Scheduling parallel and distributed processing for automotive data stream management system

Jaeyong Rho [a],*, Takuya Azumi [c], Mayo Nakagawa [b], Kenya Sato [d], Nobuhiko Nishio [b]

[a] Graduate School of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 5258577, Japan
[b] College of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 5258577, Japan
[c] Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 5608531, Japan
[d] Mobility Research Center, Doshisha University, Kyoto 6028580, Japan

HIGHLIGHTS

- Scheduling algorithms for automotive data stream management systems.
- End-to-end timing behavior in heterogeneous processor and network environments can be analyzed accurately.
- We consider load balancing and imprecise computation models to utilize limited resources more efficiently.
- The proposed algorithms improve schedule length, accuracy, and load balancing significantly compared to the previous algorithm.

ARTICLE INFO

ABSTRACT

In this paper, to analyze end-to-end timing behavior in heterogeneous processor and network environments accurately, we adopt and modify a heterogeneous selection value on communication contention (HSV_CC) algorithm, which can synchronize tasks and messages simultaneously, for stream processing distribution. In order to adapt the concepts of a static algorithm like HSV_CC to automotive data stream management system (DSMSs), one must first address three issues: (i) previous task and message schedules might lead to less efficient resource usages in this scenario; (ii) the conventional method to determine the task scheduling order may not be best suited to deal with stream processing graphs, and; (iii) there is a need to be able to schedule tasks with time-varying computational requirements efficiently. To address (i), we propose the heterogeneous value with load balancing and communication contention (HVLB_CC) (A) algorithm, which considers load balancing in addition to the parameters considered by the HSV_CC algorithm. We propose HVLB_CC (B) to address issue (ii). HVLB_CC (B) can deal with stream processing task graphs and more various directed acyclic graphs to prevent assigning a higher priority to successor tasks. In addition, to address issue (iii), we propose HVLB_CC_IC. To schedule tasks more efficiently with various computation times, HVLB_CC_IC utilizes schedule holes left in processors. These idle time slots can be used for the execution of an optional part to generate more precise data results by applying imprecise computation models. Experimental results demonstrate that the proposed algorithms improve minimum schedule length, accuracy, and load balancing significantly compared to the HSV_CC algorithm. In addition, the proposed HVLB_CC (B) algorithm can schedule more varied task graphs without reducing performance, and, using imprecise computation models, HVLB_CC_IC yields higher precision data than HVLB_CC without imprecise computation models.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Modern automotive systems incorporate a range of data from on-board and external sensors. Advanced driver-assistance systems, such as collision warning systems [13], help prevent accidents by alerting drivers to oncoming vehicles or pedestrians, which requires monitoring of the environment. Autonomous driving systems, such as the Google driverless car, should be able to determine when to change lanes and take appropriate action to avoid obstacles. To realize these functionalities, contemporary automotive systems require many types of data, such as own-vehicle and surrounding-vehicle information.

Automotive systems are distributed systems. Currently, modern luxury vehicles have more than 70 electronic control units (ECUs) connected to various in-vehicle networks [17]. These ECUs

* Corresponding author.
E-mail address: no@ubi.cs.ritsumei.ac.jp (J. Rho).

comprise different types of microcontrollers, from simple 4-bit controllers to complex 32-bit controllers, to handle different operations [9]. Various types of microcontrollers are managed by ECUs that are connected via multiple in-vehicle networks, such as local interconnect networks (LINs), controller area networks (CANs), and media-oriented system transport (MOST) networks [26]. Thus, automotive software development is complex, and as the complexity of data processing and amount of data increases, automotive software development will become increasingly complex.

Research has been conducted on the adaptation of data stream management systems (DSMSs) for automotive embedded systems to reduce data processing complexity and lower software development costs while increasing the amount of available data [22]. In current systems, automotive data from on-board and external sensors are processed and managed individually by separate electronic control units (ECUs). The duplication of data processing over multiple applications and the associated software development costs increase with increased amounts of data. Thus, researchers have focused on DSMSs for automotive system applications that can process streamed data at low latency using processes that are shared over multiple applications [5,22,34].

Existing DSMSs [1,2,4,6] are designed to run on general-purpose computers and primarily target network monitoring and financial analysis applications. Such applications require abundant resources but do not necessarily consider strict real-time constraints. In general-purpose DSMSs, dynamic distribution and stream processing optimization at runtime over multiple processors [7,23,36] are commonly adopted. These dynamic methods are difficult to implement in automotive embedded systems because time predictability for stream processing cannot be guaranteed and because a large number of modules would be required, which is not appropriate for automotive systems. Therefore, a distribution method that can guarantee strict time predictability for stream processing is required.

In distributed real-time systems, application tasks usually have precedence constraints so that the results can be used as input data for other tasks. Real-time applications can be described as directed acyclic graphs (DAGs) that have end-to-end deadlines in order to take advantage of parallel distributed processing. Parallel processing performance is highly dependent on task and message scheduling [10,19]. Scheduling distributes multiple tasks and messages to realize parallel processing and to satisfy precedence requirements. Scheduling an application on multiple processors to minimize overall scheduling length and parallel processing time is a multi-processor scheduling problem, which is recognized as an NP-complete optimization problem. Thus, heuristics are utilized to obtain near-optimal solutions than searching all possible scheduling patterns, which is not possible for realistic large-scale scheduling problems.

Most heuristic scheduling algorithms are based on list scheduling [3,12,16,21,24,25,32]. List scheduling consists of two phases: (i) a task prioritizing phase in which tasks in a DAG are listed by priority, and (ii) a processor-selection phase, in which the tasks with the highest priorities are scheduled. List scheduling is a generally accepted method because it provides high-quality scheduling and has low complexity [31]. Thus, many list scheduling algorithms have been proposed for near-optimal solutions in parallel and distributed systems.

However, achieving time accuracy in heterogeneous environments is difficult for most list scheduling algorithms because they use the average computation time for each task on heterogeneous processors, even though different computation times must be considered. Furthermore, they assume that all processors are fully connected so that communication can occur simultaneously. This can lead to inaccurate scheduling results in automotive systems. Given these ideal assumptions of existing list scheduling algorithms,

such algorithms have not been widely utilized in heterogeneous computing systems.

To address these assumptions, Xie et al. [11] proposed heterogeneous selection value on communication contention (HSV_CC) for automotive embedded systems based on list scheduling. HSV_CC can schedule tasks and messages simultaneously and considers communication contention. However, HSV_CC has several issues when applied to the automotive DSMSs. First, task and message scheduling results can lead to inefficient resource usage. Tasks and messages are likely to be assigned to specific processors and network links, and, in the worst case, some processors and links are poorly utilized. Second, HSV_CC was designed to schedule DAGs and consequently, some stream processing applications for automotive DSMSs may not be scheduled appropriately. Third, tasks whose requested processing times can vary over time have to be scheduled efficiently to apply HSV_CC to automotive DSMSs.

We propose the heterogeneous value with load balancing and communication contention (HVLB_CC) static list scheduling algorithm, which considers load balancing (LB) over multiple processors and can be applied to automotive DSMSs. When scheduling tasks and messages, both the HSV_CC parameters and parameters that balance the load among the processors are considered. In addition, this method achieves schedule length reduction by considering processor loads to utilize resources more efficiently. Our main contributions are as follows.

- We propose a list scheduling algorithm that considers processor LB to reduce parallel execution times (HVLB_CC (A)). The proposed algorithm can deal with important load balancing parameters in addition to HSV_CC parameters. We show that HVLB_CC can produce more accurate scheduling results with task and message synchronization. HVLB_CC can also be applied to real-time heterogeneous distributed computing systems.

- We address the scheduling problem that occurs in HSV_CC, i.e., when scheduling task graphs for stream processing, tasks are scheduled before the predecessor tasks are scheduled (HVLB_CC (B)). Compared to HSV_CC, the proposed algorithm can schedule stream processing graphs (SPGs) and different types of DAGs.

- We extend the HVLB_CC algorithm to execute data streams more efficiently (HVLB_CC_IC). HVLB_CC can exploit schedule holes to deal with varying requested processing times. We utilize imprecise computations for tasks with varying processing times. This approach produces more precise data than an approach that does not utilize imprecise computation models can produce.

- Compared to HSV_CC, the proposed algorithm demonstrates improved *speedup*, scheduling accuracy, and LB. Furthermore, the extended version of HVLB_CC with imprecise computations can produce more precise data results than the original can produce.

The remainder of this paper is organized as follows. A system model and a stream processing application model are described in Section 2. The problem is defined in Section 3. Section 4 details the proposed algorithm. Simulation results are presented in Section 5. In Section 6, previous work related to scheduling algorithms on heterogeneous distributed systems is reviewed. Conclusions and suggestions for future work are given in Section 7.

2. System model

In this section, we describe a DSMS for automotive embedded systems [14,33], assumptions about the target system, and a stream processing application model.

2.1. Stream processing

A DSMS processes *stream data* generated continuously in large volumes in real-time using a query. The query is registered with the system and executes continuously as new stream data arrives.
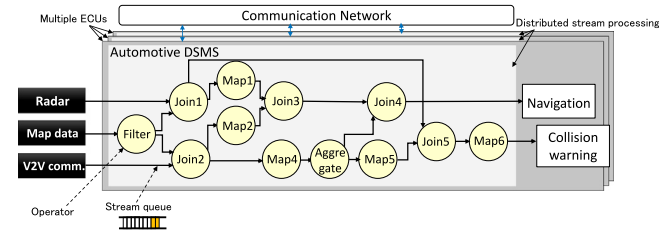
Fig. 1. Adapting DSMS to automotive systems and sample query for stream processing.

Table 1
Computation time matrix corresponding to Fig. 3.

Task	p_1	p_2	p_3
n_1	18	12	14
n_2	12	8	10
n_3	12	8	10
n_4	21	14	17
n_5	9	6	7
n_6	15	10	12
n_7	26	17	20
n_8	14	9	11
n_9	20	13	16
n_{10}	15	10	12

Fig. 2. Example of heterogeneous network topology.

By registering a query in advance, the overhead required to issue a query for each process is smaller than that required in a DBMS. This enables processing in real time. A *query* consists of one or more operators, and two consecutive operators are connected with a *stream queue*. An *operator* is a communication entity that uses a tuple, i.e., a set of data values, and outputs the computed result tuples to the next stream queue. For example, in Fig. 1, the circle and arrow in the query file represent an operator and a stream queue, respectively.

In stream processing, a precedent operator executes and produces one or more result tuples using input tuples. The result tuples are delivered to subsequent operators or applications through a stream queue. Operators used for automotive DSMSs, such as Map, Filter, and Join, are based on Borealis, which is a general-purpose DSMS [1]. The operator functionalities and other details are described in [14,34].

There are two main differences between automotive and general DSMSs. The first difference is that the automotive DSMS requires synchronous execution. The in-vehicle system has different data acquisition rates for the in-vehicle sensors and the outside vehicle communication. Therefore, the timestamps need to be aligned to synchronize execution. Second, static scheduling needs to be implemented in the in-vehicle system because it is not necessary to guarantee strict real-time processing in this case.

2.2. DSMS-based data integrated architecture

In the current automotive system architecture, data retrieved from on-board and external sensors are processed individually by separate applications embedded in an ECU because suppliers provide a product as a set consisting of software and the related sensors. Therefore, data processes can be duplicated over multiple applications in different ECUs. Furthermore, when system properties (e.g., sensor type) must be changed, large parts of the application programs must be modified. To tackle these issues, automotive DSMSs have been developed based on the data integrated architecture [14,34] shown in Fig. 1. Applications can be separated from sensor devices because data processing for multiple sensors is defined in the DSMS rather than in the sensors. Thus, data from multiple applications is accessible in a location-transparent manner. In addition, changing system properties becomes easier because applications are independent from specific sensors; thus, automotive software development costs can be reduced.

