Chain-Aware Scheduling for Mixed Timer-Driven and Event-Driven DAG Tasks

Daichi Yamazaki[1,a)] Takuya Azumi[1]

Abstract: Embedded systems, such as self-driving systems, periodically execute tasks to accurately recognize an external environment. To meet the deadline of embedded systems, the directed acyclic graph (DAG) is used for scheduling in existing studies. However, the DAG of the self-driving system is complex because the DAG is comprised of timer-driven tasks, triggered by period and event-driven tasks triggered by arriving input data. Because of periodic output, timer-driven tasks exist not only at the beginning of the end-to-end path but also in the middle. The existing studies have not considered the scheduling of the DAG that has multiple timer-driven tasks in the middle of the end-to-end path. To solve this problem, we propose DAG scheduling using chains. A chain is a sequence of tasks with data dependencies, and chains are triggered periodically. By dividing a DAG into chains and scheduling by chains, the proposed method can statically consider the scheduling order. Moreover, if the period of a task is larger than that of the predecessor task, the output of the predecessor task may not be used. Tasks whose output is not used are excluded from scheduling to execute other tasks. As a result of deleting tasks that do not contribute to the exit chains in the system, the schedulability is improved compared with a method based on an existing algorithm without deletion.

Keywords: DAG, node-reduction, multi-rate, schedulability

1. Introduction

Recently, embedded systems are active in various fields such as self-driving systems. These embedded systems have to execute tasks by deadlines. If a task misses the deadline, the process is delayed, and an accident happens. Studies of scheduling have been conducted for meeting the deadline, and response time becomes shorter [1]. However, scheduling becomes difficult because modern real-time systems have become larger and more complex [2, 3]. One approach to deal with such a problem is to model the system as a directed acyclic graph (DAG). A DAG represents tasks as nodes, with data dependencies replaced by edges. A data dependency is a relationship in which a node is triggered after all inputs have been received. By using DAG, the complex data dependencies can be mitigated, and the end-to-end latency can be shortened by assigning priorities to nodes [4–6]. However, the scheduling of a self-driving system, such as Autoware [7], is difficult because of multiple sensor nodes [8].

The self-driving system has multiple sensors, such as LiDAR and a camera for localization and perception. These sensor nodes are triggered at different periods. A node triggered by a period is called a timer node, and the successor node triggered by arriving input data is called an event node. A successor node is a node connected behind the edge of DAG. Sensor nodes are timer nodes because sensors are triggered periodically. A self-driving system has to execute timer nodes and successor event nodes having data dependencies periodically. The different periods make scheduling difficult because the triggering timing of each task is shifted. One of the solutions to this problem is using the hyper-period which is the least common multiple of the periods [9–11]. However, in order to periodically output data even if no input data come in, timer nodes exist in the middle of the end-to-end [7, 12]. The existing studies have not been considered in timer nodes in the middle of end-to-end.

Considering scheduling for a self-driving system, which has timer nodes in the middle of the end-to-end path of the system, this study proposes a scheduling method based on chains. A chain is a sequence of nodes in a row, following successor data dependencies. This proposed method decides a scheduling order for each chain to satisfy period constraints and data dependencies. Moreover, if the period of one node is larger than the predecessor node, the output of the predecessor node may not be used by not executing the successor node. By not executing the number of times the output is not used, the number of nodes to be considered in scheduling is decreased. This paper proposes a method of identifying the index whose output of a node is not used. The contributions of this study are as follows.

- The scheduling algorithm for a mixed timer and event-driven DAG with timer nodes in an end-to-end node sequence is considered.
- The reduction of execution of nodes that handle data not used as input to the execution of the exit node is considered.
- The improvement in schedulability and reduction in computation time by eliminating output not used for successor nodes is shown experimentally.

The remainder of this study is structured as follows. Section 2 describes the system model. Section 3 defines how to divide a DAG into chains. Section 4 explains the worst start time and node reduction. Section 5 discusses the evaluation. Section 6 presents related work. Section 7 describes the conclusion.

[1] Saitama University, Saitama-shi, Saitama-ken 338–8570, Japan
[a)] d.yamazaki.554@ms.saitama-u.ac.jp

2. System Model

The system model considered in the proposed method is shown in this section. A set of nodes with parameters and modeling of DAG are also defined. Moreover, this section defined a chain that is required to simplify the DAG mixed timer and event nodes.

2.1 Node Set

This section defines the system's nodes used in the derivation of scheduling. Nodes have two types of triggering. The first is the triggering by a period that the node has, and the node execution is periodically completed. A sensor node corresponds to this. Such nodes are called timer nodes, and the set of timer nodes is called V_{tm}. The assumption of this study is that the timers in timer nodes are synchronized. The second is a type of triggering by arriving at the input data. Such nodes are called event nodes, and the set of event nodes is represented as V_{ev}. The event node does not have a period. Event nodes that require multiple inputs are triggered when both inputs are received. For multiple inputs, an input with a smaller period can reach the node before an input with a larger period reaches it. Therefore, the node can be considered to be triggered when the input with a larger period arrives.

A set of all nodes in the system is represented as V. Executable nodes are contained in V, and V is expressed as $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$ in **Fig. 1**. Each node has parameters such as ($T(v_i)$, $wc(v_i)$) where $T(v_i)$ is a period. Nodes such as sensor nodes are launched in a period. $wc(v_i)$ is the worst-case execution time (WCET) of v_i. The node execution time is varied by the runtime environment because cache usage and data dependencies will affect the execution time. The execution time of a node does not exceed WCET. In this study, the nodes' execution time is fixed at WCET to simplify the study target.

2.2 Directed Acyclic Graph

To consider a system easily, the proposed method models it with a DAG. A DAG comprises nodes and edges to focus on data dependencies between the nodes. An example of a DAG model is shown in Fig. 1. A DAG is denoted $G = (V, E)$, where V is the set of nodes, and E is the set of edges between the nodes, which can be expressed as $E = \{e_{i,j} | v_i, v_j \in V\}$. The worst-case communication time of the data dependency $e_{i,j}$ is expressed $comm(e_{i,j})$.

