2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)

ROSCH:Real-Time Scheduling Framework for ROS

Yukihiro Saito*, Futoshi Sato*, Takuya Azumi‡, Shinpei Kato§, and Nobuhiko Nishio*

* Graduate School of Information Science and Engineering, Ritsumeikan University
‡ Graduate School of Science and Engineering, Saitama University
§ Graduate School of Information Science and Technology, The University of Tokyo

***Abstract*—This paper presents a real-time scheduling framework for the robot operating system (ROS) called ROSCH. ROS is an open-source software platform and a meta-operating system designed for robots. ROS provides extensive development libraries and has been widely used for autonomous driving systems. However, ROS does not guarantee real-time performance; hence, a ROS-based autonomous driving car could cause a traffic accident. Therefore, ROSCH comprises three functionalities that do not exist in the ROS to guarantee real-time performance: (1) a synchronization system, (2) fixed-priority based directed acyclic graph (DAG) scheduling framework, and (3) fail-safe function. In particular, the synchronization system guarantees that the timestamp gaps between sensor measurements will be less than or equal to the calculated value. The fixed-priority based DAG scheduling framework guarantees that end-to-end latency is less than or equal to an estimated value. Operating both mechanisms simultaneously guarantees the final output topic frequency. The fail-safe functionality provides a danger avoidance action at a deadline miss. In addition, these functionalities are designed to be compatible with legacy software. No previous method that guarantees real-time operation provides a comparable range of features while maintaining the compatibility with the existing software. Experimental results showed that the time differences between sensor timestamps remained below a calculated value. Moreover, correctly scheduled tasks incur an end-to-end latency that is within an estimated value. In addition, the throughput of the end node is achieved. Finally, the fail-safe functionality overhead is 45 μsec on an average, which is 0.004 % of the average execution time for one node.**

***Index Terms*—Robot operating system (ROS); Middleware Systems; Synchronization; DAG; Real-time scheduling; Autonomous driving system;**

I. Introduction

In recent years, autonomous driving vehicles, such as the Waymo [1], have attracted attention such as the DARPA Urban Challenge [2], [3]. In many cases, autonomous driving systems depend on the robot operating system (ROS) [4], [5]. ROS is a meta-operating system designed for robots and an open-source software platform [6]. ROS provides extensive robotics libraries [7]–[11]. Therefore, ROS has become the de facto standard platform for robot systems.

A ROS application is composed of a number of independent programs called nodes, each of which communicates with other nodes using a topic-based publish/subscribe design pattern. To communicate between nodes, a publisher node sends a message by publishing the corresponding data on a given topic. A subscriber node that is interested in the data will subscribe to appropriate topics and operate. However, ROS does not guarantee or support real-time constraints because it is originally developed as a tool to increase development efficiency for the PR2 robot [12], [13].The ROS community developed ROS 2.0 [14], [15], which improved real-time performance. However, the source code for ROS and ROS 2.0 are incompatible and a library developed for ROS cannot be used in ROS 2.0.

Many applications for an autonomous driving system have real-time constraints, i.e., tasks must be completed within a certain period. For example, obstacle distance estimation has real-time constraints in autonomous driving systems. In obstacle distance estimation, if an obstacle is not detected before a deadline, obstacle recognition is delayed and a traffic accident could occur. Therefore, the real-time performance ROS must be improved for it to be suitable for autonomous driving systems. To improve ROS's real-time performance, various studies exist and several solutions have been proposed. Hongxing et al. [16] improved the real-time performance of ROS by running ROS nodes on a real-time operating system (RTOS). Jonas [17] evaluated real-time performance using real-time ROS tests; however, these evaluations were insufficient because the tests primarily involved the ROS timer and did not include internal process tests. These prior studies did not focus on internal ROS processes, and the provided solutions operate over ROS.

Although these studies have made it possible to guarantee the real-time performance of one node, real-time performance for systems comprising a group of nodes has not yet been addressed. We clarified the requirements for real-time performance [18] on systems comprising a group of nodes by developing Autoware [19] and experimentally demonstrating its performance. Autoware is one of the most popular ROS-based open-source projects for autonomous driving systems because it includes most functionalities necessary for autonomous driving [20], [21] such as self-location estimation, environment recognition, and route planning.