2.3. Heterogeneous processors and networks

Each operator (task) has different computation times on different ECUs (processors), because different ECUs have different capabilities. Each node n_i in a DAG has a weight w_i that represents its computational volume, i.e., the amount of computation operations needed to be executed. Each processor p_u is characterized by an execution rate μ_{p_u} because the capabilities of heterogeneous processors differ.
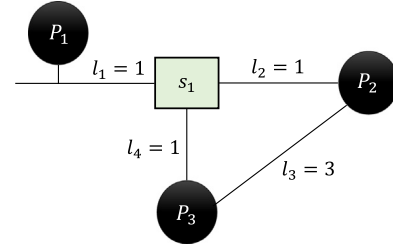
Definition 2.1. Computation time on heterogeneous processors. The processor execution rates vary according to the processor capabilities. The computation time of task n_i on p_u is given by:

$$comp(n_i, p_u) = \frac{w_i}{\mu_{p_u}}, \tag{1}$$

where μ_{p_u} is the execution rate of processor p_u.

For example, assume that the weight of n_6 is 10 and that the execution rates of processors p_1, p_2, and p_3 are 0.67, 1.0, and 0.83, respectively. Then, the computation times of n_6 on processors p_1, p_2, and p_3 are 15, 10, and 12, respectively. Computation times for other tasks can be calculated in the same manner, as shown in Table 1.

There are various types of network topologies in automotive networks, including bus, star, ring, tree, and mesh types. For example, CANs and FlexRay are configured with bus topologies; however, they can be divided by gateways to form other topologies over different domains. MOST is typically configured with a ring topology. Thus, automotive networks have a hybrid network topologies that consist of different network technologies with different bandwidths. Therefore, identical messages transmitted by different links have different communication speeds.

A heterogeneous network can be described using an undirected graph $TG = \langle P, S, L \rangle$. P is a set of heterogeneous processors, and S is a set of switches or gateways. L represents a set of v links, $L = \{l_1, l_2, \ldots, l_v\}$, and each link value represents its communication speed. A message can be expressed as $e_{i,j}^{P_{src}, P_{dest}}$ and is transmitted from task n_i to task n_j. P_{src} is a source processor assigned to n_i that sends the message to the immediate successor task, and P_{dest} is a destination processor assigned to n_j that receives the message from the predecessor task. There are one or more routes between P_{src} and P_{dest} to transmit the message $e_{i,j}^{P_{src}, P_{dest}}$. The z routes between two processors can be expressed as $R^{P_{src}, P_{dest}} = \{r_1^{P_{src}, P_{dest}}, r_2^{P_{src}, P_{dest}}, \ldots, r_z^{P_{src}, P_{dest}}\}$. Each route set in $R^{P_{src}, P_{dest}}$ consists of one or more links with different communication speeds.

An example heterogeneous automotive network topology is shown in Fig. 2 [11]. There are two routes between P_1 and P_3, denoted as $R^{P_1, P_3} = \{r_1^{P_1, P_3}, r_2^{P_1, P_3}\}$. The route sets are expressed as $r_1^{P_1, P_3} = \{l_1^1, l_4^1\}$ and $r_2^{P_1, P_3} = \{l_1^2, l_2^2, l_3^2\}$.
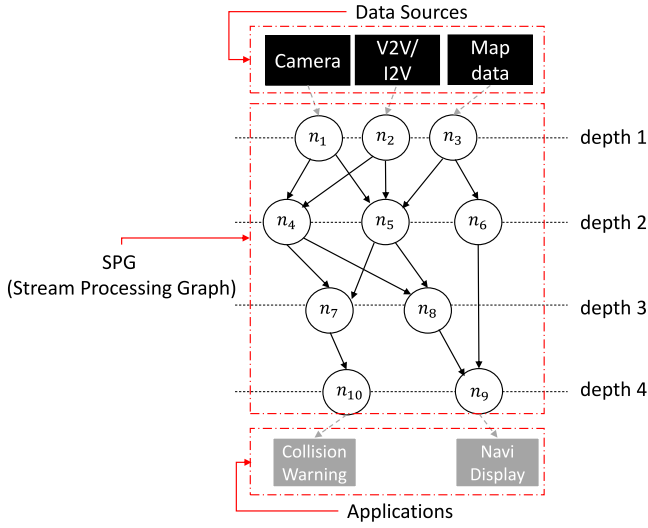
Fig. 3. Example of SPG.

Fig. 4. Gantt chart for SPG scheduling in Fig. 3 with HSV_CC algorithm (*makespan* = 73).

2.4. Stream processing application

A stream processing application can be represented by a SPG, as shown in Fig. 3, and by a DAG; thus, an SPG for a stream processing application is a type of DAG. An SPG can have tasks with a greater out-degree than the out-degree of its predecessor.

Definition 2.2. Stream Processing Graph. An SPG is a DAG consisting of a node set and a directed edge set. It can be denoted as $G = \langle V(G), E(G) \rangle$, where $V(G) = \{n_1, n_2, \ldots, n_j\}$ is a finite set of j nodes in the graph G that represents tasks, and $E(G) = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}\}$ is a finite set of directed edges that represents execution precedence between nodes.

In the SPG shown in Fig. 3, there is a set of 10 nodes $V(G) = \{n_1, n_2, \ldots, n_{10}\}$. Hereafter, the terms *node* and *task* are used interchangeably. We assume that a task can be executed only if all the predecessor operators have completed execution because the rate of data acquisition from each information source in the vehicle DSMS is different. When the data time stamps do not match in synchronous execution, incorrect results may be obtained. E is a set of directed edges $e_{i,j} \in E$ that indicates data dependencies between tasks n_i and n_j, representing that task n_j can begin execution only after task n_i completes transmission of the result. Each edge $e_{i,j} \in E$ has a weight $tpl(e_{i,j})$ that represents its communication volume, i.e., the amount of data (a tuple) to be transferred from task n_i to n_j.

The source node of an edge is called the *predecessor* and the sink node of the edge is called the *successor*. $pred(n_i)$ denotes the set of n_i's immediate predecessor tasks. $ind(n_i)$ represents n_i's in-degree, i.e., the cardinality of $pred(n_i)$. For example, in Fig. 3, $pred(n_5) = \{n_1, n_2, n_3\}$ and $ind(n_5) = 3$. We assume that a task can be executed only if all of predecessor operators have completed execution. $succ(n_i)$ is the set of n_i's immediate successor tasks and $outd(n_i)$ is n_i's out-degree, which indicates the cardinality of $succ(n_i)$. For example, $succ(n_5) = \{n_7, n_8\}$ and $outd(n_5) = 2$ in Fig. 3. A task with no predecessor task is called an *entry* task, n_{entry}, and a task with no successor is called an *exit* task, n_{exit}.

Each SPG has an end-to-end deadline. The tasks in the SPG do not have individual deadlines. Therefore, when all *exit* tasks finish execution before the DAG's end-to-end deadline, the DAG can guarantee timing constraints. The *depth* of a task is the length of the longest path from an *entry* task to the particular task. The *depth* starts at 1; thus, the *depth* of an *entry* task is 1. The *length* of
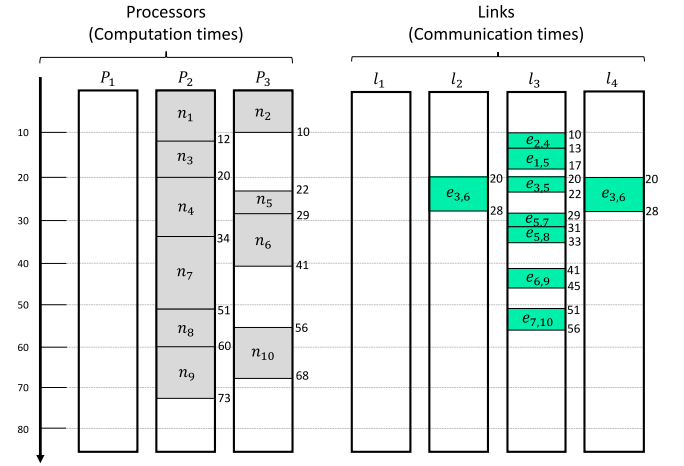
the path in the graph is equal to the number of edges on the path. For example, the longest path of n_9 is $\{n_3, n_5, n_8, n_9\}$, and its *length* is 3 because there are three edges $\{e_{3,5}, e_{5,8}, e_{8,9}\}$. Thus, the *depth* of n_9 is $1 + 3 = 4$.

Sources that generate sensor data (tuples) to an automotive DSMS can be on-board sensors such as cameras, radar, GPS systems, and external communication systems, such as vehicle-to-vehicle (V2V) and infrastructure-to-vehicle (I2V) communication systems. Various applications use stream processing results, e.g., collision warnings, automotive navigation systems, and vehicle-infrastructure cooperative right-turn collision caution signals.

3. Problem description

3.1. Inefficient resource usage

To achieve high performance in distributed and parallel stream processing, it is essential to maintain a well-balanced load among all available resources, such as processors and communication links. The previously proposed HSV_CC [11] may result in inefficient resource utilization. In the processor selection phase, each task is assigned to a processor where the minimum HSV_CC value can be obtained. The HSV_CC value can be calculated by using $HSV\_CC(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) = EFT(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) \times LDET\_CC(n_i, p_{dest})$, where EFT is the earliest finish time and $LDET\_CC$ is the longest distance exit time. $LDET\_CC$ is calculated by using $rank(n_i, p_u) - comp(n_i, p_u)$. $rank(n_i, p_u)$ is calculated by using $comp$ and $comm$; thus, the *rank* value will be smaller for higher-capability processors and communication links. Similarly, EFT will be small for each task. Therefore, the factors required to calculate HSV_CC value are highly dependent on the processor and communication link capabilities. Inefficient scheduling may occur because tasks and messages tend to be assigned to high-capability processors and communication links.

We describe a scheduling example in which tasks and messages are assigned to processors connected by links with different capabilities. As a task set, we use the task graph shown in Fig. 3 and the network topology shown in Fig. 2. Here, there are three processors (p_1, p_2, and p_3), and the computation times for 10 tasks are given in Table 1. The execution rates of the three processors are 0.67, 1.0, and 0.83.

The scheduling results of HSV_CC are shown in Fig. 4. Six tasks are assigned to p_2, and four tasks are assigned to p_3. No tasks are assigned to p_1. Communication link l_1 is not used because

no messages are transmitted to p_1. Furthermore, l_2 and l_4 only transmit messages from task n_3 to task n_6, and all other messages are transmitted through link l_3 even though l_2 and l_4 are not busy. This allocation pattern for tasks and messages, as scheduled by HSV_CC, is difficult to deal with in a situation where the amount of input sensor data and sensor fusion processing increases rapidly, because there is limited idle time after finishing a task processing and transmitting a message on the processor and communication link as most tasks and messages are concentrated on specific resources. Thus, it is evident that allocation of tasks and messages to processors and links with HSV_CC can result in inefficient resource usage.

3.2. Scheduling the stream processing graph

In HSV_CC, a task cannot be scheduled before its predecessor has been scheduled because the task computes *EFT*, which is used to select an appropriate processor based on the actual finish time of the predecessor. Assume that n_j whose $outd(n_j) = 3$ uses the results generated from n_i whose $outd(n_i) = 1$. In this case, it is a possibility that n_j could be assigned a higher priority than n_i is assigned, and n_j may be scheduled before its predecessor task has completed because, in HSV_CC, task priority is determined by $HPRV\_CC(n_i) = hrank(n_i) \times outd(n_i)$. Thus, a node with a large out-degree value can have a higher $HPRV\_CC$ value than its predecessor if the gap between their *hrank* values, which indicates the average *rank* value over all processors, is not significant. In an automotive DSMS, when stream processing must be shared among multiple applications (sink nodes), then a task located at high depth in a task graph can have a higher out-degree than the lower-depth tasks have. Thus, a high-depth task may have a higher priority than its predecessors.

