IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 December 2022, accepted 21 February 2023, date of publication 2 March 2023, date of current version 8 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3251359

RESEARCH ARTICLE

Deadline Miss Early Detection Method for Mixed Timer-Driven and Event-Driven DAG Tasks

ATSUSHI YANO, (Member, IEEE), AND TAKUYA AZUMI, (Member, IEEE)
Graduate School of Science and Engineering, Saitama University, Saitama 338-8570, Japan

Corresponding author: Atsushi Yano (a.yano.578@ms.saitama-u.ac.jp)

This work was supported in part by the Japan Science and Technology agency (JST) Precursory Research for Embryonic Science and TechnOlogy (PRESTO) under Grant JPMJPR21P1.

ABSTRACT Self-driving systems have a hard real-time nature, and the control commands of the vehicle must be output before the end-to-end deadline using sufficiently new data. It must also have the functionality to quickly shift to a safe state in the event of a deadline miss. However, the current self-driving system can only detect a deadline miss at the end of the process. To further improve safety, a method for detecting the possibility of a deadline miss in the middle of the process is required. Therefore, we represent such a real-time system as a mixed timer-driven and event-driven directed acyclic graph (DAG) and propose an early detection method for deadline misses by deriving a time constraint for each node. The experimental evaluation shows that the proposed method can detect deadline misses early for various scheduling algorithms. It also shows that the deadline miss ratio can be reduced by prioritizing the scheduling of jobs with a small margin to the time constraint determined using the proposed method for each job.

INDEX TERMS DAG, deadline miss early detection, event-driven task, self-driving system, timer-driven task.

I. INTRODUCTION

Autonomous vehicles have recently attracted significant attention worldwide. The widespread use of autonomous vehicles should bring many benefits to society, such as road capacity, fuel efficiency, emissions, and accident risk [1]. According to the standards defined by the society of automotive engineers [2], there are six levels of self-driving, from zero to five. The higher the level of self-driving, the more safety is required of the self-driving system. In levels three and above, the self-driving system must operate to guarantee safety, even in emergencies. Therefore, to achieve level five (i.e., full driving automation), research and the development of self-driving systems are underway worldwide.

Self-driving systems require hard real-time performance. If critical processes, such as automatic braking and collision warning, are not processed in time, fatal accidents with loss of life can occur. Therefore, during the development phase, the behavior and processing must be determined statically, and processes must run from the acquisition of sensor data at the entry of the system exit by the deadline [3]. The system can be statically analyzed to meet the real-time requirements by representing the self-driving system as a directed acyclic graph (DAG) with an end-to-end deadline [4], [5].

For meeting an end-to-end deadline of a DAG, efficient scheduling algorithms have been actively studied [6], [7]. Since finding the optimal scheduling of a DAG is an NP-complete problem [8], the focus of studies is on designing heuristic solutions. An efficient heuristic scheduling algorithm that considers deadlines can reduce the possibility of deadline misses in the DAG [6], [7], [8], [9]. However, multi-/many-core processors (such as Kalray MPPA [10] and Intel Xeon [11]), which are used to execute self-driving systems, may miss deadlines due to contention for machine resources such as memory and buses [12]. Therefore, self-driving systems must have the functionality to quickly transition to a safe state in the event of a deadline miss.

To guarantee safety in emergencies, a minimum risk maneuver (MRM) mode is implemented in self-driving systems [13], [14]. In the self-driving industry, the risk avoidance state in the event of a deadline miss is called a minimum risk condition (MRC), and the steering wheel operation and brake control until the vehicle moves to MRC is called MRM. For example, when MRC is to park on the shoulder, MRM

The associate editor coordinating the review of this manuscript and approving it for publication was Abderrahmane Lakas.

is to move to a safe place and stop in the shortest possible time. If the self-driving system detects a deadline miss, the possibility of a fatal accident can be reduced by promptly shifting to MRM mode. However, the current self-driving system can only detect a deadline miss at the end of the processing flow.

To improve the safety of self-driving systems, the possibility of a deadline miss must be detected in the middle of the processing flow. Therefore, this paper proposes a method for early detection of a deadline miss in a self-driving system modeled as a DAG by appropriately allocating time constraints to each process in the middle based on the deadline given at the end of the processing flow (i.e., the output of the system).

A. CONTRIBUTIONS

The main contributions of this paper are as follows.

- First, we propose the concept of dividing a DAG consisting of nodes triggered by different periods and conditions into a set of nodes with a single period.
- Then, we statically analyze the dependencies between nodes with different startup rates and provide each node in the DAG model with reasonable time constraints for early detection of deadline misses.
- Furthermore, we experimentally show that the validity of the time constraints allocated to each node and the scheduling algorithm that prioritizes the margin until the time constraint can reduce deadline misses.

The remainder of the paper is organized as follows. Section II describes a system model. Section III defines the problem model addressed in this paper. Section IV introduces the proposed method. Section V evaluates the performance of the proposed method. Existing methods relevant to this study are discussed in Section VI. Finally, Section VII presents the conclusions and future work.

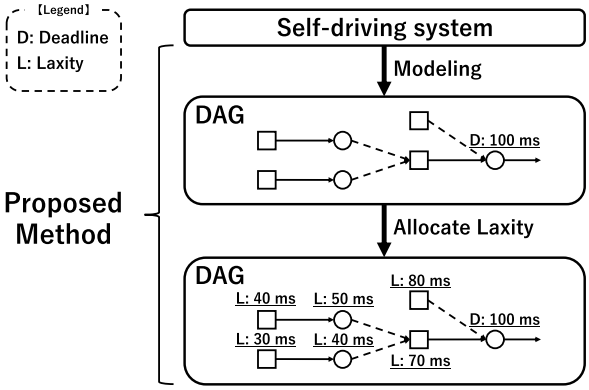


FIGURE 1. Overview of this paper.

TABLE 1. DAG notations.

Symbol	Description
$\|V\|$	Total number of nodes in a DAG
τ_i	Node
τ_i^{td}	*Timer-driven node*
τ_i^{ed}	*Event-driven node*
V^{td}	All *timer-driven nodes* in a DAG
T_i	Period of τ_i^{td}
ϕ_i	Offset of τ_i^{td}
$\langle \tau_i, k \rangle$	k-th job of τ_i
$\langle \tau_i^{td}, k \rangle$	k-th *timer-driven node* job of τ_i^{td}
$\langle \tau_i^{ed}, k \rangle$	k-th *event-driven node* job of τ_i^{ed}
ω_i	WCET of τ_i
$e_{i,j}^{tr}$	*Trigger edge* from τ_i to τ_j
$e_{i,j}^{up}$	*Update edge* from τ_i to τ_j
$comm_{i,j}$	Worst-case communication time of $e_{i,j}$
EG	Example of DAG (Fig. 2)

II. SYSTEM MODEL

This section describes the system model of this paper. This section first explains the overview of this paper as shown in Fig. 1. The target application is a self-driving system with an end-to-end deadline. This paper accurately models the constraints that a self-driving system must satisfy and the parallelism/dependencies among tasks as a DAG. To achieve early detection of deadline misses in a self-driving system, a threshold called *laxity* is calculated for each node based on the end-to-end deadline of such DAG.

A self-driving system in which processes are statically determined at the development stage can be represented as a DAG by representing each process as a node and the data flow to or from the processes as directed edges. Different types of sensor data, such as the point-cloud sensor (i.e., LiDAR), the global navigation satellite system (GNSS), and camera data, are acquired at different periods in self-driving systems. Using these sensor data, the self-driving system performs self-localization, object detection, and route planning, and finally outputs a single vehicle control command. Therefore, the DAG considered in this paper has a single exit node. Because the timing of the output of vehicle self-localization information and vehicle control commands are critical to a self-driving system's real-time performance, the nodes that output this information have a defined operating period.

A DAG consists of node and directed edge sets. Fig. 2 shows an example of the DAG considered in this paper, denoted as EG. The DAG notations in this paper are listed in Table 1. Here, a DAG is denoted as $G = (V, E)$, where V is the set of all nodes, expressed as $V = \{\tau_1, \ldots, \tau_{|V|}\}$, where $|V|$ is the total number of nodes. Here, there are two types of nodes: (i) ***timer-driven nodes*** that are triggered at predetermined periods, denoted by τ_i^{td}, and (ii) ***event-driven nodes*** that are triggered by receiving data from predecessor nodes, denoted by τ_i^{ed}. The set of all *timer-driven nodes* in the DAG is denoted by V^{td}. Each *timer-driven node* $\tau_i^{td} \in V^{td}$ is described by the tuple (T_i, ϕ_i), where T_i is the period of τ_i^{td},

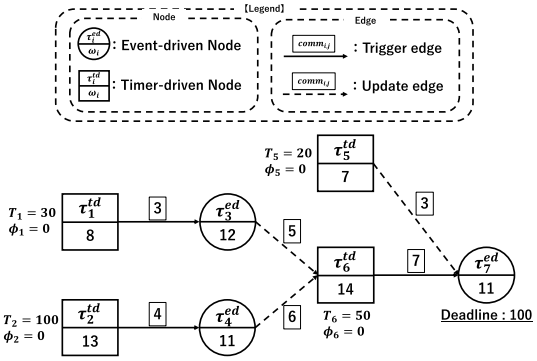


FIGURE 2. Example of DAG (*EG*).

and ϕ_i is the start time of the first execution of τ_i^{td} (i.e., offset). Each node has a worst-case execution time (WCET), denoted as ω_i. The k-th execution of τ_i is denoted by $\langle \tau_i, k \rangle$ (i.e., job). The WCET of the job is the same as that of the nodes.

