Chain-Based Fixed-Priority Scheduling of Loosely-Dependent Tasks

Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim
University of California, Riverside
{hchoi036, mkari007, hyoseung}@ucr.edu

***Abstract*—Many cyber-physical applications consist of chains of tasks. Such tasks are often loosely dependent, meaning task execution is time-triggered and independent of the update rate of input data. Since meaningful output can be obtained after processing all the intermediate tasks of a chain, the end-to-end latency of the chain is an important metric that can affect the correctness and quality of the system. In this paper, we present a *chain-based fixed-priority preemptive scheduler* for multi-core real-time systems. The scheduler identifies *effective chain instances* contributing to the generation of updated output, and employs a runtime policy to improve the end-to-end latency of chains. Based on our scheduler, an analysis method is proposed with two parts: (i) bounding the start and finish time of each job, and (ii) analyzing the end-to-end latency of effective chain instances. Experimental results show that our chain-based scheduler achieves up to 83% reduction in end-to-end latency compared to the state-of-the-art and yields a significant benefit in inter-chain distance over chain-unaware schedulers. Furthermore, our analysis method can be easily adapted to chain-unaware schedulers and provides tighter bounds than prior work.**

I. Introduction

The complex information flows of cyber-physical systems are increasingly implemented with chains of tasks. An autonomous vehicle is a good example as studied in [1, 15, 17]. Sensing tasks collect data from various sensors, e.g., LiDAR, cameras, and IMU, according to their update rate. Using the collected data, computation tasks periodically perform operations like localization, detection, and prediction, and then generate the trajectory of the vehicle. In the actuation stage, control tasks manipulate steering and throttle based on the latest results of the computation tasks. Since the final output is obtained only after processing all the intermediate tasks, bounded and predictable end-to-end latency is the key to meet the application-level requirements such as control quality and data freshness. Similar design is also observed in IoT applications, e.g., real-time smart home control [14].

Data dependency in task chains does not necessarily impose strict precedence constraints among tasks. Each task can execute and produce output at its own rate, independent of other tasks, by using the most recently updated input data. This model, we call *loosely-dependent* chains, gives flexibility in task scheduling and allows output from one task to be shared with the tasks of other chains, without having to synchronize their release times. The meaning of the end-to-end latency of a loosely-dependent chain does not differ from the conventional definition, which is the time elapsed from the release of the first task in a chain to the completion of the last task generating an updated output. In fact, the publisher-subscriber model in ROS [26] and the read-execute-write semantics in AUTOSAR [5] with shared memory have been widely used to utilize loosely-dependent chains in system design.

The end-to-end latency of loosely-dependent chains can be negatively affected by the deadline misses of intermediate tasks. However, not all the task-level deadlines need to be always met unless each has a hard real-time constraint. Timely execution of a specific job may not contribute to the generation of the final chain output if the preceding task has not updated input data or the following task does not consume the output of this job. Moreover, many sensing and control applications are known to be capable of tolerating some deadline misses as long as they do not affect the functional correctness of system-level behavior. While weakly-hard real-time systems [9, 11, 29] have been studied to capture this effect, the notion of loose dependency and end-to-end latency has not been well studied in the literature. Note that this is a non-trivial problem since each chain may consist of tasks with different periods and priorities. The complexity multiplies when tasks are shared among multiple chains.

In this paper, we propose *chain-based fixed-priority preemptive scheduling* for periodic tasks with loose dependency in a multi-core environment. The goal of this work is to minimize end-to-end latency of task chains since this is the foremost requirement in many practical applications. The main contributions of this paper are shown as follows:

- We present the design of the proposed chain-based scheduler. The offline part of the scheduler identifies *effective chain instances* which produce valid and updated chain outputs. Based on this information, the runtime part enforces execution rules to reduce the end-to-end latency of chains.
- We develop an analytical method to upper-bound end-to-end chain latency under the proposed scheduler. Our method first bounds the start and finish time of individual jobs and then analyzes the maximum latency of effective chain instances.
- We show that, with small modifications, our analysis can be used to analyze end-to-end chain latency under conventional chain-unaware fixed-priority schedulers.
- Experimental results demonstrate that the proposed chain-based scheduler yields significant improvement over chain-unaware schedulers, in terms of end-to-end latency and the distance between valid chain outputs.

2576-6996/20/$31.00 ©2020 IEEE
DOI 10.1109/ICCD50377.2020.00109

The rest of the paper is organized as follows. Sec. II reviews prior work, and Sec. III gives the system model used. The proposed chain-based scheduling and the analysis method are presented in Sec. IV and Sec. V. Evaluation results are given in Sec. VI. Sec. VII concludes the paper with future scope.

II. Related Work

Many studies have been conducted on the schedulability analysis of hard real-time tasks with data dependencies. Palencia et al. proposed approaches to analyze tasks with dynamic offset and extended the work to tasks with precedence constraints in multi-core systems [22, 23]. In [10, 28], methods to capture the upper-bound of end-to-end latency of tasks with dependency are presented based on the worst-case response time. The recent work by Kloda et al. [18], Abdullah et al. [2], and Becker et al. [5] present analytical methods to bound the end-to-end latency of a chain, which have inspired our work. However, all of these studies assume conventional chain-unaware scheduling and do not propose a new scheduler design to improve end-to-end latency.

Direct Acyclic Graph (DAG) has been widely used to represent precedence constraints among tasks [20, 27, 31]. However, DAG-based scheduling cannot be directly used to solve the scheduling problem of loosely-dependent task chains where tasks run asynchronously with different periods and priorities.

Age of Information (AoI) has recently received much attention in networking systems and cyber-physical applications [4, 16, 21, 30] and used as a metric to quantify the freshness of data over time. While scheduling approaches to improve or maintain data freshness have been proposed for real-time networking systems [13, 21], they do not provide a predictable bound on the worst-case end-to-end latency.

In the literature of weakly-hard real-time systems, there exists only a small number of papers on tasks with data dependencies. Hammadeh et al. [12] used the typical worst-case analysis (TWCA) to derive the deadline miss models of weakly-hard systems with task dependencies. Their work assumes that chains do not have shared tasks and all chains execute on a single processor. In [25], a state-based methodology is presented to model the performance of a control application in terms of data freshness and weakly-hard constraints, assuming that all tasks have the same periods.

