Master's Thesis

Scalable Parallel Computing
on NoC-based Embedded Many-Core Platform

Supervisor

Professor Toshimitsu Ushio

Assistant Professor Takuya Azumi

By

Yuya Maruyama

February 02, 2018

Division of Mathematical Science for Social System,
Department of Systems Innovation,
Graduate School of Engineering Science, Osaka University

Abstract

This thesis summarizes the development and testing of ROS-lite, a parallel computing and software development framework for embedded computing platforms based on network-on-chip (NoC) technology. Such heterogeneous computing platforms are used because embedded systems often require powerful high processing capacity and low power consumption. Since they implement nonuniform memory access (NUMA) for scalability, software for many-core platforms must be designed based on scalable data allocation and scalable parallelization. This thesis reports the results of data transfer tests on NoC implementations, microbenchmarks, and parallelization of self-driving software for implementation on an embedded many-core processor. ROS-lite, the proposed framework provides a structured communications layer for NoC components that reduces memory consumption and allows efficient software development for embedded many-core platforms. Using NUMA and the parallelization scalability of many cores, the data transfer latencies that are currently achievable between distributed memory resources are investigated. To test the practicality of embedded many-core platforms in a practical application, a module of a self-driving software suite is parallelized so that it can run on many-core processors. This pilot test achieves performance levels that are acceptable for market-ready autonomous vehicles. By highlighting many-core computing capabilities and efficient software development, I explore scalable parallel computing on NoC-based embedded many-core platforms.

Contents

Chapter 1

Introduction

The evolution of next-generation computing platforms with multi-/many-core platforms is required to satisfy the increasing demands for computational power with reasonable power consumption. This trend also applies to embedded systems such as those used in autonomous vehicles. Since autonomous driving requires fast processing, predictability, and energy efficiency for several computational modules, heterogeneous computing systems such as multi-/many-core platforms and graphical processing units (GPU), are often employed. Multi-/many-core architectures can realize high-performance and general-purpose computing with low power consumption for embedded systems [1], [2]. Recent studies have examined several applications for multi/many-core platforms [1], [3], [4], [2], [5], [6].

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 [7] [8], Tile-Gx [9] [10] and Tile-64 developed by Tilera [11], Xeon Phi developed by Intel [12], [13], and the single-chip cloud computer (SCC) developed by Intel [14]. Several of these platforms such as MPPA-256 and Tile-64 are intended for embedded systems, and studies focused on multi-/many-core platforms have gained increased attention [15], [16], [17]. Available many-core platforms are compared in Chapter 5.

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 in which the cores are allocated closely. For instance, 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 in total. Although these cores do not guarantee cache coherency, 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 separately to achieve the desired power envelope for embedded applications.

Despite these recent developments in embedded many-core platforms, several challenges persist for adapting these platforms in embedded applications [1], [3]. In NoC-based embedded many-core platforms, studies on NoC-based many-core platforms did not completely reveal the details of data transfer between distributed memory banks with NoC technology, memory-access characteristics for NUMA, and the strategies that developers

can use for parallelization in practical applications. In addition, reusing existing software on these platforms is difficult because application developers have to write platform-specific codes for NoC data transfer between clusters. Writing source code for robotics systems is difficult, in particular, when the scale and scope of embedded software applications continue to grow.

To meet these challenges, the practicality of NoC-based many-core platforms is evaluated, and a software development framework, such as the robot operating system (ROS) [18], [19] which allows reusing existing applications and efficiently developing on heterogeneous computing platforms, is proposed. The developed framework is called ROS-lite. ROS, an open-source robot operating system, is a structured communications layer above the host operating systems of a heterogeneous computing cluster. ROS is widely used in the robotics community [20], as detailed in Appendix A: ROS. Scalable parallel computing and data allocation are tested via quantitative evaluations and an applied test. The MPPA-256 serves as a reference platform. MPPA-256 [8] adopts NUMA using NoC technology and realizes numerous cores with low power consumption, and it is tested using the evaluation code, ROS-lite, and in a practical application.

Contributions: This thesis evaluates the usefulness of embedded many-core computing platforms based on NoC technology, such as MPPA-256. The data transfer methods are evaluated using microbenchmarks with matrix calculation to clarify the latencies involved in data transfer on NoC components, parallel computing, and the influence of data allocation. Next, the localization algorithm that is the core of a self-driving system is parallelized. Finally, a novel light-weight ROS architecture, called ROS-lite, is proposed herein. The advantages and disadvantages of embedded many-core computing platforms are quantified based on NoC technology with the following main innovations:

- On DMA-capable NoC components, data-transfer tests quantitatively measure the end-to-end latencies and the memory access speed.
- The scalability of parallelization is measured via the evaluation of complex matrix calculations and a real implementation of one module of a self-driving system.
- With limited memory available, the proposed software framework provides a structured communications layer and efficient development for heterogeneous computing platforms.

To the best of our knowledge, this is the first study that quantitatively evaluates data transfer and data allocation matters for many-core computing platforms. Our quantitative methods offer application developers more insight than their current reliance on intuition when designing strategies for data allocation. The demonstration of computation speed improvements for a self-driving application indicates the practical potential for NoC-based embedded many-core computing. ROS-lite provides an efficient development framework wherein ROS nodes can run on many-core platforms and communicate with each other.