E is the set of all edges in a DAG. Each edge $e_{i,j} \in E$ is the implicit communication [15] between τ_i and τ_j, and the data written by a job of τ_i are read by a job of τ_j. Since the buffer size for inter-node communication is one, old data are overwritten when the latest data arrive. There are two types of edges: (i) ***trigger edges***, which indicate that subsequent *event-driven nodes* are triggered when data arrives from the predecessor node, and are denoted by $e_{i,j}^{tr}$, and (ii) ***update edges***, which show that the data from the predecessor nodes are stored in memory and read when the subsequent nodes are triggered by a timer or *trigger edge*, and are denoted by $e_{i,j}^{up}$. $comm_{i,j}$ is the worst-case communication time for each edge. The static analysis must be used to determine the execution time of each node and the communication time of each edge. Although the worst-case value is used in this paper, other values (e.g., an average value or a best-case value) can be used depending on the user's objective.

In *EG*, the nodes $\{\tau_1^{td}, \tau_2^{td}, \tau_5^{td}, \tau_6^{td}\}$ are *timer-driven nodes*, the nodes $\{\tau_3^{ed}, \tau_4^{ed}, \tau_7^{ed}\}$ are *event-driven nodes*, the edges $\{e_{1,3}^{tr}, e_{2,4}^{tr}, e_{6,7}^{tr}\}$ are *trigger edges*, and the edges $\{e_{3,6}^{up}, e_{4,6}^{up}, e_{5,7}^{up}\}$ are *update edges*. τ_6^{td} uses data acquired from multiple predecessor nodes in different periods and outputs data in a fixed period. A real-world example of such a node is self-localization [16]. The self-localization module uses various data from LiDAR, cameras, GNSS, and IMU operating at different periods to estimate the position of the autonomous vehicle. The autonomous vehicle position is important information used by multiple subsequent modules (e.g., object detection and route planning) and must be output at a defined period to meet the requirements of subsequent modules. Thus, such nodes that need to output data independent of the timing of data arrival from their predecessor nodes are represented as *timer-driven nodes* in this paper.

III. PROBLEM MODEL

This section defines the constraints that must be satisfied by a self-driving system with a mix of *timer-driven* and *event-driven nodes* that are triggered at different periods. In a self-driving system, the timing of the generation of the data used to process the node (i.e., the timestamps of those data) must be considered in order to reduce the error in the output commands. Such timing analysis, however, is a complex problem because oversampling and undersampling occur when each node is triggered at a different period [17]. Therefore, Section III-A divides a single DAG into a set of DAGs that execute at the same period to define the complex constraints of the self-driving system and to facilitate analysis.

Section III-B then defines the constraints required for a self-driving system. The notations for the problem model are presented in Table 2.

TABLE 2. Problem model notations.

Symbol	Description
SG	Set of all *sub DAGs* in a single DAG
sg_i	*sub DAG*
Tsg_i	Period of sg_i
$getTimer(sg_i)$	Function that returns the index of the head *timer-driven node* of sg_i
$Join$	Set of all *join nodes* in a single DAG
$j\tau_i$	*Join node*
$t\tau_i$	*Tail node*
$succ^{tr}(\tau_i)$	Function that returns a set of nodes with a *trigger edge* from τ_i
$DFC[t\tau_i]$	*Data freshness* constraint of the data output from $t\tau_i$
$data(t\tau_i, k)$	Data output from the *tail node* job $\langle t\tau_i, k \rangle$
$stamp[data(t\tau_i, k)]$	Timestamp of $data(t\tau_i, k)$
D	End-to-end deadline
Tsg_{exit}	Period of the *sub DAG* including exit node
d_k	Deadline of k-th job of exit node

A. DIVIDING DAG INTO SUB DAGs

A set of nodes divided into a single-period is called a *sub DAG* and is denoted by $sg_i = \{\tau_x^{td}, \tau_y^{ed}, \ldots, \tau_z^{ed}\}$. Each *sub DAG* consists of one *timer-driven node* and zero or more *event-driven nodes*. The head *timer-driven node* of a *sub DAG* can have edges to multiple subsequent event-driven nodes. The *sub DAG* period is the same as the head *timer-driven node*. The period of sg_i is denoted by Tsg_i and is given by Eq. (1).

$$Tsg_i = T_{getTimer(sg_i)} \tag{1}$$

$getTimer(sg_i)$ is a function that returns the index of the head *timer-driven node* of sg_i.

The result of dividing *EG* into a set of *sub DAGs* is shown in Fig. 3. ***Join nodes*** are nodes that receive data from multiple *sub DAGs*; it is denoted by $j\tau_i$ ($j\tau_6$ and $j\tau_7$ in Fig. 3). The set of all *join nodes* in DAG is denoted by *Join*. ***Tail nodes*** are nodes with at least one edge to a *join node* in other *sub DAGs*; it is denoted by $t\tau_i$ ($t\tau_3$, $t\tau_4$, and $t\tau_5$ in Fig. 3). Note that there are nodes that are both *join nodes* and *tail nodes*. For example,

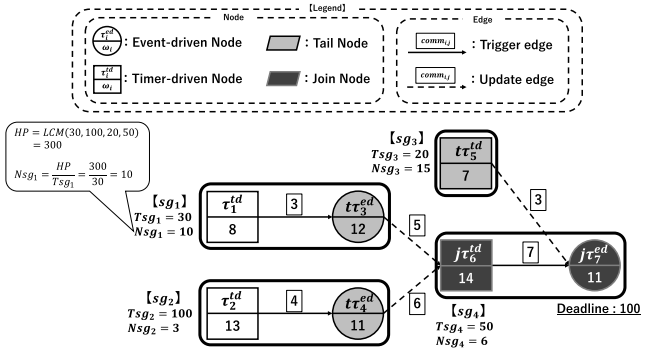


FIGURE 3. Dividing *EG* into a set of *sub DAGs*.

Algorithm 1 Dividing DAG Into *Sub DAGs*

Input: DAG, *Join*
Output: *SG*: Set of *sub DAGs* with no missing nodes and no duplicate nodes

1: $SG \leftarrow \emptyset$
2: **for all** $\tau_i^{td} \in V^{td}$ **do**
3: $sg \leftarrow \{\tau_i^{td}\}$ ▷ sg is a global variable
4: Search_succs(τ_i^{td})
5: $SG \leftarrow SG \cup sg$
6: **end for**

Algorithm 2 Search_succs(τ_i)

Input: τ_i

1: $succs \leftarrow succ^{tr}(\tau_i)$
2: **if** $succs = \emptyset$ **then**
3: return 0
4: **end if**
5: $sg \leftarrow sg \cup succs$
6: **for all** $\tau_j \in succs$ **do**
7: Search_succs(τ_j)
8: **end for**

if $j\tau_7$ in Fig. 3 has an edge to a successor *timer-driven node*, $j\tau_7$ is both a *join node* and a *tail node*.

The procedure of dividing DAG into *sub DAGs* is shown in Algorithm 1. The set of all *sub DAGs* in DAG is defined as *SG*. When Algorithm 1 finishes, *SG* contains all nodes in a DAG without duplication. Algorithm 1 calls the function *Search_succs()* shown in Algorithm 2 for all *timer-driven nodes* in DAG. Function *Search_succs()* is called recursively and adds successor nodes until there are no more *trigger edges* to successor nodes (Algorithm 2, lines 2–4). We assume that when $j\tau_i$ is an *event-driven node*, then there is only one *trigger edge* to $j\tau_i$. This is because more than 80% of the *event-driven nodes* in the target system, Autoware [16], have this specification. Therefore, when $j\tau_i$ is an *event-driven node*, $j\tau_i$ is included in only one *sub DAG* with a *trigger edge* to $j\tau_i$.

1) COMPLEXITY ANALYSIS

Algorithms 1 and 2 start at all *timer-driven nodes* and recursively search for subsequent nodes connected by *trigger edges*. From the assumption that there is only one *trigger edge* to each *event-driven node*, all nodes are searched exactly once. Thus, the time complexity of Algorithms 1 and 2 is $O(|V|)$. Similarly, the function *Search_succs()* is called only once for each node. Therefore, the number of call stacks of the function *Search_succs()* is maximized in the case of a linear DAG with a single *timer-driven node*, and the space complexity is $O(|V|)$.

B. PROBLEM DEFINITION

A self-driving system must satisfy the following constraints at runtime. Here, we assume that the system has a global clock.

1) DATA FRESHNESS CONSTRAINT

The *data freshness* constraint is an upper bound of the acceptable *data freshness* used in each *join node*. The older data used in each node in a self-driving system, the lower the accuracy of locating the vehicle and detecting obstacles [4], [18]. Therefore, sufficient new data must be used in each node. The *data freshness* is the amount of time elapsed since the timestamp associated with the data until the data are used. The timestamp of the data output by each entry node (i.e., sensor) is the execution start time of that entry node. The timestamp is carried over until the data are used by the *join node*. Meanwhile, the timestamp of the data output by a *join node* is updated to the execution start time of that *join node*, and the timestamp is carried over until the data are used by another *join node*. The *data freshness* constraint of the data output from $t\tau_i$ is denoted as $DFC[t\tau_i]$. Let us suppose that $\alpha \in \mathbb{R}^+$ and that sg_i is the *sub DAG* containing $t\tau_i$, $DFC[t\tau_i]$ is defined by Eq. (2).

$$DFC[t\tau_i] = \alpha \times Tsg_i \tag{2}$$

$DFC[t\tau_i]$ is determined according to Tsg_i. The smaller the value of α, the stricter the *data freshness* constraint, and the larger value of α, the older data are acceptable. The user can flexibly change the value of α according to the system requirements. A *join node* can be executed if there are data that meets the *data freshness* constraints from all predecessor nodes at the time the *join node* is triggered.