III. System Model

A. Task model

This paper considers a multi-core system where all CPU cores run at the same fixed clock frequency. The system runs a taskset (Φ) consisting of N periodic preemptible tasks. Each task τ_i is characterized as follows:

$$\tau_i := (BC_i, WC_i, D_i, T_i, o_i, \pi_i)$$

- BC_i: The best-case execution time of a job of τ_i
- WC_i: The worst-case execution time of a job of τ_i
- D_i: The relative deadline of τ_i ($D_i \leq T_i$)
- T_i: The period of τ_i
- o_i: The initial release offset of τ_i
- π_i: The priority of τ_i

Each task is statically allocated to one CPU core and does not migrate at runtime. The j-th job of τ_i is denoted as $J_{i,j}$. Task priorities can be assigned by any fixed-priority assignment policies, e.g., Rate Monotonic.

If a job of a task misses the deadline, it is immediately aborted (or descheduled) to prevent blocking of its next job and wasting of CPU cycles. Note that this does not affect the logical correctness of subsequent jobs since tasks can either be stateless or recover their states with low-cost rollback mechanisms developed for real-time systems [3, 8].

B. Chain model

A chain Γ^c of loosely-dependent tasks is denoted as below:

$$\Gamma^c := [\tau_s, \tau_{m_1}, \tau_{m_2}, ..., \tau_e]$$

- τ_s: The start task of a chain Γ^c.
- τ_{m_*}: The intermediate task of a chain Γ^c.
- τ_e: The end task of a chain Γ^c.

The superscript c is the identifier of the chain Γ^c. We use $\Gamma^c[i]$ to denote the i-th task of the chain, e.g., $\Gamma^c[1] = \tau_s$. Following the model widely used in prior work [2, 5, 18], we assume that all tasks in a chain use the *read-execute-write* semantics, where a task reads input before the start of execution and produces output at the end of execution, and inter-task data communication is done via shared memory/registers at negligible cost. The start task of a chain does not require input data, e.g., sensing tasks, but all other tasks use the most recently updated input data from their preceding tasks to generate valid outputs, e.g., computation and control tasks.

Each chain represents one data flow path of tasks. Fig. 1 illustrates the example. Chains 1, 2, and 3 share τ_1 as a mutual (joint) start task. Chains 2 and 3 contain mutual tasks τ_5 and τ_6 as an intermediate and an end task, respectively.

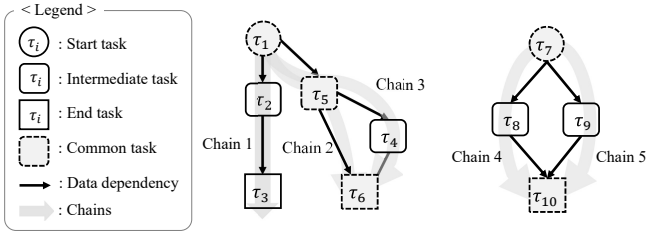
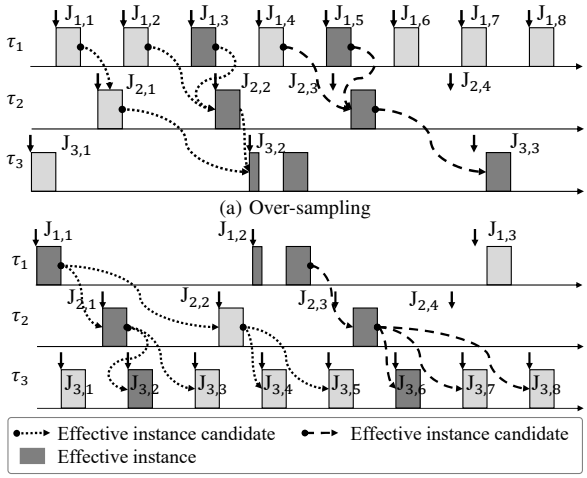


Fig. 1: Example of chains

It is worth noting that a conventional independent periodic task with a hard real-time constraint can be represented as a single-task chain. Then, our proposed scheduler and analysis can guarantee the schedulability of such tasks.

Chain instance. A chain can instantiate multiple job-level data flows at runtime. We use a *chain instance* $\mathcal{C}^c[k]$ to denote the k-th instance of the chain Γ^c. $\mathcal{C}^c[k]$ includes jobs from the start task to the end task of the corresponding chain, and $\mathcal{C}^c[k, j]$ indicates the j-th job of $\mathcal{C}^c[k]$.

Fig. 2 illustrates the execution of three tasks, τ_1, τ_2, and τ_3, of a chain $\Gamma^c = [\tau_1, \tau_2, \tau_3]$. The instances of the chain are



(a) Over-sampling

(b) Under-sampling

Fig. 2: Effective chain instances

indicated with dotted arrows. In Fig. 2a, jobs $J_{1,1}$, $J_{2,1}$, and $J_{3,2}$ form the first instance of Γ^c, i.e., $\mathcal{C}^c[1] = [J_{1,1}, J_{2,1}, J_{3,2}]$. Note that the job $J_{3,1}$ cannot be part of $\mathcal{C}^c[1]$ because it executes before valid input is ready.

In complex systems like automotive applications [17, 19], chain instances may experience the over- or under-sampling effects due to different periodicities of tasks [2]. In either case, there is at least one mutual job among the candidate chain instances that may produce the same valid output. For the over-sampling case in Fig. 2a, the first three chain instances starting with $J_{1,1}$, $J_{1,2}$, and $J_{1,3}$, respectively, generate a single final output because the end task job ($J_{3,2}$) is commonly used. For the under-sampling case in Fig. 2b, a mutual start job $J_{1,1}$ is used by the first four chain instances (finishing with $J_{3,2}$, $J_{3,3}$, $J_{3,4}$, and $J_{3,5}$) and their outputs are identical because they all use the same data from $J_{1,1}$. This means, once the first instance generates an output, the completion of the latter three instances does not yield any updated chain output.

Based on the above observations, we define an *effective* chain instance among candidate instances as follows:

Def. 1. *An effective instance of a chain Γ^c is the earliest instance producing a valid and updated final output using the most recently updated input data. The i-th effective instance of Γ^c is denoted as $E^c[i]$.*

In Fig. 2a, effective instances are: $E^c[1] = \mathcal{C}^c[3] = [J_{1,3}, J_{2,2}, J_{3,2}]$ and $E^c[2] = \mathcal{C}^c[5] = [J_{1,5}, J_{2,3}, J_{3,3}]$. In Fig. 2b, $E^c[1] = \mathcal{C}^c[1] = [J_{1,1}, J_{2,1}, J_{3,2}]$ and $E^c[2] = \mathcal{C}^c[5] = [J_{1,2}, J_{2,3}, J_{3,6}]$ are effective instances. Since $E^c[i]$ is one of the chain instances, hereafter we will use $E^c[i, j]$ to denote the j-th job of the i-th effective instance.

