공학석사학위논문

Real-time System Optimization of DAG Task Model Considering Applicability

적용성을 고려한 DAG 테스크 모델의 실시간 시스템 최적화 기법

2021년 2월

서울대학교 대학원
컴퓨터공학부
이 승 수

Real-time System Optimization of DAG Task Model Considering Applicability

적용성을 고려한 DAG 테스크 모델의 실시간 시스템 최적화 기법

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함

2020년 11월

서울대학교 대학원

컴퓨터공학부

이 승 수

이승수의 공학석사 학위논문을 인준함

2020년 12월

위 원 장	하 순 회	(인)
부위원장	이 창 건	(인)
위 원	김 태 현	(인)

Abstract

Real-time System Optimization of DAG Task Model Considering Applicability

Seungsu Lee
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Response time reduction in autonomous driving systems is very important. For this, a real-time system must be applied to the autonomous driving system. However, the application of real-time system is not so simple. With the development of the open source platform, a real-time system that can be easily applied by non-majors as various developers has become necessary. This paper presents a real-time system optimization method that can reduce response time without kernel patch, without application modifying. In this paper, an optimization method consisting of three steps of heuristic algorithm was presented. This achieves the goal by determining scheduling policy, CPU affinity, and priority, respectively. Through experiments conducted in a simulated scheduler and an actual autonomous driving system, it shows a remarkable response time reduction effect.

keywords : Optimization, Real-Time Scheduling, DAG Scheduling, Autonomous

Driving Systems

Student Number : 2019-25586

Contents

List of Figures

List of Tables

1 Introduction

Autonomous driving system is a field that has received the most attention recently. Many companies, including existing automobile companies, are developing autonomous driving systems. In general, the autonomous driving system consists of a structure from sensing nodes to actuation nodes. The response time from sensing nodes to actuation nodes is directly related to safety. For example, when the vehicle is driving at 54 km/h, if a vehicle detects a obstacle through sensing nodes and the response time takes 400ms to actuation nodes, the vehicle will advance 6m until the actual response, which is a critical problem when the distance to the Obstacle is not sufficient. As such, application of a real-time system in the autonomous driving system is essential for safety.

However, applying a real-time system is a very challenging task. In the past, research on autonomous driving systems was only open to some developers, such as those belonging to companies. Recently, research on autonomous driving systems is conducted in open source platforms such as Robot Operating System(ROS)[1], Autoware[2], and Apollo[3].For this reason, developers in various fields are participating in autonomous driving research. Among them, computer engineers can more easily apply real-time systems through related knowledge, but for computer non-majors, the need to use kernel patch and real-time scheduling API is a barrier to applying real-time system. In these cases, high-cost, high-performance HW computing units are often used as a temporary solution. However, this not only slows commercialization by increasing the cost of developing an autonomous driving system, but is not strictly using a real-time system. Therefore, the necessity of an easily applicable real-time system targeting developers in various fields has been raised.

Since most open source projects are developed based on Linux, studies on applying a real-time system in Linux have been published accordingly. However, they have the following limitations to be easily applied. A study on a Linux-based real-time operating system(RTOS)[4, 5] such as $LITMUS^{RT}$ suggested a method of applying a real-time system to Linux. It requires a kernel patch, which is very complex. In addition, it is inefficient to request such a kernel patch from not only the developer but also the user.

ROSCH(Real-Time Scheduling Framework for ROS)[6] proposed a ROS-based real-time system framework that can be used in Linux without kernel patch. However, to use this framework, the autonomous system must be developed using the real-time scheduling library and API suggested by RESCH(REal-time SCHEduler suite)[7]. In order to apply to an already developed autonomous driving system such as Autoware[2], it is difficult to use it because it requires modification of the entire system.

HLBS(Heterogeneous Laxity-Based Scheduling)[8] and HEFT(Heterogeneous Earliest Finish Time)[9] proposed a fixed priority offline DAG (Directed Acyclic Graph) schedule algorithm, which can be applied using the basic Linux scheduling tool without application modification. However, these algorithms are difficult to use practically because there is no consideration for repeated execution of the task.

In this paper, we propose a real-time system optimization method that can reduce response time without kernel patch, without application modifying. For this, we present a heuristic algorithm that can satisfy goals using only the scheduling tool provided by Linux.

This paper is organized as follows. Section 2 briefly describes the background required for the paper. Section 3 formally defines our problem. Then, Section 4 explains our proposed algorithm. Section 5 reports our experiment results. Finally, Section 6 concludes the paper.

2 Background

Linux uses CFS(Completely Fair Scheduler)[10] as the default scheduler from kernel 2.6.23. This scheduler shows responsiveness and fairness by allocating CPU time in proportion to the weight of the task. However, there are cases where the desired goal cannot be achieved by using only CFS, so various scheduling tools are provided in Linux. As a basic scheduling tool in Linux, three parameters of policy, CPU affinity, and priority can be assigned to a task.

2.1 Linux Scheduling Policy

There are three Linux Scheduling Policies to be covered in this paper. SCHED_OTHER, SCHED_FIFO, SCHED_RR. These policies do not mean system-wide scheduling policies, but are assigned to tasks.

- SCHED_OHTER

This is a default general task policy. The priority is fixed to 0, and it is scheduled with the CFS.

