 \) then
12 \( j^\text{min} \leftarrow \text{The job with the smallest } r^\text{min} \text{ in } O \);
13 \( j^\text{max} \leftarrow \text{The job with the largest } r^\text{max} \text{ in } O \);
14 \( O \leftarrow \{j^\text{min}\} \cup \{j^\text{max}\} \);
15 if \( |\alpha| = 2 \) then
16 \( \text{return } O \); // the exit condition of the recursive algorithm
else
17 \( \alpha' \leftarrow \langle \alpha_1, \alpha_2, \ldots, \alpha_{|\alpha|-1} \rangle \);
18 \text{return findSources}(O, L, \alpha');
19

Algorithm 2: Deriving the data age for a task chain

Input: Job set \( J \) start and finish times bound, Task chains \( \alpha \).

Output: Bounds on the data age for a task chain

1 \( DA^\text{min} \leftarrow \infty, DA^\text{max} \leftarrow 0 \);
2 if \( |\alpha| = 1 \) then
3 \( [DA^\text{min}, DA^\text{max}] \leftarrow [BRCT, WCRT] \text{ of task } \alpha(1) \);
4 else
5 \( \text{Sort } J \text{ by jobs' } r^\text{min} \text{ in non-decreasing order} \);
6 foreach job \( J_0 \in J \) that Task \( J(J_0) = \alpha(1) \) do
7 \( \text{// Find possible source jobs using Algorithm 1.} \)
8 \( O \leftarrow \text{findSources}(\{J_0\}, \text{LST}(J_0), \alpha) \);
9 foreach job \( J_0 \in O \) do
10 \( DA^\text{min} \leftarrow \text{min}(DA^\text{min}, \Delta(L(J_0), J_0)); \)
11 \( DA^\text{max} \leftarrow \text{max}(DA^\text{max}, \Delta(U(J_0), J_0)); \)
12 \text{return } DA^\text{min}, DA^\text{max}. \)

since by definition, \( LFT(J^*) \) (\( EFT(J^*) \)) is an upper (lower) bound on the finish time of \( J^* \) in any feasible schedule, it is also an upper (lower) bound of \( J^* \) in Eq. (4). Note that in Eq. (14), although based on the definition of delay for a single schedule, (Eq. (4)), \( \Delta(J) \) should finish after \( J^* \). However, since we have only an interval of finish times (instead of the exact finish time of \( J^* \)), we must ensure that \( \Delta(J) \) does not finish after \( J^* \) even if \( EFT(J^*) < \Delta(J) \) since it must hold that \( J^* \) finishes before \( J^* \). As a result, we ensure that the lower bound is the maximum of zero (an obvious lower bound) and the difference between the \( EFT(J^*) \) and the earliest release time of \( J^* \).

In each iteration of Algorithm 1, our pruning technique (lines 11–14) identifies two special jobs of the data-producer task in the job set \( O \), the one that has the earliest arrival time \( j^\text{min} \) and the one that has the latest arrival time \( j^\text{max} \). Then it discards the rest of the jobs since they have no impact on the bounds of data age.

Let \( J_a, J_b \in O, r^\text{min}_a < r^\text{min}_b \) be two jobs of a data producer task obtained by Algorithm 1 in lines 2 to 10. Let \( r^\text{min}(J) = \text{min}\{r^\text{min} | J \in AP(\alpha)\} \). The following lemma establishes that \( r^\text{min}(J_a) < r^\text{min}(J_b) \) and, hence, the delay calculated from the data-producers of \( J_a \) will be smaller than the delay calculated by the data producers of \( J_b \). Hence, when finding an upper bound on data age, \( J_b \) can be ignored.

Lemma 3.3. For every two jobs of a data-consumer task \( J_a, J_b \in O, r^\text{min}_a < r^\text{min}_b \), we have \( r^\text{min}(J_a) \leq r^\text{min}(J_b) \).

Proof. The proof follows the definition of \( p^\text{first} \) (Eq. (10)). The first potential data producer is a data-producer job whose \( LFT \) (or \( \text{LST}, \text{in Case 2} \)) is smaller than the \( \text{EST} \) of a data-consumer job. Since \( r^\text{min}_a < r^\text{min}_b \) and \( \text{EST}(J_a) < \text{EST}(J_b) \), the \( p^\text{first}(J_a, \alpha) \leq p^\text{first}(J_b, \alpha) \). This concludes the claim.