Lemma 1. *The end-to-end latency of an effective instance has the minimum timespan among its candidates.*

Proof: By Def. 1, the proof is obvious since an effective instance is the earliest valid instance among its candidates and uses the latest input data. ■

Algorithm 1 Synthesize chain instances

procedure FIND_EFFECTIVE_CHAIN_INSTANCES(Γ^c,$\mathcal{J}$,Φ)
  Γ^c: a chain c
  Φ: a taskset
  $\mathcal{J}$: a set of released jobs of all tasks $\in \Gamma^c$ during the hyperperiod of the taskset Φ
  $\mathcal{C}^c \leftarrow \emptyset$; $k \leftarrow 1$
  for each $J_{i,j} \in \mathcal{J}$ in ascending order of release time **do**
    if $\tau_i = \Gamma^c[1]$ **then**
      $\mathcal{C}^c[k, 1] \leftarrow J_{i,j}$
      $k \leftarrow k + 1$
    else
      for each $\mathcal{C}^c[\text{row}] \in \mathcal{C}^c$ **do** ▷ All instances of $\mathcal{C}^c$
        col $\leftarrow \tau_i$'s column in Γ^c
        if $\mathcal{C}^c[\text{row}, \text{col}] = \emptyset$ and $\mathcal{C}^c[\text{row}, \text{col} - 1] \neq \emptyset$ **then**
          $\mathcal{C}^c[\text{row}, \text{col}] \leftarrow J_{i,j}$
  for each $\mathcal{C}^c[\text{row}] \in \mathcal{C}^c$ **do** ▷ Discard ineffective instances
    if $\mathcal{C}^c[\text{row}, |\Gamma^c|] = \mathcal{C}^c[\text{row} + 1, |\Gamma^c|]$ **then**
      $\mathcal{C}^c[\text{row}] \leftarrow \emptyset$
  $E^c \leftarrow$ all non-empty rows of $\mathcal{C}^c$
end procedure

IV. Chain-based Fixed-priority Scheduling

This section presents our chain-based fixed-priority preemptive scheduler, which consists of *offline* and *runtime* components. The offline part generates effective chain instances to capture job-level data dependencies among tasks, and the runtime part governs the actual execution of each job.

A. Offline synthesis of effective chain instances

According to Def. 1, the execution of a job that does not belong to an effective instance yields no benefit in chain output and resource efficiency. Hence, we propose to statically synthesize a set of effective chain instances for use in online scheduling. In this approach, the inclusion of a job into a chain instance is determined by its release time. This is because, as we have observed $J_{3,1}$ in Fig 2a, a job cannot be part of the chain instance if its updated input data is not ready before the start of execution and the release time is the earliest start time of a job. The set of effective chain instances is constructed for one hyperperiod and is revised in Sec. V if it contains a job that is not guaranteed to meet the deadline.

Step 1. Initializing chain instances. A new chain instance $\mathcal{C}^c[i]$ is initialized when there is a job released from the first task of the chain Γ^c, i.e., if the job is $J_{u,w}$, $\mathcal{C}^c[i, 1] = [J_{u,w}]$. If τ_j is a mutual task of multiple chains, a new chain instance is initialized for each of these chains. It is worth noting that, in the under-sampling case, the job may be part of later instances of the same chain, but only the earliest instance is eligible to be an effective chain instance.

Step 2. Building chain instances. If there is a job released from the j-th task of a chain Γ^c ($j \neq 1$), it may be eligible to be part of multiple chain instances. Hence, such a job is added to all generated chain instances where $\mathcal{C}^c[i, j]$ is empty but $\mathcal{C}^c[i, j - 1]$ is occupied.

Based on the above steps, one can find effective instances E^c from all candidate instances of a given chain Γ^c. The detailed procedure is given in Alg. 1. Here, $\mathcal{C}^c$ and E^c are

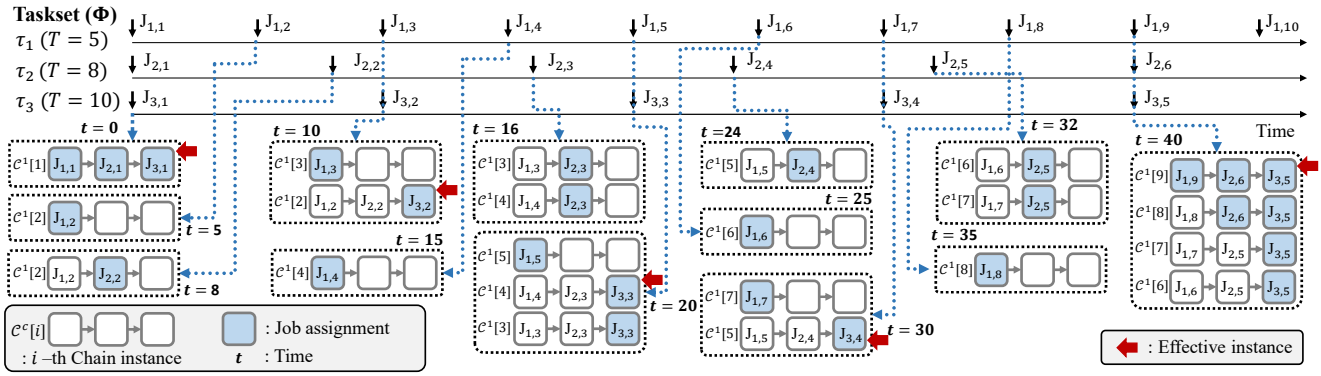


Fig. 3: Synthesis of chain instances and effective instances for the taskset (Φ)

treated as two-dimensional arrays, e.g., a job of the j-th task (column) of the k-th instance (row) is stored in $\mathcal{C}^c[k,j]$. All released jobs from the tasks of Γ^c during a hyperperiod of the entire taskset are sorted in ascending order of release times in line 4 so that chain instances can be built in a chronological order. A job of the start task of the chain initiates a chain instance from line 7 to 10, while the other jobs are allocated from line 11 to 14, as described in the above steps. Once all chain instances are generated, effective instances are found from line 15 to 18. If multiple instances have the same end job, the algorithm chooses the last one as the effective instance because that uses the latest update data, following Def. 1.

Fig. 3 illustrates the sequence of synthesizing chain instances for a given taskset (Φ). Suppose that the taskset has a chain $\Gamma^1 = [\tau_1, \tau_2, \tau_3]$. Chain instances are built using all the jobs released during one hyperperiod $t = 40$. In this example, 8 chain instances are generated and 4 of them are found to be effective instances, e.g., $\mathcal{C}^c[1], \mathcal{C}^c[2], \mathcal{C}^c[4]$, and $\mathcal{C}^c[5]$ become $E^c[1], E^c[2], E^c[3]$, and $E^c[4]$, respectively. Those chain instances will be repeated in the next hyperperiod.