Organization: The remainder of this thesis is organized as follows. Chapter 2 discusses the reference hardware model, the Kalray MPPA-256 Bostan, and the system model. The proposed ROS-lite software framework is presented in Chapter 3, which discusses the development flow, the design, and implementations. Chapter 4 explains the evaluation tests and methods and illustrates the experimental evaluation. Chapter 5 reviews the

related work that focuses on multi-/many-core systems. Chapter 6 presents conclusions and directions for future work.

Chapter 2

System Model

This chapter presents the system model used throughout the work. The many-core model of Kalray MPPA-256 Bostan is considered. First, a hardware model is introduced in Section 2.1, followed by a software model in Section 2.2.

2.1 Hardware Model

The MPPA-256 processor is based on an array of computing clusters (CCs) and I/O subsystems (IOSs) that are connected to NoC nodes with a toroidal two-dimensional topology, as shown in Figs. 2.1, 2.2 and 2.3. The MPPA many-core chip integrates 16 CCs and 4 IOSs on NoC. The architecture of Kalray MPPA-256 is presented in this section.

2.1.1 I/O Subsystems (IOS)

MPPA-256 contains the following four IOSs: North, South, East, and West. The North and South IOSs are connected to a DDR interface and an eight-lane PCIe controller. The East and West IOSs are connected to a quad 10 Gb/s Ethernet controller. Two pairs of IOSs organize two I/O clusters (IOCs), as shown in Fig. 2.1. Each IOS consists of quad IO cores and a NoC interface.

- **IO Cores**: IO cores are connected to a 16-bank parallel shared memory with a total capacity (IO SMEM) of 2 MB, as shown in Fig. 2.1. The four IO cores have their own instruction cache 8-way associative corresponding to 32 (8×4) KB and share a data cache 8-way with 128 KB and external DDR access. The sharing of the data cache of 128 KB allows coherency between the IO cores. Additionally, IO cores operate controllers for PCIe, Ethernet, and other I/O devices. They operate the local peripherals, including NoC interfaces with DMA. It is also possible to conduct an application run on the IO cores.
- **NoC Interface**: The NoC interface contains four DMA engines (DMA1-4) and four NoC routers, as shown in Fig. 2.1, and the IOS DMA engine manages transfers between the IO SMEM, IOS DDR, and IOS peripherals (e.g., PCIe interface and Ethernet controllers). The DMA engine transfers data between routers via NoC through NoC

routers and has the following three NoC interfaces: a receive (Rx) interface, a transmit (Tx) interface, and a microcore (UC). The UC is a fine-grained multithreaded engine that can be programmed to set threads sending data with a Tx interface. The UC can extract data from memory by using a programmed pattern and send the data on the NoC. After the UC is initiated, this continues in an autonomous fashion without using a processing element (PE) and an IO core.

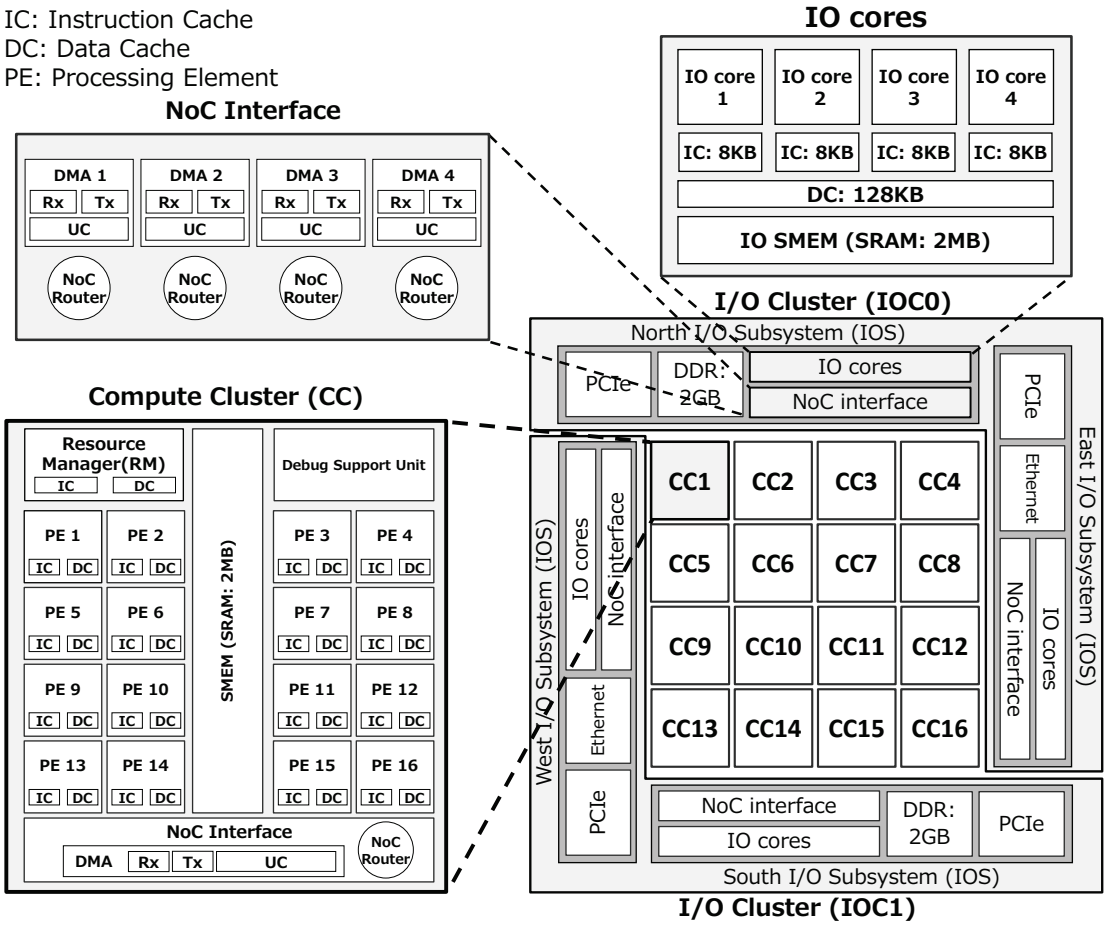

Fig. 2.1 An overview of the architecture of the Kalray MPPA-256 Bostan.