An example of *data freshness* constraint for $t\tau_4^{ed}$ and $j\tau_6^{tr}$ in *EG* when $\alpha = 1$, as shown in Fig. 4. The data output from the *tail node* job $\langle t\tau_i, k\rangle$ is denoted as $data(t\tau_i, k)$, and the timestamp of $data(t\tau_i, k)$ is denoted as $stamp[data(t\tau_i, k)]$. Using Eq. (2), we obtain that $DFC[t\tau_4]$ is 100 ms. Since $stamp[data(t\tau_4, 1)]$ is 0 ms, the job of $j\tau_6$ that starts execution by 100 ms ($stamp[data(t\tau_4, 1)] + DFC[t\tau_4]$) can use the data output from $\langle t\tau_4, 1\rangle$. Therefore, $data(t\tau_4, 1)$ can be used for $\langle j\tau_6, 2\rangle$ which starts at 50 ms and $\langle j\tau_6, 3\rangle$ which starts at 100 ms. In contrast, $\langle j\tau_6, 4\rangle$ cannot use $data(t\tau_4, 1)$ because it starts at 150 ms. Instead, it uses $data(t\tau_4, 2)$, which is newer data.

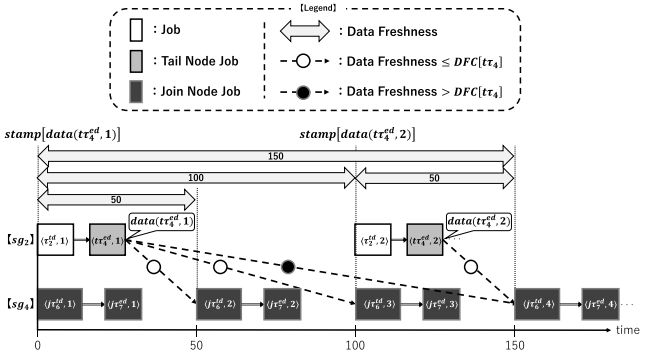


FIGURE 4. Data transmission and reception between sg_2 and sg_4 in *EG*.

TABLE 3. Notations in Section IV.

Symbol	Description
HP	Hyper-period of a DAG
JLD	Set of all the job-level dependencies in HP
$(\langle \tau_i, k\rangle \rightarrow \langle \tau_j, s\rangle)$	job-level dependency from $\langle \tau_i, k\rangle$ to $\langle \tau_j, s\rangle$
Nsg_i	Number of executions during HP for sg_i
$RST(\tau_i, k)$	Reference start time of $\langle \tau_i, k\rangle$
$RFT(\tau_i, k)$	Reference finish time of $\langle \tau_i, k\rangle$
$pred^{tr}(\tau_i)$	Function that returns a set of nodes with a *trigger edge* to τ_i
$succ(\tau_i, j)$	Function that returns a set of jobs with a job-level dependency from $\langle \tau_i, j\rangle$

2) END-TO-END DEADLINE

An end-to-end deadline is defined as the time it takes from the system's input (i.e., sensor) to the system's final output (i.e., control command). D represents the end-to-end deadline assigned to the exit node. For example, in *EG*, the exit node is τ_7^{ed}, and the end-to-end deadline is determined to be 100 ms. d_k represents the deadline given to the k-th job of the exit node. Because the execution of the exit node is dependent on the period of the *sub DAG* that includes the exit node, let Tsg_{exit} be the period of the *sub DAG* that includes the exit node and d_k is calculated as follows using Eq. (3):

$$d_k = D + (k-1) \times Tsg_{exit}. \tag{3}$$

The self-driving system must complete the execution of the exit node job through the end-to-end deadline while satisfying the *data freshness* constraints in each job. Therefore, this paper analyzes the relationship in which a predecessor node job supplies data satisfying the *data freshness* constraint to a subsequent node job (called job-level dependencies hereafter). Then, based on the obtained job-level dependencies, a reasonable time constraint is given to each job to satisfy the end-to-end deadline.

IV. PROPOSED METHOD

This paper proposes a static analysis method to compute reasonable time constraints for detecting the deadline miss at each node for an end-to-end deadline of the exit node. The proposed method consists of three main steps: (i) calculating the reference values of the start and finish time of each job, (ii) determining job-level dependencies, and (iii) allocating the time constraint to each job. The subsequent sections describe the steps of the proposed method in detail. The notations used in this section are presented in Table 3.

A. CALCULATING REFERENCE VALUES OF START AND FINISH TIME OF EACH JOB

In the first step, calculate the reference values of the start and finish times of all jobs to analyze the job-level dependencies. This step allows finding the write/read relationship of data between jobs and determining whether the data meets the *data freshness* constraint.

To cover all job-level dependencies during the running of the system, the least common multiple (LCM) of the period of *timer-driven nodes* (i.e., hyper-period) in a DAG [18] must be considered. The hyper-period of a DAG is denoted by HP and calculated using Eq. (4).

$$HP = LCM_{\forall \tau_i^{td} \in V^{td}}(T_i) \tag{4}$$

Job-level dependencies can be determined by calculating the reference values of start and finish times of all jobs in HP. The reference start time of a *timer-driven node* job $\langle \tau_i^{td}, k\rangle$ is calculated using Eq. (5), given as follows:

$$RST(\tau_i^{td}, k) = k \times T_i + \phi_i. \tag{5}$$

When $\langle \tau_i, k\rangle$ is a *timer-driven node*, $RST(\tau_i^{td}, k)$ coincides with the release time. The reference finish time of each *timer-driven node* job $\langle \tau_i^{td}, k\rangle$ is the sum of the reference start time and WCET. It is calculated using Eq. (6), given as follows:

$$RFT(\tau_i^{td}, k) = RST(\tau_i^{td}, k) + \omega_i. \tag{6}$$

The reference start time of an *event-driven node* job $\langle \tau_i^{ed}, k\rangle$ is calculated using Eq. (7), given by

$$RST(\tau_i^{ed}, k) = \max_{\tau_a \in pred^{tr}(\tau_i^{ed})} \Big(RFT(\tau_a, k) + comm_{a,i} \Big). \tag{7}$$

When $\langle \tau_i, k\rangle$ is an *event-driven node*, $RST(\tau_i^{ed}, k)$ is the time when the data were received from the predecessor job with the *trigger edge* to $\langle \tau_i, k\rangle$. Similarly, the reference finish time of each *event-driven node* job $\langle \tau_i^{ed}, k\rangle$ is calculated using Eq. (8), which is given as follows:

$$RFT(\tau_i^{ed}, k) = RST(\tau_i^{ed}, k) + \omega_i. \tag{8}$$

Note that the reference start and reference finish times calculated using the above equations may be delayed if the core is not idle when each job is triggered. Whether the proposed method correctly detects deadline misses early is evaluated in Section V.

B. DETERMINING JOB-LEVEL DEPENDENCIES

In the second step, job-level dependencies are defined in detail, and job-level dependencies of each job are analyzed. The data flow to each exit node job with an end-to-end

Algorithm 3 Determine Job-Level Dependencies

Input: *SG*, *HP*, *Join* ▷ *SG*: Set of all sub DAGs
Output: *JLD*: All the job-level dependencies during *HP*

1: $JLD \leftarrow \emptyset$ ▷ Store Job-level dependencies
2: $RSTset \leftarrow \emptyset$ ▷ Store the reference start time of each job
3: $RFTset \leftarrow \emptyset$ ▷ Store the reference finish time of each job
4: **for all** $sg_i \in SG$ **do**
5: $n \leftarrow Nsg_i$ (Eq. (9))
6: $JLD \leftarrow JLD \cup$ job-level dependencies within the sg_i
7: **for** $k \leftarrow 1 \ldots n$ **do**
8: **for all** $\tau_a \in sg_i$ **do**
9: $RSTset \leftarrow RSTset \cup RST(\tau_a, k)$ (Eqs. (5) or (7))
10: $RFTset \leftarrow RFTset \cup RFT(\tau_a, k)$ (Eqs. (6) or (8))
11: **end for**
12: **end for**
13: **end for**
14: **for all** Pairs of *tail node* job $\langle t\tau_i, k\rangle$ and *join node* job $\langle j\tau_j, s\rangle$ with edges included in different *sub DAGs* **do**
15: **if** $RFT(t\tau_i, k) \in RFTset$ and $RST(j\tau_j, s) \in RSTset$ satisfy conditions of *Definition* 1 **then**
16: $JLD \leftarrow JLD \cup (\langle t\tau_i, k\rangle \rightarrow \langle j\tau_j, s\rangle)$
17: **end if**
18: **end for**

deadline can be clarified by determining all job-level dependencies in *HP*. Due to the involvement of jobs on the data flow in the execution of exit node jobs, deadline misses can be detected early based on the execution timing of these jobs at runtime.

First, determine the job-level dependencies within each *sub DAG*. The set that stores the job-level dependencies in *HP* is denoted by *JLD*, and the job-level dependency from $\langle \tau_i, k\rangle$ to $\langle \tau_j, s\rangle$ is denoted by $(\langle \tau_i, k\rangle \rightarrow \langle \tau_j, s\rangle)$. Within a single *sub DAG*, the node-level dependencies (i.e., edges) are directly job-level dependencies. This procedure is illustrated by $sg_2 = \{\tau_2^{td}, \tau_4^{ed}\}$ of *EG*. Here, *HP* of *EG* is 300 ms according to Eq. (4). The number of executions during *HP* for sg_i is denoted as Nsg_i and calculated using Eq. (9).

$$Nsg_i = \frac{HP}{Tsg_i} \tag{9}$$

As a result, $Nsg_2 = \frac{300}{100} = 3$, and $\{(\langle \tau_2, 1\rangle \rightarrow \langle \tau_4, 1\rangle), (\langle \tau_2, 2\rangle \rightarrow \langle \tau_4, 2\rangle)$, and $(\langle \tau_2, 3\rangle \rightarrow \langle \tau_4, 3\rangle)\}$ are added to *JLD*. Because Nsg_i is always an integer by definition, no floor/ceil-operators are required.