This paper proposes a real-time scheduling framework for ROS called ROSCH that includes a synchronization system and a fixed-priority based directed acyclic graph (DAG) scheduling framework. The proposed framework is demonstrated to be effective for real-time ROS performance.

Contributions: The main technical contributions of this paper are the following:

1) We propose a synchronization system and fixed-priority based DAG scheduling framework that guarantees that

2325-1301/18/$31.00 ©2018 IEEE
DOI 10.1109/RTCSA.2018.00015

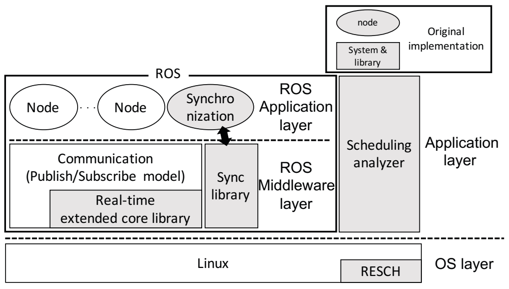

Fig. 1. System model of the proposed ROSCH framework.

the timestamp gap of different sensors and the end-to-end latency are less than a certain value.

2) We evaluate the two proposed approaches using obstacle distance estimation system applications on Autoware, demonstrating that they can guarantee the throughput of the final output topic.
3) We design and implement a fail-safe function for taking a danger avoidance action at a deadline miss.

The proposed mechanism does not exist in the ROS and can be used with legacy ROS applications without modifications.

Organization: The rest of this paper is organized as follows. We introduce our system model of the framework in Section II. Section III presents the design and implementation of the proposed framework. The advantages of using the framework are demonstrated in Section IV. Section V discusses the related work, and we provide our concluding remarks in Section VI.

II. System Model

Fig. 1 represents the system model of the proposed ROSCH framework, i.e., a synchronization system comprising the sync drivers node and sync node, a DAG scheduling framework comprising the real-time extended core library, a scheduling analyzer, and [22]. RESCH is a scheduling module implemented as a loadable kernel module. Operating the synchronization system ensures that the time differences generated between sensor timestamps are below a constant. A guarantee on the end-to-end latency was realized by the DAG scheduling framework. The throughput of the end node is ensured by operating both the synchronization system and DAG scheduling framework. The fail-safe functionality was implemented as an application programming interface (API) in the real-time extended core library. These mechanisms can be used with legacy ROS applications without modifications. The coding standards in [https://github.com/CPFL/ROSCH] must be observed when ROSCH is used.

Requirements: The requirements for proposing the real-time framework for autonomous driving systems on ROS are defined as follows.

R1: The first requirement is a guarantee that timestamp gaps between sensor data are less than a certain value.

R2: The second requirement is to guarantee the maximum value of end-to-end latency and frequency of each final output topic.

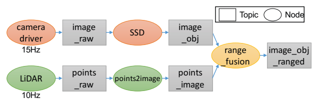

Fig. 2. Obstacle distance estimation system.

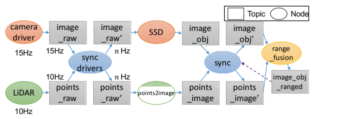

Fig. 3. Incorporation of the synchronization system into Fig. 2.

R3: The final requirement is a fail-safe procedure at a deadline miss.

R1 reason is that processing results with plural sensor data are sometimes erroneous when timestamp gaps between sensor data are large. In ROS, since each node operates independently, it is not guaranteed that the timestamp gaps falls below a certain level. R2 is for accident prediction and vehicle control. To know the possibility of an accident by end-to-end latency, it is necessary to estimate that in advance. When autonomous driving systems control the automobile, it is necessary to send a control signal at a constant cycle. R3 is due to the publish/subscribe model such as ROS. This model can be paraphrased as an event-driven dataflow system in which a node is generally launched when the predecessor nodes are completed. If a deadline miss occurs at a predecessor node, the processing of all succeeding tasks will get affected; in some cases, the system breaks down. A function to satisfy these requirements does not exist ROS.

III. Design and Implementation

This section presents the details of the synchronization system, DAG scheduling framework, and fail-safe functionality.