3.3. Stream processing with varying execution times

A scheduling approach for automotive DSMSs should deal with variable execution times because the arrival rate of the input data can be increased. For example, in Fig. 3, it is assumed that n_2 inputs stream data from an external source, such as V2V and I2V, and transforms the data format. Here, n_5 is a matching process with map data and external vehicle information. The arrival rates of input data to n_2 and n_5 can vary because the volumes and arrival times of the data streams from external sources can change depending on the environmental conditions, e.g., many V2V data streams are generated at congested urban intersections. However, static list scheduling HSV_CC considers tasks whose computation times do not vary. Therefore, a task with varying computation times is difficult to be scheduled efficiently, and data quality can decrease as the arrival rate of an external data stream increases.

4. HVLB_CC algorithm

Here, we describe the proposed HVLB_CC static list scheduling algorithm in detail, including the task prioritizing, processor selection, and scheduling phases.

As described in Section 6, concurrent communication is allowed and network heterogeneity is not considered in most existing list scheduling algorithms. Moreover, task and message scheduling are not synchronized, which can influence schedule length. Thus, such scheduling schemes are unrealistic scheduling for automotive embedded systems. As mentioned previously, HSV_CC [11] has been proposed to address these issues. However, as described in Section 3, HSV_CC cannot be applied directly to automotive DSMSs due to inefficient resource usage, the possibility that SPGs cannot be scheduled, and its inability to handle stream processing

if the execution times varies. The proposed HVLB_CC algorithm addresses these issues and can be applied to automotive DSMSs.

In the task prioritizing phase, tasks in an SPG are prioritized according to HVLB_CC values to prevent a high-depth task from being scheduled earlier than its predecessor at the lower depth. In the processor selection phase, tasks are allocated to appropriate processors while considering processor load. Tasks and messages are scheduled to the corresponding processor and communication links with specific start and finish times.

4.1. Task prioritizing phase

In the task prioritizing phase, a *rank* value must be calculated and assigned to each node. The rank value is calculated recursively by traversing the task graph (i.e., the DAG), starting from the exit node. Thus, it is sometimes referred to as the *upward rank*; however, we simply refer to it as the *rank*. The computation of the *rank* value in most existing list scheduling algorithms uses the average computation and communication times. However, calculating the *rank* value using average times cannot produce accurate scheduling results in heterogeneous computing and networking environments [11]. Therefore, we employ different computation and communication times for the different heterogeneous processors and communication links in the proposed method to calculate the *rank* value more accurately, which can be expressed as follows:

$$\begin{cases} rank(n_i, p_{src}) = comp(n_i, p_{src}) \\ \quad + \max_{n_j \in succ(n_i)} \{ rank(n_j, p_{src}) + comm_{i,j}^{p_{src}} \}; \\ rank(n_{exit}, p_{src}) = comp(n_{exit}, p_{src}); \end{cases} \quad (2)$$

where $comm_{i,j}^{p_{src}}$ denotes the communication time from n_i located on source processor p_{src} to n_j. A higher *rank* indicates that the corresponding node has higher criticality and more influence on the execution of other nodes; thus, a node with a higher *rank* will be scheduled earlier than a node with lower *rank*.

To calculate the communication speed of source processor p_{src}, we must first compute the route $speed(R^{p_{src}, p_{dest}})$ between the source processor p_{src} and the destination processor p_{dest}, which can be expressed as follows:

$$speed(R^{p_{src}, p_{dest}}) = \frac{1}{z} \sum_{t=1}^{z} speed(r_t^{p_{src}, p_{dest}}). \quad (3)$$

Here, z sub-routes exist between source processor p_{src} and destination processor p_{dest}. A route between p_{src} and p_{dest} consists of one or more sub-routes and can be expressed as $R^{p_{src}, p_{dest}} = \{ r_1^{p_{src}, p_{dest}}, r_2^{p_{src}, p_{dest}}, \ldots, r_z^{p_{src}, p_{dest}} \}$. Eq. (3) indicates the average values of the speeds for all sub-routes between p_{src} and p_{dest}, i.e., the speeds of the communication routes between p_{src} and p_{dest}.

The speed of a route consisting of one or more sub-routes can be determined using the minimum speed communication link because the message transmission time through multiple links is dependent on the link with the lowest speed. Thus, we can define communication route speed between p_{src} and p_{dest} as the average of the lowest speed value for all of the sub-routes in the communication route. This communication route speed can be expressed as follows:

$$speed(R^{p_{src}, p_{dest}}) = \frac{1}{z} \sum_{t=1}^{z} \min_{l_v \in L} \{ speed(l_v^t) \}. \quad (4)$$

Assume that there are two sub-routes between p_2 and p_3, $r_1^{p_2, p_3} = \{l_2, l_4\}$ and $r_2^{p_2, p_3} = \{l_3\}$. The speeds of the sub-routes are 1 and 3 for $r_1^{p_2, p_3}$ and $r_2^{p_2, p_3}$, respectively, because the minimum link speed in the sub-route is the communication route speed. The speeds for the other routes can be calculated in the same way and are shown in Table 3. (See Table 2.)

Table 2

rank, *hrank*, *outd*, *depth*, and *HPRV* values (Fig. 3).

Task	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
$rank(n_i, p_1)$	145.0	133.0	109.0	109.0	85.0	50.0	67.0	48.0	20.0	15.0
$rank(n_i, p_2)$	81.66	74.99	61.66	61.66	48.33	29.67	38.33	28.0	13.0	10.0
$rank(n_i, p_3)$	96.99	90.33	73.67	73.67	57.0	36.0	45.33	34.33	16.0	12.0
$hrank(n_i)$	107.9	99.4	81.4	81.4	63.4	38.6	50.2	36.8	16.3	12.3
$outd(n_i)$	2	2	2	2	2	1	1	1	0	0
$depth(n_i)$	1	1	1	2	2	2	3	3	4	4
$HPRV(n_i)$	215.8	198.8	162.8	162.8	126.8	38.6	50.2	36.8	0	0
$HPRV(n_i, G)$	107.9	99.4	81.4	20.4	15.9	9.7	5.6	4.1	0	0

Table 3

Example of communication speeds of routes.

Route	R^{p_1, p_2}	R^{p_1, p_3}	R^{p_2, p_3}	R^{p_2, p_1}	R^{p_3, p_1}	R^{p_3, p_2}
Speed	1	1	2	1	1	2

Then, we compute the data transfer speed of the source processor, $speed(p_{src})$, which is the average value of all of the communication route speeds between the source processor and one or more destination processors and can be expressed as follows:

$$speed(p_{src}) = \frac{1}{|P| - 1} \sum_{p_{dest} \in P, p_{dest} \neq p_{src}} speed(R^{p_{src}, p_{dest}}). \quad (5)$$

Note that the data transfer speeds of other source processors can be calculated using Eq. (5). For p_1, p_2, and p_3, the speeds are 1.0, 1.5, and 1.5, respectively.

The communication time, $comm_{i,j}^{p_{src}}$, required to transmit data from task n_i on p_{src} to task n_j can be computed by dividing the message size by the data transfer speed of the source processor. It can be expressed as follows:

$$comm_{i,j}^{p_{src}} = \frac{tpl(e_{i,j})}{speed(p_{src})}, \quad (6)$$

where $speed(p_{src})$ is the data transfer speed of the source processor defined by Eq. (5). If n_i and n_j are on the same processor, then the communication time between them is considered to be negligible.

In existing algorithms [15,16,18,29,31,35], the average processor speed is used because these algorithms assume that communication contention has not occurred; thus, the same communication times are used for all heterogeneous processors. In contrast, we consider communication contention by defining the processor data transfer speed, while most existing algorithms use the same data transfer times for of the heterogeneous processors. This principle is based on HSV_CC.

In addition, *rank* has different values for different heterogeneous processors, as shown in Eq. (2). The average *rank* values for all possibly assign processors represented by *hrank* are defined as follows:

$$hrank(n_i) = \frac{1}{|P|} \times \sum_{p_u \in P} rank(n_i, p_u). \quad (7)$$

While priority is given to each task, HSV_CC considers *hrank* and *outd* because considering the out-degree of a node can minimize the schedule length simultaneously [11]. The heterogeneous priority rank value (i.e., *HPRV_CC*) used for task prioritization is defined as

$$HPRV\_CC(n_i) = hrank(n_i) \times outd(n_i). \quad (8)$$

HPRV_CC is used in the proposed algorithm, which we refer to as HVLB_CC (A). In the *HPRV_CC* equation, a node with many immediate successor nodes can have a higher probability of being assigned earlier. However, Eq. (8) can restrict scheduling for various task graphs because only task graphs with $HPRV\_CC(n_p) \geq$

$HPRV\_CC(n_s)$ are used, where n_p and n_s are predecessor and successor tasks, respectively. If the *hrank* values of these two tasks do not change significantly and n_s has a large number of out-degrees, then $HPRV\_CC(n_p) < HPRV\_CC(n_s)$ can occur.

In stream processing, successor tasks located at high depths in the task graph can have greater *outd* values than those of their predecessor tasks located at lower depths. To prevent a high-depth task from being scheduled earlier than its predecessor at a lower depth, we define HPRV_CC (B) as follows:

$$HPRV\_CC(n_i, G) = hrank(n_i) \times \frac{outd(n_i)}{\max\{outd(G)\}} \times \frac{1}{depth(n_i)^2}, \quad (9)$$

where $\max\{outd(G)\}$ indicates the maximum out-degree value of the task graph G. As shown in Eq. (9), we decrease the influence of *outd* on task prioritizing and consider the *depth* of each task in the task graph to prevent assigning a higher priority to a successor task located at a higher depth.[1] The proposed algorithm that employs Eq. (9) is referred to as HVLB_CC (B).

4.2. Processor selection phase

In this phase, the ordered tasks in the priority queue are assigned to appropriate processors that can provide them with the smallest possible *HVLB_CC* values. In HSV_CC, *EFT* and the longest distance exit time with communication contention (*LDET_CC*) are important factors in processor selection. However, these two factors are highly dependent on the capabilities of processors and communication links. Inefficient schedule decisions in which tasks and messages tend to be assigned to high-capability processors and links may occur (Section 3). As a result, HSV_CC cannot prevent specific processors and links from becoming congested with tasks and messages. Thus, we propose an approach that considers processor load while allocating tasks to appropriate processors so that tasks and messages are not concentrated on specific resources. By considering both of the factors used in HSV_CC and the processor loads, the proposed algorithm can decrease the schedule length and balance the processor load better than HSV_CC can.

The earliest start time (*EST*) of a task is determined by the processor capability (i.e., the speed) and the communication route from the source processor to the destination processor. For an entry node, the value of *EST* for a node will be 0 if the entry task is scheduled on the processor at the first time. If any node has already been scheduled, *EST* equals its maximum value of 0 and available processor time. Therefore, *EST* is defined by Eq. (10). With the exception of the entry nodes, each node must consider the time at which message transmission from its predecessor is completed:

$$\begin{cases} EST(n_{entry}, p_{dest}, r_z^{p_{src}, p_{dest}}) = \max\{0, avail[p_{dest}]\}; \\ EST(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) = \max\{avail[p_{dest}], \\ \quad \max_{n_i \in pred(n_j), proc(n_i) = p_{src}, p_{src} \in P}\{MFT(e_{i,j}^{p_{src}, p_{dest}}, r_z^{p_{src}, p_{dest}})\}\}; \end{cases} \quad (10)$$

where $MFT(e_{i,j}^{p_{src}, p_{dest}}, r_z^{p_{src}, p_{dest}})$ is the time at which the message transmission on sub-route $r_z^{p_{src}, p_{dest}}$ is completed, and $avail[p_{dest}]$ is

---

[1] We explain why we use $depth(n_i)^2$ in Eq. (9) in Section 5.

the available time of processor p_{dest}. *MFT* can be determined using the finish time of the last link l_{end}^z of sub-route $r_z^{p_{src}, p_{dest}}$. Therefore, *EST* can be calculated using the link finish time (*LFT*) rather than *MFT*, as follows:

$$\begin{cases} EST(n_{entry}, p_{dest}, r_z^{p_{src}, p_{dest}}) = \max\{0, avail[p_{dest}]\}; \\ EST(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) = \max\{avail[p_{dest}], \\ \quad \max_{n_i \in pred(n_j), proc(n_i) = p_{src}, p_{src} \in P} \{LFT(e_{i,j}^{p_{src}, p_{dest}}, l_{end}^z, r_z^{p_{src}, p_{dest}})\}\}. \end{cases} \tag{11}$$

EFT can be defined as follows for a task in a communication contention environment:

$$\begin{aligned} EFT(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) \\ = EST(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) + comp(n_j, p_{dest}). \end{aligned} \tag{12}$$

According to the above timing analysis, we can synchronize and schedule tasks and messages simultaneously. The value of *EST* for a task depends on the message transmission time, which in turn depends on the actual value of *LFT*. To obtain *LFT*, we must first compute the link start time (*LST*).