- SCHED_FIFO

Priority is always greater than or equal to 1, and is always superior to tasks with SCHED_OTHER policy. If a task with a higher priority is in the run queue, the lower priority task is preempted. If a task with the same priority is in the run queue, it is executed in the First-In, First-Out(FIFO) method.

- SCHED_RR

If there is a task of the same prioirty in the run queue, it is executed in the round robin method for the given time slice. The rest of the things are the same as SCHED_FIFO.

2.2 CPU Affinity

By modifying the CPU affinity parameter, you can designate the CPU group where the task will be executed. By default, CPU affinity is set to run on all CPU cores.

2.3 Priority

A task with a higher priority preemption a task with a lower priority. For tasks using the SCHED_OHTER policy, priority is fixed to 0, and tasks using the SCHED_FIFO and SCHED_RR policies can have from 1 to 99.
In Linux, each CPU has its own run queue, so tasks in the run queue of other cores are not affected regardless of priority. In other words, only tasks in the run queue of the same CPU interfere with each other. Figure 1 shows the basic scheduling tool of Linux described above.

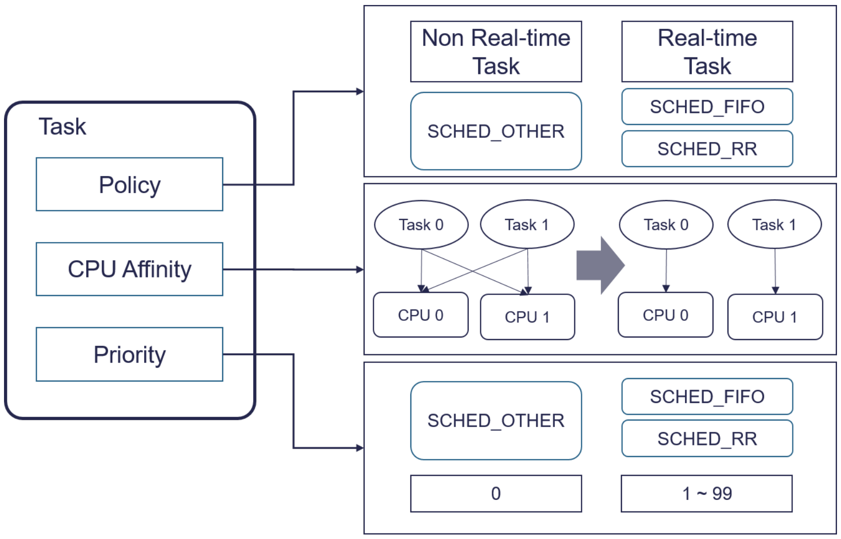


Figure 1: Basic scheduling tool of Linux

3 Problem Description

We consider a system with $CPU_1..CPU_Q$ cores. Q is number of cores in entire system. Each task is represented by $\tau_k(1 \leq k \leq n)$. The set of n tasks is represented by Γ, *i.e.*, $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$.

- Description of Autonomous Driving System

Autonomous driving systems such as Autoware can be represented as task set in the form of DAG. For example, Sensing corresponds to a root task and Actuation corresponds to a leaf task. Therefore, task contains information about DAG. This can be expressed as G_k. G_k can be represented as 2 tuples two tuples, *i.e.* $G_k = (pred_k, succ_k)$. $pred_k$ is a set of predecessor tasks, and $succ_k$ is a set of successor tasks. τ_k with $pred_k = \emptyset$ is called a root task, and τ_k with $succ_k = \emptyset$ is called a leaf task. The root task is released according to the period of the sensors, and the remaining tasks are released as event driven by the parents task. At this time, assume that the periods of all sensors are the same. Let this be called T_{system}. When there are more than one parent task of a task, all parents' events must be driven to be released. Among the paths from the root task to the leaf task, the path with deadline can be defined as a module. The time taken from the beginning to the end of the module is the response time of the module, and this corresponds to the makespan in the DAG task set. The deadline that the module must satisfy is expressed as D_{module_a}. The makespan of DAG tasks should be smaller than D_{module_a}.

- Description of Tasks Model

When all tasks are listed, tasks with deadline and should be executed in real-time can be expressed as $\Gamma^R = \{R\tau_1, \cdots, R\tau_m\}$. in sequence. Non real-time tasks without deadline such as kernel GUI tasks that are not included in the autonomous driving system or tasks related to visualization can be expressed as $\Gamma^{NR} = \{NR\tau_1, \cdots, NR\tau_l\}$. in sequence. Each task has set of its own execution times, E_k and scheduling parameter, P_k. The set of execution times E_k consists of execution times in each CPU, *i.e.* $E_k = (e_k^{CPU_1}, \cdots, e_k^{CPU_Q})$. Scheduling parameter P_k is represented as 3 tuples, *i.e.* $P_k = (plc_k, aff_k, prio_k)$. plc_k is a scheduling policy given to a task. aff_k represents the CPU affinity of the task. $prio_k$ is the priority of a task. These parameters cannot be changed during runtime. Including DAG information, the task is represented as 3 tuples, *i.e.* $\tau_k = (E_k, P_k, G_k)$.

