2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform

Takuya Azumi[1], Yuya Maruyama[2], and Shinpei Kato[3]

Abstract— **This paper proposes ROS-lite, a robot operating system (ROS) development framework for embedded many-core platforms based on network-on-chip (NoC) technology. Many-core platforms support the high processing capacity and low power consumption requirement of embedded systems. In this study, a self-driving software platform module is parallelized to run on many-core processors to demonstrate the practicality of embedded many-core platforms. The experimental results show that the proposed framework and the parallelized applications have met the deadline for low-speed self-driving systems.**

I. Introduction

The development of next-generation computing platforms with multi-/many-core platforms is a necessary response to satisfy the increasing demand for high-computational power with low power consumption, such as in autonomous vehicles. Heterogeneous computing systems, such as multi-/many-core platforms and graphical processing units (GPU), are often utilized autonomous driving requires high-speed processing, predictability, and energy efficiency for several computational modules, such as localization, path planning, and path following. Multi-/many-core architectures can achieve high-performance and general-purpose computing with low power consumption for embedded systems [1], [2]. Several applications for multi-/many-core platforms were recently examined in literature [3], [4].

Multi/many-core platforms are available as commercial off-the-shelf (COTS) multicore components, such as massively parallel processor arrays (MPPA) 256 developed by Kalray [5], Tile-Gx [6], and Tile-64 developed by Tilera [7]. Several of these platforms, such as MPPA-256 and Tile-64, are intended for embedded systems, where the research focusing on multi-/many-core platforms has attained increasing attention [8]. Nonuniform memory access (NUMA) and distributed memory devices connected with network-on-chip (NoC) components allow core scalability and reduce power consumption. Several COTS platforms include NoC technology and a cluster of many-core processors where the cores are allocated closely. For example, MPPA-256 and Tile-Gx72 use NoC components to share distributed memory, instead of shared buses. MPPA-256 contains 16 clusters of 16 cores for 256 general-purpose cores. Although cache coherency cannot be guaranteed by these cores, MPPA-256 significantly exceeds the number of cores available in other COTS, such as Tile-64 and Tile-Gx72. The clusters of cores can run independent applications to achieve the desired power envelope for embedded applications.

Despite the recent developments in embedded many-core platforms, several challenges exist for adapting these platforms in embedded applications [1]. The research on NoC-based embedded many-core platforms did not clearly elucidate the data transfer mechanism between the distributed memory banks with NoC technology, memory access characteristics for NUMA, and the strategies that can be used by developers for parallelization in practical applications. Moreover, reusing existing software on these platforms is difficult because platform-specific code for NoC data transfer between clusters needs to be written by application developers. Writing source code for robotics systems is difficult particularly when the scale and scope of the embedded software applications are continuously growing.

To meet these challenges, this paper proposes a software development framework for the embedded many-core platform called ROS-lite based on a robot operating system (ROS) [9], that allows the reuse of existing applications and efficiently developing on heterogeneous computing platforms. ROS is a structured communications layer above the host operating systems of a heterogeneous computing cluster and is widely used in the robotics community. The MPPA-256 [5] serves as a reference platform. MPPA-256 adopts NUMA using NoC technology which resulted in numerous cores with low power consumption. The test is performed using the evaluation code, ROS-lite, and in a practical application.

Contributions: This paper proposes a lightweight ROS architecture for embedded many-core platforms, called ROS-lite. The advantages of embedded many-core computing platforms are quantified based on NoC technology with the following main innovations: given the limited memory available, the proposed framework provides a structured communications layer and efficient development for heterogeneous computing platforms. ROS-lite can coexist off-load applications, such as a self-localization application [8], which uses several clusters and more than one hundred cores.

To the best of our knowledge, this is the first study on ROS nodes running on embedded many-core platforms. The demonstration for computational speed improvements for a self-driving application indicates the practical potential of NoC-based embedded many-core computing. ROS-lite provides an efficient development framework where ROS nodes can run on many-core platforms and communicate with each other.

Organization: The remainder of this paper is organized as follows. Section II discusses the reference hardware

[1] Graduate School of Science and Engineering, Saitama University
[2] Graduate School of Engineering Science, Osaka University
[3] Graduate School of Information Science and Technology, the University of Tokyo

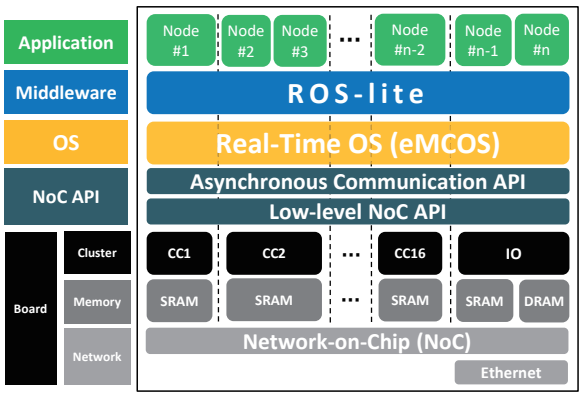

Fig. 1. System stack of ROS-lite on a many-core platform.