Lemma 3.4. The pruning step of Algorithm 1 (in lines 11–14) does not change the bounds of data age calculated by Algorithm 2.

Proof. Let \( J^* \) be the sink job for which Algorithm 1 was called in line 7 of Algorithm 2. Furthermore, let \( j^\text{min} \) and \( j^\text{max} \) be the jobs with the smallest and largest release time in \( O \).
By contradiction, assume that \( J' \) is a source job for \( J^* \) that was not in \( O \) and leads to a larger upper bound for the delay, namely, \( \bar{\Delta}^U (J', J^*) > \bar{\Delta}^U (J_{\min}, J^*) \). From Eq. (13), which is used in line 10 of Algorithm 2, it is clear that such a relation holds only if \( J' \) has a smaller release time than \( J_{\min} \). However, according to Lemma 3.3, we know that none of the pruned jobs in any intermediate level of the task chain (i.e., during any iteration of Algorithm 1) could result in a data-producer job that has an earlier release time than \( J_{\min} \). This leads to a contradiction. The proof for the lower bound follows the same reasoning.

**Theorem 3.5.** For any task chain \( \alpha \) in a schedulable task set \( \Gamma \) scheduled by a non-preemptive JLFP scheduling policy, the data age of task chain \( \alpha \) is never larger than \( DA_{\text{max}} \) (the upper bound returned by Algorithm 2), and is never smaller than \( DA_{\text{min}} \) (the lower bound returned by Algorithm 2).

**Proof.** From the discussions in Sec. 2.2, we know that the job set provided for the schedule-abstraction graph as well as Algorithm 2 includes all job-arrival patterns that could potentially contain the schedule that results in the largest (smallest) data age, respectively. From schedule-abstraction graph analysis, we also know that the start-time and finish-time intervals obtained for each job are tight.

Based on the data age definition (Eq. (5)), the upper (lower) bound of data age are safe only if no source and sink jobs in one of the chain instances of a feasible schedule have a larger (smaller) delay than the upper (lower) bound of the data age reported by Algorithm 2.

Lemmas 3.2 and 3.4 establish that for any given job \( J_i \) of a sink task in a task chain, Algorithm 1 returns the source jobs with the earliest and latest release time that could potentially provide data for \( J_i \). Moreover, \( J_i \) cannot have any other source job that has not been returned by Algorithm 1 and has an earlier release time (or a later release time) than the jobs returned by Algorithm 1. Finally, given that lines 8 to 10 update the lower-bound and upper bound of data age for any job that could potentially impact these two bounds, we conclude that the bounds of data age calculated by Algorithm 2 are safe, namely, data age of a task chain \( \alpha \) is never larger than \( DA_{\text{max}} \) and never smaller than \( DA_{\text{min}} \).

### 3.5 Discussions

**Supporting preemptive execution.** By assuming that there exists a response-time analysis that provides the start and finish time intervals of each preemptive job [18, 38], our analysis can be adapted to the preemptive task model with a few modifications. Since in the preemptive model, higher priority jobs can preempt a lower priority job, Eq. (9) does not hold and the first potential data-producer job for a job should always be calculated using Eq. (8).

**Tightness of the analysis.** As explained in Sec. 3, the start- and completion-time intervals obtained from the schedule-abstraction graph analysis of Nasiri et al. [29] are tight. However, this does not mean that the schedules in which a data-producer completes at time \( t \) are the same schedules in which a data consumer starts at time \( t + 1 \). As a result, the bounds we obtain are not tight as they may still rely on certain infeasible schedules. In Sec. 4 we will show that despite this pessimism, our analysis effectively reduces the over-estimation of data age compared to the state of the art.