It is possible that some jobs of the generated instances may turn out to miss their deadlines. In Sec. V, we will revisit this issue and present a method to revise effective instances based on the analysis results.

B. Runtime scheduling with the release-and-ready policy

At runtime, the release-and-ready (RNR) policy of the chain-based scheduler prevents unnecessarily early start of job execution. This is done by introducing two-step phases to each job: *release* and *ready*. The release phase occurs when a job is released according to its period, and the job in the release phase cannot start execution. The transition to the ready phase occurs when its previous jobs of the same chain instance(s) have completed their execution. Then, all jobs in the ready phase are scheduled based on their task priorities.

From the perspective of effective chain instances, each job can be in one of the three different categories: (1) a job is only placed in a single effective instance, (2) a job is involved in multiple effective instances of different chains, or (3) a job is not part of any effective instance. The following rules of the RNR policy indicate how the ready phase is carried out for the job in different categories.

Rule 1. Job in a single chain instance. Consider a job $E^c[i,j]$ of a single effective instance. If $j \neq 1$, the job turns into the ready phase when the immediately previous job of the same instance ($E^c[i,j-1]$) completes its execution. If $j = 1$ (the start job of the instance), the job $E^c[i,j]$ switches to the ready state when the most recent job of the previous effective instance ($E^c[i-1]$) assigned to the same CPU core as $E^c[i,j]$ has completed its execution.

Rule 2. Job in multiple instances. A job that belongs to the multiple effective instances of different chains can only switch to the ready phase when Rule 1 is satisfied for all of its effective instances.

Rule 3. Job not in effective instances. A job that does not belong to any effective instance is dropped (skipped) by the chain-based scheduler at runtime.

Lemma 2. *For a system with independent periodic hard real-time tasks, the proposed chain-based scheduler is equivalent to the conventional chain-unaware fixed-priority schedulers that solely determine task execution order by priorities.*

Proof: In the chain-based scheduler, any job of hard real-time tasks is not affected by the runtime rules 1-3 because each hard real-time task is modeled as a single-task chain (see Sec. III) and each job of that task forms an effective instance. Therefore, the chain-based scheduler yields the same scheduling decisions as the chain-unaware schedulers. ■

One may expect that the RNR policy would introduce additional interference to lower-priority tasks since it causes a self-suspension effect [7, 24] to the job waiting in the release phase for its predecessor's completion. We will show in Sec. V that our proposed analysis takes into account such self-suspending behavior by safely capturing the start and finish time of each job of a task.

V. End-to-end latency analysis

This section presents the end-to-end latency analysis of tasks with data dependency under chain-based scheduling. We first derive the start and end time of individual jobs of effective chains without preemptions, and then analyze the amount of interference due to preemptions during a given time interval of job execution. By using these, we finally analyze the end-to-end latency of effective chains in an iterative manner.

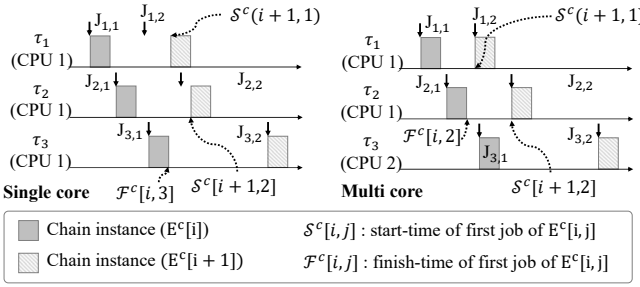


Fig. 4: Start time of a job (single vs. multi-core system)

A. Job start and finish time with no preemption

Assume there is no preemption from higher-priority tasks of other chains. With this assumption, the only factors affecting the start and finish time of a job are its own execution time and the delay introduced by the runtime chain-based scheduler.

Lower bound on job start time. To capture the maximum timespan of job execution, we need to find the earliest (lower bound) starting time of a job. Depending on the job's position within the chain instance, the analysis of start time can be done considering the following two cases: the job being the first job of the chain instance or not.

Lemma 3. *Consider the j-th job in the i-th effective instance of the chain Γ^c, denoted as $E^c[i,j]$. The lower bound on the start time of $E^c[i,j]$ is given by:*

$$\mathcal{S}^c[i,j] = \begin{cases} \max\{r, \mathcal{F}^c[i,j-1]\} & , \text{if } j \neq 1 \\ \max\{r, \max\limits_{\forall k:\Gamma^c[k]\in P}\{\mathcal{F}^c[i-1,k]\}\} & , \text{if } j = 1 \end{cases} \tag{1}$$

where r is the release time of $E^c[i,j]$ and P is the CPU core that a task $\Gamma^c[j]$ is allocated. $\mathcal{F}^c[i,j]$ is the lower bound on the finish time of $E^c[i,j]$ and will be derived by Equation 4.

Proof: If $j \neq 1$, the job $E^c[i,j]$ can be executed only after the completion of the preceding job $E^c[i,j-1]$ within the same effective instance (Rule 1 in Section IV). Hence, a lower bound on the start time of $E^c[i,j]$ can be obtained by taking a higher value between the release time and the earliest finishing time of $E^c[i,j-1]$. If $j = 1$, the start time of $E^c[i,1]$ depends on the completion time of the other job from the previous chain instance $E^c[i-1]$ executing on the same CPU (Rule 1). This is exactly captured by the inner max term in Equation 1. Thus, the proof is done. ∎

Figure 4 illustrates how the start time of a job (when $j = 1$) behaves differently in a single core and a multi-core system. In a single-core system, $J_{1,2}$ only starts after the completion of $J_{3,1}$, which is the last job of the previous chain instance. However, in a multi-core system, $J_{1,2}$ can start after the completion of $J_{2,1}$ because it is the last job of the previous chain instance to be completed on CPU 1.

Upper bound on job finish time. We also need to find the latest (upper bound) finish time of a job to capture the maximum timespan of job execution at runtime. Likewise, the upper bound of finish-time of a job can be done considering two cases as follows.

Lemma 4. *The upper bound on the finish time of a job $E^c[i,j]$ is given by:*

$$\overline{\mathcal{F}}^c[i,j] = \begin{cases} \max\{r, \overline{\mathcal{F}}^c[i,j-1]\} + WC & , \text{if } j \neq 1 \\ \max\{r, \max\limits_{\forall k:\Gamma^c[k]\in P}\{\overline{\mathcal{F}}^c[i-1,k]\}\} + WC & , \text{if } j = 1 \end{cases} \tag{2}$$

where WC is the worst-case execution time of the job $E^c[i,j]$.