2.1.2 Compute Clusters (CCs)

In MPPA-256, the 16 inner nodes of the NoC correspond to the CCs. Fig. 2.2 illustrates the architecture of each CC.

- **PEs and an RM**: In a CC, 16 PEs and an RM share 2 MB cluster local memory (SMEM), so that 17 k1-cores (a PE or the RM) share 2 MB SMEM. Users use the PEs primarily for parallel processing. Developers spawn computing threads on PEs. The PEs and an RM in the CC correspond to the Kalray-1 cores, which implement a 32-bit

5-issue Very Long Instruction Word architecture with 600 or 800 MHz. Each core is fitted with its own instruction and data caches. Each cache is 2-way associative with a capacity of 8 KB. Note that these caches do not guarantee cache coherency.

- **A Debug Support Unit and a NoC Interface**: In addition to PEs and an RM, bus masters on the SMEM correspond to a Debug Support Unit and a DMA engine in a NoC interface. A DMA engine and a NoC router are laid out in a NoC interface. The CC DMA engine also has the following three interfaces: an Rx, a Tx, and a UC. The CC DMA engine is instantiated in every cluster and connected to the SMEM.

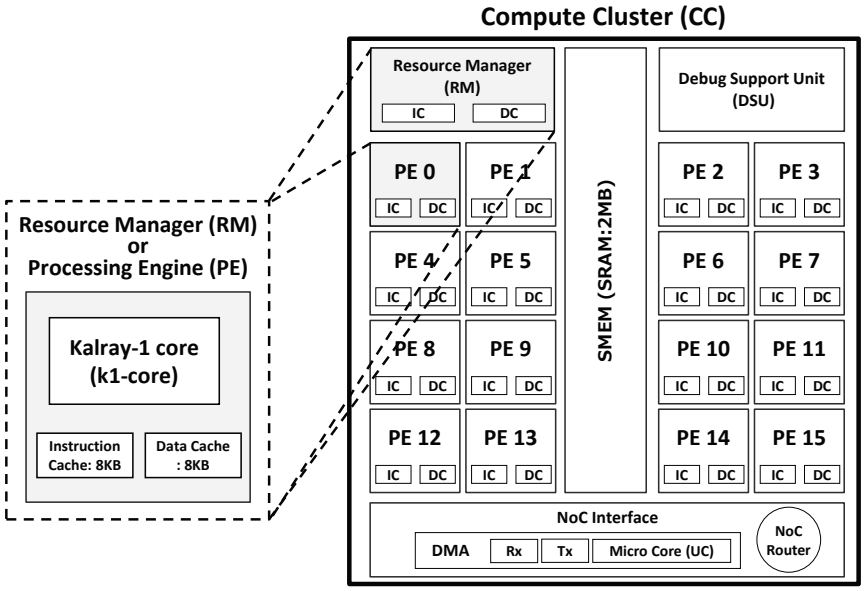

Fig. 2.2 Compute Cluster architecture.

2.1.3 Network-on-Chip (NoC)

The 16 CCs and the four IOSs are connected by NoC as shown in Fig. 2.3. Furthermore, NoC is constructed as the bus network and has routers on each node.

- **Bus Network**: A bus network connects nodes (CCs and IOSs) with torus topology [21], which involves a low average number of hops when compared to mesh topology [22], [23]. The network is actually composed of the following two parallel NoCs with bidirectional links (denoted by red lines in Fig. 2.3): the data NoC (D-NoC) that is optimized for bulk data transfers and the control NoC (C-NoC) that 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 flits (flow control digits). The NoC traffic is segmented into packets, with each packet including 1-4 header flits and 0-62 payload data flits.
- **NoC routers**: One node per CC and four nodes per I/O subsystem hold the following two routers of their own: a D-NoC router and a C-NoC router. Each RM or IO core

on a NoC node is associated with the two aforementioned NoC routers. Furthermore, DMA engines in a NoC interface on the CC/IOS send and receive flits through the D-NoC routers with the Rx interface, the Tx interface, and the UC. A mailbox component corresponds to the virtual interface for the C-NoC and enables one-to-one, N-to-one, or one-to-N low-latency synchronization. The NoC routers shown in Fig. 2.2 illustrate nodes as R1-16, R128-131, R160-163, R224-227, and R192-195 in Fig. 2.3. For purposes of simplicity, D-NoC/C-NoC routers are illustrated with a NoC router. In both D-NoC and C-NoC, each network node (a CC or an IOS) includes the following 5-link NoC routers: four duplexed links for North/East/West and South neighbors and a duplexed link for local address space attached to the NoC router. The NoC routers include FIFOs queuing flits for each direction. The data links are four bytes wide in each direction and operate at the CPU clock rates of 600 MHz or 800 MHz, with the result that each tile can transmit/receive a total of 2.4 GB/s or 3.2 GB/s, which is spread across the four directions (i.e., North, South, East, and West).