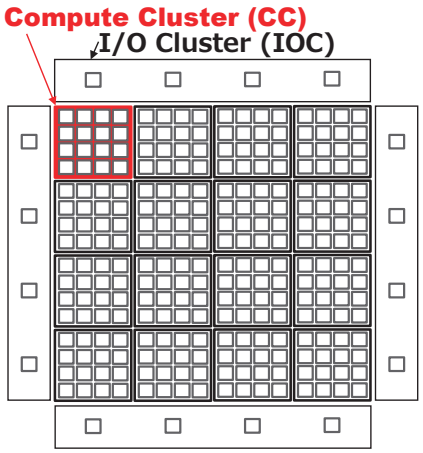

Fig. 2. Overview of the Kalray MPPA-256 Bostan architecture.

model and the software model. The proposed framework is presented in Section III, which discusses the development flow, design, and implementations. Section IV illustrates the experimental evaluation. Section V reviews the related work that focuses on the ROS framework. Finally, Section VI presents the conclusions and directions for future work.

II. System Model

This section presents the system model shown in Fig. 1. The many-core model of Kalray MPPA-256 Bostan is considered. First, a hardware model is introduced in Section II-A, followed by a software model in Section II-B.

A. Hardware Model

The MPPA-256 processor is based on an array of computing clusters (CCs) and I/O clusters (IOCs) that are connected to NoC nodes using a two-dimensional toroidal topology, as shown in Figs. 2 and 3. The MPPA many-core chip integrates 16 CCs and 4 IOCs on NoC. The Kalray MPPA-256 architecture is briefly presented in this section and is further discussed in Ref. [8].

I/O Clusters (IOCs): MPPA-256 contains the following four IOCs: North, South, East, and West. The North and South IOCs are connected to a DDR interface and an eight-lane PCIe controller. The East and West IOCs are connected to a quad 10 Gb/s Ethernet controller. Each IOC consists of quad IO cores and a NoC interface.

Compute Clusters (CCs): In MPPA-256, the 16 inner nodes of the NoC correspond to the CCs.

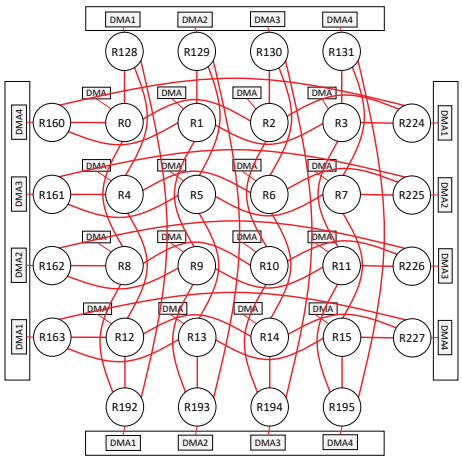

Fig. 3. NoC connections (both D-NoC and C-NoC).

Network-on-Chip The 16 CCs and the 4 IOCs are connected by NoC as shown in Fig. 3. Furthermore, NoC is constructed as a bus network and has routers on each node.

A bus network connects nodes (CCs and IOCs) using torus topology [10], which involves a low average number of hops compared to mesh topology [11]. The network is actually composed of the following two parallel NoCs with bidirectional links (denoted by red lines in Fig. 3): the data NoC (D-NoC), which is optimized for bulk data transfers and the control NoC (C-NoC), which is optimized for small messages at low latency. The NoC is implemented with wormhole switching and source routing. Data is packaged in variable-length packets that are broken into small pieces called flow control digits.

B. Software Model

The software stack used for Kalray MPPA-256 comprises a hardware abstraction layer, a library layer, an OS, and a user application. Fig. 1 shows the software stack used for Kalray MPPA-256 in the present work. The Kalray system is an extensible and scalable array of computing cores and memory. With respect to the scalable computing array of the system, several programming models or runtimes can be mapped, such as Linux, a real-time operating system, POSIX API, OpenCL, and OpenMP. Each layer is described in detail.

In the hardware abstraction layer, an abstraction package abstracts the hardware of a CC, an IOC, and a NoC. The hardware abstraction is responsible for hardware resources partitioning and controlling access to the resources from the user-space operating system libraries.

1) Operating System: Several operating systems support the abstraction package in the OS layer. Numerous cores, have difficulty in supporting previous operating systems for single/multicore(s) owing to problems involving parallelism and cache coherency [12]. Here, a real-time operating system (RTOS) supporting MPPA-256 is introduced.

eMCOS: On both CCs and IOCs, eMCOS provides minimal programming interfaces and libraries. Specifically, eMCOS is a real-time embedded operating system for many-core processors. The OS implements a distributed microkernel architecture. This compact microkernel is equipped with

minimal functions only. The eMCOS enables applications to operate a priority-based message passing, local thread scheduling, and thread management on IOCs as well as CCs.