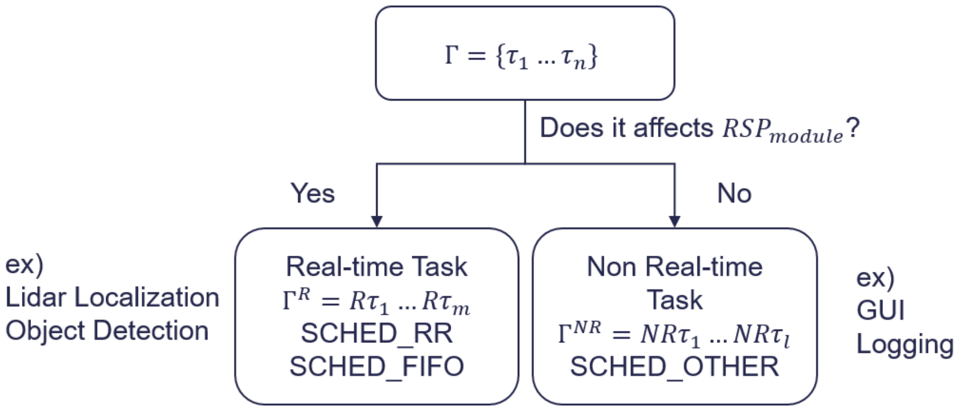
Under these assumptions, our problem is formally defined as follows:

Problem Definition: For each task τ_k in the given task set, $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, our problem is to find scheduling parameter P_k, such that all the modules of entire system can be scheduled meeting their deadlines on Q CPU cores.

4 Proposed Approach

In this section, we present a 3-step heuristic algorithm to determine P_k. Each step of the algorithm determines $plc_k, aff_k, prio_k$.

4.1 Decision of Scheduling Policy



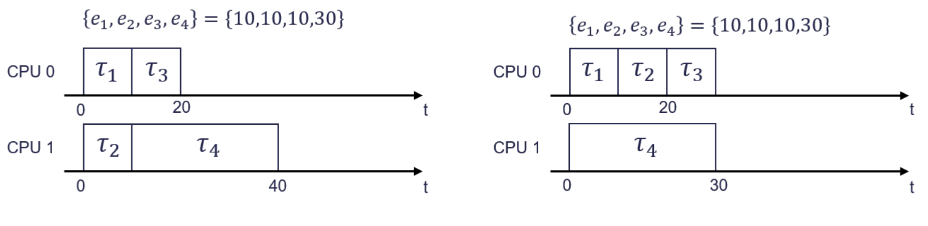
$\Gamma = \{\tau_1 \dots \tau_n\}$
Does it affects RSP_{module}?
Yes
No
ex)
Lidar Localization
Object Detection
Real-time Task
$\Gamma^R = R\tau_1 \dots R\tau_m$
SCHED_RR
SCHED_FIFO
Non Real-time Task
$\Gamma^{NR} = NR\tau_1 \dots NR\tau_l$
SCHED_OTHER
ex)
GUI
Logging

Figure 2: Decision of scheduling policy

In the first step, the scheduling policy of the task, plc_k, is determined. Receives a task set as input, classifies whether it is a real-time task or a non real-time task, and assigns a scheduling policy accordingly. As mentioned in Section 3, tasks with deadline are classified into real-time tasks, and tasks that do not are classified into non real-time tasks. Through this step, real-time tasks always have higher priority over non real-time tasks. For real-time tasks, LiDAR localization or object detection will be an example, and GUI and Logging will be the other. Figure 2 describes this step.

4.2 Assignment of CPU Affinity

4.2.1 Linux load balancing



(a) Linux load balancing (b) Assign CPU affinity

Figure 3: Load balancing scenarios

In the second step, the CPU affinity,aff_k, is determined. Since Linux uses partitioned queues, unfairness between CPU can occur, and this is resolved through load balancing. However, since load balancing only considers the weight and number of tasks, it may not be able to accurately balance. Figure 3 shows an example. Suppose CPU0 and CPU1 are homogeneous. When $\tau_1, \tau_2, \tau_3, \tau_4$ is released, Linux divides the number of tasks equally and balances them as in figure 3(a). However, considering the execution time, balancing as in figure 3(b) can reduce the makespan. Therefore, it is necessary to properly load balance through CPU affinity adjustment.

4.2.2 Proposed Algorithm for CPU Affinity Assignment

Before presenting the algorithm, the required notation and description are described in the table 1. The heuristic algorithm considering 4.2.1 is introduced on Algorithm 1.

The algorithm takes the real-time DAG task set expressed as Γ^R as input. Γ_{CPU}, which is the result of assigning the task to the CPU, and $WCRT$, Worst Case Re-

Notation	Description
Γ_{CPU_K}	The set of tasks assigned to CPU_K
$Total_{CPU_K}$	Total execution time sum of tasks assigned to CPU_K
$Subtotal_{CPU_K}$	Subtotal execution time of tasks assigned to CPU_K only in *AssignReady*
$\mathbb{L}_{CPU}$	The set of CPU_K with least $Total_{CPU_K}$
AssignReady	The set of tasks ready to be assigned
AssignFinish	The set of tasks that have already been assigned