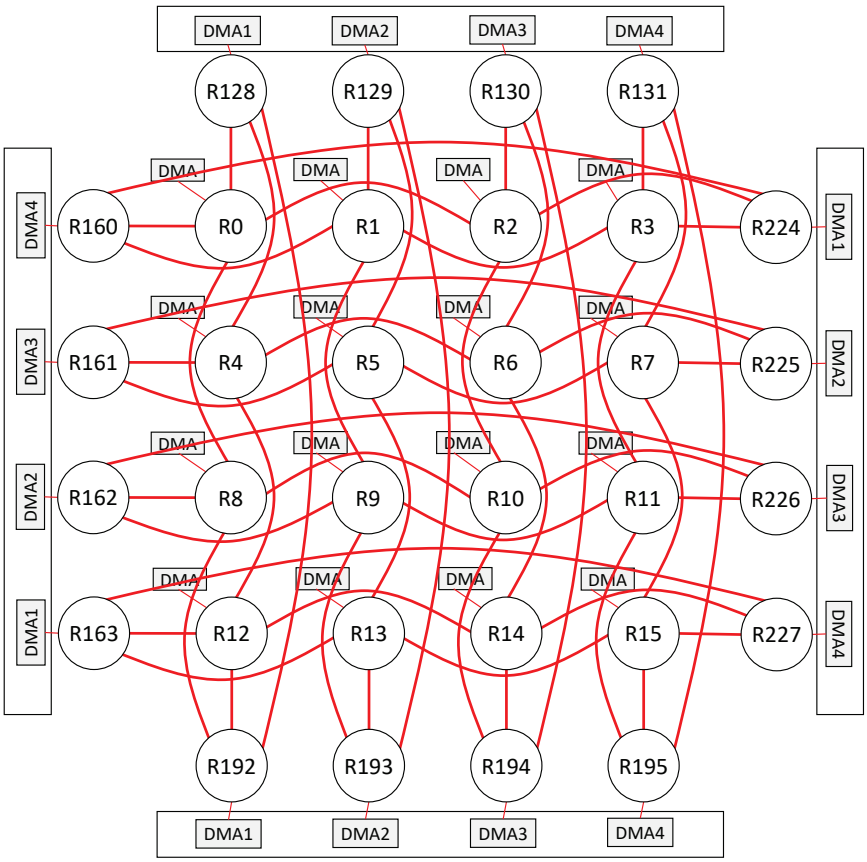

Fig. 2.3 NoC connections (both D-NoC and C-NoC).

2.2 Software Model

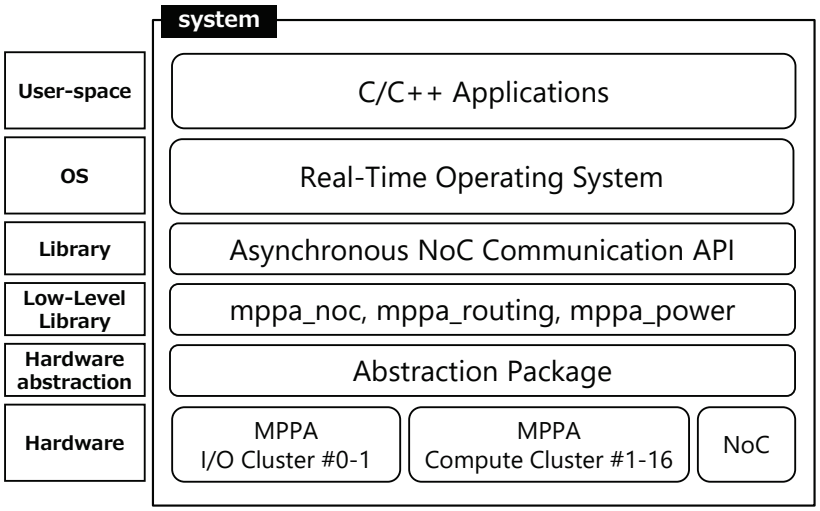

Fig. 2.4 The software stack of Kalray MPPA-256.

The software stack used for Kalray MPPA-256 is composed of a hardware abstraction layer, a library layer, an OS, and a user application. Fig. 2.4 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, it is possible to map several programming models or runtimes 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 hardware of a CC, an IOS, and NoC. The hardware abstraction is responsible for partitioning hardware resources and controlling access to the resources from the user-space operating system libraries. The abstraction sets-up and controls inter-partition communications as a virtual machine abstraction layer. The hardware abstraction runs on the dedicated RM core. All the services are provided commonly by an operating system (e.g., virtual memory and schedule) that must be provided by user-space libraries. A minimal kernel avoids wastage of resources and mismatched needs.

In a low-level library layer, the Kalray system also provides mppa_noc and mppa_routing for handling NoC. Additionally, NoC features such as routing and quality of service are set by the programmer. mppa_noc allows direct access to memory mapped registers for their configurations and uses. This library is designed to cause minimum CPU overhead and also serves as a minimal abstraction for resource allocation. mppa_routing offers a minimal set of functions that can be used to route data between any clusters of the MPPA. Routing on the network is conducted statically with its own policy. In addition, mppa_power enables spawning and waiting for the end of execution of other clusters.