2) NoC Data Transfer Methods: This section explains the data transfer methods used in MPPA-256. For scalability purposes, MPPA-256 implements a clustered architecture that reduces memory contention between numerous cores and associates memory with each cluster. 16 cores are packed in a cluster, and each cluster shares a 2 MB memory (SMEM), which reduces memory contention that frequently occurs with numerous cores and helps in increasing the number of cores. However, the clustered architecture constrains the memory that is directly accessible by each core. Communicating with cores outside the cluster requires data transfer between clusters through the D-NoC via NoC interfaces.

eMCOS message: eMCOS provides message APIs for communication between threads in the D-NoC. The destination of a message is a thread, and sending/receiving processes use asynchronous/synchronous modes. Users can also configure the receiving behavior with masks and priority-based filtering. Under the messaging APIs, data can be exchanged between threads regardless of the cluster on which the thread is running. eMCOS runs on IOC and CC to equally manage threads and messages between all clusters. eMCOS manages the message buffering on each core, and the data sent by DMA through D-NoC are copied to the buffer in eMCOS. After resolving the reception order, the data are provided by copying the non-cache access from the message buffer to the user-space addresses.

eMCOS session message: eMCOS provides message APIs for large-data transfer using DMA interfaces through D-NoC. This API is designed to rapidly send large volume of data via the UC. Users are required to open sessions that correspond to pairs of send and receive buffers. The number of segments is limited by the number of UCs on each cluster. Users transport data between clusters using sessions established in advance. The sending/receiving behaviors utilize asynchronous/synchronous modes.

C. ROS (Robot Operating System)

As previously discussed, Autoware is based on ROS [13], [9], which is a component-based middleware framework developed for robotics research. ROS is designed to enhance the modularity of robot applications at a fine-grained level and is suitable for distributed systems, while allowing for efficient development. Given that autonomous vehicles require many software packages, ROS provides a strong foundation for Autoware's development.

In ROS, the software is abstracted as *nodes* and *topics*. The *nodes* represent the individual component modules, whereas the *topics* mediate the inputs and outputs between *nodes*, as illustrated in Fig. 4. ROS *nodes* are usually standard programs written in C++, which can interface with other installed software libraries.

Communication among *nodes* follows a publish/subscribe model, which is a strong approach to modular development. In this model, *nodes* communicate by passing *messages* via

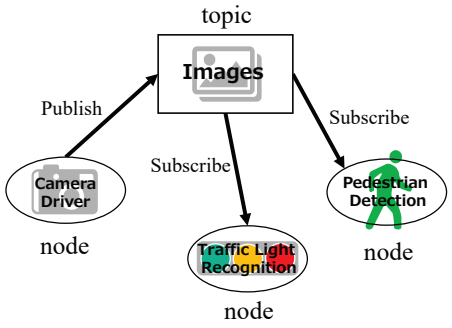

Fig. 4. Publish/subscribe model in ROS.

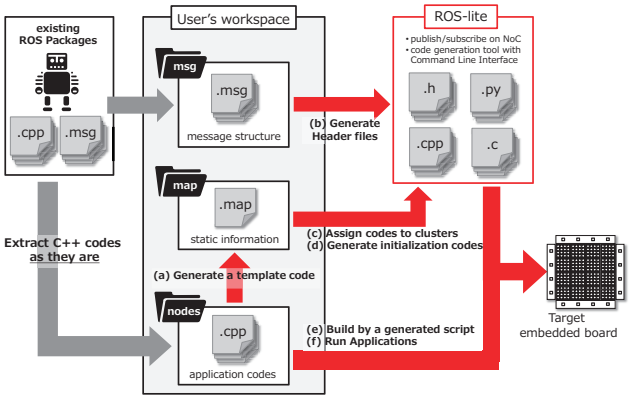

Fig. 5. Development flow of ROS-lite framework.

a *topic*. A *message* has a simple structure (almost identical to structs in C) and is stored in .msg files. *Nodes* identify the content of the *message* based on the *topic* name. When a *node* publishes a *message* to a *topic*, another *node* subscribes to the *topic* and uses the *message*. For example, in Fig. 4, the "Camera Driver" *node* sends *messages* to the "Images" *topic*. The *messages* in the *topic* are received by the "Traffic Light Recognition" and "Pedestrian Detection" *nodes*. The *topics* are managed using first-in, first-out queues when accessed by multiple *nodes* simultaneously. At the same time, ROS *nodes* can implicitly launch several threads. However, the issues concerning real-time processing must be addressed.

III. Proposed Framework

The proposed framework, ROS-lite, is a structured communications layer that enables efficient application development for many-core platforms. Although the existing ROS is well-developed and widely used, it does not support NoC communication on many-core platforms as well as the required features for embedded platforms, such as RTOSs and limited memory. ROS-lite addresses these challenges and provides a development environment where ROS nodes run on each core and communicate with each other. ROS-lite makes porting existing software and new development on embedded platforms more efficient by adapting and extending numerous existing ROS applications. Given that ROS-lite is based on the existing ROS framework, it can communicate with external ROS nodes on another platform. The tests reported in this study used eMCOS as an RTOS and MPPA-256 as an embedded many-core platform.