LST, the time at which message $e_{i,j}^{p_{src}, p_{dest}}$ begins to be transmitted through sub-route $r_z^{p_{src}, p_{dest}}$, can be expressed as follows:

$$\begin{cases} LST(e_{i,j}^{p_{src}, p_{dest}}, l_1^z, r_z^{p_{src}, p_{dest}}) = \max\{AFT(n_i), avail(l_1^z)\}; \\ LST(e_{i,j}^{p_{src}, p_{dest}}, l_{x+1}^z, r_z^{p_{src}, p_{dest}}) \\ \quad = \max\{LST(e_{i,j}^{p_{src}, p_{dest}}, l_x^z, r_z^{p_{src}, p_{dest}}), avail(l_{x+1}^z)\}; \end{cases} \tag{13}$$

where $AFT(n_i)$ is the actual finish time of task n_i, and $avail(l_z^x)$ is the available time of link l_z^x. When *LST* is calculated using Eq. (13), we can obtain *LFT* as follows:

$$\begin{cases} LFT(e_{i,j}^{p_{src}, p_{dest}}, l_1^z, r_z^{p_{src}, p_{dest}}) \\ \quad = LST(e_{i,j}^{p_{src}, p_{dest}}, l_1^z, r_z^{p_{src}, p_{dest}}) + CTML(e_{i,j}^{p_{src}, p_{dest}}, l_1^z); \\ LFT(e_{i,j}^{p_{src}, p_{dest}}, l_{x+1}^z, r_z^{p_{src}, p_{dest}}) \\ \quad = \max\{LFT(e_{i,j}^{p_{src}, p_{dest}}, l_x^z, r_z^{p_{src}, p_{dest}}), \\ \quad\quad LST(e_{i,j}^{p_{src}, p_{dest}}, l_{x+1}^z, r_z^{p_{src}, p_{dest}}) + CTML(e_{i,j}^{p_{src}, p_{dest}}, l_{x+1}^z)\}; \end{cases} \tag{14}$$

where $CTML(e_{i,j}^{p_{src}, p_{dest}}, l_{x+1}^z)$ indicates the communication time to transmit message $e_{i,j}^{p_{src}, p_{dest}}$ through link l_{x+1}^z. This time can be expressed as follows:

$$CTML(e_{i,j}^{p_{src}, p_{dest}}, l_x^z) = \frac{tpl(e_{i,j}^{p_{src}, p_{dest}})}{speed(l_x^z)} \tag{15}$$

where $speed(l_x^z)$ represents link l_x^z's communication speed and $tpl(e_{i,j}^{p_{src}, p_{dest}})$ is the amount of a message transmitted from the source processor to the destination processor.

In HSV_CC, the *downward* factor (i.e., *EFT*) and the *upward* factor (Eq. (16)) are considered:

$$LDET\_CC(n_i, p_u) = rank(n_i, p_u) - comp(n_i, p_u). \tag{16}$$

By considering both factors in the processor selection criteria, HSV_CC achieves overall schedule lengths that are superior to those of existing algorithm [11]. Note that LDET_CC will be 1.0 for exit nodes.

Definition 4.1. Balancing Parameter (BP)

$$BP(p_{dest}, \alpha) = 1.0 + \left(\frac{\sum_{k=1}^m comp(n_k, p_{dest})}{period} \right) \times \alpha. \tag{17}$$

Here, m is the number of tasks assigned to the destination processor p_{dest}, and the numerator indicates the cumulative processing time of m tasks. *period* represents the DAG (application) period and deadline. *BP* begins at 1.0, and the CPU utilization of each processor is calculated based on the cumulative computation times of the assigned tasks to p_{dest}. The parameter α is used to adjust the amount of *BP* influence on the value of *HVLB_CC*.
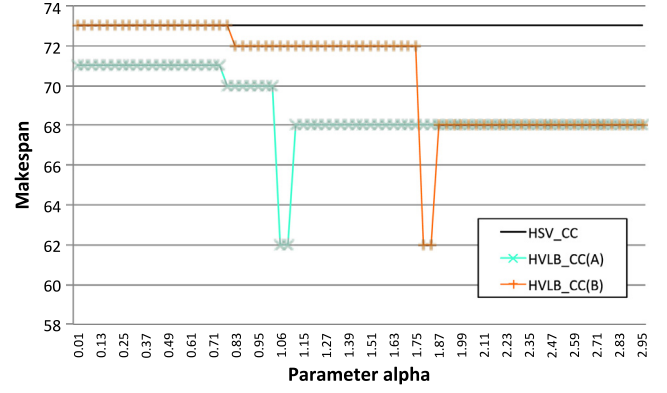

Fig. 5. Effect of parameter α on schedule length in proposed algorithms.

Definition 4.2. Heterogeneous Value with Load Balancing and Communication Contention

$$\begin{cases} HVLB\_CC(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) \\ \quad = EFT(n_j, p_{dest}, r_z^{p_{src}, p_{dest}}) \times LDET\_CC(n_i, p_{dest}) \times BP(p_{dest}, \alpha); \quad (18) \\ HVLB\_CC(n_{exit}, p_{dest}, r_z^{p_{src}, p_{dest}}) = EFT(n_{exit}, p_{dest}, r_z^{p_{src}, p_{dest}}). \end{cases}$$

The proposed algorithm can consider *downward* and *upward* simultaneously and can balance loads over multiple processors. This is the most significant improvement over HSV_CC. *BP* helps balance the processor load by giving low-load processors the chance to be selected. For a task scheduled at last, we consider only the *EFT* value to reduce the overall schedule length.

4.3. Scheduling algorithm

We present a heuristic task and message scheduling approach (i.e., HVLB_CC) to solve the unbalanced assignment problem that occurs in HSV_CC. To consider LB while scheduling dependent tasks over multiple processors, the HVLB_CC algorithm searches for the optimal task and message assignments by adjusting α which is described in Definition 4.1.

As shown in Fig. 5, we can search for the optimal assignment that produces a value of *makespan* smaller than the value that can be produced by HSV_CC. We initiated α at zero and increased its value from 0.01 to 3.0. In addition, the *CCR* was fixed to 1 in this example. For HVLB_CC (A), when α is in the range [0.0, 0.79], *makespan* is 73, which is the same as its value for HSV_CC. *makespan* is 72 in the range [0.8, 1.76], and we can obtain the minimum *makespan* value of 62 in the range [1.77, 1.83]. From 1.84 to 3.0, *makespan* is constant at 68. Similarly, the minimum values of *makespan* are 71, 70, 62, and 68 in the ranges [0.0, 0.78], [0.79, 1.05], [1.06, 1.1], and [1.11, 3.0], respectively, for HVLB_CC (B). Then, the proposed algorithms selected the scheduling results that could produce the minimum *makespan* value with α in the ranges [1.77, 1.83] and [1.06, 1.1] for HVLB_CC (A) and HVLB_CC (B), respectively.

The proposed HVLB_CC approach is described in **Algorithm 1**. The task order in the priority queue in HVLB_CC (A) is $\{n_1, n_2, n_3, n_4, n_5, n_7, n_6, n_8, n_9, n_{10}\}$, which is the same as it is in HSV_CC. In HVLB_CC (B), the task order in the priority queue is $\{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}\}$. The α values that can record the minimum *makespan* values are 1.77 and 1.06 in HVLB_CC (A) and HVLB_CC (B), respectively. The minimum values of *makespan* in scheduling results obtained using HVLB_CC (A) and HVLB_CC (B) are both 62, while *makespan* is 73 for HSV_CC. Thus, the proposed HVLB_CC algorithm yields smaller values of *makespan* and improves the load balance between three processors in comparison

Algorithm 1: HVLB_CC Algorithm

Input: Application (SPG), the number of processors, and network topology

Output: Assignment result for the SPG, i.e., start times and finish times are determined for all tasks and messages

1   Calculate *rank*, *hrank*, *LDET_CC*, and *HPRV_CC* for each task;
2   Calculate *depth* for each task; // if HVLB_CC (B) is used
3   Enqueue tasks into a priority queue according to a nonincreasing order of *HPRV_CC*;
4   $\alpha = 0$;
5   **for** *cycle* $\leftarrow$ 1 **to** k **do**
6   **while** *not all tasks in the priority queue are scheduled*
7   **do**
8   Dequeue the task n_i has the maximum *HPRV_CC*;
9   Compute *BP* using Eq. (17) in all processors;
10   Compute the *HVLB_CC* using Eq. (18) for n_i in all processors;
11   Schedule n_i on to the corresponding processor and the messages to the corresponding links that minimize *HVLB_CC*;
12   Mark n_i as a scheduled task;
13   **end**
14   Compute the *makespan* of the assignment result;
15   **if** *the makespan is the minimum* **then**
16   *minimum alpha* $\leftarrow \alpha$;
17   **end**
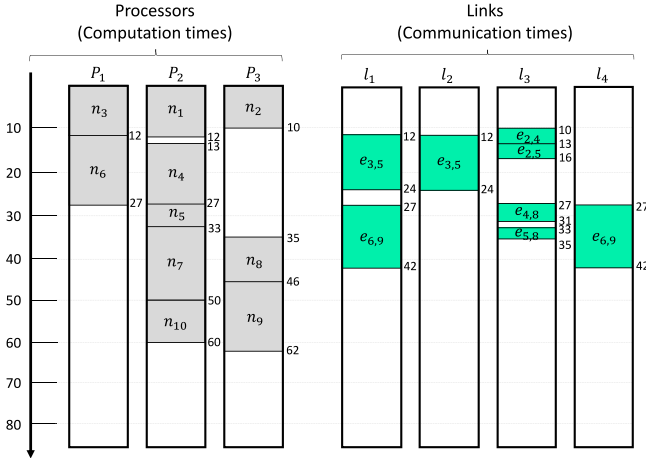18 $\alpha \leftarrow \alpha + 0.01$;
19 **end**

Fig. 6. Gantt chart of SPG scheduling in Fig. 3 with the proposed HVLB_CC (*makespan* = 62).

to the HSV_CC algorithm. A Gantt chart of the scheduling results obtained using the proposed HVLB_CC algorithm is shown in Fig. 6. Note that the scheduling results at the minimum *makespan* values for HVLB_CC (A) and HVLB_CC (B) are the same.

Here, we discuss the time complexity of the proposed heuristic HVLB_CC algorithm, which can be evaluated as follows. Assume that there are *n* computation tasks and *p* networked processors. *rank*, *hrank*, and *HPRV_CC* must be calculated for all of the tasks, and these values must be compared for all of the processors, which can be performed within $O(p \times n)$. Note that scheduling must traverse all of the tasks and can be performed within $O(n)$. The scheduling to find the minimum *makespan* value in the range of α can be performed within $O(k)$. Therefore, the complexity of the proposed HVLB_CC algorithm is $O(p \times n^2 \times k)$ whereas that of HSV_CC is

$O(p \times n^2)$ [11]. However, the proposed algorithm can produce more accurate schedule results and decrease *makespan*.

In the time complexity of the proposed algorithm, parameter *k* increases in the range of α, which is used to adjust the value of *BP* (the balancing parameter) relative to that of HVLB_CC. The parameter *k* in $O(p \times n^2 \times k)$ will not affect the entire time complexity significantly compared to the value of parameter n^2 because n^2 requires greater computational time. Even though the time complexity of the proposed algorithm is greater than that of HSV_CC, our algorithm can provide more accurate scheduling results and decreased makespan.