2.2.1 Operating System

Several operating systems support the abstraction package in the OS layer. It is difficult for numerous cores such as MPPA to support previous operating systems for single/multicore(s) owing to problems involving parallelism and cache coherency [24], [25]. Here, the following real-time operating systems (RTOSs) supporting MPPA are introduced:

- **RTEMS**: Real-Time Executive for Multiprocessor Systems (RTEMS) is a full-featured RTOS prepared for embedded platforms. RTEMS supports several APIs and standards, and most notably supports the POSIX API. The system provides a rich set of features, and an RTEMS application is mostly a regular C or C++ program that uses the POSIX API. RTEMS runs on the IOC except for the CC.
- **NodeOS**: On the CC, the MPPA cluster operating system utilizes a runtime called NodeOS. The OS addresses the need for a multicore OS to conform to the maximum possible extent to the standard POSIX API. The NodeOS enables a user code by using the POSIX API to run on PEs on the CC. First, NodeOS runtime starts on PE0 prior to calling the user main function. Subsequently, `pthread` is called on other PEs.
- **eMCOS**: On both CCs and IOSs, eMCOS provides minimal programming interfaces and libraries. Specifically, eMCOS is a real-time embedded operating system developed by eSOL (a Japanese supplier for RTOSs) and is the first commercially available many-core RTOS for use in embedded systems. The OS implements a distributed microkernel architecture. This compact microkernel is equipped with only minimal functions. The eMCOS enables applications to operate priority based message passing, local thread scheduling, and thread management on IOSs as well as CCs.

RTMES and NodeOS are provided by Kalray and eMCOS is released by eSOL.

2.2.2 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. Sixteen cores are packed in a cluster, and each of these clusters share 2 MB memory (SMEM), as shown in Fig. 2.2. This reduces memory contention that frequently occurs with numerous cores and helps to increase the number of cores. However, the clustered architecture constrains the memory that can be directly accessed by each core. Communicating with cores outside the cluster requires transferring data between clusters through the D-NoC via NoC interfaces.

In the lowest-level layer, each cluster on MPPA-256 contains hardware for a NoC interface, which has DMA units that handle data receiving and sending: Rx, Tx, and UC interfaces.

An Rx interface is installed on the receiving side to receive data with DMA. A D-NoC Rx resource must be allocated and configured the Rx to wait for receiving the data. The DMA in each NoC interface contains 256 D-NoC Rx resources.

The Tx and UC interfaces (explained in Sections 2.1.1 and 2.1.2) manage the sending side that programs use to send data between clusters. The UC is a network processor programmed to set threads to send data in DMA. The UC executes programmed patterns and sends data through the D-NoC without a PE or an RM. The UC interface results in higher data transfer throughput compared with the direct activation of the Tx interface. However, the DMA in each NoC interface contains only eight D-NoC UC resources. Both the interfaces use a DMA engine to access memory and copy data. Regardless of whether a UC interface is used, a D-NoC Tx resource must be allocated and configured to send data. In addition, D-NoC UC resources must be allocated and configured, if a UC interface is used.

Several options are available for data transfer between clusters on D-NoC. Kalray provides mppa_noc and mppa_async libraries and eMCOS provides two type of message APIs. Table 2.1 lists the features of each method for comparison. The specifications are different for each method; therefore an appropriate method must be chosen according to the application. For an internal implementations, all methods are based on mppa_noc which is a low-level NoC communication library that provides interfaces to handle DMA resources, such as Rx, Tx, and UC interfaces for data transfer. A detailed description of each available NoC data transfer method is given below.

Table 2.1 Comparison of NoC Data Transfer Methods

	Provider	Model	Resource Management	Buffer Allocation	DMA Type	Equality of IOC and CC
mppa_noc	Kalray	send/receive	self	user-space	Tx/UC	equal
mppa_async	Kalray	read/write	auto	user-space	Tx	non-equal
eMCOS message	eSOL	send/receive	auto	kernel-space	Tx	equal
eMCOS session message	eSOL	send/receive	auto	user-space	UC	equal

- **mppa_noc**: This is the lowest-level NoC communication library for D-NoC and C-NoC components. This library handles all interfaces (Rx, Tx, and the UC). Users allocate DMA resources and configure them for NoC transport. After the data is transferred, users must free the resources. Although users must manage DMA resources, the library offers fine-grain control over interfaces, including transport routes on NoC components and send/receive buffers.
- **mppa_async**: The asynchronous communication API is used for asynchronous operations that involve on one-sided communications between the CC local memory banks and I/O cluster DDR memory. This API is designed to support applications that require remote memory access, whether the DDR memory or the SMEM of other CCs. Users must create segments that correspond in whole or in part to the other CC's local memory banks. These segments are managed by servers run on the IOC and CC. Users initiate an asynchronous write or read process between the local memory and a designated memory segment. Two addresses are supplied, one local and one remote from the memory segment.
- **eMCOS message**: eMCOS provides message APIs for communication between threads via Tx interfaces in the D-NoC. The destination of a message is a thread,

and sending/receiving processes use asynchronous/synchronous modes. Users can also configure receiving behavior with masks and priority-based filtering. Under the message 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. Message buffering on each core is managed by eMCOS, and the data sent by DMA through D-NoC are copied to the buffer in eMOCS. After resolving the reception order, data are provided by copying non-cache access from the message buffer to user-space addresses.