Table 1: Notations and descriptions used in the algorithm

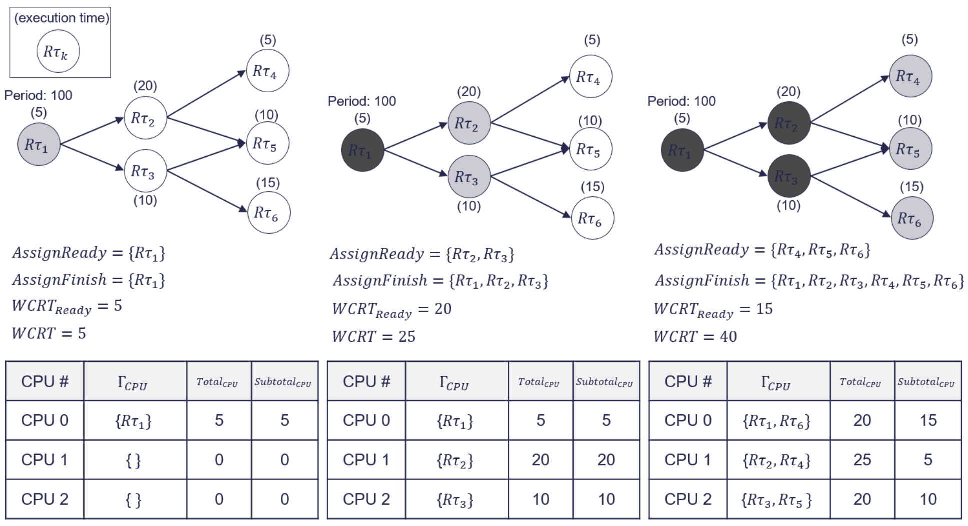
sponse Time that can be guaranteed, are taken as outputs. The algorithm first initializes the variables from Line 1 to Line 5. In the **while**-loop from Line 6 to Line 33, tasks are assigned until all real-time DAG tasks are included in *AssignFinish*. It means all real-time DAG tasks are assigned. *AssignReady* is initialized in Line 7. From Line 9 to Line 13, the **for**-loop finds assignable tasks. From Line 10 to Line 12, the **if**-conditional branch checks whether tasks are assignable. If all of the predecessors of $R\tau_i$ have been assigned, tasks are marked as assignable and included in *AssignReady*. In Line 15, $\mathbb{S}ubtotal_{CPU}$ is initialize because a new *AssignReady* set is ready. From Line 16 to Line 30, the **for**-loop assigns tasks in *AssignReady* in the order of $\max_{e_j \in E_j}(E_j)$'s largest. From Line 17 to Line 18, the CPU with the least $Total_{CPU}$ is a candidate to be assigned. This equalizes the sum of the CPU's execution time. The **if**-conditional branch from Line 19 to Line 20, If the number of candidates is 1, the candidate is determined as the CPU_K to be assigned. The **else**-conditional branch from Line 22 to Line 24, if there are multiple candidates, the CPU with the least $Subtotal_{CPU_L}$ is determined as CPU_K. This acts as a tie break and makes the sum of execution time equal to local within *AssignReady*. From Line 25 to 26, the task is assigned to the CPU_K by determine aff_j as CPU_K and including $R\tau_j$ in Γ_{CPU_K}. In

Algorithm 1 The DAG execution time fairness balancing algorithm

```
Input: $\Gamma^R$(real-time DAG task set)
Output: $\Gamma_{CPU}$(assigned task set),
        $WCRT$(Worst Case Response Time)
1: $\Gamma_{CPU} = \{\Gamma_{CPU_1}, \cdots, \Gamma_{CPU_Q}\} \leftarrow \{\emptyset, \cdots, \emptyset\}$
2: $\mathbb{T}otal_{CPU} = \{Total_{CPU_1}, \cdots, Total_{CPU_Q}\} \leftarrow \{0, \cdots, 0\}$
3: $\mathbb{S}ubtotal_{CPU} = \{Subtotal_{CPU_1}, \cdots, Subtotal_{CPU_Q}\} \leftarrow \{0, \cdots, 0\}$
4: $AssignFinish = \emptyset$
5: $WCRT = 0$
6: while $AssignFinsh = \Gamma^R$ do
7:    $AssignReady = \emptyset$
8:    /* step 1: Classification of tasks that can be assigned */
9:    for $R\tau_i \in \Gamma^R$ do
10:      if $pred_i \subset AssignFinish$ then
11:         $AssignReady = AssignReady \cup \{R\tau_i\}$
12:      end if
13:   end for
14:   /* step 2: Assignment of $AssignReady$*/
15:   $\mathbb{S}ubtotal_{CPU} \leftarrow \{0, \cdots, 0\}$
16:   for $R\tau_j \in AssignReady$, in largest $\max_{e_j \in E_j}(E_j)$ order do
17:      $\mathbb{L}_{CPU} \leftarrow \emptyset$
18:      $\mathbb{L}_{CPU} = \mathbb{L}_{CPU} \cup \{CPU_L | CPU_L$ that has least elements of $\mathbb{T}otal_{CPU}\}$
19:      if number of $\mathbb{L}_{CPU}$ is 1 then
20:         $CPU_K \leftarrow CPU_L$
21:      else
22:         /* tie breaker */
23:         $CPU_K \leftarrow CPU_L$ that has least element of $\mathbb{F}_{CPU}$
24:      end if
25:      $aff_j \leftarrow CPU_K$
26:      $\Gamma_{CPU_K} = \Gamma_{CPU_K} \cup \{R\tau_j\}$
27:      $AssignFinish = AssignFinish \cup \{R\tau_j\}$
28:      $Total_{CPU_K} = Total_{CPU_K} + e_j^{CPU_K}$
29:      $Subtotal_{CPU_K} = Subtotal_{CPU_K} + e_j^{CPU_K}$
30:   end for
31:   $WCRT_{Ready} \leftarrow Subtotal_{CPU_W}$ that greatest element of $\mathbb{S}ubtotal_{CPU}$
32:   $WCRT = WCRT + WCRT_{Ready}$
33: end while
34: return $\Gamma_{CPU}$ and $WCRT$
```