For the model considered in this paper, the key problem is the job-level dependency from *tail node* jobs to *join node* jobs. Therefore, job-level dependencies are defined based on the reference start time, reference finish time, and *data freshness* constraints. The job-level dependency from *tail node* jobs to *join node* jobs is defined as follows:

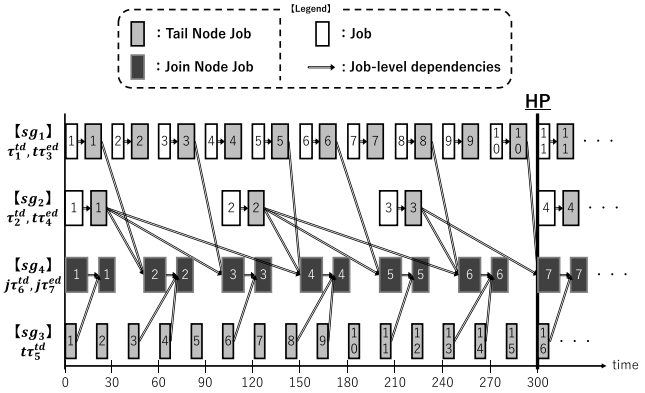


FIGURE 5. Job-level dependencies in *EG* when $\alpha = 1.7$ in Eq. (2) for all *join nodes*.

Definition 1: Suppose that $e_{\tau_i,\tau_j} \in E$ *and* $t\tau_i$ *is a tail node, and* $j\tau_j$ *is a join node. There exists a job-level dependency from* $\langle t\tau_i, k\rangle$ *to* $\langle j\tau_j, s\rangle$ *that is if*

1) $RFT(t\tau_i, k) + comm_{i,j} \leq RST(j\tau_j, s)$ *and*
2) $RST(j\tau_j, s) - stamp[data(t\tau_i, k)] \leq DFC[t\tau_i]$.

Here, suppose that $\langle t\tau_i, k\rangle \in sg_i$, $stamp[data(t\tau_i, k)]$ is given by Eq. (10).

$$stamp[data(t\tau_i, k)] = RST\,getTimer(sg_i) \tag{10}$$

$RST\,getTimer(sg_i)$ is the reference start time of the head *timer-driven node* of the sg_i containing $\langle t\tau_i, k\rangle$. The *join node* job can use data that arrive from *tail node* jobs before the reference start time (condition 1 of *Definition* 1) and satisfy the *data freshness* constraint (condition 2 of *Definition* 1).

The job-level dependencies between *tail node* jobs and *join node* jobs can be determined using *Definition* 1 and Eqs. (5), (6), (7), and (8). This procedure is shown in Algorithm 3. First, reference start and finish times are calculated for all jobs that occur within *HP* (Algorithm 3, lines 4–13). Next, job-level dependencies within a single *sub DAG*, (i.e., other than dependencies between *tail node* jobs and *join node* jobs), are stored in *JLD* (Algorithm 3, line 6). Finally, the dependencies between the *tail node* job and the *join node* job are determined based on the calculated reference start and finish times (Algorithm 3, lines 14–18).

1) COMPLEXITY ANALYSIS

The total number of edges in the input DAG and the maximum number of jobs in one node are denoted as $|E|$ and $|J|$, respectively. Here, let *SG* be the set of all *sub DAGs*, $|J| = \max_{sg_i \in SG} Nsg_i$. The time complexity of lines 4–13 in Algorithm 3 is $O(|V| \times |J|)$, and since the maximum number of pairs of *tail node* and *join node* is $|E|$, the time complexity of lines 14–18 in Algorithm 3 is $O(|E| \times |J|^2)$. Since the input graph is a multi-period DAG, $|E| \geq |V| - 2$ and $|J| \geq 2$. Therefore, $O(|E| \times |J|^2) > O(|V| \times |J|)$, and the overall time complexity of Algorithm 3 is $O(|E| \times |J|^2)$. The space complexity of Algorithm 3 depends on the number of job-level

dependencies stored in *JLD*; thus, the space complexity is $O(|E| \times |J|^2)$.

The job level dependencies in *EG* after applying Algorithm 3 to *EG* when $\alpha = 1.7$ in Eq. (2) for all *join nodes* are shown in Fig. 5.[1] Note that there are jobs that have dependencies with multiple subsequent jobs (e.g., $\langle \tau_4^{ed}, 1 \rangle$), and jobs that do not have dependencies with subsequent jobs (e.g., $\langle \tau_3^{ed}, 2 \rangle$). Having no dependency with successor jobs indicates that the job's sent data do not satisfy the *data freshness* constraint for any successor jobs.

C. ALLOCATING TIME CONSTRAINT TO EACH JOB

In the third step, a reasonable time constraint (i.e., *laxity*) is given to each job based on *JLD* and the end-to-end deadline. The *laxity* value represents the margin to the end-to-end deadline for each job [9], [19] and can be calculated by recursively subtracting the communication time between jobs and the job execution time from the data flow's exit node job deadline. Therefore, the *laxity* value of any job is the start time threshold to meet the end-to-end deadline for the exit node job at the end of the data flow connected by job-level dependencies. At runtime, if a job's start time exceeds the statically computed *laxity* value, it can be predicted that successor jobs will not be supplied with data to satisfy the *data freshness* constraint, resulting in a deadline miss.

Eq. (11) defines the *laxity* of the k-th exit node job.

$$laxity(\tau_i, k) = d_k - \omega_i \tag{11}$$

$laxity(\tau_i, k)$ indicates the latest time at which $\langle \tau_i, k \rangle$ must be started. If *laxity* is calculated using WCET, a deadline miss invariably occurs for a given d_k when the execution start of $\langle \tau_i, k \rangle$ is later than $laxity(\tau_i, k)$.

The *laxity* of the k-th job except for the exit node is defined in Eq. (12).

$$laxity(\tau_i, k) = \min_{\langle \tau_a, b \rangle \in succ(\tau_i, k)} \Big(laxity(\tau_a, b) - comm_{a,i} \Big) - \omega_i \tag{12}$$

When there are multiple successor jobs, the *laxity* is calculated based on the successor job with the minimum margin. This is to ensure that all successor jobs are started before the *laxity*. Note that if there are no subsequent jobs (i.e., no dependencies are determined for the job), the *laxity* cannot be calculated and *NULL* is returned.

From the features of *laxity* and *Definition* 1, the *laxity* value for each job can be used as a reasonable time constraint for early detection of deadline misses. When the actual start time of job execution exceeds the statically calculated *laxity* value at runtime, the proposed method provides early detection and notification of future deadline misses. The behavior of the system following notification of deadline miss early detection is beyond the scope of this paper and is not discussed.

Let m be the remainder of dividing the index of each job by Nsg_i. The calculation results of *laxity* for the example in Fig. 5 are presented in Table 4. Jobs that do not have dependencies with subsequent jobs do not have the *laxity*. As shown in Fig. 5, the dependencies of each job are repeated in *HP*. Therefore, Table 4 can be used to calculate the *laxity* value of all jobs after *HP* (e.g., $laxity(\tau_2, 4) = 84 + 300 = 384$ ms).

[1] We assume that the first job of *join nodes* is also supplied with data that satisfy the *data freshness* constraint.

TABLE 4. Calculation results of *Laxity* for each job of *EG*.

sub DAG	node	m	*laxity* - $n \times HP$ [ms]	node	m	*laxity* - $n \times HP$ [ms]
sg_1	τ_1	1	90	τ_3	1	101
		2	-		2	-
		3	140		3	151
		4	-		4	-
		5	190		5	201
		6	240		6	251
		7	-		7	-
		8	290		8	301
		9	-		9	-
		10	340		10	351
sg_2	τ_2	1	84	τ_4	1	101
		2	184		2	201
		3	284		3	301
sg_3	τ_5	1	79	τ_5	9	229
		2	-		10	-
		3	129		11	279
		4	129		12	-
		5	-		13	329
		6	179		14	329
		7	-		15	-
		8	229			
sg_4	τ_6	1	68	τ_7	1	89
		2	118		2	139
		3	168		3	189
		4	218		4	239
		5	268		5	289
		6	318		6	339

m: "job index" $mod\ Nsg_i$
n: $\lfloor$ "job index" $/\ Nsg_i \rfloor$

Early detection of deadline misses is possible by having information on the number of executions and *laxity* for each node in the system. From a practical perspective, when *HP* becomes huge or the number of nodes in a DAG is large, it is difficult to keep these values in each node due to overhead and memory constraints. After implementing the proposed method, only the entry node of each path needs to have a *laxity* value. Each node subtracts the communication time (i.e., the time between a current time and the transmission time of the predecessor node) from the received *laxity*, and detects a deadline miss if the current time exceeds the *laxity* value. After each node finishes executing, each node subtracts the execution time from the value of *laxity* it holds and transfers the *laxity* value and transmission time to successor nodes. Therefore, the proposed method can be easily implemented even in large-scale systems.

D. THE WAY TO USE THE PROPOSAL LAXITY VALUE

This section describes two possible uses of the *laxity* values calculated in the three steps of the proposed method (Sections IV-A, IV-B, and IV-C). The first use is a start time threshold of each job at runtime for early detection of deadline misses, as described in Section IV-C. For the DAG that represents the system, the proposed method calculates *laxity* statically for all jobs with job-level dependency in *HP*, as shown in Table 4. When the actual start time of the job execution exceeds the value of its *laxity*, early detection of a deadline miss is performed at runtime.