A. Synchronization system

As shown in Fig. 2 in the obstacle distance estimation system in Autoware [19], there is a case wherein a node combines multiple topics in a system that uses ROS. A gap in the timestamps (time difference) of sensors, such as the camera and LiDAR, occurs when the range_fusion node receives topic data from multiple nodes because the periods for SSD and points2image differ. There should be no significant difference between the timestamps of sensors for nodes that subscribe to and combine some topics, such as the range_fusion node. If there is a large time difference, range_fusion cannot estimate the distance and recognize cars correctly; range_fusion may incorrectly recognize a nearby car as a distant car. Therefore, a large time difference will cause traffic accidents. The sync drivers node and the sync node added to Fig. 2 controls the sensor topics published by the sensor driver nodes such as the camera driver and LiDAR nodes on the obstacle distance estimation system (Fig. 3).

The sync drivers node adjusts the publishing period of the sensor topics and publishes to the two sensor topics simultaneously. Note that n, which is the period of the final output topic publication, is a divisor of the longest sensor topic cycle periods. For example, when n is a divisor of the LiDAR driver period, the sync drivers node publishes based on the publishing timing of the LiDAR driver, which is a divisor of 10. As a result, the camera driver timestamp has a gap of

66.67 ms that is one cycle of the camera driver time with respect to the actual time. The sync node is placed before the range_fusion node, which subscribes to two or more topics. The sync node buffers the image_obj and points_image topics in a ring buffer. The pair of latest values generated by the sync drivers node are published from the buffered data to the range_fusion node. When the range_fusion node publishes to the image_obj_ranged topic, the image_obj_ranged topic publishes a publish request to the sync node. The cycle period of the final output topics is updated at this time. The timestamp gap is the gap from the actual time, and the processing/output rate is n Hz. Developers can implement the sync node in about five lines of code using the template library. In addition, because the ROS cLBSan change the name of a topic that is published or subscribed to at runtime, the synchronization system can be introduced without changing any other existing nodes.

Note that the sync node may not be needed in some cases when the fixed-priority based DAG scheduling framework is used. The estimation of the end-to-end latency of this scheduling framework can be less than the publishing interval. As a result, buffering does not occur at the sync node and the sync node may become an extra overhead. In this time, sync node does not buffer data and may become an extra overhead. However, even in that case, sync node is effective for keeping the timestamp gap within a certain value when a node suffers from a deadline miss.

B. *Fixed-priority based DAG scheduling framework*

The publish/subscribe model used in ROS can be described by nodes and directed edges. Therefore, a system on ROS is described by a DAG, and DAG scheduling can be used. For example, heterogeneous laxity-based scheduling (HLBS) is an offline DAG scheduling algorithm that can meet multiple deadline constraints and consider the execution order restriction.

An online estimation of a node's WCET for scheduling is difficult because of the increased complexity of CPU architecture in recent years. In addition, we do not know the number of cores used because many applications using ROS use rich libraries and multiple threads until these operate. Therefore, the node actual execution time must be used and offline scheduling must be considered.

For these reasons, we propose a fixed-priority based DAG scheduling framework. In this framework, scheduling comprises the following three stages.

1) Execution time measurement
2) Automatic analysis of the core assignment
3) Fixed-priority scheduling

First, each nodes execution time is measured in an environment in which any number of cores are used. The value of the node WCET corresponding to the number of used cores is estimated. Next, which cores are allocated to each node are analyzed using the estimated WCET. The DAG scheduling algorithm for analysis uses exHLBS, which is an extended version of HLBS [23] that can handle multithread nodes, and is implemented in the scheduling analyzer. Finally, fixed-priority scheduling is performed using the analysis result. When a deadline miss occurs, the fail-safe functionality is provided by the real-time extended core library.

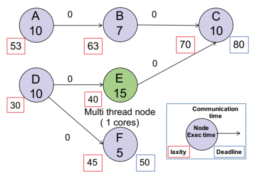

Fig. 4. DAG including multithreaded nodes (one used cores).

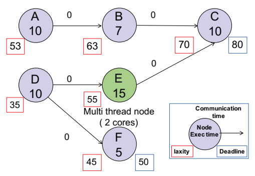

Fig. 5. DAG including multithreaded nodes (two used cores)

1) Execution time measurement: The execution time measurement is automatically performed by describing the node information to be measured in the config file. This function uses the real-time extended core library and RESCH [22].The scheduling framework sets the highest priority processor affinity by using the system call (sched_setaffinity) and measures the execution time of the callback function of each node in each number of cores less than or equal to the setting maximum value. In addition, the estimated WCET of each node is the maximum execution time in the measurement result.