Proof: In Lemma 3, we found the job that affects the start time of $E^c[i,j]$. Let's denote it as J_E. Then, by Rule 1 in Sec. IV, an upper bound on the finish time of $E^c[i,j]$ can be reached by adding the worst-case execution time of $E^c[i,j]$ to the latest finishing time of J_E. Thus, it is proved. ∎

Using Lemmas 3 and 4, an upper bound on start time, $\overline{\mathcal{S}}^c[i,j]$ and an lower bound on finish time, $\mathcal{F}^c[i,j]$, can be trivially derived as follows:

$$\overline{\mathcal{S}}^c[i,j] = \begin{cases} \max\{r, \overline{\mathcal{F}}^c[i,j-1]\} & , \text{if } j \neq 1 \\ \max\{r, \max\limits_{\forall k:\Gamma^c[k]\in P}\{\overline{\mathcal{F}}^c[i-1,k]\}\} & , \text{if } j = 1 \end{cases} \tag{3}$$

$$\mathcal{F}^c[i,j] = \begin{cases} \max\{r, \mathcal{F}^c[i,j-1]\} + BC & , \text{if } j \neq 1 \\ \max\{r, \max\limits_{\forall k:\Gamma^c[k]\in P}\{\mathcal{F}^c[i-1,k]\}\} + BC & , \text{if } j = 1 \end{cases} \tag{4}$$

where BC is the best-case execution time of $E^c[i,j]$.

B. Interference on job execution

Now we consider interference caused by preemptions of higher-priority tasks. The following lemma holds due to the RNR policy of the chain-based scheduler.

Lemma 5. *A job $J_{t,i}$ of a higher-priority task τ_t in a chain $\Gamma^{c'}$ does not interfere (preempt) a lower-priority task τ_j in another chain Γ^c, if and only if, the higher-priority task τ_t is also part of the chain Γ^c and its job $J_{t,i}$ belongs to one of the effective instances of E^c, i.e., $\tau_t \in \Gamma^{c'} \wedge \tau_t \in \Gamma^c \wedge J_{t,i} \in E^c$.*

Proof: Suppose that we compute interference imposed on a job of τ_j in a chain Γ^c. If a higher-priority task τ_t ($\pi_t > \pi_j$) of another chain $\Gamma^{c'}$ is also involved in Γ^c, three possible cases arise: (i) a job of τ_t belongs only to an effective instance of $\Gamma^{c'}$, (ii) it belongs only to an effective instance of chain Γ^c, and (iii) it is a mutual job of effective instances of both chains. For the first case, such a job needs to be considered as an interference source because its execution is independent of the chain Γ^c. For the last two cases, the execution of such a job is taken into account in the start time of τ_j by Rule 1 and 2 in Sec. IV. Thus, the proof is done. ∎

Based on this lemma, we analyze the interference imposed on a job by a single higher-priority task of a different chain. Note that below assumes that the start and finish time of higher-priority jobs are known. We will show in the next subsection how these are derived in an iterative fashion.

Interference from a single higher-priority task. At first, we give Alg. 2 that finds out all jobs of a given higher-priority task τ_k from a different chain $\Gamma^{c'}$ that overlap with a given execution time window $t = [s, f]$ of a lower-priority job under

Algorithm 2 Interference from a higher-priority task τ_k

1: **function** I_TASK(t,τ_k,$\Gamma^{c'}$,E^c)
2: $t = [s, f]$: a given time window of a victim job of E^c
3: τ_k: a higher-priority task causing interference
4: $\Gamma^{c'}$: a chain that contains τ_k
5: $j \leftarrow \tau_k$'s position in $\Gamma^{c'}$
6: $\mathcal{J} \leftarrow \emptyset$
7: **for each** effective instance $E^{c'}[i]$ of $\Gamma^{c'}$ **do**
8: **if** $E^{c'}[i,j] \notin E^c$ **then**
9: finish $\leftarrow \overline{\mathcal{F}}^{c'}[i,j]$
10: start $\leftarrow \mathcal{S}^{c'}[i,j]$
11: **if** $[s,f] \cap$ [start, finish] **then**
12: $\mathcal{J} \leftarrow \mathcal{J} \cup E^{c'}[i,j]$
13: **return** $\mathcal{J}$ ▷ a set of jobs of τ_k that overlap with the t
14: **end function**

analysis. The algorithm iterates over all effective instances that the higher priority task is involved (line 7). Then it obtains the maximum timespan of a job of τ_k ($E^{c'}[i,j]$) using its latest finish time $\overline{\mathcal{F}}$ and the earliest start time $\mathcal{S}$. If the execution of this job overlaps with t, then $E^{c'}[i,j]$ is inserted to the set $\mathcal{J}$ that is returned at the end of the algorithm.

Lemma 6. *The maximum temporal interference imposed on a job of a chain Γ^c by a single higher-priority task τ_k during a given time window t is upper-bounded by:*

$$W^c(t,\tau_k) = \begin{cases} 0 & , \text{if } \substack{\tau_k \in \Gamma^c \wedge \\ c' \neq c : \tau_k \notin \Gamma^{c'}} \\ \left| \bigcup_{\substack{\forall c' : c' \neq c \wedge \\ \tau_k \in \Gamma^{c'}}} \text{I\_TASK}(t, \tau_k, \Gamma^{c'}, E^c) \right| \times WC_k & , o.w. \end{cases} \tag{5}$$

where $\text{I\_TASK}(t, \tau_k, \Gamma^{c'})$ *is given in Alg. 2.*

Proof: If a higher-priority task τ_k is only engaged in the chain Γ^c which involves the target job $E^c[i,j]$, preemption cannot happen by Lemma 5, thus, interference is 0. Otherwise, all other chains that includes τ_k, except the chain Γ^c, need to be considered to capture the maximum interference imposed on a job of Γ^c. By using Alg. 2, we can find all the jobs of τ_k in the chain $\Gamma^{c'}$ that overlap with the execution of the victim job. Taking the union of all such jobs eliminates redundant jobs that are involved in multiple chain instances. Then, the maximum temporal interference can be upper-bounded by multiplying WC_k by the number of elements in the union set. ■

Job execution with interference. By considering interference from each of higher-priority tasks, we derive an upper bound on the maximum execution timespan of a job from the start of execution to the end of execution.