The priority of each job in scheduling is the second application. The *laxity* is a metric that represents the margin to the end-to-end deadline for each job, and heuristic task scheduling algorithms that prioritize the *laxity* have been proposed [9], [19]. Prioritizing jobs with lower margins, i.e., jobs with lower *laxity* values, lower the *deadline miss ratio*. The scheduling algorithm for prioritizing jobs with small *laxity* values calculated by the proposed method is called the proposed least *laxity* first (LLF).

V. EVALUATION

This section experimentally demonstrates the performance of early detection of deadline misses and proposed LLF using *laxity* calculated by the proposed method. In the experiment, proposed LLF and the following three scheduling algorithms are used: (a) earliest deadline first (EDF) [20], i.e., higher priority for jobs with shorter time to implicit deadline, (b) LLF algorithm when calculating the *laxity* at each node of a single-period DAG transformed using the method proposed by Saidi et al. [21], and (c) *laxity*-based heuristic scheduling algorithm for multi-period DAGs proposed by Igarashi et al. [9]. In the algorithms (a) and (c), which do not consider *event-driven nodes*, the *event-driven nodes* are assumed to be *timer-driven nodes* that execute in the period of the *sub DAG* to which they belong. Here, all jobs are dynamically scheduled and executed non-preemptively[2] [22], [23].

A. EXPERIMENTAL SETUP

The performance evaluation is conducted using the reference system of Autoware.Auto[3] [24], which is an open-source self-driving system and DAGs randomly generated by task graphs for free (TGFF) [25] converted to the model considered in this paper.

The reference system converted to the DAG model considered in this paper is shown in Fig. 6. One node in ROS is converted to one node in a DAG, and nodes that output data periodically (i.e., nodes with timer callbacks) are represented as *timer-driven nodes*, while other nodes are represented as *event-driven nodes*. All data inputs to the *timer-driven node* are considered *update edges*, and if the *event-driven node* has only one data input, it is considered a *trigger edge*. When an event-driven node has multiple inputs, the edge from the predecessor node with the longest period is represented as a *trigger edge* and the remaining edges are *update edges*. The period of the *timer-driven nodes* is set to a default value, and the WCET of each node and the worst-case communication time are set by referring to the value measured by CARET.[4] All numerical values in Fig. 6 are in milliseconds.

The experimental parameters and their values are presented in Table 5. In all experiments, the first end-to-end deadline of each DAG is set to be the maximum period of the *timer-driven nodes* in that DAG. The simulation framework is written in Python and executed on a system with the following configuration: (i) Intel® Core i7-10875H CPU @2.30 GHz, (ii) 16 GiB memory, and (iii) Ubuntu 20.04 LTS OS (64-bit).

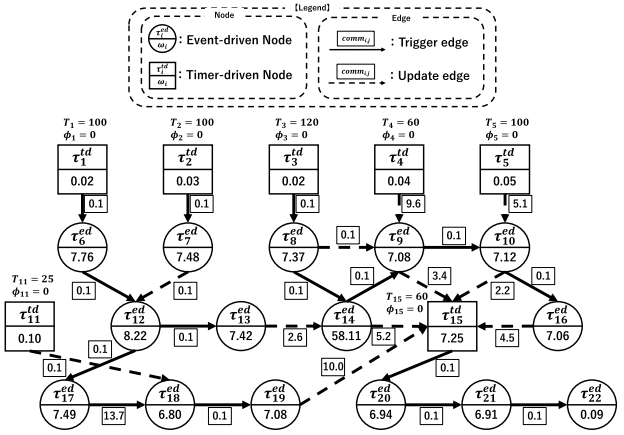


FIGURE 6. Reference system converted to DAG.

TABLE 5. Experimental parameters.

Section V-C	
Number of cores	8
Average CPU usage [%]	65, 70, 75, 80, 85, 90, 95
α in Eq. (2) for all *join nodes*	2.0, 2.1, 2.2, 2.3, 2.4, 2.5
Section V-D	
Number of cores	2, 3, 4, 5, 6, 7, 8
α in Eq. (2) for all *join nodes*	1.0, 1.2, 1.4, 1.6, 1.8, 2.0
Number of nodes	10–500
Number of entry nodes	3, 4, 5
Number of exit nodes	1
Period of *timer-driven nodes* [ms]	10, 20, 30, 40, 50, 60, 80, 100, 120

B. PERFORMANCE METRICS

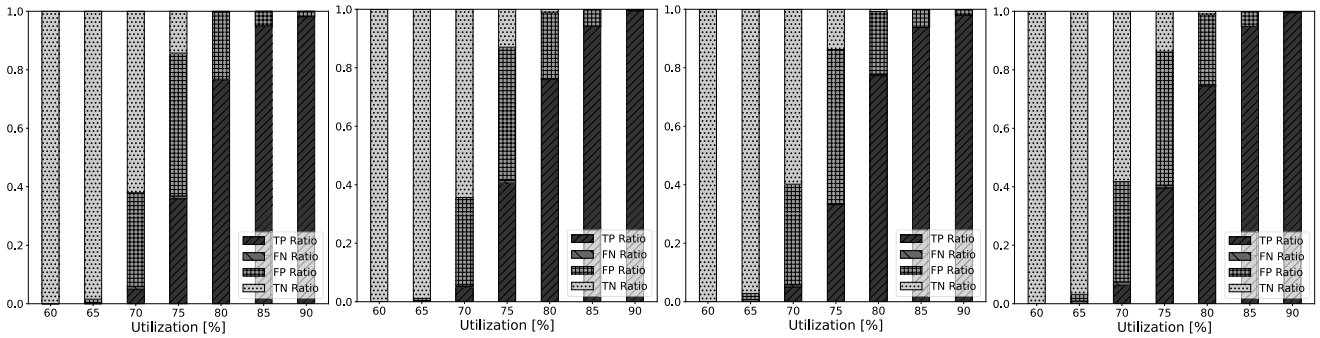
This section presents the metrics used in the evaluation. First, four terms used in the calculation of metrics are listed.

- *True Positive (TP):* The TP is the case in which a deadline miss is detected early because the actual execution start time of the job exceeded its *laxity* value at runtime, and the deadline miss actually occurred when the process was continued. The TP shows that the proposed method accurately and early detects that a deadline miss occurs.
- *False Positive (FP):* The FP is the case in which a deadline miss is detected early, but no deadline miss occurred when the process was continued. The FP indicates that early detection of deadline misses is wrong. In self-driving systems, a large FP ratio harms normal driving because it increases the number of safety functionalities (e.g., MRM mode) that are mistakenly activated even though no deadline miss has occurred.

[2]The proposed method is applicable regardless of preemptive or non-preemptive.

[3]https://github.com/ros-realtime/reference-system

[4]https://github.com/tier4/caret



(a) EDF (with proposed threshold). (b) Saidi et al. [21] (with proposed threshold). (c) Igarashi et al. [9] (with proposed threshold). (d) Proposed LLF.

FIGURE 7. TP, FP, FN, and TN ratios with increasing utilization.

- *False Negative (FN):* The FN is the case in which a deadline miss is not detected early, but a deadline miss occurred in subsequent processing. The FN indicates the failure of the proposed method to detect deadline misses early.
- *True Negative (TN):* The TN is the case in which a deadline miss is not detected early and did not occur in subsequent processing.

The proposed method's performance is measured using the metrics listed below.

- ***Accuracy:*** The *Accuracy* is the ratio of accurate classifications to all predictions made by the proposal *laxity* and is defined as follows.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{13}$$

The *Accuracy* is 1.0 if a deadline miss occurs in all cases where a deadline miss is detected early and if no deadline miss occurs in all cases where no deadline miss is detected early ($0 \leq Accuracy \leq 1.0$).

- ***Precision:*** The *Precision* is the ratio of actual deadline misses to cases where deadline misses are detected early by the proposal *laxity*. The *Precision* is calculated by Eq. (14).

$$Precision = \frac{TP}{TP + FP} \tag{14}$$

- ***Recall:*** The *Recall* is the ratio of cases where a deadline miss is detected early in the proposal *laxity* to cases where a deadline miss occurs. Eq. (15) describes the *Recall*.

$$Recall = \frac{TP}{TP + FN} \tag{15}$$

- ***F-measure:*** The *F-measure* is the harmonic mean of the *Precision* and the *Recall* and is derived in Eq. (16).

$$F-measure = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{16}$$

Since the *Recall* and the *Precision* are in a trade-off relationship, the balance between the *Recall* and the *Precision* is evaluated by the *F-measure*.

- ***Deadline Miss Ratio:*** The *deadline miss ratio* is the percentage of actual deadline misses at runtime.
- ***Earlier Time:*** The *earlier time* is the time of earlier detection of a deadline miss in the TP case.
- ***Run Time:*** The *run time* is the time taken by the proposed method.
- ***Memory Usage:*** The *memory usage* the peak memory usage of the proposed method. The peak memory usage is measured using *memory_profiler*[5] library in Python.

C. EXPERIMENT USING REFERENCE SYSTEM OF AUTOWARE.AUTO

This section quantitatively evaluates whether the proposed *laxity*, regardless of the scheduling algorithm, allows for the early detection of deadline misses. The *laxity* value of each job in *HP* used as a threshold for early detection is calculated based on WCET. During scheduling simulation, the execution time of each job is varied between the best-case execution time and WCET using measured data from the reference system's actual operation.

Experiment 1: The changes in the TP, FP, FN, and TN ratios for 15,000 runs when α in Eq. (2) is set to a random value in the range of 2.0 to 2.5 and the CPU utilization is randomly increased with eight processor cores are shown in Fig. 7. The "(with proposed threshold)" suffix in the figures indicates that the proposed *laxity* value is used as a threshold for early detection of deadline misses. As can be seen, deadline misses starts to occur at the utilization of 65%, and the more utilization increases, the more deadline misses occur. The *Accuracy*, *Precision*, *Recall* and *F-measure* are calculated from the values of TP, FP, FN, and TN at each utilization in Fig. 7.