2) Automatic analysis of core assignment: The scheduling analyzer analyzes the allocation for each core using the estimated WCET. Note that in this paper, it is assumed that the topic communication time on the local host is extremely small compared to the execution time of nodes because the coding conventions stipulate that zero copy between publishing and subscribing called nodelet are used for large topic data. In addition, the assumption environment in this paper is a single autonomous driving system; therefore,s the cost of a distributed communication is excluded. Thus, the description will advance by topic communication time sake as 0 ms. Note that the algorithm itself supports the case wherein communication cost exists.

HLBS assumes a single thread and uses the time left until the deadline, called the laxity of each node. This paper proposes exHLBS based on HLBS. When there is a multithreaded node, exHLBS calculates the laxity according to the number of used cores. For example, if the multithreaded node E uses one core (Fig. 4), the execution time is assumed to be 30 ms. When node E use two cores and the execution time is assumed to be shorter than node E that use one core, the laxity of nodes will change as shown in Fig. 5. In this way, a DAG that includes laxity is generated by considering the number of multithreaded nodes and the number of used cores.

Next, we allocate cores to the generated DAG. The target node is the node whose laxity is the lowest is allocated cores that can complete execution the node in the shortest time. Then, if there is time between the arranged tasks, that time is also included as a task placement candidate. In the DAG of Fig. 4, we will allocate tasks in the order of D, E, F, A, B, and C. The result of this allocation is shown in Fig. 6. In the DAG of Fig. 5, we will allocate in the order of D, F, A, E, B, and

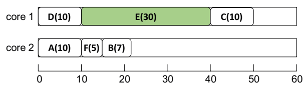

Fig. 6. Results of core allocation using parallel-exHLBS in Fig. 4

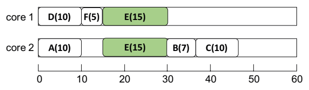

Fig. 7. Results of core allocation using parallel-exHLBS in Fig. 5

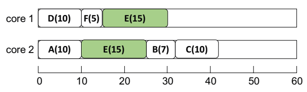

Fig. 8. Core allocation result with bestfit-exHLBS for the system in Fig. 5.

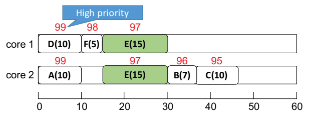

Fig. 9. Priority added result for the system in Fig. 7.

C. Multithreaded node E is allocated cores according to the set number of cores and the processing begins simultaneously by a plurality of cores (Fig. 7). As a result, allocating two cores to node E shortens the makespan to 3 ms.

exHLBS supports parallel-exHLBS that is restricted to allocations at the same starting time and bestfit-exHLBS that is removes this restriction. For example, the results when bestfit-exHLBS is applied to the example in Fig. 5, as shown in Fig. 8. The theoretical value of the makespan is likely to be shorter for bestfit-HLBS than parallel-exHLBS. However, when the elapsed time reaches 25 ms, the thread group running on core 2 of node E in Fig. 8 is forcibly live-migrated to core 1. Live migration has a large overhead and greatly reduces the hit rate of the in-core cache. As a result, because the execution time measurement environment is different, the execution time may increase. For this reason, parallel-exHLBS is adopted as the default method for core allocation and bestfit-exHLBS is available as an option.

3) Fixed-priority scheduling: Scheduling is performed using the core allocation result of each node. Developers can easily implement arbitrary scheduling algorithms for ROS using RESCH. This time, static priority scheduling is used for the scheduling. It is easy to implement, highly reliable and has a low scheduling overhead. High priority is given to the nodes that are allocated earlier in each core, and the priority decreases as the allocation time increases. Hence, we can observe the allocation order of Fig. 7. Incidentally, at this point, it is assumed that the number of threads of the multithreaded node is unknown and the thread ID cannot be obtained. The multithreaded nodes are unified with the lowest priority assigned to each core. The priority allocation results are shown in Fig. 9.