A. ROS-lite Toolchain

This section describes ROS-lite in detail. The development flow of ROS-lite is shown in Fig. 5. ROS-lite can run

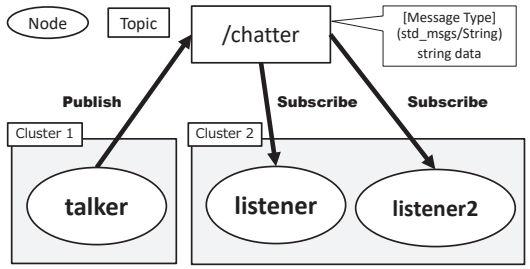

Fig. 6. Publish/subscribe model in ROS-lite.

```
- name: talker
  cluster: 1
  publish: [/chatter]
  publish_type: ["std_msgs::String"]
  subscribe: []
  subscribe_type: []
- name: listener
  cluster: 2
  publish: []
  publish_type: []
  subscribe: [/chatter]
  subscribe_type: ["std_msgs::String"]
- name: listener2
  cluster: 2
  publish: []
  publish_type: []
  subscribe: [/chatter]
  subscribe_type: ["std_msgs::String"]
```

Fig. 7. Map file description (`.map`).

application code written as ROS nodes. We assume that the applications are written in C++ similar to ROS. Application developers can adapt the source code (application code and message structure files) from existing ROS packages without changes. For task mapping, developers edit a map file (`.map`) and assign ROS nodes to the CCs on many-core platforms. Developers are only required to edit the map file to modify task mappings. The ROS nodes as processes on cores are described in terms of a publish/subscribe model, following the message structures defined in the message files (`.msg`). ROS nodes can communicate with each other via either intra-cluster or inter-cluster routes. Intra-cluster communications are handled via shared memory, whereas the inter-cluster communications are handled via NoC.

ROS-lite toolchain provides a code generation and a build system with a command-line interface to allow efficient development. First, similar to (a) in Fig. 5, a template of a map file (`.map`) is generated from the application source code. The map file (`.map`) contains node names, the assigned cluster, and the information of topics that are published and subscribed. Figure 7 shows an example of the map file specifications. These descriptions, except for the assigned cluster number, can be interpreted from ROS nodes to provide code generation for the template code. Application developers can then focus on the number of assigned clusters. Second, similar to (b) in Fig. 5, the header files that define the message structure are generated from message files (`.msg`), similar to the original ROS. The generated header files are lighter than when they are in ROS because only the part required by ROS-lite is generated. This code-generation module is based on the original ROS script, and the message files (`.msg`) can be described as they are in ROS. The message files (`.msg`) do not require any modifications. Third, similar to (c) and (d) in Fig. 5, the initialization code for launching processes of ROS nodes is generated from a map file (`.map`). ROS nodes are automatically launched as processes scheduled by the RTOS on user-assigned clusters. Application developers do not have to write code except for the ROS nodes that are used. Finally, similar to (e) in Fig. 5, a build script is generated from the user-defined map file (`.map`). The source code of the ROS nodes is built separately for each user-assigned cluster because the executable files are loaded into separate memory banks in each cluster. This process is conducted by the build script, which does not require any modification when the task mapping is changed by modifying the map file (`.map`). The code for the ROS nodes is allocated in a single directory, and each node is automatically built in each user-assigned cluster.

A simple example to understand the ROS-lite framework is provided herein. One node publishes a *chatter* topic and two nodes subscribe to the topic, as shown in Fig. 6. Messages in the *chatter* topic are defined as a string type named *data*. The publisher node is launched in *cluster 1*, and two subscriber nodes are launched in *cluster 2*, as described by the assigned cluster number in the map file (`.map`) shown in Fig. 7. Application developers can change the node mapping by modifying the cluster number field. Information on the topics in the map file (`.map`) is used to initialize the relation between the topics and the nodes so that ROS-lite directs the process of matching the nodes with topic names. The fields for node name and topic information are generated from the source code of the ROS nodes. The initialization script for the node relations in ROS-lite is generated from the map file (`.map`). The field for the assigned cluster numbers must be filled by the application developers.

B. Design and Implementation

We tested ROS-lite with an implementation on MPPA-256 as shown in Fig. 1. MPPA-256 is an example of NoC-based clustered many-core processors with distributed memory architecture. If memory is limited, ROS-lite can run on this platform similar to an embedded application. ROS-lite allows NoC-based message transfer with low memory consumption. ROS-lite is a general framework and can be used with other many-core processors.

In our test, eMCOS is used as the RTOS for ROS-lite because it has rich messaging functions, as described in Section II-B.2 and can run well on CCs and IOCs. eMCOS manages all threads under ROS-lite. The light-weight threads that are generated are not core-affinity threads. Threads are not allocated to a specific core within a cluster but are migrated within the specific cluster following the eMCOS scheduling policy. Therefore, eMCOS improved internal thread management.

ROS-lite does not have a master server similar to ROS ' s **ros master**, which helps the original framework in avoiding single failures. Nodes in ROS-lite use map files to statically initialize the topic information that specifies each node that

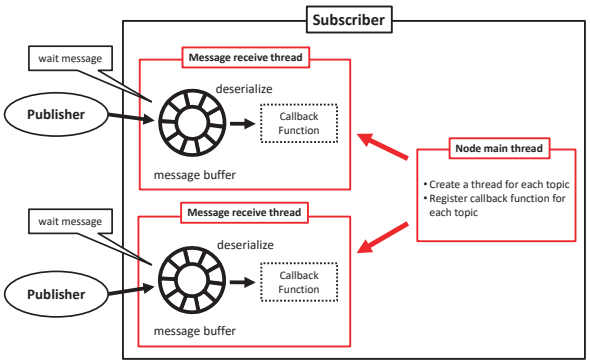

Fig. 8. Structure of the subscriber node.