First, *Accuracy* results as utilization increases are shown in Fig. 8. The *Accuracy* decreases from 65% utilization to its lowest value at 75%, after which it increases as utilization increases. The proposed *laxity* is calculated using WCET, which accounts for this result. The proposed *laxity* is pessimistic because most jobs are executed in less execution time

[5]https://github.com/pythonprofilers/memory_profiler

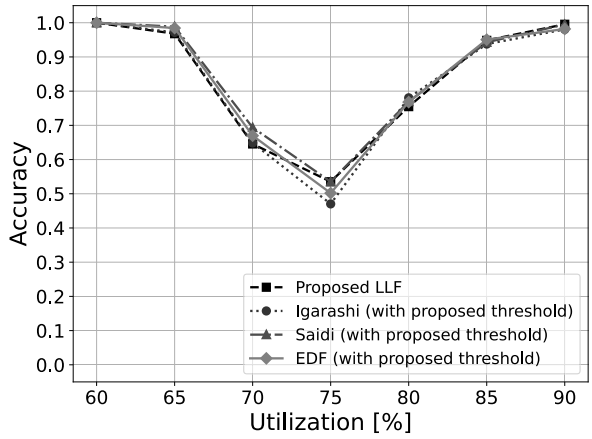


FIGURE 8. ***Accuracy*** **result with increasing utilization.**

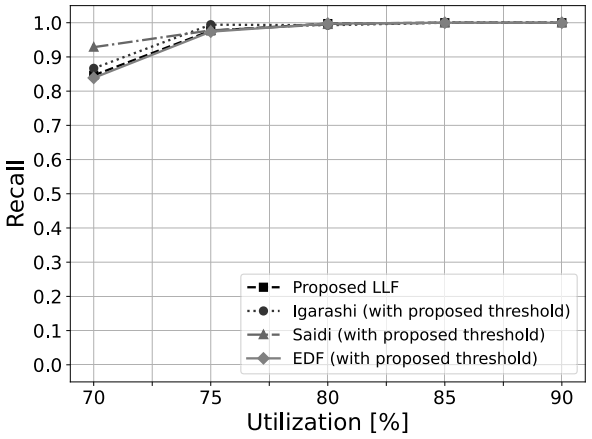


FIGURE 10. ***Recall*** **result with increasing utilization.**

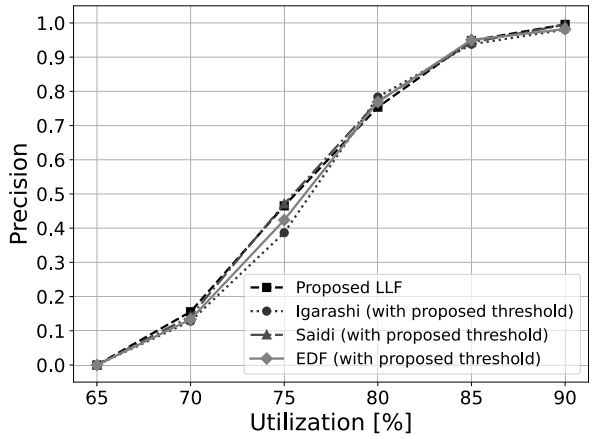


FIGURE 9. ***Precision*** **result with increasing utilization.**

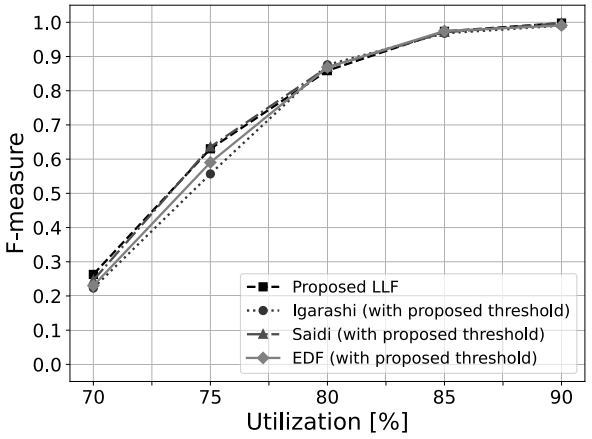


FIGURE 11. ***F-measure*** **result with increasing utilization.**

than WCET at runtime. As a result, 65% to 75% of utilization, FP cases occur frequently and *Accuracy* declines. At 80% or higher utilization, the *Accuracy* increases because more cases of deadline miss occur and FP becomes TP. The *Accuracy* is approximately 100% for any scheduling algorithm with the utilization of 90%. Thus, the more stressed the CPU resources are, the more accurate the prediction based on the proposed *laxity* will be.

The *Precision* results as utilization increases are shown in Fig. 9. The *Precision* increases almost monotonically with increasing utilization in all algorithms. This is also due to the proposed *laxity* being calculated using the pessimistic WCET. The mean values of the *Precision* for all utilization in algorithms (a), (b), (c), and (d) are 0.55, 0.54, 0.55, and 0.54, respectively, indicating that the early detection of deadline misses is correct more than 50% on average.

The *Recall* results as utilization increases are shown in Fig. 10. As can be seen, the *Recall* is greater than 0.8 at all utilizations and nearly equal to 1.0 at 75% and above. In other words, there are few cases in which an actual deadline miss cannot be detected early as it occurs (i.e., the FN cases). In the FN case, the self-driving system detects the deadline misses only at the end of the process, increasing the risk that the vehicle will not be transferred to a safe state in time, resulting in an accident. As a result, the proposed method's high *Recall* is critical for improving the safety of self-driving systems. The few FN cases that exist here are caused by differences between reference and actual start/finish times. Because of these errors, which are caused by CPU resource status at runtime, the job-level dependencies statically analyzed by the proposed method are not satisfied, resulting in deadline misses.

TABLE 6. ***Earlier time*** **result.**

	Average [ms]	Maximum [ms]
EDF	71	458
Saidi et al. [21]	54	392
Igarashi et al. [9]	42	273
Proposed LLF	50	328

Finally, the *F-measure* results as utilization increases are shown in Fig. 11. The *F-measure*, as well as the *Precision* and the *Recall*, increases monotonically with increasing utilization. Thus, the proposed *laxity's* classification performance improves with increased utilization.

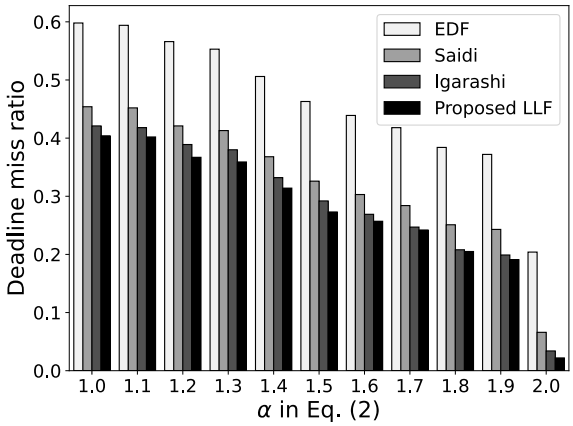


FIGURE 12. ***Deadline miss ratio*** **with different values of α in Eq. (2) for all** ***join nodes.***

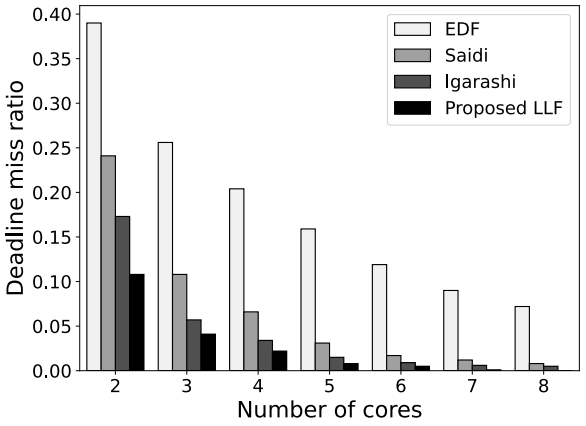


FIGURE 13. ***Deadline miss ratio*** **with the different number of cores.**

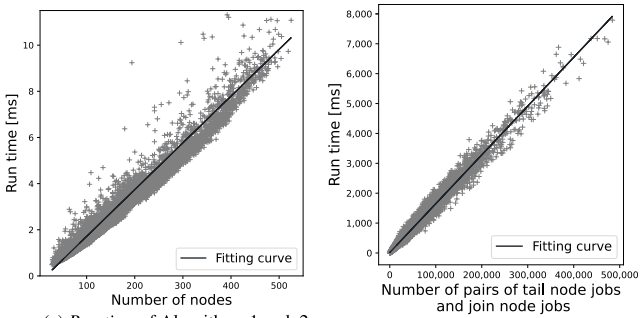
Experiment 2: The *earlier time* results of each algorithm for 15,000 runs are presented in Table 6. The average *earlier time* for algorithms (a), (b), (c), and (d) is approximately 71, 54, 42, and 50 ms, respectively. Here, when an autonomous vehicle travels at 50 mph,[6] it moves forward more than 3.0 ft[7] in 42 ms. Therefore, early detection of a deadline miss is an essential factor in preventing accidents.

D. EXPERIMENT USING RANDOM DAGs

This section compares the *deadline miss ratio* of the proposed LLF, which prioritizes jobs with a smaller proposed *laxity*, with the existing algorithm using random DAGs.