Theorem 1. *The maximum execution timespan I_l of a job $J_l = E^c[i,j]$ with interference from higher-priority tasks is upper bounded by the following recurrence:*

$$I_l^{m+1} \leftarrow WC_l + \sum_{\forall \tau_k : \pi_k > \pi_l \wedge \tau_k \in P} W^c([s, s + I_l^m], \tau_k) \tag{6}$$

where s is the start time of the job under analysis ($s = \mathcal{S}^c[i,j]$), WC_l is the worst-case execution time of $E^c[i,j]$, and P is the CPU core $E^c[i,j]$ is assigned to.

Proof: Obvious from Lemma 6. ■

Algorithm 3 Bound start and finish time of jobs

1: **procedure** BOUND_START_FINISH($\Phi, \mathcal{S}, \mathcal{F}, \overline{\mathcal{S}}, \overline{\mathcal{F}}, E$)
2: **repeat**
3: $flag \leftarrow false$
4: $V_{prev} \leftarrow (\mathcal{S}, \overline{\mathcal{S}}, \mathcal{F}, \overline{\mathcal{F}})$ ▷ Store previous values
5: **for each** $\Gamma^c \in \Gamma$ **do**
6: **for each** $E^c[i] \in E^c$ **do**
7: **for each** $E^c[i,j] \in E^c[i]$ **do**
8: ▷ Based on Eqs. (1), (2), (3), and (4),
9: Update $\mathcal{S}^c[i,j]$, $\overline{\mathcal{S}}^c[i,j]$, $\mathcal{F}^c[i,j]$, and $\overline{\mathcal{F}}^c[i,j]$
10: ▷ Using I_l in Eq. (6) with $I_l^0 = \overline{\mathcal{F}}^c[i,j] - \mathcal{S}^c[i,j]$,
11: $\overline{\mathcal{F}}^c[i,j] \leftarrow \overline{\mathcal{S}}^c[i,j] + I_l$
12: **if** $\overline{\mathcal{F}}^c[i,j] >$ absolute deadline **then**
13: $\overline{\mathcal{F}}^c[i,j] \leftarrow$ absolute deadline
14: Mark this job *unschedulable*
15: **if** $V_{prev} \neq (\mathcal{S}, \overline{\mathcal{S}}, \mathcal{F}, \overline{\mathcal{F}})$ **then**
16: $flag \leftarrow true$ ▷ Continue until converge
17: **until** $flag = true$
18: **end procedure**

C. Job start and finish time with preemption

Our analysis so far has assumed that there is no interference when deriving the start and finish time of a job (Sec. V-A) or the start and finish time is known when analyzing interference from higher-priority tasks (Sec. V-B). We now present Alg. 3 that bounds the start and finish time of each job in effective instance with interference.

Alg. 3 iteratively applies the equations provided in the previous subsections until the start and finish time of all jobs converge (lines 2-17). For each chain, effective instances and jobs within each instance are checked in chronological order. First, the start and finish time of each job is updated in line 9 based on the equations in Sec. V-A. This is to capture the results of preceding instances and jobs of the same chain. Then, in line 11, the upper bound on finish time is recalculated by the sum of $\overline{\mathcal{S}}^c[i,j]$ and I_l, where I_l is to capture interference due to the increased job timespan of $I_l^0 = \overline{\mathcal{F}}^c[i,j] - \mathcal{S}^c[i,j]$. Based on $\overline{\mathcal{F}}^c$, the algorithm checks if the job is not guaranteed to meet the deadline (line 12). Since a deadline-missing job is descheduled at runtime (see Sec. III), the job's finish time is capped to the absolute deadline and marked unschedulable.

D. End-to-end latency of effective instance

Since job-level dependencies are identified in Sec. IV-A, the maximum end-to-end latency L^c of a chain Γ^c is given by:

$$L^c = \max_{\forall i} \overline{\mathcal{F}}^c[i, N_c] - \mathcal{S}^c[i,1] \tag{7}$$

where N_c is the number of tasks in Γ^c and i is the index of effective chain instances of Γ^c.

Inter-chain distance. In addition to the end-to-end latency, it is often important to assess how often new, updated chain outputs are generated. Hence, we define the inter-chain distance of a chain as the distance between the completions of

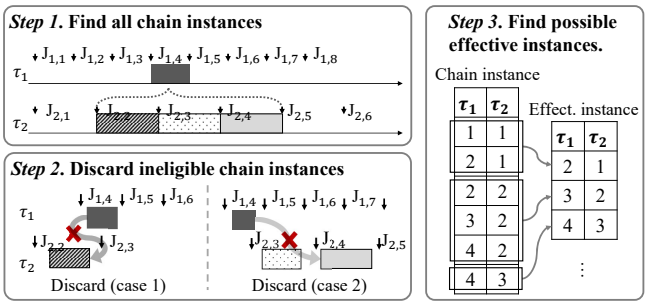


Fig. 5: Effective chain instances in chain-unaware scheduling

two adjacent effective instances. The maximum inter-chain instance $\mathcal{D}^c$ of Γ^c is bounded by:

$$\mathcal{D}^c = \max_{\forall i} \overline{\mathcal{F}}^c[i, N_c] - \mathcal{F}^c[i-1, N_c] \tag{8}$$

Revising effective instances. If any job of an effective instance is marked unschedulable, we revise the corresponding instance by substituting with another job of the same task, which is not a part of other effective instances of the same chain. If no other job can replace the deadline non-guaranteed job, that instance is marked as unschedulable and will not be considered for end-to-end analysis. However, if all effective instances of the chain are marked unschedulable, we try regenerating at least one effective instance that contains only the jobs guaranteed to meet the deadline. In this case, effective instances of other chains are also regenerated and Alg. 3 is ran again to capture interference correctly.

E. End-to-end latency of chain-unaware schedulers

Our analysis framework can be adapted for conventional chain-unaware schedulers with small modifications as below.

Part 1. Bounding job start and finish time. The start time of a job is bounded by taking the maximum among the release time of the job and the finish time of higher-priority jobs with execution overlaps. The finish time of a job is captured by considering the maximum interference from all higher-priority tasks. Then, these procedures are iterated until all start and finish time of individual jobs converge, as done in Alg. 3.

Part 2. Finding possible effective instances. Since chain-unaware schedulers do not respect job-level dependency, we find out all effective chain instances that can possibly happen at runtime. The high-level idea is depicted in Fig. 5. Among all chain instances where preceding and successor jobs have partial overlaps (Step 1), we discard obviously ineligible ones (Step 2), e.g., the start time of a preceding job greater than the finish time of a successor job. Then, the remainders are considered effective instances in our analysis (Step 3).

VI. Evaluation

We first evaluate the impact of execution time variations on end-to-end latency bounds under our analysis. We then compare our work against the start-of-the-art schemes, and explore the performance characteristics of the chain-based scheduler in a multi-core environment.