Data dependencies E represent the relationship of sending data to successor nodes, and an event node is executed after the execution of the predecessor node is finished. Nodes in front of the data dependency are called predecessor nodes, and nodes behind are called successor nodes. However, event nodes that have multiple predecessor nodes in a multi-period system work as follows. Arriving data from the predecessor nodes launched for shorter periods are used only to update data but not to trigger the successor nodes. However, arriving data from the predecessor node with the longest period is used for triggering the successor nodes.

2.3 Chain

A chain is a node sequence. A chain starts with a timer or an event node that branches off from another chain. Further, a chain ends before it hits a node defined in another chain. This prevents that the same node is included in multiple chains. The i-th chain is defined by $\Gamma_i = \{v_{c1}, v_{c2}, ..., v_{c(n-1)}, v_{cn}\}$, where v_{c1} is the node at the beginning of the chain, v^i_{cn} is the node at the end of the chain, and each node is connected before and after nodes by data dependencies.

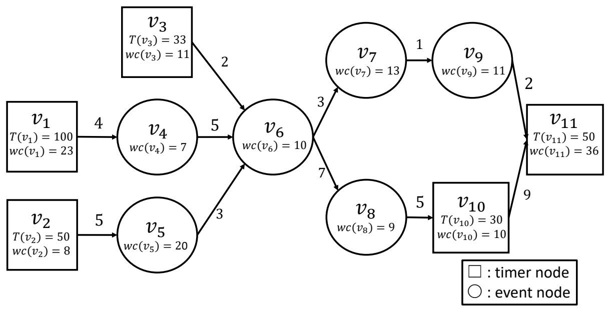
v_3 $T(v_3) = 33$ $wc(v_3) = 11$
v_1 $T(v_1) = 100$ $wc(v_1) = 23$
v_4 $wc(v_4) = 7$
v_6 $wc(v_6) = 10$
v_7 $wc(v_7) = 13$
v_9 $wc(v_9) = 11$
v_{11} $T(v_{11}) = 50$ $wc(v_{11}) = 36$
v_2 $T(v_2) = 50$ $wc(v_2) = 8$
v_5 $wc(v_5) = 20$
v_8 $wc(v_8) = 9$
v_{10} $T(v_{10}) = 30$ $wc(v_{10}) = 10$
2
4
5
3
3
7
1
5
2
9
5
□ : timer node
○ : event node

Fig. 1 Directed acyclic graph

Moreover, the chain Γ_i has parameters such as ($E(\Gamma_i)$, $T(\Gamma_i)$, $WC(\Gamma_i)$, $\phi(\Gamma_i)$). $E(\Gamma_i)$ represents the edge set of the node sequence comprising a chain Γ_i, which can be expressed by the equation $E(\Gamma_i) = \{e^i_{c1,c2}, e^i_{c2,c3}, ..., e_{c(n-1),cn}\}$. $T(\Gamma_i)$ is a period to trigger the chain Γ_i execution. $WC(\Gamma_i)$ is the WCET of the chain Γ_i, which can be derived by the following equation.

$$WC(\Gamma_i) = \sum_{v_j \in \Gamma i} wc(v_j) + \sum_{e_{k,l} \in E(\Gamma_i)} comm(e_{k,l}) \qquad (1)$$

$\phi(\Gamma_i)$ is an offset of the chain Γ_i. The offset is a value to shift the trigger time of the chain Γ_i. The trigger time of the chain is the time when the first node in the chain is triggered. If the offset is not zero, the trigger time of the chain is shifted by $\phi(\Gamma_i)$.

If an edge exists from a node in one chain Γ_i to the first node in another chain Γ_j, Γ_i is called the predecessor chain of Γ_j, and Γ_j is called the successor chain of Γ_i. The communication time between chains is represented by $COMM(\Gamma_i, \Gamma_j)$, that is the communication time between the node of Γ_i and the first node of successor chain Γ_j.

Job parameters are required to analyze scheduling using timer nodes [13]. A chain job is each execution of the chain repeated in the period, and the chain job is similar to a job of a node. The chain job is created for the x-th time in the chain Γ_i is denoted as $\Gamma_{i,x}$. The chain job $\Gamma_{i,x}$ is assumed to trigger at $T(\Gamma_i) * (x - 1)$.

3. Generating Chain

This paper uses a chain that is a node sequence following data dependencies. Using a chain as the scheduling unit reduces the number of scheduling units compared to using a node as the scheduling, and that avoids the complex scheduling order of multi-period systems. Therefore, this paper proposes how to divide a DAG into chains in this section.

3.1 Start Point and End Point of Chain

The chain starts with two types of nodes. The first type is a timer node. The second type is an event node that was not selected as the branch destination of the chain when the chain branched. All nodes in a chain are executed by the timer node's period. The reason for this is that the timer node is the start point of the chain, and the other nodes in the chain are triggered at finishing the execution of the predecessor node. The end point of the chain is the predecessor node of another timer node because the chain is limited to a single period.

3.2 Confluence

A confluence node is an event node that has multiple data dependencies from predecessor nodes. In Fig. 1, a confluence node v_6 has multiple predecessor nodes such as v_3, v_4, and v_5. If a DAG

Table 1 Parameters and functions in Algorithms 1 and 2

$classify(DAG)$	A function to classify timer nodes from DAG
V_{tm}	A List of timer nodes
$Divide(DAG, V_{tm})$	A function to generate chains from DAG
$chainList$	A list of chains generated from DAG
$calcWST(chainList)$	A function to find the trigger indexes that contribute to the exit chain for each chain in *chainList* and to calculate the worst start time for that chain jobs
$Simulate(WSTList)$	A function to simulate core allocation and preemption in order of decreasing worst start time and return scheduling result
$result$	Scheduling results of chain jobs
$timerNodes$	A list of timer nodes that are not included in any chains
$maxPeriod(V)$	A function to return the node with the largest period from the node set V as input
$LongSeq(v)$	A function to return the node sequence with the longest WCET starting from node v and ending at the next timer node or exit
$long$	Output of the function $LongSeq(v)$
$OtherSeq(long)$	A function to return node sequences that are not chosen successors of junction nodes of *long*

is divided as described in Section 3.1, the same confluence node is defined in multiple chains. If chains define the scheduling unit doubly, the scheduling order is decided considering the extra execution of a confluence node.