Experiment 3: The *deadline miss ratio* results of each algorithm for 1,000 random DAGs with four processor cores and varying the value of α in Eq. (2) for all *join nodes* are shown in Fig. 12. As can be seen, for any value of α, the proposed LLF achieves the lowest *deadline miss ratio*. As the value of α increases, i.e., the *data freshness* constraint becomes looser, the *deadline miss ratio* becomes smaller for all algorithms. When α is 2, the *deadline miss ratio* for algorithms other than EDF is less than 10%. Meanwhile, if the value of α increases further, the *data freshness* constraint is met in any scheduling order. Therefore, if the user does not want to allow nodes in the system to use old data, the value of α should be at most 2.

Experiment 4: The *deadline miss ratios* for each algorithm when the value of α in Eq. (2) for all *join nodes* are set to 2, and the number of processor cores is varied from 2 to 8 in 1,000 random DAGs are shown in Fig. 13. As can be seen, the proposed algorithm achieves the lowest *deadline miss ratio* on any number of cores. EDF shows a significantly higher *deadline miss ratio* than other algorithms. This is because EDF executes jobs that output data that are not used in the execution of subsequent jobs (i.e., overwritten data). Algorithms except for EDF can lower the *deadline miss ratio* because they statically analyze the job-level dependency and prioritize jobs that output data necessary for subsequent job execution.



(a) *Run time* of Algorithms 1 and 2. (b) *Run time* of Algorithm 3.

FIGURE 14. ***Run time*** **results of algorithms.**

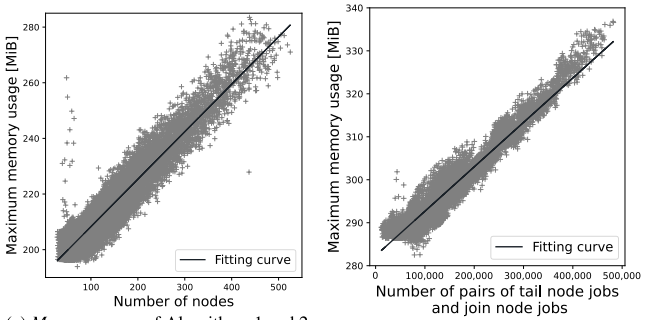
TABLE 7. **Overall *run time* result.**

Average [ms]	Minimum [ms]	Maximum [ms]
2,560.5829	0.0006	20,214.0857

Experiment 5: The *run time* results of each algorithm for 10,000 random DAGs are shown in Fig. 14. As can be seen, the *run time* in Algorithms 1 and 2 increases linearly with the number of nodes in the DAG. Even with 500 nodes, Algorithms 1 and 2 complete processing in approximately 10 ms. The *run time* of Algorithm 3 is shown to be proportional to the number of pairs of *tail node* jobs and *join node* jobs. This value is the number of loops in line 14 of Algorithm 3. Algorithm 3 can be computed in approximately 8,000 ms at a maximum. The *run time* result for the overall proposed method is shown in Table 7. The average *run time* and maximum *run times* are approximately 2,560 and 20,214 ms, respectively, to compute the *laxity* value for all jobs in *HP*. Since this calculation is performed statically, a maximum *run time* of 20,214 ms is acceptable. Therefore, the proposed method can be applied to systems with hundreds of nodes.

Experiment 6: The *memory usage* results of each algorithm for 10,000 random DAGs are shown in Fig. 15.

[6]approximately 80 kph.
[7]approximately 0.96 m.



(a) *Memory usage* of Algorithms 1 and 2.

(b) *Memory usage* of Algorithm 3.

FIGURE 15. ***Memory usage*** **results of algorithms.**

TABLE 8. **Overall *memory usage* result.**

Average [MiB]	Minimum [MiB]	Maximum [MiB]
312.4327	283.4893	410.9432

Similar to *run time*, *memory usage* for Algorithms 1 and 2 is proportional to the number of nodes, and *memory usage* for Algorithm 3 is proportional to the number of pairs of *tail node* jobs and *join node* jobs. Algorithms 1 and 2 use at most approximately 283 MiB of memory, and Algorithm 3 uses at most 338 MiB. The *memory usage* result for the overall proposed method is shown in Table 8. The average *memory usage* and maximum *memory usage* are approximately 312 and 410 MiB, respectively. 410 MiB is sufficiently acceptable, and all of the proposed *laxity* is calculated statically, with only a reference to its value at runtime. Thus, the proposed method is applicable to systems with 500 nodes in terms of memory usage.

The experimental results demonstrate that the *laxity* value calculated by the proposed method can be used for the early detection of deadline misses in arbitrary scheduling algorithms. Additionally, scheduling jobs with smaller *laxity* values more preferentially is shown to reduce the *deadline miss ratio* more than existing algorithms for both the reference system of Autoware.Auto and random DAGs.

VI. RELATED WORK

This section introduces existing methods related to the proposed method and compares them. Table 9 compares the proposed method with existing methods.

A. DETERMINATION OF JOB-LEVEL DEPENDENCIES IN MULTI-PERIOD DAG

In a DAG consisting of multiple dependent periodic tasks, determining job-level dependencies allows the worst-case end-to-end latency and task prioritization to be derived. Determining job-level dependencies in *HP* is synonymous with converting a multi-period DAG to a single-period DAG, and various methods have been proposed.

The conversion of multi-period DAGs to single-period DAGs is based on the period of the task that transmits and receives the data [4], [17], [18], [21]. Becker et al. [17] proposed a heuristic solution to determine job-level dependencies to meet the end-to-end timing requirements of the system. They calculate all possible job-level dependencies using the earliest/latest data read times and earliest/latest data output times for each periodic task. Saidi et al. [21] proposed a method to convert a multi-period DAG into a single-period DAG for correct data exchange between tasks operating at different periods. They proposed the equations that derive the number of edges to be added between two nodes based on the period of the predecessor and successor nodes. By traversing all node pairs in a multi-period DAG and applying these equations, the conversion to a single-period DAG is achieved. Verucchi et al. [18] proposed a method to convert one multi-period DAG into multiple single-period DAGs and obtained the DAG that optimizes the schedulability, the age latency, and the reaction latency. Their method generates a set of single-period DAGs in the following four steps: (i) generate all jobs triggered in *HP*, (ii) add synchronization nodes that guarantee job periods and deadlines, (iii) compute permutations of all possible combinations of dependencies between jobs and map each pattern to one single-period DAG, (iv) remove redundant edges. Kordon and Tang [4] proved that the maximum age latency could be calculated by replicating each job of a multi-period DAG a fixed number of K times. They first proved that the maximum age latency can be computed by extending all nodes of the original DAG in *HP*. To reduce the computational complexity, they proved that the maximum age latency can be computed by only partially extending the original DAG, and proposed an algorithm to minimize this extension number K.

Although these methods properly determine job-level dependencies for multi-period DAGs, they consider DAGs consisting only of periodic tasks. Therefore, these studies cannot be directly applied to the model with a mixture of

TABLE 9. **Proposed method vs. existing methods.**

	DAG	MPA	EDP	MTE	DED
QL-HEFT [26]	✓				
HMDS [8]	✓				
Becker et al. [17]	✓	✓			
Saidi et al. [21]	✓	✓	✓		
Verucchi et al. [18]	✓	✓	✓		
Kordon et al. [4]	✓	✓	✓		
Klaus et al. [15]	✓	✓	✓		
Igarashi et al. [9]	✓	✓	✓		
SAFLA [27]	✓	✓		✓	
PiCAS [23]	✓	✓		✓	
Casini et al. [5]	✓	✓	✓	✓	
Tang et al. [3]	✓	✓	✓	✓	
Blaß et al. [28]	✓	✓	✓	✓	
Proposed method	✓	✓	✓	✓	✓

MPA: Multi-period application
EDP: Nodes with edges from multiple predecessor nodes triggered at different periods.
MTE: Mixture of *Timer-driven* and *event-driven nodes*
DED: Deadline miss early detection method

timer-driven and *event-driven nodes*, which is considered in this paper.

B. SCHEDULING FOR MULTI-PERIOD DAG

In a system consisting of multiple periodic tasks, scheduling while preserving all job-level dependencies in multi-core platforms is a complex problem. Klaus et al. [15] argued for a tradeoff between tight data age guarantees, synchronization overhead, and schedulability in multi-core environments. They also proposed a solution to protect the job-level dependencies without explicit synchronization. Their method is the first to consider a cross-chain; however, the chain consists of only periodic tasks. In the *RTSS 2021 Industry Challenge* [22], scheduling strategies and analysis techniques for multi-period DAGs with various timing constraints have been proposed. The model of the presented self-driving system has the following constraints, including the *data freshness* and end-to-end deadline constraints considered in this paper: (i) control commands must be output within a defined time from the input of sensor data, (ii) control commands must be generated using sufficiently fresh data (iii) the difference in the timestamps of the data must not exceed a predefined threshold in components that merge different sensor data.

The above studies consider models in which all tasks in the DAG execute periodically. However, in reality, the module considered as one task in *RTSS 2021 Industry Challenge* consists of multiple tasks [16]. Since these tasks can be regarded as a chain of *event-driven nodes* triggered by *timer-driven nodes*, the model targeted by this paper must be considered.