4.4. Imprecise computation model

As mentioned earlier, the proposed HVLB_CC_IC scheduling algorithm utilizes imprecise computations. We assume that a premature task can record its latest intermediate result when it terminates execution. Note that not all of the tasks in an SPG are applied in the imprecise computations model; instead, only those tasks whose input data rates vary are employed. An example of a task for which the amount of required processing fluctuates is a task that requires V2V communication to exchange data that changes in amount according to the surrounding situation. The computation time of each task adopted in the imprecise computation model is assumed to consist of a *mandatory part* mp_i, which is followed by an *optional part* op_i. This can be defined as follows:

$$comp(n_i, p_u) = mp_i + op_i \tag{19}$$

where $0 < mp_i < comp_i$. For example, the process required to determine the navigation in the navigation system is the mandatory part of the model and the road-to-vehicle communication is the optional part. An execution of the *optional part* can begin after the processing *mandatory part* has been processed. In case an imprecise computation model task finishes executing only its *mandatory part* or terminates before completing execution of the *optional part*, then the results of the task are *imprecise*. In contrast, the results of a task are *precise* when the task processes all of the *mandatory* and *optional* parts.

4.4.1. Searching for schedule holes

The schedule results produced by the proposed algorithm can have possible idle time slots called schedule holes. Due to the precedence constraints between tasks, schedule holes may occur in the schedule of a particular processor. For example, for a task that receives two pieces of input data from each predecessor task, the times at which the pieces of data are generated by the two predecessor tasks are generally not the same. Thus, if the data arrives from one predecessor task much earlier than it does from the other task, the processor idle time slots can occur between the time at which the data is sent from the predecessor task and the time at which all of the required data arrive at the successor task. Note that not all of the processor idle time slots are available for execution of an *optional part*.

Schedule holes that can be utilized for an *optional part* of a task can be calculated using the following equations. If a processor with an allocated predecessor node n_p is the same as the processor with successor node n_s, the schedule hole between them can be calculated as follows:

$$condition1(n_p) = \min_{n_s \in succ(n_p), proc(n_p)=proc(n_s)} \{ \min \{EST(n_s), EST(n_{np})\} - (EST(n_p) + comp(n_p, p_{src})) \} > 0 \tag{20}$$

where n_{np} is a node executed after n_p on the $proc(n_p)$.

If a processor with an allocated precedence node differs from the processor with the successor node, the schedule hole between them can be calculated as follows:

$$condition2(n_p) = \min_{n_s \in succ(n_p), proc(n_p) \neq proc(n_s)} \{ \min \{EST(n_s), EST(n_{np}), LST''(e_{p,s}^{p_{src}})\} - (EST(n_p) + comp(n_p, p_{src})) \} > 0, \tag{21}$$

where $LST''(e_{p,s}^{p_{src}})$ is the recalculated value of *LST* that is delayed to time that can be maximized the schedule hole and must not influence the data receive time to successor node.

A task with one successor task allocated to the same processor or a different processor with its own task must satisfy either *condition 1* or *condition 2*. For a task with more than two successor tasks, schedule holes occur for the execution of the *optional part* when both *condition 1* and *condition 2* are satisfied. The maximum amount of available processor time slots will be the minimum value of the results of *condition 1* and *condition 2*.

5. Performance evaluation

The performance evaluation was conducted in five parts. We compared the existing algorithm (HSV_CC) and the proposed algorithms (HVLB_CC (A) and HVLB_CC (B)) in terms of (i) the schedule length ratio (*SLR*) and *speedup* with increasing number of tasks, (ii) LB, (iii) the value of *SLR* with increasing communication to computation ratio (*CCR*), (iv) the scheduling failure rate, and (v) the data precision in HVLB_CC with and without an imprecise computation model.

5.1. Experimental metrics

The performance comparisons of the algorithms were based on *SLR* and *speedup* which are used in [11], and LB [20]. *SLR* is a normalized schedule length, and it is computed by dividing schedule length (*makespan*) by the minimum execution time for all of the tasks in the critical path (CP) among multiple processors. It is defined as follows:

$$SLR = \frac{makespan}{\sum_{n_i \in CP} \min_{p_u \in p} \left[comp_{(n_i, p_u)} \right]}. \tag{22}$$

The *speedup* value for a given schedule is computed by dividing the minimum sequential execution time by the *makespan* value of the parallel execution time and is defined as follows:

$$speedup = \frac{\min_{p_u \in p} \left[\sum_{n_i \in N} comp(n_i, p_u) \right]}{makespan} \tag{23}$$

where the minimum sequential execution time is computed by assigning all of the tasks to a single processor that can minimize the cumulative computation time.

LB is computed by dividing *makespan* by the ratio of the sum of processing time of each processor and the number of processors (*Avg*) and can be defined as follows:

$$LB = \frac{makespan}{Avg} \tag{24}$$

where *Avg* is the average execution time over all processors. *Avg* can be calculated by using

$$Avg = \frac{\sum_{k=1}^{n} \sum_{n_i \in N_k} comp(n_i, p_k)}{n} \tag{25}$$

where n is the number of processors. N_k represents a set of tasks allocated to the processor p_k. Note that the numerator in this equation indicates the sum of computation times of m allocated tasks for all n processors.

5.2. Simulation setup

The task graphs for our experiments were generated by using Task Graphs For Free (TGFF) version 3.5 [8]. TGFF can generate random DAGs for various parameters, such as number of tasks and maximum in-degree and out-degree. We set the experimental parameters as follows: number of tasks $n = \{10, 20, 30, 40, 50\}$,

maximum in-degree $i = 2$, and maximum out-degree $o = 3$, and the minimum number of entry and exit nodes was 2. In experiments (i)–(iii), we generated random task graphs with the constraint $outd(n_p) \geq outd(n_s)$, where n_s uses the results from n_p, because HSV_CC fails to schedule task graphs without the out-degree constraint. These parameters were determined by considering the actual operator sets used in an automotive DSMS.

CCR is the communication-to-computation ratio. A low value of *CCR* in a task graph can be considered as computation intensive processing (application). If $CCR > 1$ or higher, the communication time is greater than the computation time. In addition, the heterogeneity of communication becomes more obvious. On the other hand, the processing is communication-intensive if *CCR* is high. If $CCR < 1$, computation dominates the systems. In these experiments, we set $CCR = \{0.1, 0.5, 1.0, 5.0, 10.0\}$.

Three processors with different capabilities were used in all of the experiments. The execution rate μ of each processor was selected from [0.67, 0.83, 1.0]; thus, there were six execution rate patterns for the processors.[2] Note that the network topology[3] shown in Fig. 2 was applied in all of the experiments.

5.3. Experiment 1: Comparison of scheduling accuracy and performance

Here, we discuss the *SLR* and *speedup* results for a varying number of tasks to verify the superiority of the proposed algorithms in terms of scheduling accuracy and performance. For each number of tasks, we randomly generated 100 task graphs with variable computation times according to the execution times of the three processors. Note that *CCR* was set to 1. The number of tasks increased from 10 to 50 at increments of 10. These numbers were determined based on a real stream processing case in an automotive DSMS. In each task graph, α, which can adjust the influence of LB on the processor selecting phase, varies from 0 to 20 in 0.01 increments. The scheduling results that could minimize *makespan* by increasing α were chosen in the proposed HVLB_CC (A) and HVLB_CC (B) algorithms. Fig. 7 shows the comparative results for *SLR* and *speedup*.

We first present the experimental results for *SLR* that were obtained by HSV_CC and the proposed HVLB_CC (A) and HVLB_CC (B) algorithms with three processors. As shown in Fig. 7(a)–(f), HVLB_CC (A) and HVLB_CC (B) always outperform HSV_CC in terms of *SLR* in the three processor execution time rate patterns. The worst values of *SLR* obtained using HSV_CC are 3.49, 3.92, 3.62, 3.61, 3.84 and 3.79, corresponding to processor execution rates of [1.0, 0.67, 0.83], [0.83, 0.67, 1.0], [0.67, 0.83, 1.0], [1.0, 0.83, 0.67], [0.83, 1.0, 0.67], and [0.67, 1.0, 0.83] respectively. When the processor execution rate is [1.0, 0.67, 0.83], *SLR* is the lowest value because the tasks and messages are scheduled on high-capability processors and communication links. Thus, *makespan* can be decreased because the tasks are executed in less time due to the high-capability processors. When the processor execution rates are [0.83, 0.67, 1.0], the tasks are likely to be assigned to p_1 because it is the higher-capability processor. In these two processor patterns, the tasks and messages are nearly equally assigned to the three processors; however, *makespan* increases because some tasks are executed to the lowest-capability processor.

The worst *SLR* values obtained using HVLB_CC (A) and HVLB_CC (B) are 3.06, 3.15, 2.99, 3.29, 3.04 and 3.19, and 3.11, 3.12, 2.99, 3.11, 2.99 and 3.12, corresponding to processor execution rates of [1.0, 0.67, 0.83], [0.83, 0.67, 1.0], [0.67, 0.83, 1.0],

---

[2] To simplify the comparison, the experimental results of only three patterns of the processor execution rates are given in this paper.

[3] The network topology is the same as that one presented in [11] to compare the scheduling performance with that of the existing algorithm.
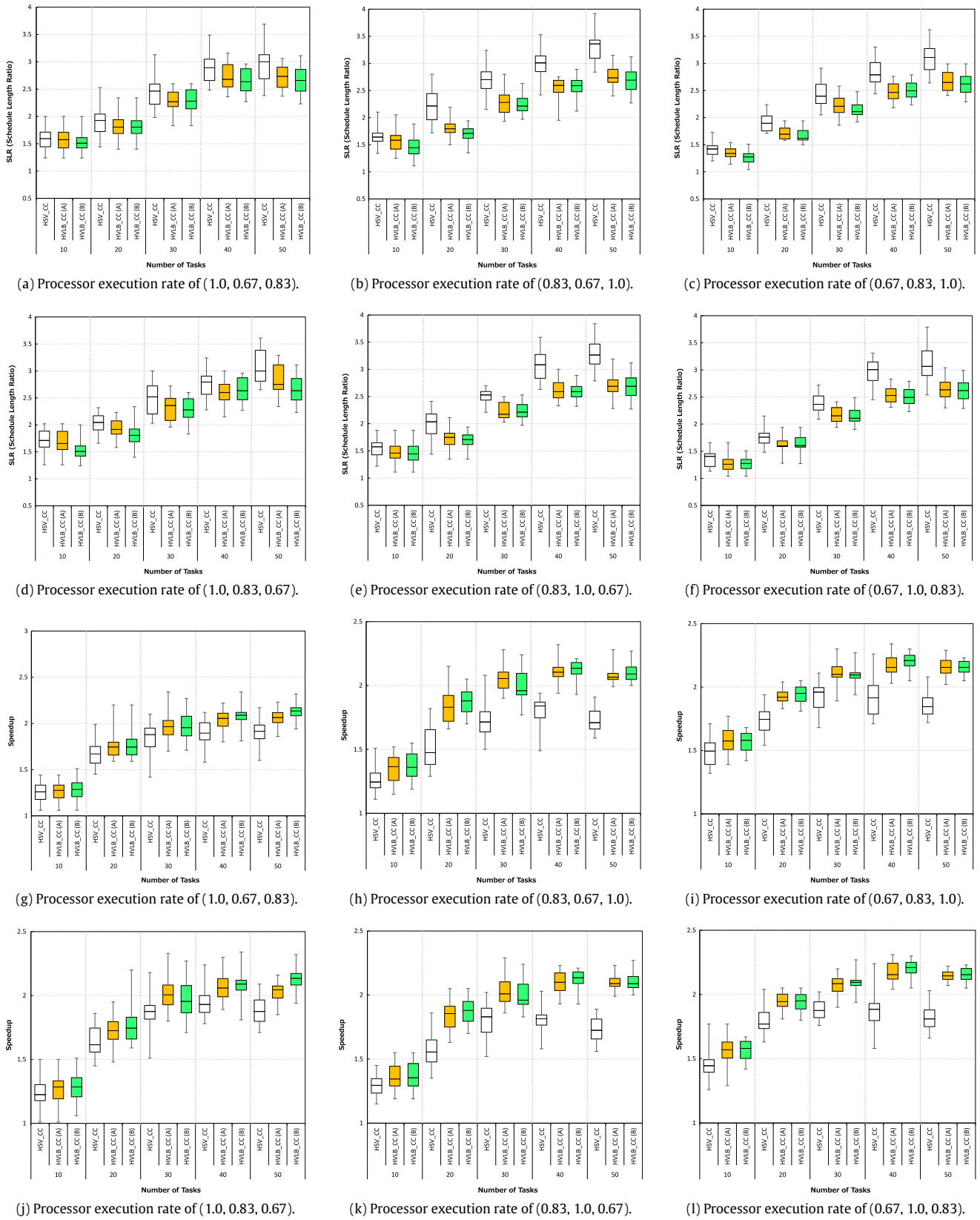
(a) Processor execution rate of (1.0, 0.67, 0.83).