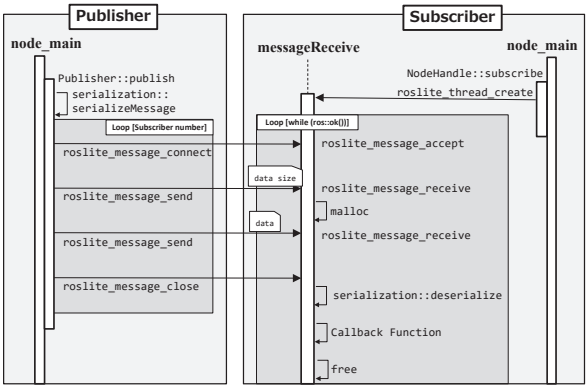

Fig. 9. Design of the publish/subscribe sequence of Fig. 8 without message buffering.

subscribes to each topic outside its cluster. Initialization scripts are generated off-line by the command-line interfaces. We adopted this off-line initialization process given that many specifications on the execution time will be fixed ahead of time for most embedded systems. By eliminating the master server, ROS-lite avoids the unnecessary communication between clusters and a single failure point.

Next, the internal design of the publish/subscribe transport for ROS-lite is discussed. A subscriber's main thread creates a **message receive function** for each topic, as shown in Fig. 8. The subscriber nodes handle the message buffering and behavior with one process in ROS-lite. The design in Fig. 8 can be implemented easily with the message function in eMCOS. The message function already implements message buffering and NoC transport between threads, but the buffer is allocated in kernel-space and cannot be dynamically configured. ROS-lite can use session messaging to address this problem. Figure 9 shows the implementation design of publish/subscribe transport in ROS-lite. Prior to data transport, the publisher nodes serialize the data following the message structure. The serialization policy of ROS-lite is based on the original ROS. After serialization, the publisher node sends messages to the subscriber nodes listed in the topic information, which are initialized by the map file off-line. After receiving the serialized data size, the subscriber node allocates memory and creates segments for data transport. Frequent **alloc** and **free** of heap memory for each message size conserve the limited memory of the embedded

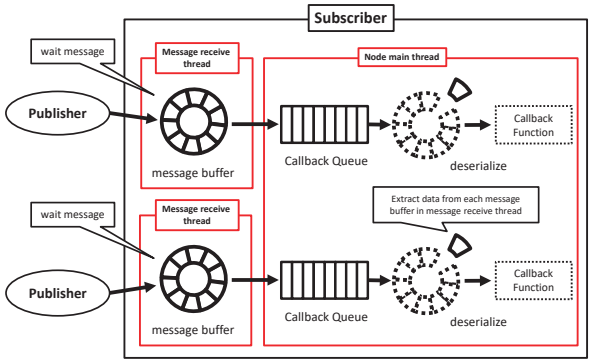

Fig. 10. Structure of subscriber node with callback queue.

system. Although this transaction increases overhead, it gives flexibility for ROS-lite to handle variable-length messages, such as vectors.

To prevent the simultaneous execution of callback functions and design and ensure that each node only runs as one process, the design shown in Fig. 10, must be used instead of the design shown in Fig. 8. A callback queue manages the order of callback functions to be executed at the subscriber node. This queue can be implemented with eMCOS message buffering. eMCOS messaging is suitable for small-size data transport, and the data pushed for the queue is small because it only contains pointers to the callback function and received messages.

ROS-lite supports the original ROS source code to enhance porting and development efficiency for applications running on many-core processors. ROS-lite provides similar interfaces to the ROS API for application developers who are not familiar with NoC interfaces. The code of talker and listener can be used for the scenario in Fig. 6.

C. Case Study

This work adopts a portion of a self-driving system, and this section demonstrates the parallelization potential of the MPPA-256. As shown in Fig. 11, the node set achieves slow self-driving mobility as level 3 or 4, where the is determined in advance such as that of a bus. These nodes are based on Autoware [14] version 1.8. We selected an algorithm for vehicle self-localization, path planning, path following, sensor fusion written in C++ in Autoware, open-source software for urban self-driving [14], and a parallelized part of the self-driving software.

The demonstration is divided into two parts: ROS-lite nodes and an off-load node.

1) ROS-lite nodes: ROS-lite nodes comprise path planning nodes (**lane_rule**, **lane_select**, and **waypoint_filter**, and **velocity_set**) and path following nodes (**pure_pursuit** and **twist_filter**).

lane_rule generates traffic waypoints, including velocity information based on the waypoints read by waypoint_loader. **lane_select** selects the driving lane. **waypoint_filter**, which is part of obstacle_avoid, generates filtered waypoints for path following. **velocity_set** determines the final velocity using sensor information. **pure_pursuit** calculates the ve-