**Computational complexity.** In our analysis, there are two factors that influence time complexity: (i) the number of jobs in the observation window (\(|J|\)) and (ii) the length of the task chain to analyze (\(|\alpha|\)). Algorithm 1 calls itself at most \(|\alpha|\) times (where, \( |\alpha| \) is the length of task chain). In each iteration, the complexity of both for-loops (in lines 5 and 8) is \( O(|J|) \) because in the worst-case, the number of data-consumer jobs (\( C \)) and the number of jobs in \( O \) is about the total size of the job set. Hence, the complexity of Algorithm 1 is \( O(|\alpha| \cdot |J|) \). Algorithm 2 determines the source jobs for jobs of the last task in the task chain. In which the number of jobs of the last task in the task chain is \( Y \), where \( Y = OW/T_x \), \( OW \) is the observation window and \( T_x \) is the period of last task in the task chain. Therefore, Algorithm 2 calls Algorithm 1 for \( Y \) times. Putting these together, the complexity of our analysis for a given task chain \( \alpha \) is \( O(|\alpha| \cdot |J| \cdot Y) \). Since the number of jobs in a hyperperiod could be exponential to the number of tasks, our method is pseudo-polynomial w.r.t. the number of tasks. The complexity could be improved by deriving a direct equation to obtain only the job with the earliest and latest release in \( P_{\text{est}} \) (since the pruning step will discard the intermediate jobs).

### 4 EMPirical Evaluation

We conduct experiments on an industrial case study (Sec. 4.1) as well as synthetic task sets (Sec. 4.2) to answer the following questions: (i) does our analysis reduce the pessimism of derived bounds of data age compared to the state of the art? And (ii) is the runtime of our analysis practical?

#### 4.1 Case Study

We considered an automotive application [22, 40] (detailed in Fig. 2 and Table 1) as a case study. We compare our method with Becker et al. [6] and Becker et al. [8]. The former paper does not incorporate scheduling policy information and the latter paper relies on WCRTs.

We considered the observation window of 350ms, extracted the jobs within this window, and calculated their start and finish time intervals using the schedule-abstraction graph [29] for the EDF scheduling policy. The obtained response-times are also used for Becker et al. [8] method which needs information about the task response times. However, since these baselines do not assume any uncertainty in the execution time of the tasks, we considered two setups for the task sets in Table 1: one with execution-time variation and one without. The second and third columns of Table 2 show the result of our algorithm in these two setups. For the method of Becker, we only used the setup without execution-time variation.

As shown in Table 2, our method reduces the pessimism of data age bounds for the given task chains by up to 30% (27.65% on average for the four task chains) in comparison to Becker et al. [8].

#### 4.2 Evaluation via Synthetic Task Sets

We design four experiments to study the impact of system utilization (\( U \)), the number of processing elements (\( m \)), the number of tasks in the task sets (\( n \)), and the length of task chains on the effectiveness of our solution. In each experiment, we generate 100 schedulable task sets for each data point reported in Fig. 7.

**Task set generation.** To generate a periodic task set with \( n \) tasks, we select \( n \) period values based on the probability distribution
of periodic automotive benchmark applications [26]. Using the RandFixedSum method of Emberson et al. [14], we generate a set of \( n \) random utilizations that sum to the target system utilization. We assumed the system is scheduled by the EDF policy; hence, the priority of each job is equal to its absolute deadline. Moreover, to be able to compare our results to other work in the state of the art, we assume that tasks do not have release jitter (since none of the methods we compare against supports release jitter). After generating the tasks, we assign them to PEs using the worst-fit partitioning. We consider that PEs are identical since our analysis is not affected by the type of PEs (as migration is not allowed).

**DPG generation.** To generate random data-propagation graphs, we go through each task and randomly assign an edge from that task to another task with a probability of 0.4. We made sure that the edges do not form a cycle by processing the tasks in a random but consistent order and adding edges in one direction. We limit the maximum branching factor of a node (i.e., the number of immediate successors of a node) to 4, the maximum number of input nodes to 5, and the maximum depth of the graph to 10 to form task graphs that are not too far in terms of structure from the ones of our case study. A generated DAG contains between two and fifteen task chains, and a task could be a part of several task chains at once.

**Baselines.** We compare our analysis to the schedule-agnostic data-age analysis of Becker et al. [6], a recent work of Becker et al. [8] that uses WCRT information, the data age analysis of Dürr et al. [13] that uses WCRT information, and the work of Kloda et al. [24], which obtains the reaction latency (an upper bound on data age). To have a fair comparison between the three methods [8, 13, 24] which require information about the WCRT of the jobs, we use the accurate WCRTs derived by the SAG analysis (with partial-order reduction) [36, 37] which is the most accurate existing job-level response time analysis method.