(b) Processor execution rate of (0.83, 0.67, 1.0).

(c) Processor execution rate of (0.67, 0.83, 1.0).

(d) Processor execution rate of (1.0, 0.83, 0.67).

(e) Processor execution rate of (0.83, 1.0, 0.67).

(f) Processor execution rate of (0.67, 1.0, 0.83).

(g) Processor execution rate of (1.0, 0.67, 0.83).

(h) Processor execution rate of (0.83, 0.67, 1.0).

(i) Processor execution rate of (0.67, 0.83, 1.0).

(j) Processor execution rate of (1.0, 0.83, 0.67).

(k) Processor execution rate of (0.83, 1.0, 0.67).

(l) Processor execution rate of (0.67, 1.0, 0.83).

Fig. 7. The *SLR* and *speedup* values of HSV_CC and HVLB_CC at the varying processor execution rate and *CCR* = 1.0.

[1.0, 0.83, 0.67], [0.83, 1.0, 0.67], and [0.67, 1.0, 0.83], respectively. These results show that the proposed scheduling algorithms outperform HSV_CC in terms of worst-case SLR. In HSV_CC, the processor selection criteria are based on *EFT* and *LDET* which are largely influenced by the processor execution rate. Thus, the tasks and messages are intended to be allocated to specific processors
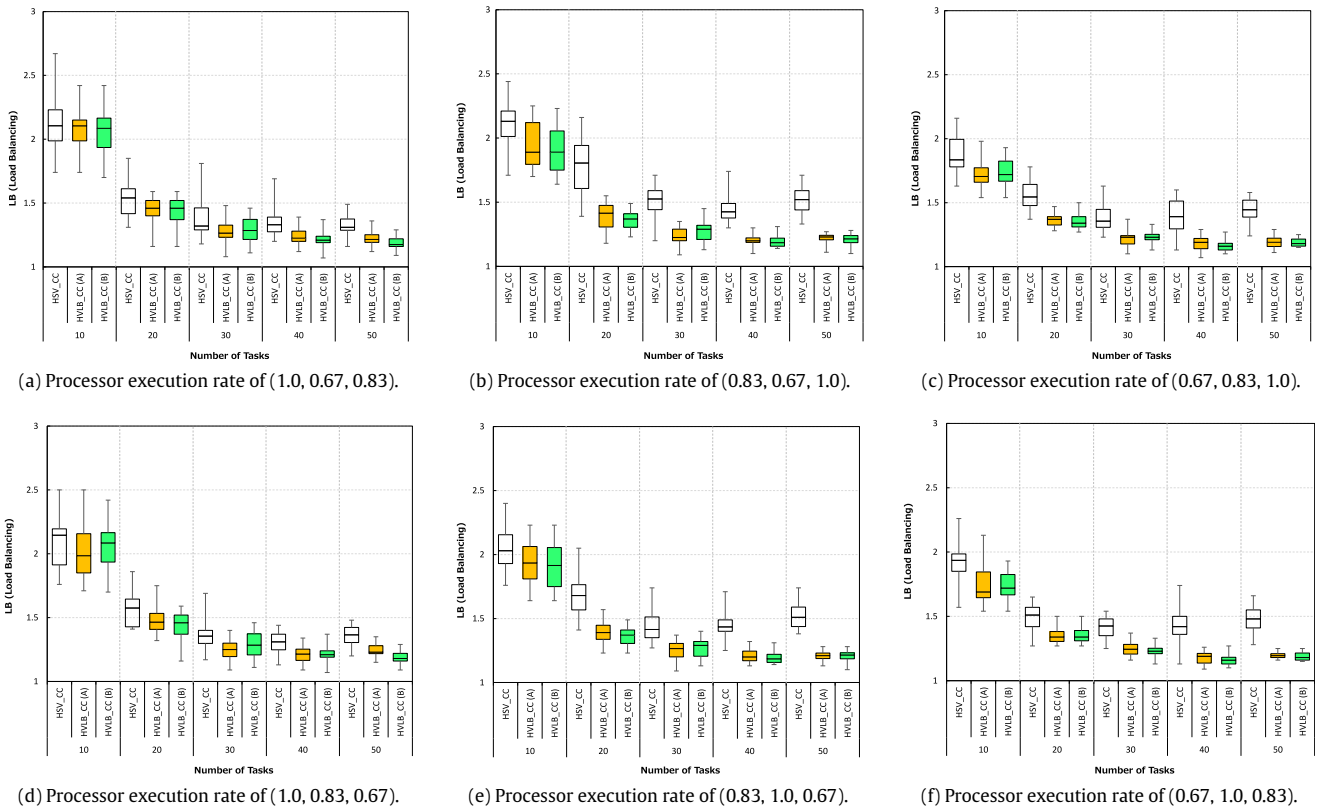
(a) Processor execution rate of (1.0, 0.67, 0.83).

(b) Processor execution rate of (0.83, 0.67, 1.0).

(c) Processor execution rate of (0.67, 0.83, 1.0).

(d) Processor execution rate of (1.0, 0.83, 0.67).

(e) Processor execution rate of (0.83, 1.0, 0.67).

(f) Processor execution rate of (0.67, 1.0, 0.83).

Fig. 8. The load balancing values of HSV_CC vs HVLB_CC with increasing the number of tasks.

and communication links. However, in the proposed HVLB_CC (A) and (B) algorithms, allocating tasks and messages to specific processors and links can be avoided because, with α, considering and balancing the processor loads during the processor selection phase can affect the schedule length positively. Fig. 7(h)–(m) show the results for *speedup* with varying number of tasks that were obtained using HSV_CC and the proposed HVLB_CC (A) and (B) algorithms. The best *speedup* value is 2.26 when the processor execution rates are [0.67, 0.83, 1.0] for HSV_CC. For HVLB_CC (A) and (B), the best *speedup* values are 2.34 and 2.34 when the processor execution rates are [1.0, 0.67, 0.83], and the proposed algorithms outperform HSV_CC for all numbers of tasks and all processor execution rates because, as mentioned above, HVLB_CC (A) and (B) schedule tasks by considering the processor loads and minimizing *makespan* by adjusting α.

5.4. Experiment 2: Comparison of load balance over processors

Here, we analyze the LB results with a varying number of tasks to confirm the load balance between the processors. For each number of tasks, we randomly generated 100 task graphs with variable computation times according to the execution times of the three processors. In addition, *CCR* was set to 1.0. The number of tasks was increased from 10 to 50 in increments of 10, and, in each task graph, α was selected to minimize *makespan*.

As shown in Fig. 8, the proposed HVLB_CC (A) and (B) algorithms show better performance in terms of LB with a varying number of tasks and for all of the processor execution rates, because the proposed algorithms consider both of the processor load in the processor selection phase and the factors considered by HSV_CC (i.e., *EFT* and *LDET*). In HSV_CC, tasks and messages are intended to be allocated to specific processors and communication links because both *EFT* and *LDET* are highly dependent on the processor execution rate. However, in the proposed HVLB_CC (A) and (B)

algorithms, allocating tasks and messages to higher-speed processors and links can be avoided. The proposed algorithms yield balanced processor loads, and can decrease *makespan* by utilizing processors and links that are not sufficiently used in HSV_CC.

5.5. Experiment 3: Comparison of scheduling accuracy with varying communication-to-computation ratio

In this experiment, we observed *SLR* while varying *CCR*. The number of tasks was fixed to 20. The processor execution rates of the three processors were set to 0.83, 1.0, 0.67. We randomly generated 100 task graphs with variable computation times according to the execution time rate of three processors, and *CCR* was varied from 0.1 to 10.

Fig. 9 shows the *SLR* values for HSV_CC, HVLB_CC (A), and HVLB_CC (B) with different *CCR* values.

For *CCR* = 0.1 and 0.5, the communication time is of low significance compared to the computation time (i.e., these problems are computation-intensive). The worst-case *SLR* values of the proposed HVLB_CC algorithms are 16.3% and 18.1% better than that of HVC_CC when *CCR* is 0.1 and 0.5, respectively. Similarly, the best-case *SLR* values of the proposed HVLB_CC algorithms are 8% and 9.6% better than that of HVC_CC when *CCR* is 0.1 and 0.5, respectively. For *CCR* = 1, the *SLR* values of HVLB_CC (A) and HVLB_CC (B) are the same. Compared to the *SLR* values of HSV_CC, the proposed algorithms can improve by 20.7% and 17% in the worst and best-case scenarios, respectively. For *CCR* = 5 and 10, the tasks in the task graphs have longer data transfer times than computation times. For the worst-case *SLR*, HVLB_CC (A) can improve by 3.9% and 1.75% compared to HSV_CC when *CCR* is 5 and 10, respectively. Similarly, HVLB_CC (B) can improve by 1.54% and 1.75% compared to HSV_CC when *CCR* is 5 and 10, respectively. For the best-case *SLR*, HVLB_CC (A) and (B) can improve by 11.1% and 5.5% when *CCR* is 5 and 10, respectively. Note that the gap between
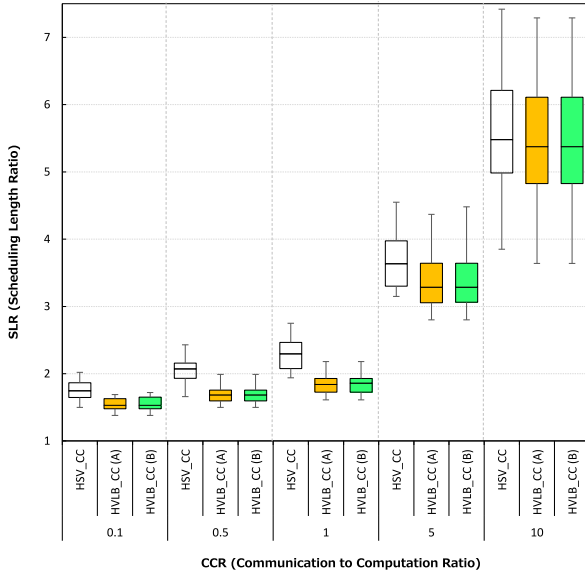
Fig. 9. *SLR* values for HSV_CC and HVLB_CC with increasing CCR.

Fig. 10. *SFR* s of HSV_CC and HVLB_CC.

Table 4
Computation time matrix used in *Experiment 5*.

Task	p_1	p_2	p_3
n_1	26	17	20
n_2	26	17	20
n_3	14	9	11
n_4	12	8	10
n_5	17	11	13
n_6	30	20	24
n_7	9	6	7
n_8	27	18	22
n_9	27	18	22
n_{10}	30	20	24

HSV_CC and HVLB_CC relative to *SLR* decreases when *CCR* becomes greater than 1 because tasks tend to be assigned to high-capability processors first when *CCR* is greater than 1, and successor tasks are not mostly assigned to different processor with a precedence task. A higher *CCR* value increases the communication time more than the computation time, which causes *EFT* to increase for the task.