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

Chapter 3

Proposed Framework

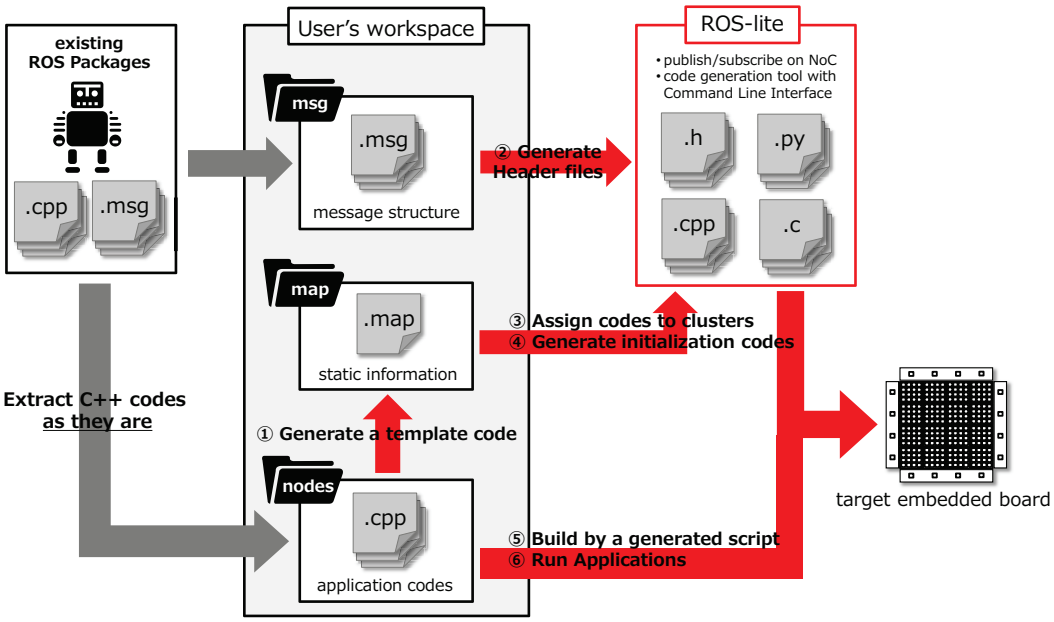

Fig. 3.1 Development flow of ROS-lite framework.

The proposed framework, ROS-lite, is a structured communications layer that enables an 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 in which ROS nodes run on each core and communicate with each other. By adapting and extending numerous existing ROS applications, ROS-lite makes porting existing software and new development on embedded platforms much more efficient. For more information about ROS in general, refer to Appendix A. As it is based on the existing ROS framework, ROS-lite can communicate with external ROS nodes on another platform. The tests reported herein used eMCOS as an RTOS and MPPA-256 as an

embedded many-core platform.

3.1 ROS-lite Toolchain

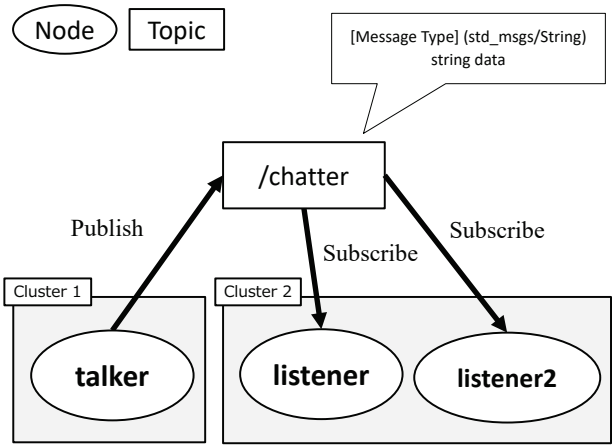

Fig. 3.2 The publish/subscribe model in ROS-lite.

```
1  - name: talker
2    cluster: 1
3    publish: [/chatter]
4    subscribe: []
5  - name: listener
6    cluster : 2
7    publish: []
8    subscribe: [/chatter]
9  - name: listener2
10   cluster : 2
11   publish: []
12   subscribe: [/chatter]
```

Fig. 3.3 Map file Description (`.map`).

This section describes ROS-lite in detail. The development flow for ROS-lite is shown in Fig. 3.1. ROS-lite can run application code written as ROS nodes. I assume that the applications are written in C++, similar to ROS. Application developers can adapt source code (application code and message structure files) from existing ROS packages without changing it. For task mapping, developers edit a map file (`.map`) and assign ROS nodes to the CCs on many-core platforms. Developers are required to only 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 message files (`.msg`). ROS nodes can communicate with each other via either intra-cluster or inter-cluster routes. These cores handle intra-cluster communications via shared memory, whereas the inter-cluster are handled via NoC.

ROS-lite toolchain provides a code generation and a build system with a command-line interface to allow an efficient development. First, as in ① of Fig. 3.1, a template of a map file (`.map`) is generated from application source code. The map file (`.map`) contains node names, the assigned cluster, and the information of topics that are published and subscribed to. Fig. 3.3 can refer for an example of the map file specifications. These descriptions, except for the assigned cluster number, can be able to be interpreted from ROS nodes to provide code generation for template code. Application developers can then focus on the number of assigned clusters. Second, as in ② of Fig. 3.1, header files that defined the message structure are generated from message files (`.msg`), as in the original ROS. Note that 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 message files (`.msg`) can be described as they are in in ROS. The message files (`.msg`) do not require any modifications. Third, in ③ and ④ of Fig.