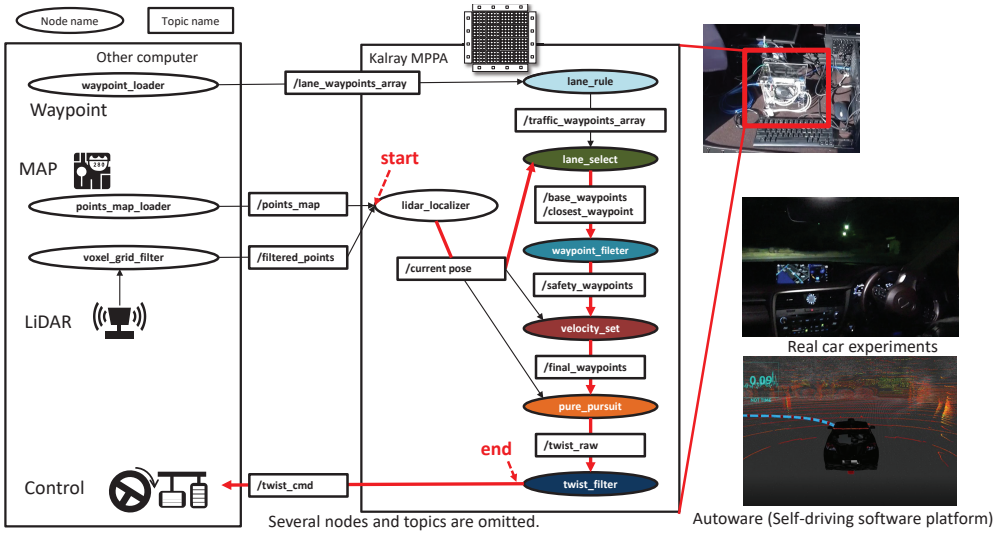

Fig. 11. Node diagrams for Autoware on MPPA-256.

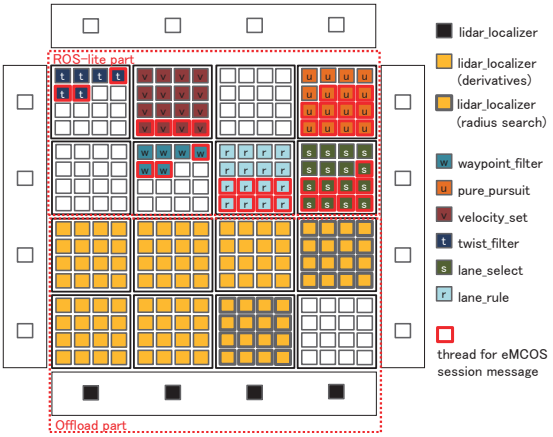

Fig. 12. A situation of self-driving system.

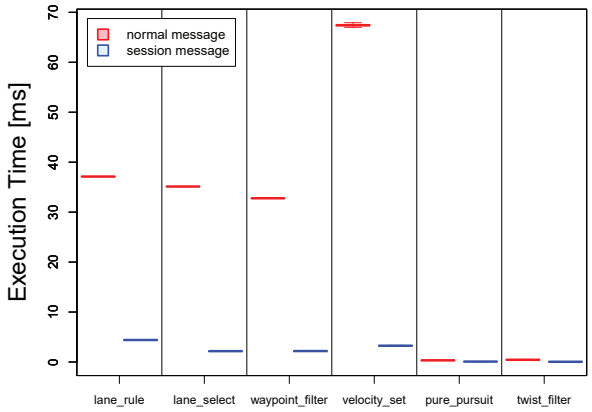

Fig. 13. Execution time of each node.

locity and the steering angle of the next movement based on current_pose and final_waypoints. **twist_filter** checks the twist_raw and adjusts the steering angle for safety.

Each node is allocated to one cluster using ROS-lite as shown in the upper part of Fig. 12. For this demonstration, a subscriber and an eMCOS session message thread were allocated to one core[1]. ROS-lite generates the code between each node based on the mapping information.

2) Off-load node: The self-localization adopts the normal-distribution transform matching algorithm implemented in the Point Cloud Library [15]. The self-localization based on LiDAR sensor (**lidar_localizer**) is one of the heaviest computations in the self-driving systems. A diagram depicting the self-driving system [16] is shown in Fig. 11.

The self-localization algorithm is primarily divided into two processes: *radius_search*, which searches for several nearest neighbor points for each query and calculates the distance, and *derivatives*, which calculates the derivative to determine the convergence of the matching operation. To parallelize *radius_search*, the algorithm of the nearest neighbor search needs to be redesigned because the data to be searched exceeds 1 MB.

This demonstration parallelized *derivatives* onto five CCs and *radius_search* onto two CCs, and the remainder of the algorithm was executed in parallel on the IOC with its four cores. To parallelize the remainder of *computeTransform* in CCs, the algorithm of the nearest neighbor search must be redesigned because the data to be searched exceeds 1 MB.

IV. Evaluations

First, this section examines the two types of evaluations: ROS-lite NoC data transfer evaluation and latency characteristics of RTOS interfaces. Subsequently, we performed practical self-driving applications to examine the practicality of the NoC-based embedded many-core platform. Finally, we arranged the lessons learned from the evaluations previously discussed. The following evaluations were conducted on real hardware boards with eMCOS.

A. Experimental results

The execution time of each node is shown in Fig. 13 using eMCOS message and eMCOS session message. In the case of sending light data (**pure_pursuit** and **twist_filter**), we can select eMCOS message given that the results are almost similar. In the case of sending heavy data, such as waypoints

[1]Given that computing power is surplus, multiple nodes can be run in one cluster.