Note that Linux system calls (sched_setscheduler) must be used when using the ROS fail-safe functionality described later because it is difficult to manage the thread ID in RESCH and the thread ID could not be managed in time for the implementation.

C. ROS fail-safe function

The fail-safe functionality is implemented into the ROS middleware layer as an API using the real-time extended core library. In ROS, the poll manager thread and main thread perform the majority of the work. The poll manager thread is a thread of the ROS middleware layer and transmits and receives topic data. The main thread is the thread in which the code of the user application runs. When the publish method of the ROS API is called, we request the poll manager thread to send the topic data and the main thread performs the following operation. If the fail-safe functionality is enabled, the code of the user application will run on the newly created worker thread. The main thread continues to poll until the deadline and monitors the termination of the worker thread. If a deadline miss occurs, the main thread lowers the worker thread to background processing (processing in CFS) and executes the user-defined hook function. Basically, within the hook function, only topics that were not published until the deadline are distributed, and the topic published in the worker thread is discarded. However, whether the topic published in the worker thread after the deadline miss should be discarded or not depends on the application. Therefore, developers are able to choose whether to do this or not.

The application developer implements the fail-safe functionality according to the application's specification by preparing a hook function that is executed at the time of a deadline miss. This is because the fail-safe functionality has different specifications to satisfy for each application.

TABLE I
EVALUATION ENVIRONMENT

Hardware	CPU	CPU name	Core-i7 6800K
		Clock frequency	3.4 GHz (3.8 GHz)
		Cores/Threads	6/12
	Memory	Capacity	32GB
	GPU	GPU name	GTX-1080
Software	Operating System		Ubuntu 14.04 64bit
	Linux kernel version		3.13
	ROS version		Indigo
	OpenCV version		2.4.13
	PCL version		1.7.2

IV. EVALUATION

The real-time performance of ROSCH was evaluated to ensure that the following four requirements are met:

- The sensor timestamp gap is guaranteed to be less than a certain value.
- The end-to-end latency is guaranteed to be less than a certain value.
- The throughput of the end node is guaranteed to be a certain rate.
- The overhead of the fail-safe functionality at deadline miss is reasonable.

The specifications of the experimental setup are listed in TABLE I. The system used for evaluation is the localization & obstacle detection system implemented the synchronization illustrated in Fig. 10. This system simulates a real environment using ROSBAG, which is a log data playback functionality for ROS. In this evaluation, ROSBAG publishes the two sensor topics (image_raw and points_player topic) from the camera driver and LiDAR nodes.

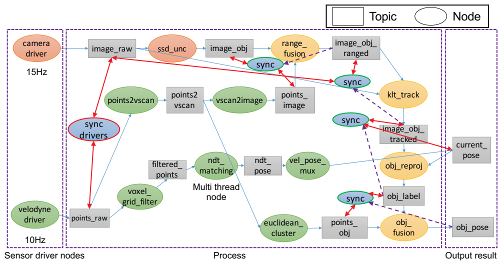

Fig. 10. Localization & obstacle detection system introducing synchronization.

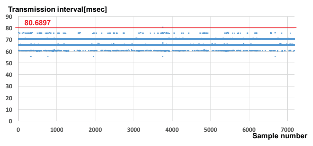

Fig. 11. Publishing interval of camera driver.

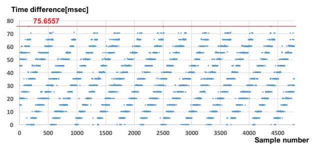

Fig. 12. Timestamp gap of Fig. 10.

A. Guarantee that the sensor timestamp gap is less than a certain value

The timestamp gap of the sensing data between LiDAR and camera was measured when the synchronization system was used. First, we measured the publishing interval of the camera driver (Fig. 11). The camera driver basically operates at 15 Hz, but the publishing interval was not always 66.67 ms. Therefore, it is necessary to set the publishing period to the estimated maximum interval of the camera driver node. Because the maximum publish interval was 80.6897 ms, it can be observed that the sensor timestamp gap is less than 80.6897 ms. As shown in Fig. 12, the maximum gap is 75.6557 ms, which is less than the predicted calculation result. Moreover, the sensing data timestamp gap between current_pose and obj_pose were the same and synchronized.