3.1, initialization code for launching processes of ROS nodes are 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, in ⑤ of Fig. 3.1, 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 build script. The build script does not require any modifications 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 ROS-lite framework is provided herein. One node publishes a *chatter* topic and two nodes subscribe to the topic, as shown in Fig. 3.2. 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. 3.3. Application developers can change the node mapping by modifying of the cluster number field. Information about 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. These fields, for node name and topic information, are generated from source codes of the ROS nodes, and then initialization script for the node relations in ROS-lite are generated from the map file (`.map`). Note that the field for the assigned cluster numbers must be filled by application developers.

3.2 Design and Implementation

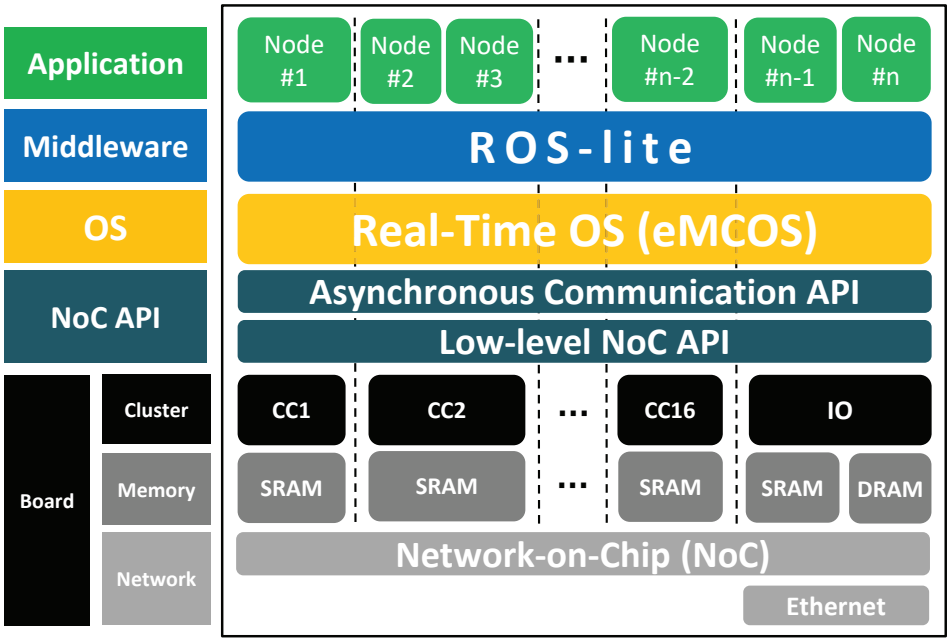

Fig. 3.4 System stack of ROS-lite on a many-core platform.

I tested ROS-lite with an implementation on MPPA-256 as shown in Fig. 3.4. MPPA-256 is an example of NoC-based clustered many-core processors with distributed memory architecture. ROS-lite can run on this platform, if memory is limited, as it would be in an embedded application. ROS-lite allows NoC-based message transfer with low memory consumption. Note that ROS-lite is a general framework, and other many-core processors can be used with it.

In our test, eMCOS is used as the RTOS for ROS-lite because eMCOS has rich messaging functions, as described in Section 2.2.2, and can run well on CCs and IOCs. All threads under ROS-lite are managed by eMCOS. The light-weight threads so 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. Internal thread management is thereby improved by eMCOS.

ROS-lite does not have a master server like ROS 's `ros master`, which helps the original framework to avoid single failures. Nodes in ROS-lite use map files to statically initialize topic information that specifies each node that subscribes to each topic outside its own cluster. Initialization scripts are generated off-line by the command-line interfaces. I adopted this off-line initialization process since many specifications about execution time

will be fixed ahead of time for most embedded systems. By eliminating the master server, ROS-lite avoids unnecessary communication between clusters and a single failure point.

I next discuss the internal design of publish/subscribe transport for ROS-lite. A subscriber's main thread creates a `message receive function` for each topic, as shown in Fig. 3.5. Subscriber nodes handle message buffering and behavior with one process in ROS-lite. The design in Fig. 3.5 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 configured dynamically. ROS-lite can use mppa_async and session messaging to address this problem. ROS-lite adopts mppa_async because the session message s constrained by the session number by DMA resources. Session messaging is not a suitable scenario that requires creating many sessions. Fig. 3.6 shows the implementation design of publish/subscribe transport in ROS-lite. Before data transport, the publisher nodes serialize data following the message structure. ROS-lite's serialization policy is based on the original ROS. After serialization, the publisher node sends messages to the subscriber nodes listed in topic information, which are initialized by the map file off-line. 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 system. Segments of mppa_async for transport between clusters are created and destroyed with each transport. Although this transaction increases overhead, it allows ROS-lite to flexibly handle variable-length messages, such as vectors.

To prevent the simultaneous execution of callback functions and design ensure that each node only runs as one process, it is necessary to use the design shown in Fig. 3.7, not the design of Fig. 3.5. A callback queue manages the order of callback functions to be executed at the subscriber node. This queue can be implemented simply 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. Fig. 3.8 shows an example of ways in which the design in Fig. 3.7 can be implemented.

ROS-lite supports original ROS source code to enhance porting and development efficiency for applications running on many-core processors. ROS-lite provides the same interfaces as the ROS API for application developers who are not familiar with NoC interfaces. Figs. 3.9, 3.10, and 3.11 provide source code for the scenario in Fig. 3.2.