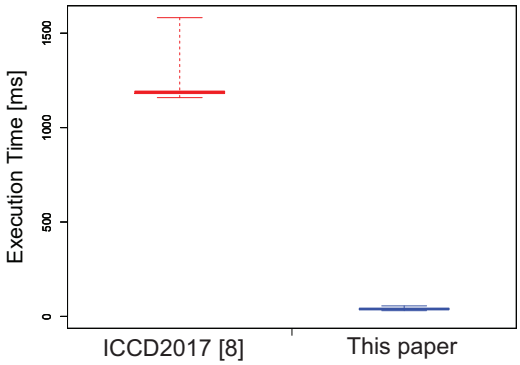

Fig. 14. lidar_localizer result.

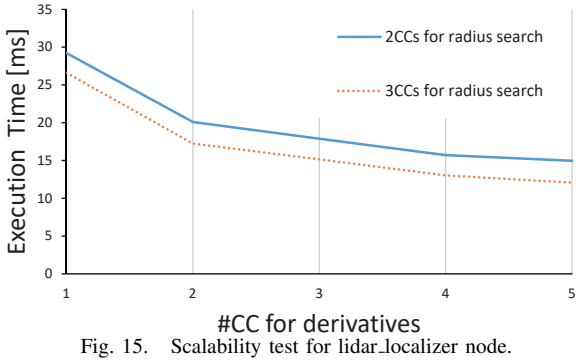

Fig. 15. Scalability test for lidar_localizer node.

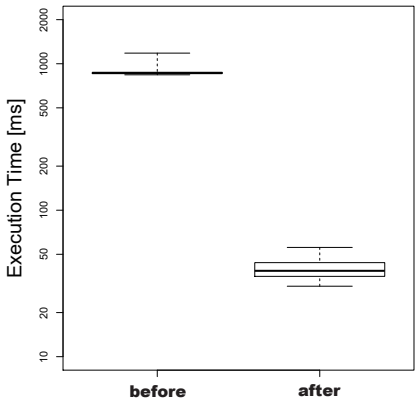

Fig. 16. End-to-end result.

TABLE I

MEMORY CONSUMPTION ON CC IN CASE OF PURE_PURSUIT.

	text (B)	bss (B)	data (B)	total (B)
RTOS (eMCOS)	241,142	53,882	2107	297,131
ROS-lite	39,857	8	24	39,889
ROS-node	46,167	0	4	46,171

and a map, eMCOS session message is faster than eMCOS message.

As shown in Fig. 14, the execution time of related work (ICCD 2017 [8]) and the proposed framework for the **lidar_localizer** node, which is one of the heaviest nodes of self-driving applications. The result shows about 50 times faster than the related work. The related work utilized all CCs and each CC, including one core, because of memory constraints for point cloud map data. On the other hand, we utilized only seven CCs (five CCs for *derivatives* and two CCs for *radius_search*) because we adopt the swapping method, which can exchange the data between SRAM on each CC and DRAM on the IOC for the point cloud map data.

We have also evaluated the range between one to five CCs for *derivatives* in the case of two CCs and three CCs for *radius_search*. At least two CCs are needed where one CC for the search part of *radius_search*, and the other is for the merge part of *radius_search*. If each CC has 4 MB, we can allocate both parts of *radius_search* to one CC. As shown in Fig. 15, the scalability of the **lidar_localizer** node can be observed. The execution time of Fig. 15 comprises one loop, including *derivatives* and *radius_search*. **lidar_localizer** usually converges within five times.

As shown in Fig. 16, the execution time is the end to end time, including the **lidar_localizer**, **path planning** nodes (**lane_rule**, **lane_select**, and **waypoint_filter**, and **velocity_set**), and path following nodes (**pure_pursuit** and **twist_filter**). The left of Fig. 16 shows the execution result before parallelization, which denotes that all applications are running on the four cores on IOC. The right of Fig. 16 shows the execution result after parallelization when the nodes are mapped to cores, as shown on the upper side of Fig. 12. The query can be assumed to be 10 Hz in many autonomous driving systems. Thus, this tuning successfully meets the deadline (100 ms), which is almost the same deadline for related work [17]. This parallelized algorithm was executed on simulated and real car experiments in our test course and worked successfully. The steering, accelerator, and brake are automatically controlled based on the results of MPPA-256.

Table I shows memory consumption on CC in case of **pure_pursuit**. RTOS, ROS-lite, stack size for threads consumes around 300 KB, 40 KB, and 500 KB, respectively. Therefore, given that each CC has 2 MB memory, ROS nodes can use 1 MB for topic data, such as waypoints and current position.

B. Lessons Learned

Based on the result of the execution time, as shown in Fig. 13, we select eMCOS session message for NoC communication. However, eMCOS session message generates many threads, which are indicated by the red bold lines in Fig. 12. Given that the execution time of eMCOS message is almost similar to that of eMCOS session message, we should select eMCOS message, such as the case of pure_pursuit and twist_filter, as shown in Fig. 13.

In this demonstration nodes, some code needs to be changed due to the limited resources such as waypoints and map data. This means that it is necessary to support the swapping algorithm from SRAM of CC to DRAM of IO cluster. We can obtain insights and guidelines for users and developers of NoC-based embedded many-core platforms through MPPA-256. From the practical application viewpoint, given that we parallelize the vehicle self-localization algorithm of the self-driving system and path planning, and path following, NoC-based embedded many-core systems can be practically used in real environments. We attempted

TABLE II
COMPARISON OF PREVIOUS WORK ON ROS FRAMEWORKS

	Embedded	RTOS	Modification	Many-core	NoC
ROSCH [18]			✓		
RT-ROS [19]		✓	L		
mROS [20]	✓	✓	L		
ROS 2 [21]	✓	✓	L		
micro-ROS [22]	✓	✓	L		
this paper	✓	✓	L	✓	✓

∗In a table, "L" means "limited."

to perform parallel data transfer from an IOC to CCs and parallel computing on the IOC and CCs using the CC SMEM as scratch pad memory. We have also conducted demonstration experiments with real environments and confirmed the practicality of NoC-based embedded many-core systems.