It can be observed that the improvement becomes significantly greater when *CCR* is decreased from 10 to 1 and when *CCR* is increased from 0.1 to 1. The greatest improvements with the proposed algorithm are possible with a *CCR* value of 1. Note that very little improvement is observed when *CCR* is 10. As a result, the HVLB_CC (A) and (B) algorithms can consider the heterogeneity of both computation and communication for task ordering.

5.6. Experiment 4: Comparison of scheduling failure rate

As mentioned in Section 3, HSV_CC is restricted in scheduling for various DAGs. In this experiment, we confirmed that how much the rate could be improved by considering the depth factor in a DAG with the proposed HVLB_CC (B) algorithm.

We generated 1000 random task graphs. In the stream processing system, an operator, such as *Join* or *Union*, has two in- and out-degrees, and a user-defined operator can have a maximum in-degree of three. Thus, we generated random DAGs whose maximum in-degrees were 2 and out-degrees were 3. To confirm successful scheduling for HSV_CC and HVLB_CC (B), the scheduling failure rate (*SFR*) is defined as follows:

$$SFR = \frac{number\ of\ failed\ DAGs}{total\ number\ of\ requested\ DAGs} \times 100. \qquad (26)$$

Fig. 10 shows the *SFR* s of HSV_CC, HVLB_CC (*depth*), and HVLB_CC (*depth*^{2}). The *SFR* of HSV_CC is 78%, and the *SFT* values of HVLB_CC (*depth*) and HVLB_CC (*depth*^{2}) are 29% and 0%, respectively. As a result, HSV_CC is very restricted when the DAGs are random, and it can only schedule perfectly for DAGs with $outd(n_p) \geq outd(n_s)$, where n_p and n_s are predecessor and successor tasks, respectively. We observed the change in *SFR* while increasing the impact of the depth factor. When we use Eq. (9) with *depth*, there are still DAGs that experience scheduling failure. In contrast, when we increase the depth factor as *depth*^{2} in Eq. (9), all DAGs are scheduled successfully. Thus, we select *depth*^{2} in Eq. (9).
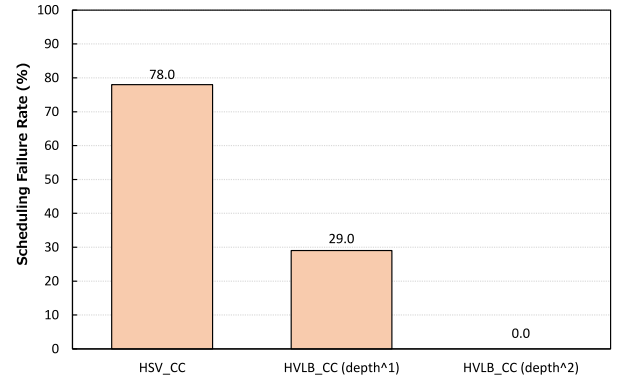
5.7. Experiment 5: Comparison of data precision with increasing input arrival rate

Here, we discuss the data precision improvement that can be realized with the proposed scheduling utilizing the imprecise computation model with respect to data arrival rate λ in the range [1, 2] in increments of 0.1. λ can be calculated by dividing the total computation time for the *mandatory* and *optional* parts by the computation time for the *mandatory* part, where only computation time for the *optional* part can be varied. The data precision is the percentage value obtained by dividing the total processing time by the requested processing time.

We conducted a simulation of the task graph shown in Fig. 3 using the computation time matrix shown in Table 4. Note that *CCR* was fixed at 1. In the schedule results obtained using HVLB_CC, the schedule holes for n_2, n_5, and n_8 were found to be 9, 5, and 12, respectively, by using Eqs. (20) and (21). In Fig. 3, it has been assumed that n_2, whose arrival rate can be increased, is processing the receiving data streams from external communications and changing the data format to be used in the DSMS. n_5 is assumed as matching processing with map data from n_3 and external-vehicles information from n_2. Thus, we changed n_2 and n_5 into imprecise computation models and input data arrival rates. We conducted two sets of experiments: one set for the case in which imprecise computation models are not allowed (i.e., the original version of HVLB_CC) and another set for the imprecise computation case (i.e., for the alternative version, HVLB_CC_IC). The simulation results were analyzed as follows.

Fig. 11 shows a comparison between the original and alternative versions of HVLB_CC. With HVLB_CC_IC, the data precisions of n_2 and n_5 are 100% until the input data arrival rates become 1.5 and 1.4, respectively, and their precisions start to decrease at input data arrival rates of 1.6 and 1.5, respectively. This is because the available schedule holes for n_2 and n_5 are 9 and 5; thus, when the requested computation time increases beyond the available schedule hole, each task starts to terminate the execution
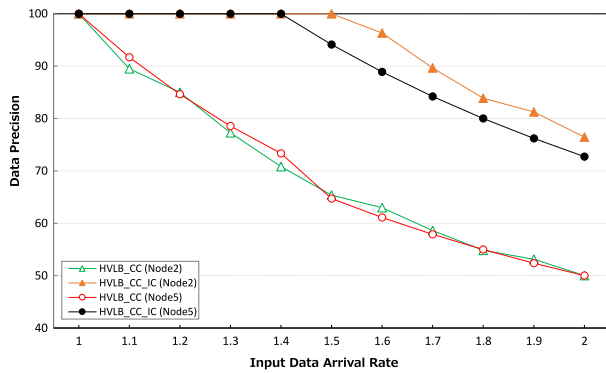
Fig. 11. Effect of imprecise computation model.

of *optional* parts. In contrast, the data precisions of n_2 and n_5 decrease from 1 in HVLB_CC without imprecise computations are not allowed, because HVLB_CC without imprecise computations does not exploit any schedule hole, and each task terminates execution after finishing its *mandatory* part. Compared to HVLB_CC without imprecise computations, HVLB_CC with the imprecise computation model exhibits the greatest improvement in terms of data precision.

6. Related work

In the automotive field, DSMSs have been researched for applications in automotive embedded systems in order to reduce data processing complexity and software development costs. In a previous study [34], a scheduling method that uses the earliest deadline first for an automotive embedded DSMS was implemented on each distributed processor. In [22], distributed stream processing on single- and multi-core processors was investigated to process numerous streams of data from external communications more efficiently. However, these methods assume that an optimal distribution pattern for multiple operators (tasks) is already known; thus, the distributions of operators and messages on multiple processors and networks have not previously been described in detail. StreamCar [5] is a stream processing platform that facilitates the combination of sensor data. However, this method does not consider strict real-time constraints, which are important in automotive systems, and does not include a scheduling method.

Most DAG scheduling algorithms are based on list scheduling which maintains a list of tasks in a DAG according to their priorities. This process has two phases: (i) a task prioritizing phase to assign a priority to each task and to queue the tasks according to their priorities and (ii) a processor selection phase to select a task based on its order of priority and to assign the selected task to a suitable processor.

The heterogeneous earliest-finish-time (HEFT) algorithm [31] calculates the *upward rank* value for each task, and the task with the highest *upward rank* value is selected first by traversing the graph upwards. The critical-path-on-a-processor (CPOP) algorithm [31] uses both *upward* and *downward rank* values, and these values are used to prioritize each task. This algorithm is similar to the HEFT algorithm, but all of the tasks on critical paths are assigned to a single processor that can minimize their execution times. These algorithms select processors for the tasks according to their *EFT* values, in which the computation and communication times are calculated as average values. They primarily concentrate on task scheduling to minimize the schedule length without considering message scheduling. Furthermore, they assume simple system models in which all of the processors are fully connected and have concurrent inter-processor communication, even though

communication contention can influence the execution time in parallel processing [27]. Therefore, it is difficult to obtain accurate schedule results with these algorithms, especially in automotive embedded systems in which time accuracy is strictly required.

Clustering-based scheduling algorithms [15,16,18,29,35] primarily target networked on-chip multiprocessor architectures that are generally intended for unbounded numbers of processors. Tasks that are clustered according to some criteria in the DAG are assigned to multiple processors. However, these clustering-based scheduling algorithms commonly use dynamic task allocation methods. When a task graph arrives at the system, it must first enter a *global waiting queue* and will subsequently be moved to a *local waiting queue* on a processor. This dynamic method is difficult to apply to automotive embedded systems due to the potential for unpredictable communication times. Thus, time predictability of stream processing cannot be guaranteed. Moreover, these algorithms assume simple system models in which communication contention is not considered because particular communication links between two processors are assumed to be prepared, which indicates that communication contention can be ignored. To obtain accurate and efficient scheduling, both task and message scheduling must be considered simultaneously. Therefore, the main weaknesses of clustering-based algorithms are high-complexity and unpredictable timing constraints because they primarily target unlimited numbers of processors.

Some studies consider the communication contention problem in list scheduling algorithms. Sinnen et al. proposed a communication contention-awareness scheduling algorithm based on list scheduling [27,28], and they showed that this algorithm could produce significantly more accurate scheduling than could a contention-unaware algorithm in a heterogeneous computing environment in which all processors had the different processing abilities. Their algorithms addressed contention by providing appropriate communication routing between processors. Tang et al. proposed a list scheduling method with contention on communication links in heterogeneous computing environments with an arbitrary processor network (APN) [30]. They concentrated on a network topology that consisted of processors connected by the APN, in which constant $p(p-1)/2$ (p represents the number of processors) communication routes between processors existed. For in-vehicle networks, several types of network protocols, such as CAN and MOST, are inter-connected via gateways. Thus, there are multiple routes between processors, and different numbers of communication routes exist between them. Moreover, strict timing constraints cannot be guaranteed in existing scheduling algorithms with communication contention because they do not consider the synchronization of tasks and messages and they assume that communication delays are possible. Thus, it is difficult to measure accurate end-to-end worst-case execution times (WCETs), which are the most important factors in automotive fields.

Xie et al. [11] proposed HSV_CC based on list scheduling in consideration of communication contention problems in heterogeneous automotive embedded systems. They assumed that the number of communication routes between different processors could differ and selected the most appropriate communication route. Moreover, they could provide task and message synchronization, which is important for measuring accurate end-to-end WCET. However, it is difficult to apply HSV_CC to an automotive DSMS for the following reasons. First, it can cause inefficient resource allocation in which tasks and messages are likely to be assigned to specific processors and network links. In the worst case, some processors and links are poorly utilized even though other processors and links have sufficient idle time slots. Second, HSV_CC scheduling was designed to schedule DAGs. Thus, it is possible that some stream processing applications for automotive DSMSs may not be scheduled appropriately. Third, HSV_CC, which assumes

J. Rho et al. / J. Parallel Distrib. Comput. 109 (2017) 286–300

299

Table 5
HVLB_CC vs previous work.

	Heterogeneity of processors	Heterogeneity of networks	Communication contention	Accurate time predictability	Load balance over resources	Schedulability of stream processing	Varying data input arrival rate
HEFT, CPOP [31]	✗						
Automotive DSMS [22,34]						✗	✗
Clustering-based [15,16,18,29,35]	✗				✗		
Comm. contention-based [27,28]			✗		✗	✗	
APN-based [30]	✗		✗	✗			
HSV_CC [11]	✗	✗	✗	✗			
HVLB_CC	✗	✗	✗	✗	✗	✗	✗

that the processing time of each task is constant, is inappropriate for automotive DSMSs. In stream processing, the requested processing time for a task can vary. For example, the amount of input data for map matching processing using stored map data and the external vehicle information acquired by external communications can vary over time. This type of processing is difficult to be scheduled efficiently with static HSV_CC. Static list scheduling algorithms such as HSV_CC or HEFT can be applied to dynamic scheduling to handle tasks whose requested processing time can vary. However, a dynamic scheduling method is not desired in automotive DSMSs due to strict time predictability and resource restrictions.