Taskset generation. We use randomly-generated tasksets for the evaluation. Based on the timing parameters of automotive benchmarks in [19], task period is chosen from $\{1, 2, 5, 10, 20, 50, 100, 200\}$ ms, with an associated probability of $\{0.04, 0.03, 0.03, 0.32, 0.32, 0.04, 0.21, 0.01\}$ for each period. For each taskset, task utilization is obtained by the UUniFast algorithm [6], then multiplied by the chosen period to obtain the worst-case execution time. The evaluation uses tasksets schedulable by RM because the latest work [2, 5] compared with our work assumes all tasks meet the deadlines.

Varying execution time. The difference between the best-case and the worst-case execution times (BC and WC) of a task can affect the tightness of analysis because these values are used to compute the range of start and finish time of each job. Thus, we evaluate the impact of varying BC under the chain-based scheduler (CBS) and the conventional chain-unaware RM scheduler with our analysis method (SFA-RM). We use 500 tasksets for each setting. The number of tasks N per taskset is selected from $\{3, 5, 7\}$ and each taskset has a single randomly-ordered chain including all N tasks.

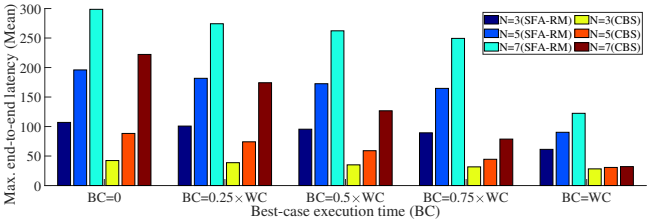


Fig. 6: End-to-end latency by best-case execution time

Fig. 6 shows the average of the maximum end-to-end latency with different BCs. As BC decreases (moving left on the x-axis), the bounded range of start and finish time of each job increases, resulting in increased timespan of chain instances. This result indicates that a tighter end-to-end bound can be achieved when the execution time is more deterministic and less varying. In the rest of this section, we choose BC=WC for the ease of analysis and simulation-based experiments.

Comparison with the state-of-the-art. We now compare our work with the latest approaches proposed for conventional chain-unaware scheduling [2, 5]. Below summarizes the list of methods used:

- **Abdullah et al.** [2]: the analysis of reaction latency of cause-effect chains in fixed-priority preemptive scheduling
- **Becker et al.** [5]: the analysis of end-to-end latency of cause-effect chains with specified job-level dependencies
- **SFA-RM**: Start- and Finish-time based Analysis under chain-unaware Rate Monotonic scheduling (our work)
- **CBS**: the proposed analysis framework under Chain-Based Scheduler (our work)

The comparison is carried out under two different chain setups: a single chain, and multiple chains with a mutual task. Since Abdullah et al. [2] assume preceding tasks in a chain should have lower priority than succeeding tasks in a multi-core system, we limit the comparison to a uniprocessor system.

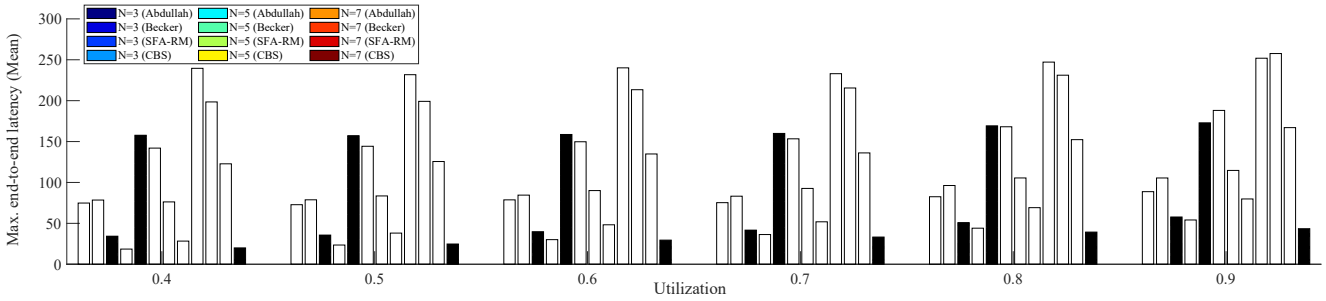


Fig. 7: End-to-end latency of a chain that consists of N tasks

We use 500 tasksets with 7 tasks each for each utilization level. Each taskset has a chain with N tasks, and tasks that do not belong the chain are hard real-time tasks, which are modeled as single-task chains in CBS. Fig. 7 shows the average of the maximum end-to-end latency for the chain with N tasks under the four approaches. As expected, we can observe the overall latency rises as N or the utilization increases. For CBS, the latency at $N = 5$ is higher than $N = 7$ because all jobs of higher-priority single-task chains are regarded as interference. Nonetheless, the results under our two approaches significantly outperform the others. In particular, when the utilization is 0.9 with $N = 7$, CBS yields 43 in the end-to-end latency while Abdullah et al. and Becker et al. have 251 and 257, respectively, i.e., CBS achieves up to 83% reduction in the end-to-end latency.

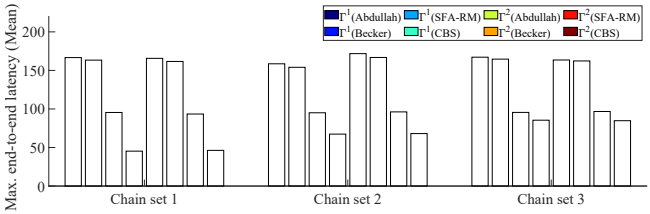


Fig. 8: Latency of multiple chains including a mutual task

Fig. 8 shows the average maximum latency for multiple chains that include a mutual task. Each taskset has the utilization of 0.8 with 9 tasks that form two chains. In order to identify the effect of the mutual task's position in a chain, three types of chain sets are considered. In *chain sets 1* and *2*, the two generated chains for each taskset share the start and the end task, respectively, and in *chain set 3*, an intermediate task is shared. As expected, we observe that our two approaches significantly outperform the others for all chain sets.

Maximum inter-chain distance. We now evaluate the maximum distance between chain instances under SFA-RM and CBS. For a comparison with simulation results, we use a smaller number of tasksets with specific parameters: 100 tasksets, each with utilization of 0.8 and 4 tasks. Task period is uniformly distributed in the range of $[1, 20]$.