Thus far, in Section IV, we have quantitatively clarified the characteristic of data transfer and parallel computing on NoC-based embedded many-core platforms. We can obtain insights and guidelines for users and developers of NoC-based embedded many-core platforms through MPPA-256. The results of localization, which is one of the heaviest applications, show the scalability of the application.

V. RELATED WORK

This section compares many-core platforms and discusses previous work related to the ROS framework. Previous work on several types of ROS frameworks and comparisons with of these work to ROS-lite are described.

ROSCH [18] is a ROS-based real-time framework that supports the synchronization of topic messages. RT-ROS [19] provides an integrated real-time/nonreal-time task execution environment. It is constructed using Linux and the Nuttx Kernel. mROS [20] is a lightweight runtime environment of ROS1 nodes for embedded boards. ROS 2 [21] is a next-generation ROS that uses Data Distribution Service (DDS) as a communication middleware and is targeted at real-time embedded systems such as an autonomous driving vehicle. Thus, it supports real-time constraints. However, we have to port the existing ROS applications. micro-ROS [22] supports DDS for eXtremely Resource Constrained Environments (DDS-XRCE) instead of classical DDS to run ROS 2 nodes on microcontrollers. Table II briefly summarizes the characteristics of several related frameworks and compares them to ROS-lite. Currently, these ROS frameworks do not support NoC-based many-core platforms.

VI. CONCLUSIONS

This paper proposed a lightweight framework called ROS-lite for the NoC-based embedded many-core platform. Parallelization of a real application proved the practicality of NoC-based embedded many-core platforms. The ROS-lite runs with low memory consumption and achieves that ROS nodes running on each core on many-core platforms and communicate with each other. ROS-lite achieves running several applications for self-driving systems by the deadline (100 ms). Moreover, ROS-lite can coexist with the data-intensive off-load application, which uses several CCs such as the localization of self-driving systems.

In future work, we will support ROS 2 application based on DDS, which is a rea-ltime publish/subscribe model.

ACKNOWLEDGMENTS

This work was partially supported by JST PRESTO Grant Number JPMJPR1751, the New Energy and Industrial Technology Development Organization (NEDO), and eSOL.

REFERENCES

[1] M. Becker, D. Dasari, B. Nicolic, B. Akesson, T. Nolte *et al.*, "Contention-free execution of automotive applications on a clustered many-core platform," in *Proc. of ECRTS*, 2016, pp. 14–24.
[2] Q. Perret, P. Maurère, É. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Mapping hard real-time applications on many-core processors," in *Proc. of RTNS*, 2016, pp. 235–244.
[3] S. Igarashi, Y. Kitagawa, T. Ishigooka, T. Horiguchi, and T. Azumi, "Multi-rate DAG scheduling considering communication contention for NoC-based embedded many-core processor," in *Proc. of DS-RT*, 2019.
[4] B. Paolo, B. Marko, N. Capodieci, C. Roberto, S. Michal, H. Přemysl, M. Andrea, G. Paolo, S. Claudio, and M. Bruno, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, vol. 52, pp. 299 – 311, 2017.
[5] B. D. de Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Proc. of DATE*, 2014, pp. 1–6.
[6] C. Ramey, "TILE-Gx100 manycore processor: Acceleration interfaces and architecture," in *Proc. of HCS*, 2011, pp. 1–21.
[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, "Tile64-processor: A 64-core SoC with mesh interconnect," in *Proc. of ISSCC*, 2008, pp. 88–598.
[8] Y. Maruyama, S. Kato, and T. Azumi, "Exploring scalable data allocation and parallel computing on NoC-based embedded many cores," in *Proc. of ICCD*, 2017.
[9] "ROS.org," http://www.ros.org/.
[10] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. of DAC*, 2001, pp. 684–689.
[11] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS," in *Proc. of ISSCC*, 2007, pp. 98–99.
[12] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.
[13] 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 *Proc. of IEEE ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.
[14] "Autoware: Open-source software for urban autonomous driving," https://gitlab.com/autowarefoundation/autoware.ai.
[15] "Point Cloud Library (PCL)," http://pointclouds.org/.
[16] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proc. of ICCPS*, 2018.
[17] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proc. of ASPLOS*, 2018.
[18] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "ROSCH: Real-time scheduling framework for ROS," in *Proc. of RTCSA*, 2018.
[19] 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, 2015.
[20] H. Takase, T. Mori, K. Takagi, and N. Takagi, "mROS: A lightweight runtime environment of ROS 1 nodes for embedded devices," *Journal of Information Processing*, vol. 61, no. 2, 2020.
[21] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proc. of EMSOFT*, 2016, pp. 5:1–5:10.
[22] "micro-ROS," https://micro-ros.github.io/.