Line 27, $R\tau_j$ is included in *AssignFinish*, indicating that the assign is complete. From Line 28 to Line 29, the execution time of the assigned task is added to $Total_{CPU_k}$ and $Subtotal_{CPU_k}$, respectively. In Line 31, the largest F_{CPU_W} value in the *AssignReady* assignments from Line 16 to Line 30 is set as the $WCRT_{Ready}$. Since *AssignReady* tasks can be executed simultaneously, it means that it can be completed in $WCRT_{Ready}$ even in the worst case when only *AssignReady* tasks are considered. In Line 32, the $WCRT_{Ready}$ is added to the $WCRT$ and updated. This means that if you add all of the $WCRT_{Ready}$, it can be guaranteed that $WCRT$ for all tasks. Finally, in Line 34, output of algorithm Γ_{CPU} and $WCRT$ are returned.

4.2.3 Algorithm applied to example DAG task set



$AssignReady = \{R\tau_1\}$
$AssignFinish = \{R\tau_1\}$
$WCRT_{Ready} = 5$
$WCRT = 5$

CPU #	Γ_{CPU}	$Total_{CPU}$	$Subtotal_{CPU}$
CPU 0	$\{R\tau_1\}$	5	5
CPU 1	{ }	0	0
CPU 2	{ }	0	0

$AssignReady = \{R\tau_2, R\tau_3\}$
$AssignFinish = \{R\tau_1, R\tau_2, R\tau_3\}$
$WCRT_{Ready} = 20$
$WCRT = 25$

CPU #	Γ_{CPU}	$Total_{CPU}$	$Subtotal_{CPU}$
CPU 0	$\{R\tau_1\}$	5	5
CPU 1	$\{R\tau_2\}$	20	20
CPU 2	$\{R\tau_3\}$	10	10

$AssignReady = \{R\tau_4, R\tau_5, R\tau_6\}$
$AssignFinish = \{R\tau_1, R\tau_2, R\tau_3, R\tau_4, R\tau_5, R\tau_6\}$
$WCRT_{Ready} = 15$
$WCRT = 40$

CPU #	Γ_{CPU}	$Total_{CPU}$	$Subtotal_{CPU}$
CPU 0	$\{R\tau_1, R\tau_6\}$	20	15
CPU 1	$\{R\tau_2, R\tau_4\}$	25	5
CPU 2	$\{R\tau_3, R\tau_5\}$	20	10

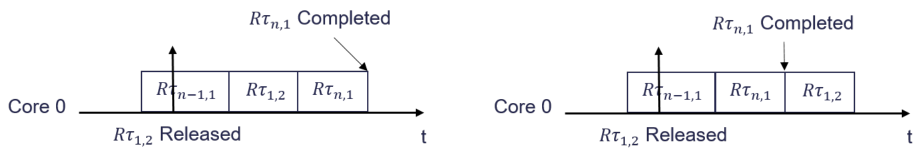
Figure 4: Example of affinity assignment

Figure 4 shows an example of affinity assignment. Three snapshots are taken every Line 32 of the algorithm 1. Through this, change based on *AssignReady* can be checked. Suppose CPU 0 to CPU 2 are homogeneous. In the first and second snapshot, the task with the smallest e_k is assigned to the CPU with the least $Total_{CPU}$, In the third snapshot, after assigning $R\tau_5$ and $R\tau_6$, $Total_{CPU}$ are all equal to 20. At this time, as a tie breaker, $R\tau_4$ is assigned to CPU 1 with the smallest $Subtotal_{CPU}$ which is 0.

4.3 Prioritization of DAG tasks

In the last step, the priority of the task, $prio_k$, is determined. the makespan of DAG tasks set is practically guaranteed through prioritization.

4.3.1 The makespan Delay of DAG Task set in Practical Case



(a) Higher priority to predecessor tasks (b) Higher priority to successor tasks

Figure 5: Priority scenarios

As with HEFT [9] and HLBS [8], many DAG scheduling algorithms ignore the effect of the next period's job on the current job's scheduling. However, in practical situations such as when using an autonomous driving system, the gap between the period and the expected the makespan is not enough, so the job of the next period often affects the scheduling of the job of the current period. The figure 5 shows examples. The figure 5 depicts the situation in which the job of the current period was not completed and the job of the next period was released. $R\tau_{n-1,1}, R\tau_{n,1}$ are the jobs of the current period, and $R\tau_{1,2}$ is the job of the next period. Figure 5(a) is a scheduling when a high priority is given to predecessor tasks, and Figure 5(b) is a scheduling

Algorithm 2 Successor tasks priority algorithm

```
    Input: $\Gamma^R$(real-time DAG task set)
    Output: $\mathbb{P}rio$(determined priority set)
1: $\mathbb{P}rio = \{prio_1, \cdots, prio_m\} \leftarrow \{1, \cdots, 1\}$
2: for $R\tau_i \in \Gamma^R$, in topological order do
3:     if $pred_i$ is $\emptyset$ then
4:         $prio_i \leftarrow 1$
5:     else
6:         $prio_i \leftarrow \max_{R\tau_j \in pred_j}(prio_j) + 1$
7:     end if
8: end for
9: return $\mathbb{P}rio$
```

when a high priority is given to successor tasks. In the case of Figure 5(a), $R\tau_{1,2}$ is executed first even though $R\tau_{n,1}$ has not been completed, and this delays the makespan of the current period job. Figure 5(b) gives high priority to successor tasks to prevent delay. If the makespan delay that often occurs is not considered, it can lead to a critical damage.