**Metrics.** We report the upper bound of the data age of each task chain normalized by the hyperperiod of the task chain, namely, it is divided by the least common multiple of the periods of the tasks in the chain. The average values of normalized data age are then shown in Fig. 7. Later in Fig. 8, we report the runtime (CPU time) of our method for two experiments that focus on larger system sizes.

**Execution platform.** We instrumented the open-source code of schedule-abstraction graph approach [31, 32] to also report the start-time intervals per job, in addition to the finish-time intervals. We performed the experiments on SurfSara, the Dutch national high-performance computing cluster on AMD EPYC 7601 processors clocked at 2.2GHz with 1TB of RAM.

**Experiment1 (impact of system utilization).** We consider a system with four PEs and 15 tasks per task set. We vary the total utilization \( U \) from 0.4 to 3.6 in steps of 0.4. Fig. 7(a) shows the impact of system utilization on the average data age. Despite the fact that the data age increases as the utilization increases, our method is able to produce significantly less-pessimistic upper bounds on data age than the state of the art. Namely, it reduces the bounds by up to 38% and on average by 31.8% compared to Becker et al. [8]. In comparison to Kloda’s method [24], we improve the tightness by up to 40.3%, and on average 30%. This experiment also shows that the method of Dürr et al. [13] is sensitive to system utilization. For example, for \( U = 0.4 \) Dürr et al. [13] is 8% more pessimistic than Becker et al. [8], while at \( U \approx 3.6 \), this difference grows to 30%.

**Experiment2 (impact of the number of PEs).** In this experiment, we vary the number of PEs \( m \) from 2 to 8. We keep system
Fig. 8: CPU time as a function of (a) the number of tasks, and (b) the maximum length of task chains.

As our method, and three other methods [8, 13, 24] that require the WCRT information of the tasks have the same overhead of a method that provides the response-time bounds for each job, we separately reported the runtime of the SAG analysis with partial-order reduction [36, 37] in order to get more accurate insight into the runtime of data-age analysis methods. It is worth mentioning that, we selected SAG analysis because it is providing the most accurate response-time bounds for each job. Though our method is slightly slower than Kloda’s [24] and Dürr’s [13] methods, our method obtains a far less pessimistic upper bound on data age. Furthermore, our experiments show that even with the SAG’s runtime, the average runtime of our solution in setup (i) is 902 seconds which is 1.8 times faster than Becker et al. [6] method.

5 CONCLUSION

In this paper, we presented a novel data-age analysis of multi-rate task chains with the presence of uncertainties in the release time and execution time of the tasks. We demonstrated these uncertainties by timing intervals on the start-time and completion-time of the jobs in an observation window. This enabled us to develop a new analysis that explores possible data dependencies between jobs for a collection of schedules (instead of only one, as opposed to the literature). To speed up the analysis, we introduced pruning rules to rule out jobs that would not impact the bounds of data age.

Using an industrial case study together with synthetic task sets, we demonstrated that our method reduces over-estimations of data age of all experiments between 30% and 40% compared to the state-of-the-art analyses. Moreover, unlike previous approaches, our method scales well in the length of task chains. We found that some of the existing analyses of data age [6, 8, 13] perform worse than the upper bound derived from the reaction latency (calculated by the Kloda method [24]). As such, our work delivers the first direct analysis of data age, deriving tighter upper and lower bounds than previous approaches. In the future, we intend to extend our analysis to include more complex timing constraints that require the consideration of multiple chains simultaneously (e.g., when input delivery to a set of chains must occur contemporaneously).
ACKNOWLEDGMENTS
This work made use of (i) the Dutch national e-infrastructure with support of the SURF Cooperative under grant agreement no EINF-2663, (ii) the EU ECSEL Joint Undertaking under grant agreement no 101007260 (project TRANSACT), and (iii) funding from the Eindhoven Artificial Intelligence Systems Institute (EAISI), Eindhoven University of Technology, PO Box 513, the Netherlands.

REFERENCES