A solution to this problem is defining that a confluence node is included only by the chain with the maximum trigger period in chains connecting the confluence node. The reason for not triggering on the smaller period is that data from a larger period chain cannot be in time for a confluence node. If a confluence node triggered each smaller chain period, data from a larger period chain are unchanged in the next trigger time. The output of a confluence node is not completely updated. Therefore, a confluence node is triggered by the maximum period.

3.3 Junction

A junction node is defined as a node that has multiple successor event nodes. In Fig. 1, v_6 is a junction node and has successor nodes v_7 and v_8. If chains are generated by following the successor node from the start node of the chain, one branch destination must be selected at a junction node to prevent a double definition of the chain. To avoid this case, a junction node connects node sequences to make a chain with the longest WCET. A junction node v_6 has two branch destinations, v_7, v_9 node sequence and v_8 node sequence. The branch destination of v_6 is selected v_7, v_9 node sequence. The reason for making a chain to be the longest WCET is to create a chain with a critical path. A critical path is the longest execution time path to protect node dependencies. Making the critical path chain and prioritizing the execution of the chain, the proposed method tries to reduce the scheduling makespan. In the evaluation section, it was found that the method of selecting the branch with the longest WCET had a lower deadline miss rate than the "min_branch" method of selecting the branch with the shortest WCET at the branch end of the chain.

Branches that are not chosen successors of junction nodes become the start nodes to make new chains. The period of a newly generated chain is the same as the period of the predecessor chain. This reason is that an event node of the branch destination that starts when the data arrive will also have the same period as that of the entry node. The new chain is triggered by that period, but

Algorithm 1: The overview of proposed method

```
  input : DAG
  output: Scheduling result
  // Classifying timer nodes from DAG
1 $V_{tm} = classify$(DAG)
  // Generating chains from DAG
2 $chainList = Divide$(DAG, $V_{tm}$)
  // Calculating the chain jobs contributing to
     the exit chain and their worst start time
3 $WSTList = calcWST(chainList)$
  // Simulating scheduling and assigning chain
     jobs to core
4 $result = Simulate(WSTList)$
5 return $result$
```

when the chain starts is not decided. The lack of deciding start time leads to not delivering data between the junction node and the start node of the new chain as the node dependencies. Adding a new parameter offset to have a start time to chain, the offset is equal to the WCET of the chain containing the junction node.

3.4 Setting Chain Parameter

The parameters of a chain are defined after the DAG is divided into chains. The WCET of a chain is obtained by Eq.(1). A period is determined by the start node of a chain. If the start node is a timer node, the period of the timer node becomes the period of the chain. If the start node is an event node, the predecessor chain of the start node is considered. The largest period of the predecessor chain is used as the period of the chain because all inputs of the start node will be updated every period.

The derivation of the offset is determined by the predecessor chain of the start node of a chain. When the start node is a timer node, it triggers according to its period, and the offset is zero. When the start node is an event node, the chain triggers every maximum period of the predecessor chain. The chain must be executed when the predecessor node has finished execution and the data have been updated. Assuming that the predecessor chain is Γ_j, an offset of Γ_i is derived by a following equation:

$$\phi(\Gamma_i) = \phi(\Gamma_j) + WC(\Gamma_j) \qquad (2)$$

3.5 Dividing Order

Dividing a DAG into chains is the first step of the proposed method. The overall flow of the proposed method is presented in Algorithm 1. The parameters using in Algorithm 1 are shown in **Table 1**. Algorithm 1 takes a DAG as input and finds core allocation and scheduling results for each chain job. First, Algorithm 1 classifies timer nodes from a DAG for dividing chains. Next, *Divide* function divides the DAG into chains. Each chain generates multiple chain jobs depending on the period. The function *calcWST* identifies chain jobs with output that is used to process the exit chain and gives priority for scheduling to chain jobs. Finally, the function *Simulate* simulates scheduling and gives the scheduling results and core allocations.

The procedure of the function *Divide* is explained in Algorithm 2 and Table 1. Rules of dividing a DAG are based on Sections 3.1, 3.2, and 3.3. First, Algorithm 2 enumerates the timer nodes as the entry nodes of a chain. Then, one of the timer nodes with the maximum period is selected from the candidate *timerNodes* for the entry. The chain with the longest WCET is created from the selected node (lines 4-5 in Algorithm 2). The reason for choosing the timer node with the maximum period is

Algorithm 2: *Divide* : Generating chains from DAG

```
   input  : DAG, $V_{tm}$
   output: $chainList$
1  $chainList \leftarrow []$
2  $timerNodes \leftarrow V_{tm}$
   // make node sequences
3  while $timerNodes \neq \emptyset$ do
4      $maxPeriodNode \leftarrow maxPeriod(timerNodes)$
5      $long \leftarrow LongSeq(maxPeriodNode)$
6      $others \leftarrow OtherSeq(long)$
7      add $long$ and $others$ to $chainList$
8      remove $maxPeriodNode$ from $timerNodes$
9  end
10 return $chainList$
```

to consider the confluence described in Section 3.2. By creating from the chain with the largest period, a chain with a larger period can include the confluence node in the chain earlier.

When a junction node exists in a chain, the execution time of each branching destination is compared, and the longer one is defined to continue the chain (line 5 in Algorithm 2). All unchosen branching nodes are treated as the starting point of a new chain (line 6 in Algorithm 2). Finally, the created chains are added to the chain list, and the entry node of the chain removes from the *timerNodes*. (lines 7-9 in Algorithm 2). At the end of all loops, the definition of all chains is complete. In Fig. 1, DAG can be separated to $\Gamma_1 = \{v_1, v_4, v_6, v_7, v_9\}$, $\Gamma_2 = \{v_2, v_5\}$, $\Gamma_3 = \{v_3\}$, $\Gamma_4 = \{v_8\}$, $\Gamma_5 = \{v_{10}\}$, and $\Gamma_6 = \{v_{11}\}$.