Table 5 shows a comparison between HVLB_CC and previous methods. HVLB_CC can be applied to parallel and distributed environments consisting of heterogeneous processors connected to in-vehicle networks that contain different technologies with different bandwidths. To predict execution times for stream processing accurately, the proposed algorithm schedules tasks and messages simultaneously in consideration of the communication contention constraint. For application in automotive DSMSs, HVLB_CC considers LB, and can deal with SPGs. Furthermore, the alternative version of the proposed algorithm can handle tasks with varying input data arrival rates by applying an imprecise computation model.

7. Conclusion

This paper has presented heuristic scheduling algorithms based on static-list scheduling for automotive DSMSs. We have addressed three issues in applying the existing static list scheduling algorithm called HSV_CC, which is proposed for heterogeneous embedded systems, to stream processing distribution: (i) previous task and message schedules can lead to less efficient resource usages in a stream processing scenario, (ii) the conventional method to determine the task scheduling order may not be best suited to deal with stream processing graphs, and (iii) tasks with time-varying computational requirements are needed to be scheduled efficiently.

To address (i), we proposed the HVLB_CC (A) algorithm, which considers LB in addition to the parameters considered by the HSV_CC algorithm. We proposed HVLB_CC (B) to address issue (ii). HVLB_CC (B) can deal with stream processing task graphs and more various DAGs to prevent assigning higher priorities to successor tasks. To schedule tasks more efficiently with various computation times (iii), HVLB_CC_IC utilizes schedule holes left in processors. These idle time slots can be used to execute for *optional* parts and thereby generate more precise result by applying imprecise computation model. Our experimental results demonstrate that the proposed algorithms achieve minimum schedule lengths, accuracies, and LB values significantly better than those obtainable by using the existing HSV_CC algorithm. In addition, the proposed HVLB_CC (B) algorithm can schedule more varied task graphs (including SPG) without reducing performance. Furthermore, using

imprecise computation models, HVLB_CC_IC obtains higher precision data than HVLB_CC can without imprecise computation models. Therefore it is possible to improve the accuracy of applications that use V2V communication.

In the future, we plan to extend the proposed algorithm so that it can be applied to task graphs with multiple periods and deadlines. In an automotive DSMS, sensor data from different sources (e.g., radar, camera, and external communication) generally arrive to the system at different periods and exit tasks that produce the final processed sensor data for applications should be able to have different deadlines. Furthermore, DSMS mechanisms such as data buffering, window aggregation, and operator state should be considered. To experimentally validate the proposed scheduling algorithm, we also plan to conduct experiments using data from real-world data sets and analyze the experimental results.
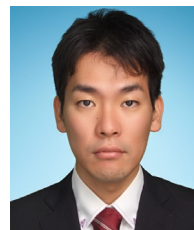
Acknowledgment

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research, CIDR, 2005, pp. 277–289.

[2] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: A new model and architecture for data stream management, VLDB J. (ISSN: 1066-8888) 12 (2) (2003) 120–139.

[3] G. Apostolos, Y. Tao, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, J. Parallel Distrib. Comput. (ISSN: 0743-7315) 16 (4) (1992) 276–291.

[4] D. Arvind, A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, J. Widom, STREAM: The stanford stream data manager (demonstration description), in: Proceedings of the ACM SIGMOD International Conference on Management of Data, vol. 26, 2003, p. 665.

[5] A. Bolles, H.-J. Appelrath, D. Geesen, M. Grawunder, M. Hannibal, J. Jacobi, F. Koster, D., S. Nicklas, StreamCars: A new flexible architecture for driver assistance systems, in: Proceedings of IEEE Intelligent Vehicles Symposium (IV), ISBN: 978-1-4673-2119-8, 2012, pp. 252–257.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah, TelegraphCQ: Continuous dataflow processing for an uncertain World, in: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research, CIDR, 2003.

[7] S. Das, S. Antony, D. Agrawal, A. Abbadi, Thread cooperation in multicore architectures for frequency counting over multiple data streams, Proc. VLDB Endow. (ISSN: 2150-8097) 2 (1) (2009) 217–228.

[8] R.P. Dick, D.L. Rhodes, W. Wolf, TGFF: task graphs for free, in: Proceedings of the 6th International Workshop on Hardware/Software Codesign, 1998, pp. 97–101.

[9] Different Types of Microcontrollers are Used in Automobile Applications. Available from https://www.elprocus.com/different-microcontrollers-used-in-automobiles.

[10] G. F. Dror, R. Larry, C.S. Kenneth, W. Parkson, Theory and practice in parallel job scheduling, in: Book of Job Scheduling Strategies for Parallel Processing, Vol. 1291, 1997, pp. 1–34.

[11] X. Guoqi, L. Renfa, L. Keqin, Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems, J. Parallel Distrib. Comput. (ISSN: 0743-7315) 83 (C) (2015) 1–12.

[12] T. Hagras, J. Janecek, A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems, Parallel Comput. (ISSN: 0167-8191) 31 (7) (2005) 653–670. Heterogeneous Computing.

[13] Intelligent Transport Systems (ITS); V2X Applications; Part 3: Longitudinal Collision Risk Warning (LCRW) Application Requirements Specification, ETSI TS, Vol. 101, 2013, pp. 539-543.

[14] S. Katsunuma, S. Honda, K. Sato, Y. Watanabe, Y. Nakamoto, H. Takada, Real-time-aware embedded DSMS applicable to advanced driver assistance systems, in: Proceedings of the IEEE 33rd International Symposium on Reliable Distributed Systems Workshops, SRDSW, 2014, pp. 106–111.

[15] S. Kim, J.C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proceedings of the International Conference on Parallel Processing, vol. 3, 1988, pp. 1–8.

[16] S.C. Kim, S. Lee, J. Hahm, Push-pull: Deterministic search-based DAG scheduling for heterogeneous cluster system, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 18 (11) (2007) 1489–1502.

[17] A. Lara, From complex mechanical system to complex electronic system: the case of automobiles, in: Automotive Technology and Management, vol. 14, 2014, pp. 65–81.

[18] J.-C. Liou, M.A. Palis, An efficient task clustering heuristic for scheduling DAGs on multiprocessors, in: Proceedings of Workshop on Resource Management, Symposium of Parallel and Distributed Processing, 1996, pp. 152–156.

[19] M. Muthucumaru, D.B. Tracy, J. Howard, Heterogeneous distributed computing, in: Proceedings of Encyclopedia of Electrical and Electronics Engineering, vol. 8, 1999, pp. 679–690.

[20] F.A. Omara, M.M. Arafa, Genetic algorithms for task scheduling problem, J. Parallel Distrib. Comput. (ISSN: 0743-7315) 70 (1) (2010) 13–22.

[21] B. Rashmi, D. Agrawal, Improving scheduling of tasks in a heterogeneous environment, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 15 (2) (2004) 107–118.

[22] J. Rho, T. Azumi, H. Oyama, K. Sato, N. Nishio, Distributed processing for automotive data stream management system on mixed single- and multi-core processors, ACM SIGBED Rev. (ISSN: 1551-3688) 13 (3) (2016) 15–22.

[23] A. Safaei, A. Sharifrazavian, M. Sharifi, M. Haghjoo, Dynamic routing of data stream tuples among parallel query plan running on multi-core processors, Distrib. Parallel Databases (ISSN: 0926-8782) 30 (2) (2012) 145–176.

[24] B. Sanjeev, C.S. Prashanth, Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length, in: Proceedings of the IEEE International Conference on Parallel Processing Workshops, 2003, pp. 97–103.

[25] B. Savina, K. Padam, S. Kuldip, Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs, J. Parallel Distrib. Comput. 65 (4) (2005) 479–491.

[26] S. Seo, J. Kim, S. Hwang, K. Kwon, J. Jeon, A reliable gateway for in-vehicle networks based on LIN, CAN, and FlexRay, ACM Trans. Embed. Comput. Syst. (ISSN: 1539-9087) 11 (1) (2012).

[27] O. Sinnen, L.A. Sousa, Communication contention in task scheduling, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 16 (6) (2005) 503–515.

[28] O. Sinnen, L.A. Sousa, F. Sandnes, Toward a realistic task scheduling model, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 17 (3) (2006) 263–275.

[29] G.L. Stavrinides, H.D. Karatza, Scheduling real-time DAGs in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes, Future Gener. Comput. Syst. 28 (7) (2012) 977–988.

[30] X. Tang, K. Li, D. Padua, Communication contention in APN list scheduling algorithm, in: Proceedings of Science in China Series F, Information Sciences, vol. 52, 2009, pp. 59–69.

[31] H. Topcuouglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 13 (3) (2002) 260–274.

[32] T. Xiaoyong, L. Kenli, L. Renfa, V. Bharadwaj, Reliability-aware scheduling strategy for heterogeneous distributed computing systems, J. Parallel Distrib. Comput. (ISSN: 0743-7315) 70 (9) (2010) 941–952.

[33] M. Yamada, K. Sato, H. Takada, Implementation and evaluation of data management methods for vehicle control systems, in: Proceedings of the IEEE Vehicular Technology Conference, VTC Fall, 2011, pp. 1–5.

[34] A. Yamaguchi, Y. Nakamoto, K. Sato, I. Yoshiharu, Y. Watanabe, S. Honda, H. Takada, AEDSMS: Automotive embedded data stream management system, in: Proceedings of the 31st International Conference on Data Engineering, ICDE, 2015, pp. 1292–1303.

[35] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. (ISSN: 1045-9219) 5 (9) (1994) 951–967.

[36] Y. Zhou, B. Ooi, K.-L. Tan, J. Wu, Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System, in: Proceedings of the 14th International Conference on Cooperative Information Systems, vol. 4275, ISBN: 3-540-48287-3, 2006, pp. 54–71.

Jaeyong Rho received his B.E. and M.S. degrees from Ritsumeikan University, Japan in Information Science and Engineering, in 2014 and 2016 respectively. His research interests include real-time scheduling theory, parallel and distributed computing systems, and deep learning.

Takuya Azumi is an Assistant Professor at the Graduate School of Engineering Science, Osaka University. He received his Ph.D. degree form the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists for Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor at the College of Information Science and Engineering, Ritsumeikan University. From 2011 to 2012, he was a Visiting Scholar in University of California, Irvine. His research interests include real-time systems and component based development. He is a member of IEEE, ACM, IPSJ, IEICE, and JSSST.

Mayo Nakagawa was born in 1994. He is enrolled in the Information Science and Engineering, Ritsumeikan University, 2013. His research interest is real-time embedded systems and scheduling algorithms.

Kenya Sato received the B.E. and M.E. degrees from Osaka University, Japan, in 1984 and 1986 respectively. He received the Ph.D. degree from Nara Institute of Science and Technology, Japan, in 2000. In 1986–1991, he was a research engineer at Information and Electronics Research Laboratory, Sumitomo Electric Industries, Japan, and in 1991–1993, he was a visiting researcher at Computer Science Department, Stanford University, California, USA. Dr. Sato served as a chief technologist at Automotive Multimedia Interface Collaboration, Inc., Michigan, USA, in 2001–2003. Since 2004, he is an associate professor in Department of Information Systems Design, Doshisha University, Kyoto, Japan. His research interests include network architecture, distributed systems, embedded systems, and ITS. He is a member of the IEEE computer society and the ACM.

Nobuhiko Nishio was born in 1962. He graduated in Mathematical Engineering and Information Physics, from School of Engineering, The University of Tokyo in 1986. He is an M.S. from Graduate School of The University of Tokyo in 1988. After quitting Ph.D. course, he worked at Keio University Shonan Fujisawa Campus from 1993 to 2003. He obtained his Ph.D. in Media and Governance in 2000. He was an Associate Professor at College of Science and Engineering, Ritsumeikan University in 2003. Since 2005, he is Professor at College of Information Science and Engineering, Ritsumeikan University (Current profession). He was Visiting Scientist at Google Inc. from 2007 to 2008. He majored in autonomous, distributed and collaborative systems, ubiquitous computing and networking and sensing systems. He has been awarded the Yamashita Memorial Research Prize from IPSJ. He is Regular member of ACM and IEEE Computer Society.