B. Guarantee that end-to-end latency is less than a certain value

The end-to-end latency obtained by the fixed-priority based DAG scheduling framework was measured. We set the data playback speed of ROSBAG to 10 % and measured the end-to-end latency until the obj_pose topic and current_pose topics were published.

First, as described in Section III-B, the execution time of each node was measured. One thousand samples were collected and their WCETs were estimated. In this system, the ndt_matching node is a multithreaded node. We measured the execution time in each case and found that the number of cores changed from 1 to 12 and determine the number of cores, 7, to operate ndt_matching with the shortest WCET. An automatic analysis with parallel-exHLBS was performed using these estimated WCETs, as shown in Fig. 13. WCETs are obtained by rounding to the whole number because these values can only be specified on the order of ms because of the specifications of Linux system calls. Moreover, we set the deadline to 90 ms for the obj_pose topic and to 70 ms for the

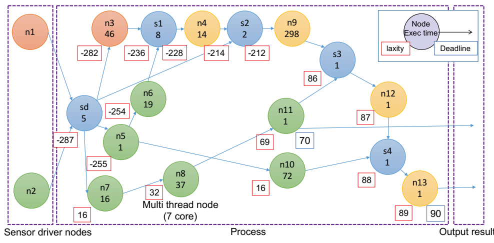

Fig. 13. Localization & obstacle detection system introducing synchronization in DAG

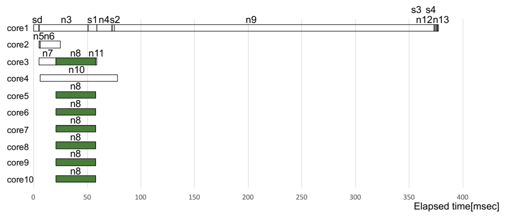

Fig. 14. Core allocation results for Fig. 13.

current_pose topic. The core allocation resulting is shown in Fig. 14.

The results of measuring the end-to-end latency until the obj_pose topic and current_pose topics were published as shown Figs. 15 in and 16, respectively. The end-to-end latency until the obj_pose topic was published was around 100 to 200 ms, but could be as much as 254.5 ms. This figure is far from the worst value of analysis result value of 377 ms. This is because the analysis made estimates using the maximum execution time of all the nodes. The end-to-end latency until the current_positive topic was published was around 40 to 60 ms. This time is close to the estimate because the number of nodes leading up to the current_pose topic and the overall execution time is small compared to obj_pose topic. The maximum value was 71.9 ms and more than the estimation value. The reason is thought to be because communication overhead is large when the voxel_grid_filter node handled a large topic without using a nodelet.

C. Guarantee on the throughput of the end node

We evaluated the operation frequency of the LiDAR driver node, which affects the operation frequency of the whole system. Moreover, we showed that we could estimate the end-to-end The operation cycle of the whole system is 10 Hz according to the operation cycle of the LiDAR driver node. Therefore, the frequency of whether the system can operate at 10 Hz is influenced by the operation frequency of the LiDAR driver node. Therefore, to measure the accuracy of the throughput of the final output topic publishing frequency, the operation frequency of the LiDAR driver node was measured. We measured the timestamp intervals of the points_topic topic from the LiDAR driver node. The results are shown in Fig. 17. The maximum interval was 116.14 ms, the minimum was 85.57 ms, and the average was 100.09 ms. Therefore, the frequency of publishing showed a value really close to 10 Hz and the system operation frequency was conjectured constant.

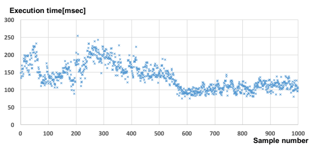

Fig. 15. End-to-end latency for the obj_pose topic.

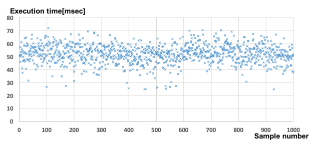

Fig. 16. end-to-end latency for the current_pose topic.

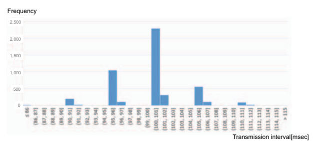

Fig. 17. Publishing intervals for the LiDAR driver (histogram).