4. Scheduling Method

This section explains the scheduling algorithm that is proposed in this paper. The proposed method defines the worst start time that is used as the priority for scheduling. In addition, this section proposes how to find jobs with the number of triggering in the chain where such output is not used. By excluding the unused jobs from the scheduling order, the number of nodes to be considered in the scheduling is decreased.

4.1 Scheduling Assumptions

Scheduling assumptions are defined in this subsection. The division of the DAG into chains and the determination of the scheduling order are performed statically. The scheduling order is determined in the order of earliest to latest, using the worst start time as an index. If all cores are full, the chain with the latest worst start time is preempted from the chain to which the core is assigned. This study is also based on the assumption that the system has one exit node in the DAG. For example, a self-driving system, such as Autoware, combines steering wheel operation and acceleration amount into a single topic [7].

The deadline constraint is defined to consider the proposed method. The deadline for a multi-period system is set only for the chain at the exit of the system, and this study assumes that the deadline is equal to the period of the exit chain. In addition, each chain must update its data to produce a valid output. Each chain other than the exit chain is considered a deadline miss if the data do not arrive before the successor chain triggers.

4.2 Worst Start Time

The worst start time is defined for each chain job to determine the scheduling order. The scheduling order is determined in the order of earliest to latest, using the worst start time. The worst start time is an absolute time and is found by subtracting WCET

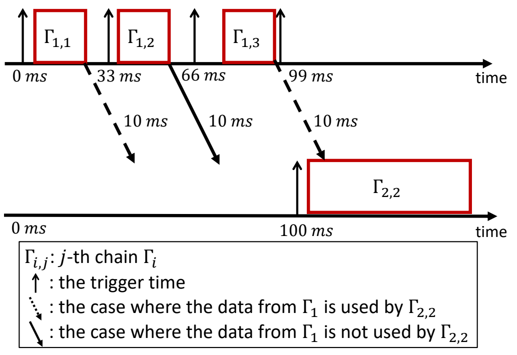
$\Gamma_{1,1}$
$\Gamma_{1,2}$
$\Gamma_{1,3}$
$0\ ms$
$33\ ms$
$66\ ms$
$99\ ms$
time
$10\ ms$
$10\ ms$
$10\ ms$
$\Gamma_{2,2}$
$0\ ms$
$100\ ms$
time
$\Gamma_{i,j}$: j-th chain Γ_i
: the trigger time
: the case where the data from Γ_1 is used by $\Gamma_{2,2}$
: the case where the data from Γ_1 is not used by $\Gamma_{2,2}$

Fig. 2 Relationship of using data from Γ_1 to $\Gamma_{2,2}$

from the next trigger time of the chain, and the worst start time for the j-th triggering of the chain Γ_i is defined as follows:

$$WST(\Gamma_i, j) = T(\Gamma_i) * j + \phi(\Gamma_i) - WC(\Gamma_i) \qquad (3)$$

Scheduling in the order of the worst start time is synonymous with scheduling in the order of the deadlines of jobs in each chain. The reason deadline is synonymous is that the end time of each chain job period is used to derive the worst start time. The WCET is subtracted from the end time of the period of each chain job to derive the worst start time. Therefore, the worst start time of a chain job with a long execution time will be earlier. Because chain jobs with longer execution times are prioritized, the effect of prioritizing the critical path can be expected. The worst start time can be considered as an index that can consider both the deadline and execution time.

4.3 Chain Job Index

Depending on the trigger period of a chain, the output of the chain may not be used. Due to the fact that a chain is triggered in each period, data may arrive two or more times before the next trigger of the chain. If multiple data arrive, the most recent data is used to process the most recent environmental information [9]. Such a case occurs when the period of the successor chain is larger than that of the current chain. An example of the unused output of the chain job is illustrated in **Fig. 2**. The example has a chain $\Gamma_1, (T(\Gamma_1) = 33, WC(\Gamma_1) = 21)$ and a chain $\Gamma_2, (T(\Gamma_2) = 100, WC(\Gamma_2) = 64)$ exist, and a worst-case communication time is 10 ms from chain Γ_1 to chain Γ_2.

Because the first chain job $\Gamma_{2,1}$ does not receive data from chain Γ_1, the data used for the second chain job $\Gamma_{2,2}$ is considered. Because the chains receive the data at the start time of the period, chain Γ_1 must deliver the data at the trigger time of the successor chain job. The start time of the period becomes the deadline for receiving data, and chain Γ_1 must deliver data at the trigger time of the chain job $\Gamma_{2,2}$. Therefore, the latest data of the chain Γ_1 job processed by the time when the communication time is subtracted from the trigger time of the chain job $\Gamma_{2,2}$ are used. In Fig. 2, the data of the chain job $\Gamma_{1,3}$ does not arrive before the trigger time of the chain job $\Gamma_{2,2}$, and the data of the chain job $\Gamma_{1,2}$ are used as the latest data. Because chain job $\Gamma_{1,1}$ is not the latest data, chain $\Gamma_{1,1}$ is not used in $\Gamma_{2,2}$.

The output is not used for the chain job on the first trigger where no data are used. By not executing the chain job, the idle time of the core can be increased. In addition, by knowing in advance which chain jobs will not be executed, the number of node executions in the chain can be decreased. This will reduce the number of nodes that need to be considered within the hyper-

Table 2 Parameters and functions in section 4.4 and Algorithm 3

Γ_i	i-th chain in *chainList*
wstList	A list of the worst start time and the trigger index for chain jobs that contribute to the exit chain
hp	The hyper-period for all chain periods
$exitJdg(chainList)$	A function to find the exit chain from *chainList* and return the exit chain
exitNum	The number of exit chain triggering in hyper-period
$updateWST(wstList, \Gamma_i)$	A function to update the worst start time and trigger index information *wstList* of the chain Γ_i, based on the result calculated by Eqs.(4), (5), (7), and (8)
$predChains(\Gamma_i)$	A function to return a list of predecessor chains of chain Γ_i
$succReadyChains(\Gamma_i)$	A function to return a list of chains that are successors of chain Γ_i and for which the trigger index has already been calculated
calcCandidate	Next candidates for the chain that can calculate the worst start time and the trigger index that do not contribute to output

period and hopefully eliminate the complexity of the scheduling order. The above method needs to find the chain job of the trigger index whose output will not be used before determining the scheduling order.