4.3.2 Proposed Algorithm for Determine Priority

The priority decision Algorithm 2 using the idea of 4.3 is introduced on the next page. Initialize $\mathbb{P}rio$ on Line 1. From Line 2 to Line 8, priorities are assigned to all real-time DAG tasks in topological order in the **for**-loop. The **if**-conditional branch from Line 3 to Line 5, if there is no predecessor tasks, 1 is given as a priority. This corresponds to root tasks. The **else**-conditional branch from Line 5 to Line 7, the priority is determined by adding 1 to the highest priority among predecessor tasks. Therefore, priority increases from root tasks to leaf tasks. It returns the priority set determined in Line 9.

4.3.3 Prioritization example of DAG task set

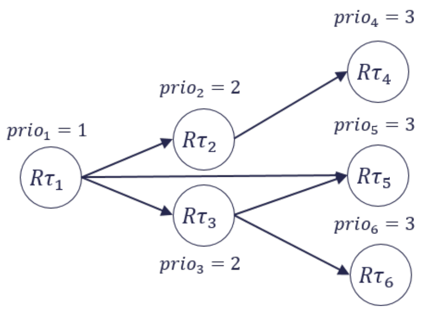


Figure 6: Example of prioritization

Figure shows an example of prioritization. The root task, $R\tau_1$, is given 1 as a $prio_1$. Next, according to the topological order, the priority of $R\tau_2$ becomes 2 by adding 1 to *max* of predecessor's *prio*. In the case of $R\tau_5$, 1 is added to the *max* of predecessor's *prio*, so 3 is given as priority.

5 Evaluation

In this section, we evaluate the optimization method presented through two experiments. First, compare the makespan on the schedule simulator by using *Random DAG Tasks Generator*. Compare *Random*, *Exhaustive*, and *Ours* to find the difference in average makespan and deadline miss rate. Second, on the autonomous system, Autoware[2], we conduct a real driving experiment using a minicar. When using only Linux CFS and using our method, we measure the makespan and compare it with the deadline for driving. As a result, we check whether the actual driving was successfully performed.

5.1 Simulated Scheduler Evaluation

5.1.1 Evaluation Setup

Parameter	**Description**	**Value**
T_{system}	Period of DAG task set	100ms
$\#_{RealTime}$	Number of real-time tasks	10
$\#_{Dummy}$	Number of dummy tasks	10
$Depth$	Depth of DAG	uniform[3,7]
e_{total}	Total execution time of system	[10ms, 200ms], (in increments of 10ms)
U_{total}	Total utilization of system depends on the value of e_{total}	[0.1, 2.0] (in increments of 0.1)
$Portion_{RT}$	The portion of real-time tasks across the system	[0.2, 0.8] (in increments of 0.3)

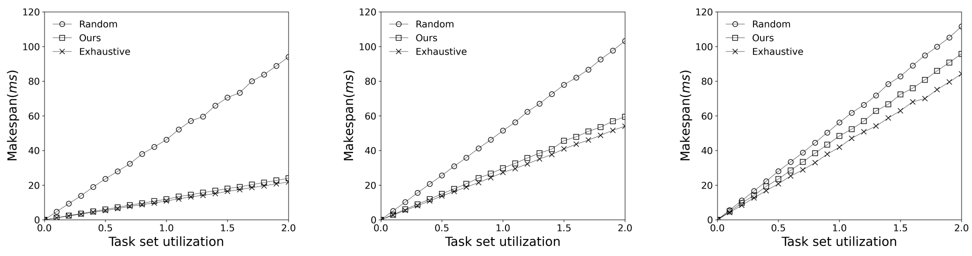
Table 2: Parameter of *Random DAG Tasks Generator*

A real-time DAG task set and a dummy task set were created through the *Random DAG Tasks Generator*. The dummy task set corresponds to a non-real time task set.

Table 2 describe parameters for task. Since the period is 100ms, when e is increased by 10ms, utilization also increases by 0.1. For each utilization, we measure the average of makespan, increasing by 0.1 with 1000 working sets. We compare the resulting from the following three different optimization methods:

- *Ours*: Scheduling by the method proposed in this paper.
- *Random*: Scheduling by randomly setting the scheduling parameter P_k.
- *Exhaustive*: Scheduling by searching all possible cases through exhaustive test and selecting the best case.