C. METHODS CONSIDERING CHAINS CONSISTING OF TIMER-DRIVEN AND EVENT-DRIVEN NODES

Research on processing chains with a mixture of *timer-driven* and *event-driven* nodes has been conducted in the field of the robot operating system (ROS) [29], which is widely used as a robot development support framework. Casini et al. [5] first proposed a response time analysis of the ROS 2 processing chain. They also proposed a real-time scheduling model for ROS 2. Blaß et al. [28] extended existing work [5] to include a response time analysis that considered both execution time variance and scheduler starvation. They incorporated into the analysis execution time showing the cumulative execution time of n consecutive instances of the task to address execution time variance. In addition, the pessimism of the existing analysis was reduced by utilizing the feature that ROS 2 Executor is starvation-free. Tang et al. [3] also identified flaws in existing work [5] and improved the accuracy of the response time analysis by taking into account the behavior of the ROS 2 executor. They further demonstrated experimentally that raising the priority of the last task in the processing chain can reduce not only the upper bound of response time but also the actual response time. Choi et al. [23] proposed a priority-driven chain-aware scheduler (PiCAS) for ROS 2 in a multi-core environment. PiCAS traverses all chains in ascending priority order, and each task in a chain is assigned a higher priority the closer it is to the tail of the chain. Experimental results demonstrate that PiCAS can reduce end-to-end latency better than the ROS default scheduler.

These studies provide accurate timing analysis and schedulers that reduce end-to-end latency in ROS environments. However, these studies have not determined the job-level dependencies. Therefore, their methods cannot be used directly for the early detection of deadline misses.

D. FIXED PRIORITY HEURISTIC SCHEDULING ALGORITHMS

The *laxity* is also used as a priority in heuristic list-based scheduling. Igarashi et al. [9] proposed a heuristic scheduling algorithm on a clustered many-core platform that avoids contentions using the logical execution time model. Their target model is a mixture of *timer-driven* and *event-driven nodes*. However, their algorithm cannot strictly consider the job-level dependencies because they consider *event-driven nodes* to be *timer-driven nodes* triggered by the largest period of the predecessor nodes.

A similar concept to *laxity* is *rank* [30], and many authors have proposed scheduling algorithms that use *rank* values as priorities. Zhao et al. [26] proposed QL-HEFT that uses reinforcement learning to compute *rank* values in a cloud computing environment. Since reinforcement learning is difficult to fall into locally optimal solutions due to its policy of maximizing the sum of long-term rewards, QL-HEFT performed better than existing studies, especially for large-scale DAGs. Senapati et al. [8] proposed a low-overhead heuristic scheduling algorithm called HMDS on heterogeneous platforms. HDMS determines priority using a metric called the predicted finish time, which estimates the total cost required to complete the execution of all dependent nodes of a given task. In addition, HMDS incorporates four pruning mechanisms and outperforms existing algorithms with low computational complexity. QL-HEFT and HMDS are designed for single-period DAGs and do not apply to multi-period DAGs.

Co-scheduling of multiple independent DAGs has been studied to increase the efficiency of resource use in cyber-physical systems consisting of multiple distributed subsystems. Roy et al. [27] first proposed a co-scheduling method for multiple independent single-period DAGs on a heterogeneous platform called SAFLA. SAFLA generates a static schedule that minimizes energy consumption while avoiding shared bus contention and guaranteeing deadlines for all DAG instances during an *HP*. Evaluation using benchmarks demonstrated that deadline miss ratios and energy consumption of SAFLA were lower than existing methods under various scenarios. However, SAFLA does not consider intermediate *timer-driven nodes* such as τ_6^{td} in *EG*.

VII. CONCLUSION

In this paper, we have proposed a static analysis method for early detection of deadline misses for real-time systems with *data freshness* constraints and end-to-end deadlines. The

proposed method can reasonably allocate the *laxity* to each job in a DAG with such constraints, which is a mixture of *timer-driven* and *event-driven nodes*. The experimental evaluation showed that the proposed method could detect deadline misses accurately and early. Additionally, the scheduling algorithm with the *laxity* as priority calculated using the proposed method could reduce the *deadline miss ratio* compared to existing algorithms.

In future work, we will develop a method for early detection of deadline misses that considers the effects of the reference start time and finish time delays for each job. We will also consider the difference in timestamps of the data from the predecessor job at the *join node* job [22], [31]. Dealing with the increased computational complexity of the proposed method due to extremely long *HP* is also part of our plan.

REFERENCES

[1] D. Milakis, B. van Arem, and B. van Wee, "Policy and society related implications of automated driving: A review of literature and directions for future research," *J. Intell. Transp. Syst.*, vol. 21, no. 4, pp. 324–348, Jul. 2017.
[2] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, SAE Int., Warrendale, PA, USA, 2018.
[3] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ROS2 executors," in *Proc. RTSS*, 2020, pp. 231–243.
[4] A. Kordon and N. Tang, "Evaluation of the age latency of a real-time communicating system using the LET paradigm," in *Proc. ECRTS*, 2020, pp. 1–21.
[5] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *Proc. ECRTS*, 2019, pp. 1–23.
[6] S. K. Roy, R. Devaraj, A. Sarkar, and D. Senapati, "SLAQA: Quality-level aware scheduling of task graphs on heterogeneous distributed systems," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5, pp. 1–31, Sep. 2021.
[7] S. K. Roy, R. Devaraj, A. Sarkar, K. Maji, and S. Sinha, "Contention-aware optimal scheduling of real-time precedence-constrained task graphs on heterogeneous distributed systems," *J. Syst. Archit.*, vol. 105, May 2020, Art. no. 101706.
[8] D. Senapati, A. Sarkar, and C. Karfa, "HMDS: A makespan minimizing DAG scheduler for heterogeneous distributed systems," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5s, pp. 1–26, Oct. 2021.
[9] S. Igarashi, Y. Kitagawa, T. Ishigooka, T. Horiguchi, and T. Azumi, "Multi-rate DAG scheduling considering communication contention for NoC-based embedded many-core processor," in *Proc. DS-RT*, Oct. 2019, pp. 1–10.
[10] T. Azumi, Y. Maruyama, and S. Kato, "ROS-lite: ROS framework for NoC-based embedded many-core platform," in *Proc. IROS*, Oct. 2020, pp. 4375–4382.
[11] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon phi," in *Proc. ICPE*, Mar. 2014, pp. 137–148.
[12] S. Igarashi, T. Fukunaga, and T. Azumi, "Accurate contention-aware scheduling method on clustered many-core platform," *J. Inf. Process.*, vol. 29, pp. 216–226, Mar. 2021.
[13] B. Coll-Perales, J. Schulte-Tigges, M. Rondinone, J. Gozalvez, M. Reke, D. Matheis, and T. Walter, "Prototyping and evaluation of infrastructure-assisted transition of control for cooperative automated vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 7, pp. 6720–6736, Jul. 2022.
[14] J. Schindler, R. Markowski, D. Wesemeyer, B. Coll-Perales, C. Böker, and S. Khan, "Infrastructure supported automated driving in transition areas—A prototypic implementation," in *Proc. CAVS*, Nov. 2020, pp. 1–6.
[15] T. Klaus, M. Becker, W. Schröder-Preikschat, and P. Ulbrich, "Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads," in *Proc. RTAS*, May 2021, pp. 66–79.
[16] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proc. ICCPS*, Apr. 2018, pp. 287–296.
[17] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proc. RTCSA*, Aug. 2016, pp. 159–169.
[18] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate DAGs from multi-rate task sets," in *Proc. RTAS*, Apr. 2020, pp. 226–238.
[19] Y. Suzuki, T. Azumi, N. Nishio, and S. Kato, "HLBS: Heterogeneous laxity-based scheduling algorithm for DAG-based real-time computing," in *Proc. CPSNA*, Oct. 2016, pp. 83–88.
[20] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. New York, NY, USA: Springer, 1998.
[21] S. E. Saidi, N. Pernet, and Y. Sorel, "Automatic parallelization of multi-rate FMI-based co-simulation on multi-core," in *Proc. TMS*, 2017, pp. 1–13.
[22] L. Shaoshan, Y. Bo, G. Nan, D. Zheng, and A. Benny, "Industry challenge," in *Proc. Ind. Session (RTSS)*, 2021, pp. 1–47.
[23] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New design of priority-driven chain-aware scheduling for ROS2," in *Proc. RTAS*, May 2021, pp. 251–263.
[24] T. Shuhei and T. Azumi, "ROS 2 framework for embedded multi-core platform," in *Proc. APRIS*, 2021, pp. 29–30.
[25] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Workshop CODES/CASHE*, 1998, pp. 97–101.
[26] Z. Tong, X. Deng, H. Chen, J. Mei, and H. Liu, "QL-HEFT: A novel machine learning scheduling scheme base on cloud computing environment," *Neural Comput. Appl.*, vol. 32, no. 10, pp. 5553–5570, May 2020.
[27] S. K. Roy, R. Devaraj, and A. Sarkar, "SAFLA: Scheduling multiple real-time periodic task graphs on heterogeneous systems," *IEEE Trans. Comput.*, early access, Jul. 22, 2022, doi: 10.1109/TC.2022.3191970.
[28] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *Proc. RTSS*, Dec. 2021, pp. 41–53.
[29] A.-M. Hellmund, S. Wirges, Ö. S. Taş, C. Bandera, and N. O. Salscheider, "Robot operating system: A modular software framework for automated driving," in *Proc. ITSC*, Nov. 2016, pp. 1564–1570.
[30] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[31] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "ROSCH: Real-time scheduling framework for ROS," in *Proc. RTCSA*, Aug. 2018, pp. 52–58.

ATSUSHI YANO (Member, IEEE) received the B.E. degree from the Graduate School of Science and Engineering, Saitama University, in 2019, where he is currently pursuing the master's degree. His research interests include embedded systems and real-time scheduling.

TAKUYA AZUMI (Member, IEEE) received the Ph.D. degree from the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists with the Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor with the College of Information Science and Engineering, Ritsumeikan University. From 2014 to 2018, he was an Assistant Professor with the Graduate School of Engineering Science, Osaka University. He is currently an Associate Professor with the Graduate School of Science and Engineering, Saitama University. His research interests include real-time operating systems and component-based development. He is a member of ACM, IPSJ, IEICE, and JSSST.