The proposed method considers only the worst case of using the latest data to find a chain job whose data are unused. The worst case is when the chain job finishes execution at the end of the period with the WCET, such as $\Gamma_{1,3}$ in Fig. 2. In case the execution time is shorter, or the execution is finished with more time to spare at the end of the period, a method could use newer data than the proposed method. Moreover, the worst case may occur, and the data cannot be supplied to the succeeding chain because of the optimistic estimate of the trigger index of the chain job. Finding the chain job whose data will be used in the worst case can prevent such a situation.

4.4 Derivation of Worst Start Time and Trigger Index

A method to find the worst start time and the trigger index for each chain job is defined in this subsection. The calculation is started from the exit chain because the proposed method finds the trigger index of each chain whose output is used for the exit chain. In this study, the value of the worst start time of chain Γ_i that contributes to the *exitIdx*-th job of the exit chain Γ_{exit} is represented using $WST(\Gamma_i, exitIdx)$. The parameter *exitIdx* is the information that this worst start time is related to the *exitIdx*-th time of the exit chain. The worst start time of the exit chain Γ_{exit} is defined as follows:

$$WST(\Gamma_{exit}, exitIdx) = T(\Gamma_{exit}) * exitIdx + \phi(\Gamma_{exit}) - WC(\Gamma_{exit}) \quad (4)$$

The worst start time is used for finding the trigger index of the chain. To ensure that the output of the system is not lost, The exit chain of the system must have all chain jobs executed during the hyper-period. The trigger index of the chain Γ_i that is used for the *exitIdx*-th job in the exit chain Γ_{exit} is represented using $jobIdx(\Gamma_i, exitIdx)$. The $jobIdx(\Gamma_{exit}, exitIdx)$ of the exit chain Γ_{exit} is defined as follows:

$$jobIdx(\Gamma_{exit}, exitIdx) = exitIdx \quad (5)$$

The following explains how to calculate the worst start time and trigger index for chains other than the exit chain Γ_{exit}. The worst start time is the worst value that can be reached in time for the chain to finish processing by the triggering time of the successor chain. The successor chain of the chain Γ_i is necessary for this calculation, and the successor chain Γ_{succ} is defined as follows:

$$\Gamma_{succ} \in succReadyChains(\Gamma_i) \quad (6)$$

The worst start time can be obtained by subtracting the time taken by the current chain from the start time of the successor chain. The worst start time $WST(\Gamma_i, exitIdx)$ other than the exit chain is defined as follows:

Algorithm 3: *calcWST* : Calculating the worst start time of the chain that contributes to the exit chain output

```
input : chainList
output: A list of the worst start time and the trigger index for
        chain jobs that contribute to the exit chain
1  $wstList \leftarrow []$,   $\Gamma_{exit} \leftarrow exitJdg(chainList)$
2  $hp \leftarrow$ hyper period of chains in $chainList$
3  $exitNum \leftarrow hp/T(\Gamma_{exit})$
4  for $exitIdx = 1, \cdots, exitNum$ do
5      Calculate $WST(\Gamma_{exit}, exitIdx)$ // Eq.(4)
6      Calculate $jobIdx(\Gamma_{exit}, exitIdx)$ // Eq.(5)
7      $wstList \leftarrow updateWST(wstList, \Gamma_{exit})$
8  end
9  add $predChains(\Gamma_{exit})$ to $calcCandidate$
10 foreach $\Gamma_i \in calcCandidate$ do
11     foreach $\Gamma_{succ} \in succReadyChains(\Gamma_i)$ do
12         for $idx = 1, \cdots, exitNum$ do
13             Calculate $WST(\Gamma_i, idx)$ // Eq.(7)
14             Calculate $jobIdx(\Gamma_i, idx)$ // Eq.(8)
15             $wstList \leftarrow updateWST(wstList, \Gamma_i)$
16         end
17     end
18     add $predChains(\Gamma_i)$ to $calcCandidate$
19     remove $\Gamma_i$ from $calcCandidate$
20 end
21 return $wstList$
```

$$\begin{aligned}WST(\Gamma_i, exitIdx) = {} & T(\Gamma_{succ}) * (jobIdx(\Gamma_{succ}, exitIdx) - 1) \\ & + \phi(\Gamma_{succ}) - WC(\Gamma_i) - COMM(\Gamma_i, \Gamma_{succ}) \quad (7)\end{aligned}$$

The trigger index can be derived by dividing the worst start time by the period. If the chain has an offset, the start time will be shifted. Therefore, it is necessary to perform subtraction to eliminate the shift before the calculation. The trigger index $jobIdx(\Gamma_i, exitIdx)$ other than the exit chain is defined as follows:

$$jobIdx(\Gamma_i, exitIdx) = \left\lfloor \frac{WST(\Gamma_i, exitIdx) - \phi(\Gamma_i)}{T(\Gamma_i)} \right\rfloor + 1 \quad (8)$$

The flow of deriving the worst start time and trigger indexes retroactively from the exit chain is shown in Algorithm 3. Moreover, parameters and functions used in Algorithm 3 are explained in **Table 2**. Algorithm 3 finds the worst start time and trigger indexes of the exit chain and puts their predecessor chains in the list of candidates for the calculation (lines 4-9 in Algorithm 3). After calculating the parameters of the exit chain, Algorithm 3 calculates the parameters of the chains in the candidate list *calcCandidate* (lines 10-17 in Algorithm 3). After the parameters of a chain have been calculated, its predecessor is added to the list of candidates for the calculations, and the process is repeated to obtain the worst start times and trigger indexes for all chains (lines 18-19 in Algorithm 3).