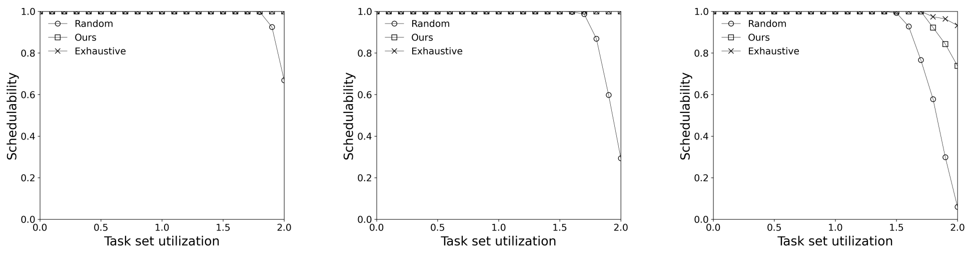
5.1.2 Evaluation Result



(a) $Portion_{RT} = 0.2$ (b) $Portion_{RT} = 0.5$ (c) $Portion_{RT} = 0.8$

Figure 7: Simulation makespan results

Figure 7 shows the simulation makespan results. Figure 7(a), 7(b). 7(c) are measured makespan when $Portion_{RT}$ is 0.2, 0.5, and 0.8, respectively. In each graph, the x-axis is the utilization of all task sets, and the y-axis is the makespan for real-time taskset. In all graphs, makespan was measured in the order of *Random*, *Ours*, and *Exhaustive*. *Exhaustive* is the best solution considering all cases, so makespan lower than this cannot be measured. As the $Portion_{RT}$ decreases, the room for optimization increases,



(a) $Portion_{RT} = 0.2$ (b) $Portion_{RT} = 0.5$ (c) $Portion_{RT} = 0.8$

Figure 8: Simulation schedulability results

and the effect of reducing makespan increases. In the case of Random, all task sets are not classified, so all $Portion_{RT}$ show a similar graph pattern. Nevertheless, when the $Portion_{RT}$ is small, the probability of completing the real-time tasks beforehand is high, so we can see that the makespan is slightly reduced. In the case of *Ours* and *Exhaustive*, the load of the real-time task set increases according to the $Portion_{RT}$, so the makespan is measured differently even at the same U_{total}.

Figure 8 shows the simulation schedulability results. In each graph, the x-axis is the utilization of all task sets, and the y-axis is the schedulability for real-time taskset. In all graphs, *Ours* has higher schedulability than *Random* and *Exhaustive* than *Ours*. As utilization increases, makespan increases, so schedulability decreases. In all graphs, the reason that schedulability does not decrease significantly even when utilization is close to the number of cores is because there is a deadline only in the real-time task set, so it actually affects as much as $Portion_{RT} * U_{total}$. In figures 8(a) and 8(b), only a few deadline misses occurred in the case of *Random*. In figure 8(c), deadline misses were also observed in *Random* and *Ours*. At this time, in the case of *Random*, schedulability is significantly lowered, while *Ours* has a graph similar to *Exhaustive* and maintains schedulability.

5.2 Autonomous System Evaluation

5.2.1 Evaluation Setup

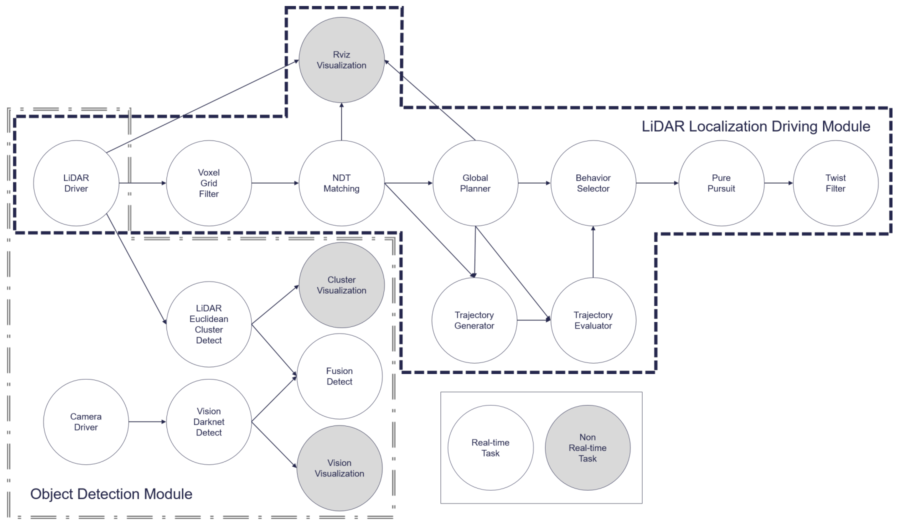
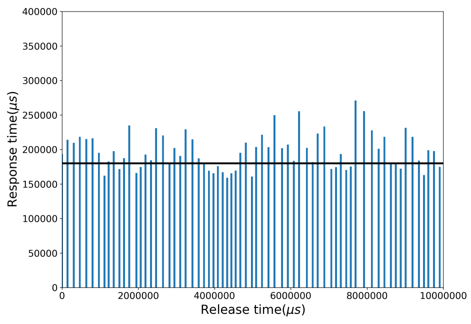


Figure 9: DAG task set structure of Autoware

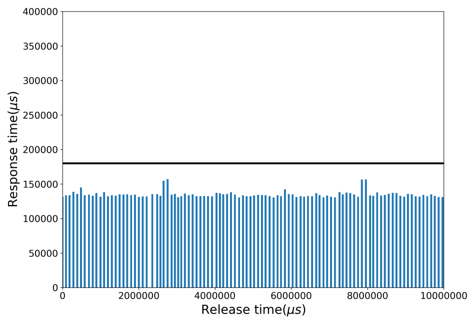
For practical verification of the proposed optimization method, an experiment using an RC minicar that has $1/10$ scale of a real car was designed. Nivida TX2 embedded board[11] is installed on the minicar and used as a hardware computing unit. TX2 board has 6 cores, which is composed of two CPU clusters. It consists of 2 Nvidia Denver2 cores and 4 ARM Cortex-A57 cores, and the clock frequency is fixed to 2.0 GHz each. Linux kernel 4.4.197-tegra is installed on the TX2 board. On top of that, ROS, a middleware, and Autoware, an autonomous driving system, were installed, and a module for actual autonomous driving was constructed.