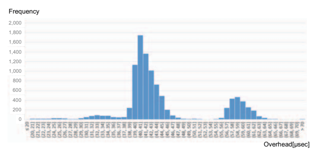

Fig. 18. ROS fail-safe overhead (histogram).

D. *Overhead of fail-safe at deadline miss*

The overhead from the deadline miss of a callback function to calling a fail-safe function was measured. The deadline was set to 2 ms and the callback function is made to take at least 10 ms for processing. The evaluation results are shown in Fig. 18. The time used to call the node was a maximum of 107 μsec, minimum of 15 μsec, and the average of 45 μsec. We compared the evaluation result and the average execution time of the node of Fig. 10. This average execution time was 11.24 ms per node. Therefore, the fail-safe functionality overhead in average node execution time of the obstacle detection system was 0.004 %.

V. Related Work

rosc: rosc [24] is a dependency-free ROS client implementation in ANSI C that aims to support small embedded systems and any operating system. It is available as an alpha release. The developers had planned real-time extensions to rosc; however, the development of rosc has ended.

μROS: μROS [25] is a middleware for embedded systems that need to communicate with ROS in a lightweight and quick manner. μROS can run on RTOSs such as ChibiOS/RT. However, the ROS node code does not work with the μROSnode and it is difficult to port from ROS. In addition, at present, μROS is an alpha release and has not been in development for very long.

rosrt: rosrt [26] is a package that contains real-time safe tools for interacting with ROS. When used together with Xenomai [28], which is an RTOS, it guarantees that the CPU time is not affected by other processes. However, rosrt is currently experimental and the API is not stable because it was implemented for the Willow Garage PR2 [12], which is a robotics research and development platform.

RT-ROS: RT-ROS [27] is based on the hybrid operating system platform RGMP (RTOS and GPOS on multiprocessors). Nodes that require real-time constraints can run on RTOSs. However, RT-ROS is not open-source and does not appear to support libraries related to ROS, which means that it is necessary to modify existing ROS node programs.

ROS 2.0: ROS 2.0 [14], [15] is a next-generation ROS that uses DDS [29] as a communication middleware. It is targeted at real-time embedded systems such as an autonomous driving vehicle; thus, it supports real-time constraints. Although it can communicate with existing ROS nodes to enable mixed systems, the ROS node code does not work with ROS 2.0. In other words, the legacy code is not supported for real-time constraints.

TABLE II
Comparison of the Proposed Framework and Prior Work

	Libraries (Open CV, PCL, etc.)	Modification less	Synchronization	Open source
rosc [24]		x		x
μROS [25]				x
rosrt [26]	x			x
RT-ROS [27]	Not available to all	Not available to all		
ROS 2.0 [14], [15]	x			x
The proposed framework (ROSCH)	x	x	x	x

Table II compares the proposed framework (ROSCH) and previous work. The table shows that rosc and μROS do not support libraries related to ROS such as Open CV and PCL. If μROS, rosrt ,and ROS 2.0 are used, developers must modify the existing ROS node programs. RT-ROS is not open-source and does not appear to support the libraries and modifications. Note that ROSCH has all the extensions shown in Table II. ROSCH is open-source and does not require the modification of existing ROS node programs.

VI. Conclusions

We presented a real-time scheduling framework for ROS called ROSCH that meets the four real-time requirements occurring in ROS. We verified whether the synchronization system, fixed-priority based DAG scheduling framework, and fail-safe functionality could satisfy the requirements. The timestamp gap of the sensor fell within the criterion in Section III-A. The end-to-end latency from the data input to the final output can also be estimated. However, because the guaranteed delay was pessimistically estimated, it deviated from the actual execution time. In addition, because some of the coding conventions stipulated in this paper are not followed, the estimated end-to-end latency could be exceeded. We can guarantee throughput of the final output topic publishing frequency by running the entire system at regular intervals. However, the accuracy of the operation depends on the sensors and driver nodes, and there are some variations in their publishing intervals. The fail-safe overhead increased the total overhead by the small value of 0.004 %. These results show that the synchronization system and fixed-priority based DAG scheduling framework are effective for real-time performance. In the future, we plan to develop WCET estimation using probability, pipelining of exHLBS, and early prediction of deadline misses using laxity.

Our prototype system and application programs used in the performance evaluation are all open-source, and may be downloaded from [https://github.com/CPFL/Autoware] and [https://github.com/CPFL/ROSCH].