A core allocation for a chain is conducted after finishing Algorithm 3. Each chain job is put in a ready queue when the chain is triggered. Chains with a timer node are triggered by their own period, and chains without a timer node are triggered when the execution of the chain with the largest period in the preceding chains finishes. The chain job in the ready queue is sorted in decreasing order of the worst start time. When a core is free, the chain job at the head of the queue is assigned to the free core. If the execution of the chain job is finished, the core that is allocated to the chain job is released. In addition, the proposed method treats preemption. When the worst time of the chain job at the head of the queue is smaller than the worst time of the chain job already allocated to the core, the allocations of those chain jobs are exchanged. In Algorithm 1, the function $Simulate$ runs the simulation according to this assignment rule and returns the scheduling results.

5. Evaluation

This section explains the evaluation of the proposed method and the existing studies. The assumptions are explained, and an evaluation using random DAGs is presented.

5.1 Preparation

In the evaluation, the proposed method is compared with two existing algorithms that schedule on a node-by-node basis. The existing algorithm is extended to accommodate the DAG that includes timer nodes and event nodes, and each node is prioritized by laxity [5]. The laxity represents the time to deadline and can be calculated for exit nodes by subtracting the execution time of the exit node from the deadline. The laxity of other nodes is obtained by subtracting their own execution time and communication time with the successor node from the laxity of the successor node.

Since this evaluation includes the timer node, it is necessary to determine how many times data from the job will be used, as in Section 4.3. Two existing studies were chosen for comparison that could determine which data from the job is used. The first existing study decided the job dependencies between jobs with close tentative release time [3], and it is called "Igarashi." The tentative release time is obtained by multiplying the node period and the trigger index. The second existing study is based on the value of the periods of dependent nodes to determine the dependence of jobs on each other [14]. This method is called "Salah." The calculation method differs depending on whether the period of the predecessor node is larger, smaller, or the same as the period of the successor node.

Judging the deadline miss is defined for use in evaluation. The target DAG has only one exit chain from the assumptions in Section 4.1, and that chain has a relative deadline equal to the period. The exit chain Γ_{exit} has the deadline that repeats in the period. Therefore the k-th execution has a deadline of $k * T(\Gamma_{exit}) + WC(\Gamma_{exit})$. A deadline miss is detected when the time that the last node in the chain has completed exceeds the deadline. In addition, a deadline miss of a chain other than the exit chain occurs when data is not delivered to a successor chain with the dependencies determined in Section 4.4. The deadline miss of a chain other than the exit chain is judged when the end time of each chain exceeds $WST(\Gamma i, exitIdx) + WC(\Gamma_i)$. If a deadline miss occurs even once, the DAG is treated as a deadline miss DAG, and the evaluation is based on the rate of deadline misses out of the total DAGs handled in the evaluation.

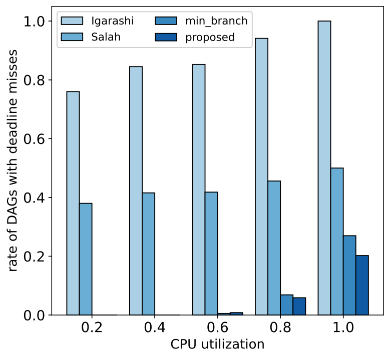

Fig. 3 Evaluating the rate of DAGs with deadline misses with varying CPU utilization

5.2 Random DAG Evaluation

An evaluation using randomly generated DAGs was conducted to prove the generality of the proposed method. The DAG handled by the proposed method must have timer nodes. In this evaluation, DAGs are generated by using a random DAG generation tool [15] that also generates nodes that are triggered in periods. Parameters for random DAG generation used in the evaluation are shown below. The number of nodes was varied by 20, 40, 60, 80, and 100. The number of timer nodes was also varied to 2, 4, 6, 8, or 10. The period ranges from 10 ms to 100 ms in 10 ms increments. The execution time of the chain is set not to exceed the period. The node execution time takes a random value from the range of 2 ms to 100 ms, and the communication time takes a random value from the range of 2 ms to 20 ms. A total of 2,550 files were generated with 80 random DAGs for each node and each number of timer nodes.

Due to the scheduling assumption, DAGs for which the chain execution time exceeded the period were then omitted. DAGs that do not reach the exit node due to a low number of chain trigger indexes are also omitted. The remaining DAGs were used in the evaluation. The maximum number of timer nodes was decided with reference to the number of nodes to be triggered in a period, which was an issue at the RTSS 2021 Industry Challenge [12].

The CPU utilization was varied, and the rate of DAGs with deadline misses was measured. The measurement results are shown in **Fig. 3**. In addition to comparisons with the existing studies, this evaluation also compares the "min_branch" method, in which a chain is created by selecting the branch destination with the shortest WCET. The rate of DAGs with deadline misses is the rate of deadline misses among all DAGs used in the evaluation. If the percentage is lower, the schedulability is higher. For node-by-node scheduling studies, the time when the execution of the last node in the chain finishes is compared to the deadline. The CPU utilization was derived by dividing the execution time of each chain by the period and then dividing that value by the number of cores. The CPU utilization was evaluated every 0.2 on a horizontal axis from 0.2 to 1.0. The results show that the rate of DAGs with deadline misses of "proposed" is lower than "Igarashi" and "Salah." The rate of deadline misses increases for all methods as the CPU utilization increases.

The reason the "proposed" method had a lower rate of DAGs that missed deadlines than the "Igarashi" and "Salah" is that only chains that contribute to the exit chain were scheduled. By not

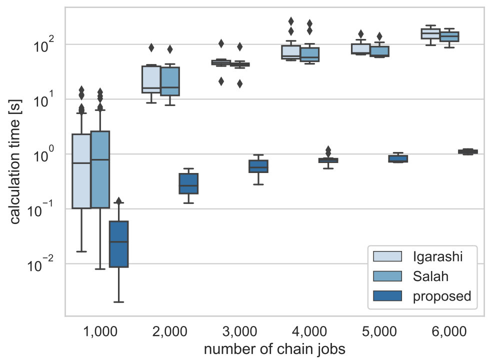

Fig. 4 Evaluating computation time with varying the number of chain jobs

executing the nodes that do not contribute to the exit chain, the idle time of the core increased. This allowed us to prioritize the execution of chain jobs that contribute to the exit chain. The increase in the number of missed DAGs with each increase in the CPU utilization can be attributed to the fact that the advantage of more free core time was lost as the execution time became longer.