Fig. 9 shows the results of the experiment (lower is better). Since our analysis considers all possible chain instances for SFA-RM, it cannot analytically bound inter-chain distance. Hence, we report only the maximum observed inter-chain

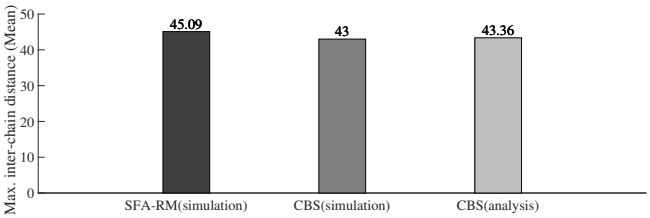


Fig. 9: Inter-chain distance under SFA-RM and CBS

distance in simulation. On the other hand, CBS can find the maximum inter-chain instance by using Eq. (8) and the results from both simulation and analysis are reported. As can be seen, CBS gives a smaller inter-chain distance than SFA-RM, meaning that updated end-to-end outputs are more frequently produced under CBS. This is interesting given that CBS skips some jobs that do not belong to effective instances.

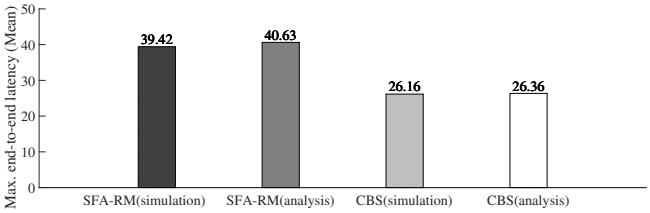


Fig. 10: Latency under SFA-RM and CBS

Fig. 10 shows the maximum end-to-end latency under the same experimental setting as the above. We observe that CBS outperforms SFA-RM in end-to-end latency and the pessimism of our analysis is very small for both SFA-RM and CBS.

End-to-end latency in multi-core processors. Lastly, we consider end-to-end latency in multi-core systems. We use 500 tasksets that are generated in the same way as in Fig 6 except that each taskset consists of 9 tasks with utilization of 0.9. In addition, each taskset has of two chains: $\Gamma^1 = [\tau_5, \tau_9, \tau_4, \tau_1, \tau_6]$ and $\Gamma^2 = [\tau_7, \tau_2, \tau_8, \tau_3]$. For task-to-core allocation, the worst-fit decreasing heuristic is used to balance load across cores.

Fig. 11 shows the maximum end-to-end latency of two chains in various multi-core settings. The latency of each chain reduces as the number of cores increases, because each core is less contended for by tasks. After all tasks are assigned to individual cores ($N = 9$), the end-to-end latency does not reduce any more. Based on all these experiments, we conclude

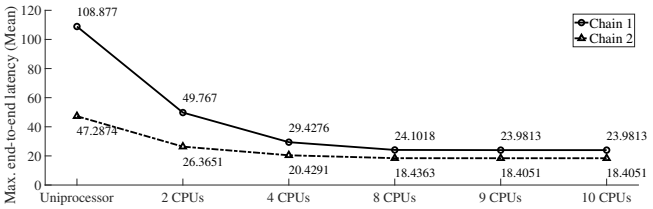


Fig. 11: End-to-end latency in uni- and multi-core systems

that our chain-based scheduler yields significant benefit in end-to-end latency of chained tasks without sacrificing the update rate of chain output.

VII. Conclusion

In this paper, we presented chain-based fixed-priority preemptive scheduling of tasks with loose data dependency. The proposed scheduler consists of offline and runtime parts. The offline part makes use of the notion of effective chain instances to capture job-level data dependency. At runtime, the scheduler governs the actual execution of each job based on the release-and-ready policy to improve the end-to-end latency of a chain. Our analysis framework bounds the end-to-end latency by analyzing the start and finish time of each job of effective instances in an iterative manner. Furthermore, it has been shown that our analysis can be easily adapted for chain-unaware schedulers. The evaluation results have demonstrated that our chain-based scheduler outperforms the state-of-the-art, achieving up to 83% of reduction in end-to-end latency, and yields a shorter update rate of chain output. For future work, we plan to apply the proposed scheduling approach to robotic platforms and weakly-hard real-time systems since our work can bring significant benefit to such systems. We are also interested in investigating the timing unpredictability caused by shared memory resources such as caches, memory buses, and DRAM banks, in multi-core platforms.

Acknowledgment

We gratefully acknowledge the support from the Office of Naval Research (ONR) grant N00014-19-1-2496.

References

[1] Apollo autonomous driving. http://apollo.auto, accessed May 2019.

[2] J. Abdullah, G. Dai, and W. Yi. Worst-case cause-effect reaction latency in systems with non-blocking communication. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.

[3] M. Asberg et al. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2013.

[4] O. Ayan et al. Age-of-information vs. value-of-information scheduling for cellular networked control systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2019.

[5] M. Becker et al. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016.

[6] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[7] J.-J. Chen et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

[8] H. Choi and H. Kim. Work-in-progress: A unified runtime framework for weakly-hard real-time systems. In *Brief Presentation Session of IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2019.

[9] H. Choi, H. Kim, and Q. Zhu. Job-class-level fixed priority scheduling of weakly-hard real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2019.

[10] A. Davare et al. Period optimization for hard real-time distributed automotive systems. In *Design Automation Conference (DAC)*, 2007.

[11] J. Goossens. (m, k)-firm constraints and dbp scheduling: impact of the initial k-sequence and exact schedulability test. In *International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[12] Z. A. Hammadeh et al. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[13] S. Han et al. Online scheduling switch for maintaining data freshness in flexible real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[14] S. Heo et al. RT-IFTTT: Real-time iot framework with trigger condition-aware flexible polling intervals. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[15] S. Kato et al. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018.

[16] S. Kaul, R. Yates, and M. Gruteser. Real-time status: How often should one update? In *IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[17] J. Kim et al. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[18] T. Kloda, A. Bertout, and Y. Sorel. Latency analysis for data chains of real-time periodic tasks. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 360–367, 2018.

[19] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[20] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[21] N. Lu, B. Ji, and B. Li. Age-based scheduling: Improving data freshness for wireless real-time traffic. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing (Mobihoc)*, 2018.

[22] J. Palencia and M. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium (RTSS)*, 1998.

[23] J. Palencia and M. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[24] P. Patel, I. Baek, H. Kim, and R. Rajkumar. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[25] P. Pazzaglia et al. Beyond the weakly hard model: Measuring the performance cost of deadline misses. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[26] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[27] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[28] J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2016.

[29] Y. Sun and M. D. Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):171, 2017.

[30] Y. Sun, E. Uysal-Biyikoglu, R. D. Yates, C. E. Koksal, and N. B. Shroff. Update or wait: How to keep your data fresh. *IEEE Transactions on Information Theory*, 63(11):7492–7508, 2017.

[31] Y. Xiang and H. Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.