Figure 9 shows the autonomous driving DAG task set structure. The whole system consists of two modules, LiDAR localization driving module and object detection module, and the response time of each module is obtained by the measurement of makespan. Tasks essential for driving were classified as real-time tasks, and tasks that were not essential were classified as non real-time tasks. In addition, such as kernel GUI tasks are added to the non real-time task. After measuring makespan through the Linux basic CFS and our approach, we compare the results.

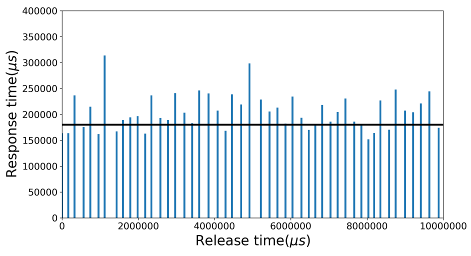
5.2.2 Evaluation Result



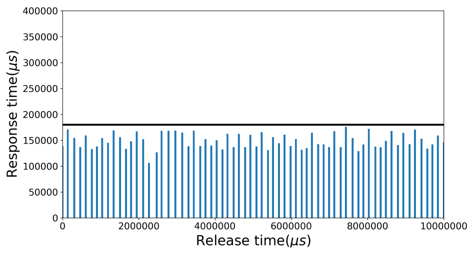
(a) Driving module on CFS



(b) Driving module on *Ours*



(c) Detection module on CFS



(d) Detection module on *Ours*

Figure 10: Autoware results

Figure 10 shows the response time measured in each module. In each graph, the x-axis is the release time of each module and the y-axis is response time of each module. The horizontal line in the graph represents the deadline for driving success, and the response time above this line represents a deadline violation. Figure 10(a) and 10(c) are measured on CFS, and deadline violations are observed. Figure 10(b) and 10(d) are graphs measured in*Ours*. In this graph, no job is observed above the horizontal line, which means that the deadline has always been satisfied. In the actual driving test, when the deadline was not satisfied on the CFS, the driving failed, but *Ours* confirmed that the driving was successful.

6 Conclusion

This paper presents a real-time system optimization method that can reduce response time without kernel patch, without application modifying. To simplify this, it is described as a problem that reduces the makespan of the DAG task set. The proposed approach determines the scheduling parameter for the task through a heuristic algorithm of three steps. First, in the first step, a scheduling policy is determined by classifying a real-time task and a non real-time task. In the second step, CPU affinity is assigned through an algorithm aimed at fair load balancing. In the third step, prioritization is performed to resolve the response time delay that can practically occur when the interval between response time and period is not enough. Through evaluation conducted by simulated scheduler and Autoware, it showed significant response time reduction effect close to optimal solution. In the future, we plan to extend our heuristic approach to the optimal algorithm.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, 2009, p. 5.

[2] 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 *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[3] B. A. team, "Apollo: Open source autonomous driving," https://github.com/ApolloAuto/apollo, 2017, online; Accessed: 2020-12-14.

[4] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "$litmus^{RT}$: A testbed for empirically comparing real-time multiprocessor schedulers," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006, pp. 111–126.

[5] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *Proceedings Real-Time Systems Symposium*. IEEE, 1997, pp. 298–307.

[6] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "Rosch: real-time scheduling framework for ros," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018, pp. 52–58.

[7] S. Kato, R. Rajkumar, and Y. Ishikawa, “A loadable real-time scheduler suite for multicore platforms,” *Technical Report CMU-ECE-TR09-12*, 2009.

[8] Y. Suzuki, T. Azumi, S. Kato *et al.*, “Hlbs: Heterogeneous laxity-based scheduling algorithm for dag-based real-time computing,” in *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2016, pp. 83–88.

[9] H. Topcuoglu, S. Hariri, and M.-y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[10] I. Molnar, “Completely fair scheduler,” https://lwn.net/Articles/230501/, 2007.

[11] N. Corporation, “Jetson tx2,” https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/, 2019.

요약(국문초록)

자율주행 시스템에서 응답시간 감소는 매우 중요하다. 이를 위해 실시간 시스템은 반드시 자율주행 시스템에 적용되어야 한다. 하지만, 실시간 시스템의 적용은 그리 간단하지 않다. 오픈 소스 플랫폼이 발전하면서, 다양한 개발자와 같은 비전공자를 위한 쉽게 적용 가능한 실시간 시스템이 필요해지게 되었다. 이 논문에서는 커널 패치 없이, 어플리케이션의 수정없이 응답시간을 감소시킬 수 있는 실시간 최적화 기법을 제시한다. 이 논문에서는 세 단계의 휴리스틱 알고리즘으로 구성된 최적화 기법을 제시한다. 이는 각각 스케쥴링 정책, 중앙 처리 장치 선호도, 우선순위를 결정하여 목표를 달성한다. 시뮬레이션 된 스케줄러와 실제 자율주행 시스템에서 진행된 실험에서, 뛰어난 응답시간 감소 효과가 있음을 보인다.

주요어 : 최적화, 실시간 스케줄링, 유향 비순환 그래프 스케줄링, 자율주행 시스템
학 번 : 2019-25586