A method of “proposed” had a lower deadline miss rate than “min_branch.” If the chain is created to have a longer WCET, the worst start time is smaller and the priority of that chain will be higher. The execution of chains with critical paths was prioritized, causing a reduction in deadline misses. On the other hand, when CPU utilization was 0.6, the deadline miss rate was slightly higher for “proposed” than for “min_branch.” When a chain existed in which the period and WCET were almost equal in “proposed,” deadline misses occur in “proposed” and not in “min_branch.” Because this chain dominated one core, the number of cores allocated to other chains that had to execute in parallel was reduced, resulting in deadline misses. In “min_branch,” no chain with approximately equal period and WCET existed since the chain chooses the branch destination with the shortest WCET. “Proposed” may have a higher deadline miss rate than “min_branch” when the chain’s WCET and period are equal.

The relationship between the number of chain jobs and computation time is shown in **Fig. 4**. The number of chain jobs is the sum of the number of chains generated during the hyper-period. The computation time is expressed on a logarithmic scale, with an increase of one tick representing a value 10 times greater. The computation time is the time from determining the priority of each method to determining the scheduling order of each node and simulating. The number of cores was set to four, and the computation time was measured by varying the number of chain jobs. The computation time of “proposed” was shorter than that of “Igarashi” and “Salah” for all numbers of chain jobs. Further, as the number of chain jobs increased, the difference in computation time increased.

The reason the computation time of “proposed” is shorter than that of “Igarashi” and “Salah” is that the number of decisions on the scheduling order is reduced because chain jobs that do not contribute to the output of the exit chain are not executed. The condition that chain jobs that do not contribute to the output of the exit chain are created is when the data dependency from the chain with the smallest period to the chain with the largest period is present. In addition, the greater the difference between the periods of the smaller and larger chains, the greater the number of

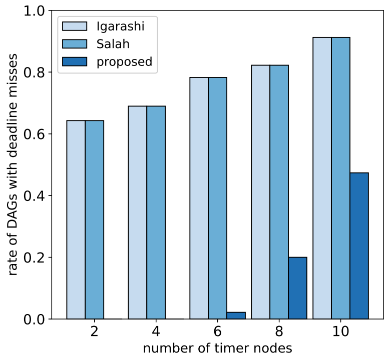

Fig. 5 Evaluating the rate of DAGs with deadline misses with varying the number of timer nodes

Table 3 Comparison with related work

	MPD	MTE	NDC	GN	AD	JD
Delete chain jobs [9]	✓			✓	✓	✓
ROS 2 Chain’s criticality [16]	✓	✓		✓	✓	
Response time analysis [17]	✓	✓		✓	✓	
Data age constraint [18]	✓				✓	✓
A single-period DAG [19]	✓				✓	
Schedulability test [20]	✓				✓	
Age latency [21]	✓				✓	
Two-level GFP [22]	✓	✓		✓		
this study	✓	✓	✓	✓	✓	✓

MPD: Multi-periods DAG
MTE: Mixed V_{tm} and V_{ev}
NDC: Non duplicated in chain
GN: Grouping nodes
AD: Available for any DAG
JD: Considering job dependencies

chain jobs that are not executed. In Fig. 4, the computation time of “proposed” does not become large as the number of chain jobs increases. The longer hyper-period is, the more chain jobs are generated. Because the proposed method is scheduling in hyper-period, a longer hyper-period leads to an increase in the number of times chain jobs are deleted. Because the number of deletion opportunities increases, the computation time reduces as the number of chain jobs increases. Another factor in the lower computation time is the per-chain scheduling. When scheduling per node, the priority of each node must be compared and assigned to a core. The longer the hyper-period, the greater the number of jobs on a node, and the greater the number of priority comparisons. By grouping the nodes into chains and scheduling each chain, the number of priority comparisons is reduced, resulting in less computation time.

The change in the percentage of DAG deadline misses when the number of timer nodes is varied was also experimented with, and the results are shown in **Fig. 5**. The rate of DAGs that miss deadlines for all parameters was lower for “proposed” method than for “Igarashi” and “Salah.” The rate of misses increased as the number of timer nodes increased. This is due to the increase in the number of chains that must be executed in parallel. Since timer nodes are triggered at each period, their activation timings may overlap. The increase in the number of timer nodes is thought to have caused the number of chains with overlapping startup timings to increase, exceeding the number of chains that can be executed in parallel, resulting in deadline misses.

6. Related Work

This section discusses differences between the proposed method and existing studies that used DAG with multiple periods. First, the method for using a chain, which is a sequence of nodes with dependencies as in this study, is introduced. Next, the results of various studies for DAG with only multiple timer nodes are presented. Finally, the research on scheduling nodes together, as the proposed method treated the node sequence as a chain, will be mentioned. A table comparing existing studies and the proposed method is shown in **Table 3**.

Nodes with data dependencies are treated as a chain in studies. H. Choi et al. [9] used a chain as a sequence of nodes from entry to exit in a DAG composed of timer nodes. They found jobs that do not contribute to the output generation of the final chain and aimed to minimize the end-to-end latency of the chain. In ROS 2 [23], the conventional research cannot be applied, because ROS 2 causes the unique scheduling behavior. H. Choi et al. [16] further made priority decisions based on the chain's criticality and timing requirements under the frame of ROS 2. D. Casini et al. [17] proposed a scheduling algorithm for ROS 2, treating end-to-end as a processing chain, and proposed a response time analysis of the processing chain. In existing studies, nodes may be defined doubly in the chain due to merging and branching. When scheduling each chain, the multiple definitions of nodes lead to the execution of an extra large number of nodes. To prevent multiple definitions of nodes, the proposed method divided the chain by merging and branching.