Acknowledgement

This work was partially supported JST PRESTO Grant Number JPMJPR1751.

References

[1] Waymo, "Waymo," https://waymo.com/.

[2] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[3] M. Buehler, K. Iagnemma, and S. Singh, *The DARPA urban challenge: autonomous vehicles in city traffic*. Springer, 2009, vol. 56.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[5] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, 2013.

[6] W. G. Inc, "Willow Garage Inc," https://www.willowgarage.com/.

[7] C. Ainhauser, L. Bulwahn, A. Hildisch, S. Holder, O. Lazarevych, D. Mohr, T. Ochs, M. Rudorfer, O. Scheickl, T. Schumm, and F. Sedlmeier, "Autonomous Driving needs ROS," 2013.

[8] M. Aeberhard, T. Kuhbeck, B. Seidl, and et al., "Automated Driving with ROS at BMW," 2015.

[9] kwc, "Robots Using ROS: Stanford Racing's Junior," http://www.ros.org/news/2010/03/robots-using-ros-stanfords-junior.html.

[10] ——, "Robots Using ROS: Marvin autonomous car (Austin Robot Technology UT Austin)," http://www.ros.org/news/2010/03/robots-using-ros-marvin-autonomous-car.html.

[11] T. Foote, "Driverless Development Vehicle with ROS Interface," http://www.ros.org/news/2016/02/driverless-development-vehicle-with-ros-interface.html.

[12] J. Barry, M. Bollini, A. Holladay, L. Kaelbling, and T. Lozano-Pérez, "Planning and control under uncertainty for the PR2," in *Proc. of IROS PR2 Workshop*, 2011, extended abstract.

[13] "PR2," http://www.willowgarage.com/pages/pr2/overview.

[14] "ROS 2.0," http://design.ros2.org/.

[15] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[16] H. Wei, Z. Huang, Q. Yu, M. Liu, Y. Guan, and J. Tan, "RGMP-ROS: A real-time ROS architecture of hybrid RTOS and GPOS on multi-core processor," in *Proc. of 2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 2482–2487.

[17] J. Sticha, "Validating the Real-Time Capabilities of the ROS Communication Middleware," in *Bachelor ' s Thesis in Informatics*, 2014.

[18] Y. Saito, T. Azumi, S. Kato, and N. Nishio, "Priority and Synchronization Support for ROS," *Proc. of 2016 IEEE 4th International Conference on Cyber-Physical Systems Networks and Applications (CPSNA)*, pp. 77–82, 2016.

[19] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS2018)*, 2018.

[20] M. O'Kelly, H. Abbas, S. Gao, S. Shiraishi, S. Kato, and R. Mangharam, "APEX: Autonomous Vehicle Plan Verification and Execution," *SAE World Congress*, vol. 1, 2016.

[21] H. Mengwen, E. Takeuchi, Y. Ninomiya, and S. Kato, "Robust virtual scan for obstacle detection in urban environments," in *Proc. of IEEE Intelligent Vehicles Symposium (IV)*, 2016, pp. 683–690.

[22] S. Kato, R. raj Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Department of Electrical and Computer Engineering and Carnegie Mellon University, Tech. Rep., 2009.

[23] Y. Suzuki, T. Azumi, N. Nishio, and S. Kato, "HLBS: Heterogeneous laxity-based scheduling algorithm for DAG-Based real-time computing," *in Proc. of 2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 83–88, 2016.

[24] N. Ensslen, "Introducing rosc," in *ROS Developer Conference 2013*, 2013.

[25] M. Migliavacca, A. Zoppi, M. Matteucci, and A. Bonarini, "uROSnode running ROS on microcontrollers," in *Proc. of ROS Developer Conference 2013*, 2013.

[26] "rosrt," http://wiki.ros.org/rosrt.

[27] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "RT-ROS: A real-time ROS architecture on multi-core processors," *Future Generation Computer Systems*, vol. 56, pp. 171–178, 2016.

[28] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *Xenomai*. Betascript Publishing, 2010.

[29] G. Pardo-Castellote, "OMG Data-distribution Service: Architectural Overview," in *Proc. of the 2003 IEEE Conference on Military Communications - Volume I*, ser. MILCOM'03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 242–247.