Studies of DAG that consist only of timer nodes triggered by period have been conducted. T. Klaus et al. [18] proposed a scheduling method that satisfies the data age constraint for periodic nodes. As another scheduling method, M. Verucchi et al. [19] proposed a method to generate a single-period DAG that satisfies the given constraints from multi-period DAG. This method optimized the schedulability and end-to-end latency. S. Baruah et al. [20] introduced a method for scheduling with EDF and determining whether all deadlines can be met. A. Kordon et al. [21] showed how to calculate the age latency accurately. They also proved that the calculation of age latency does not require computation during hyper-period, which shortens the computation time. These results were applied to a DAG consisting of only timer nodes, and the difference between these studies and the proposed method is that it does not include event nodes.

Similar to this study, scheduling methods that deal with node sets exist. DAGs are treated as nodes, and the scheduling of multiple DAG has been studied. Each DAG has a set of periods and deadlines, and the nodes in each DAG are called subnodes. R. Pathan et al. [22] considered the scheduling of a set of DAG nodes. They proposed a two-level GFP scheduling algorithm and the analysis method by assigning a fixed priority to each DAG node and the sub-nodes within the DAG node. The proposed method differs from existing studies in that it can split a DAG with a mixture of arbitrary timer nodes and event nodes.

7. Conclusion

In this study, we proposed a scheduling algorithm that determines the order of each chain in a DAG where timer and event nodes are mixed. Chains are defined as a node sequence that is divided to satisfy the period and data dependencies of each node and can be converted into a set of nodes that are triggered by the period. The proposed method finds chain jobs unused for the exit chain from the chain jobs. In the evaluation, the proposed method showed better schedulability and shorter computation time than a method based on an existing algorithm without deletion.

The future work is considering the input timing of a chain and adjusting the start timing. In this study, an assumption is made that the data from the predecessor chain are received by the time the chain period starts. The timing of when a chain needs data will be closer to the output timing of the predecessor chain, and newer data will be treated as input.

References

[1] Senapati, D., Sarkar, A. and Karfa, C.: HMDS : A Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems, *TECS*, Vol. 20, No. 5s, pp. 1–26 (2021).

[2] Baruah, S.: Scheduling DAGs When Processor Assignments Are Specified, *in Proc. of RTNS* (2020).

[3] Igarashi, S., Kitagawa, Y., Ishigooka, T., Horiguchi, T. and Azumi, T.: Multi-rate DAG Scheduling Considering Communication Contention for NoC-based Embedded Many-core Processor, *in Proc. of DS-RT* (2019).

[4] Bittencourt, L. F., Sakellariou, R. and Madeira, E. R.: DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm, *in Proc. of PDP* (2010).

[5] Jiang, X., Guan, N., Long, X., Tang, Y. and He, Q.: Real-time scheduling of parallel tasks with tight deadlines, *Journal of Systems Architecture*, Vol. 108, p. 101742 (2020).

[6] Igarashi, S., Ishigooka, T., Horiguchi, T., Koike, R. and Azumi, T.: Heuristic Contention-Free Scheduling Algorithm for Multi-core Processor using LET Model, *in Proc. of DS-RT* (2020).

[7] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y. and Azumi, T.: Autoware on Board: Enabling Autonomous Vehicles with Embedded systems, *in Proc. of ICCPS* (2018).

[8] Igarashi, S., Fukunaga, T. and Azumi, T.: Accurate Contention-aware Scheduling Method on Clustered Many-core Platform, *Journal of Information Processing*, Vol. 29, pp. 216–226 (2021).

[9] Choi, H., Karimi, M. and Kim, H.: Chain-Based Fixed-Priority Scheduling of Loosely-Dependent Tasks, *in Proc. of ICCD* (2020).

[10] Bhuiyan, A., Guo, Z., Saifullah, A., Guan, N. and Xiong, H.: Energy-Efficient Real-Time Scheduling of DAG Tasks, *TECS*, Vol. 17, No. 5, pp. 1–25 (2018).

[11] Medina, R., Borde, E. and Pautet, L.: Scheduling Multi-periodic Mixed-Criticality DAGs on Multi-core Architectures, *in Proc. of RTSS* (2018).

[12] Liu, S., Yu, B., Guan, N., Dong, Z. and Akesson, B.: Industry Challenge, *in Proc. of RTSS* (2021).

[13] Becker, M., Dasari, D., Mubeen, S., Behnam, M. and Nolte, T.: End-to-end timing analysis of cause-effect chains in automotive embedded systems, *Journal of Systems Architecture*, Vol. 80, pp. 104–113 (2017).

[14] Saidi, S. E., Pernet, N. and Sorel, Y.: Automatic Parallelization of Multi-rate Fmi-based Co-simulation on Multi-core, *in Proc. of TMS/DEVS* (2017).

[15] Azumi-Lab: RD-Gen, `https://github.com/azu-lab/RD-Gen`.

[16] Choi, H., Xiang, Y. and Kim, H.: PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2, *in Proc. of RTAS* (2021).

[17] Casini, D., Blaß, T., Lütkebohle, I. and Brandenburg, B.: Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling, *in Proc. of ECRTS* (2019).

[18] Klaus, T., Becker, M., Schröder-Preikschat, W. and Ulbrich, P.: Constrained Data-Age with Job-Level Dependencies: How to Reconcile Tight Bounds and Overheads, *in Proc. of RTAS* (2021).

[19] Verucchi, M., Theile, M., Caccamo, M. and Bertogna, M.: Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets, *in Proc. of RTAS* (2020).

[20] Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L. and Wiese, A.: A Generalized Parallel Task Model for Recurrent Real-time Processes, *in Proc. of RTSS* (2012).

[21] Kordon, A. and Tang, N.: Evaluation of the Age Latency of a Real-Time Communicating System using the LET paradigm, *in Proc. of ECRTS* (2020).

[22] Pathan, R., Voudouris, P. and Stenström, P.: Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms, *TPDS*, Vol. 29, No. 4, pp. 915–928 (2017).

[23] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the Performance of ROS2, *in Proc. of EMSOFT* (2016).