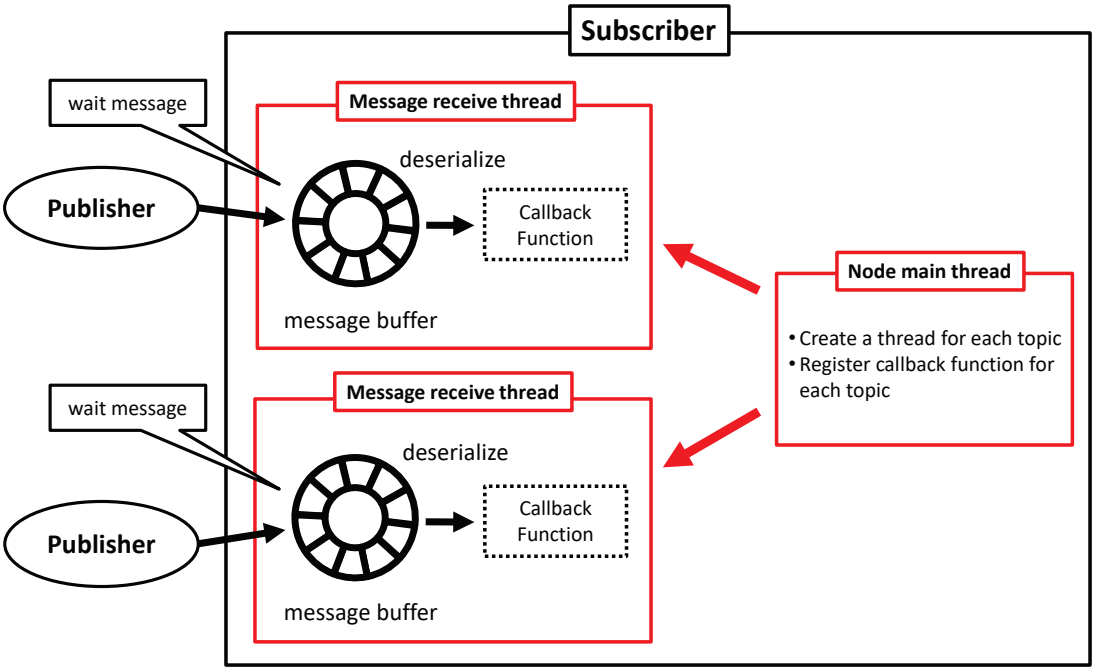

Fig. 3.5 Structure of subscriber node.

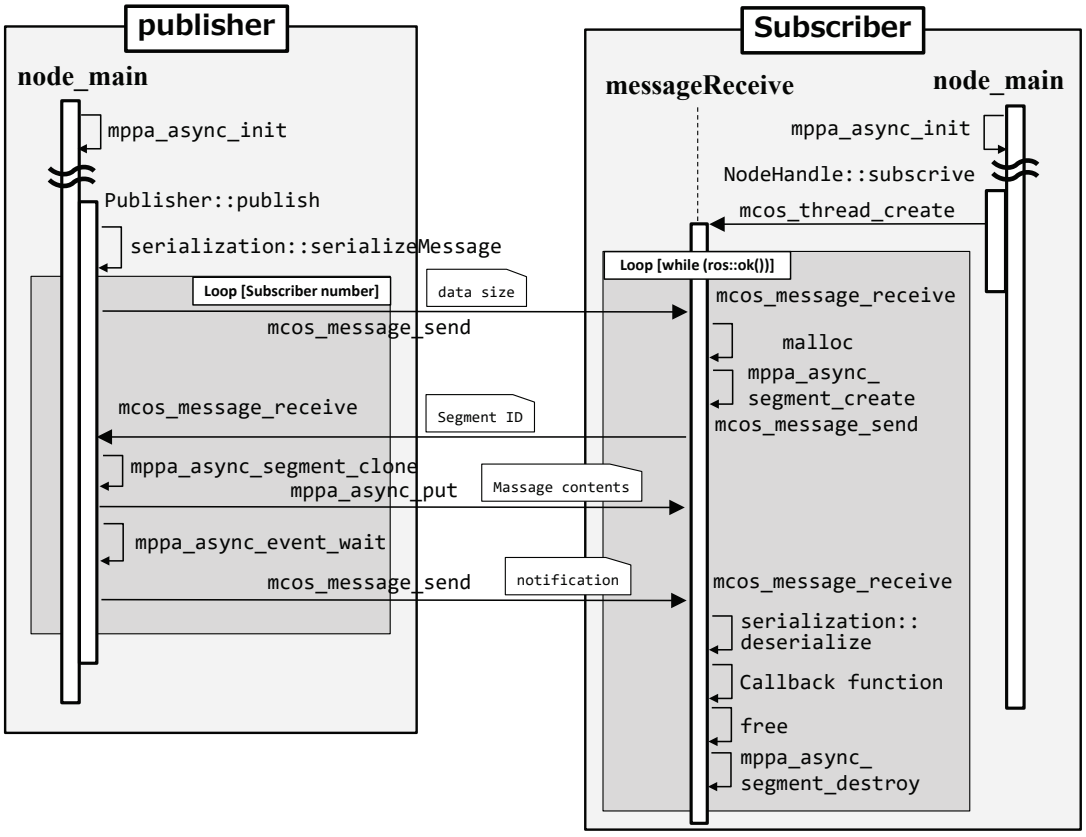

Fig. 3.6 Design of publish/subscribe sequence of Fig. 3.5 without message buffering.

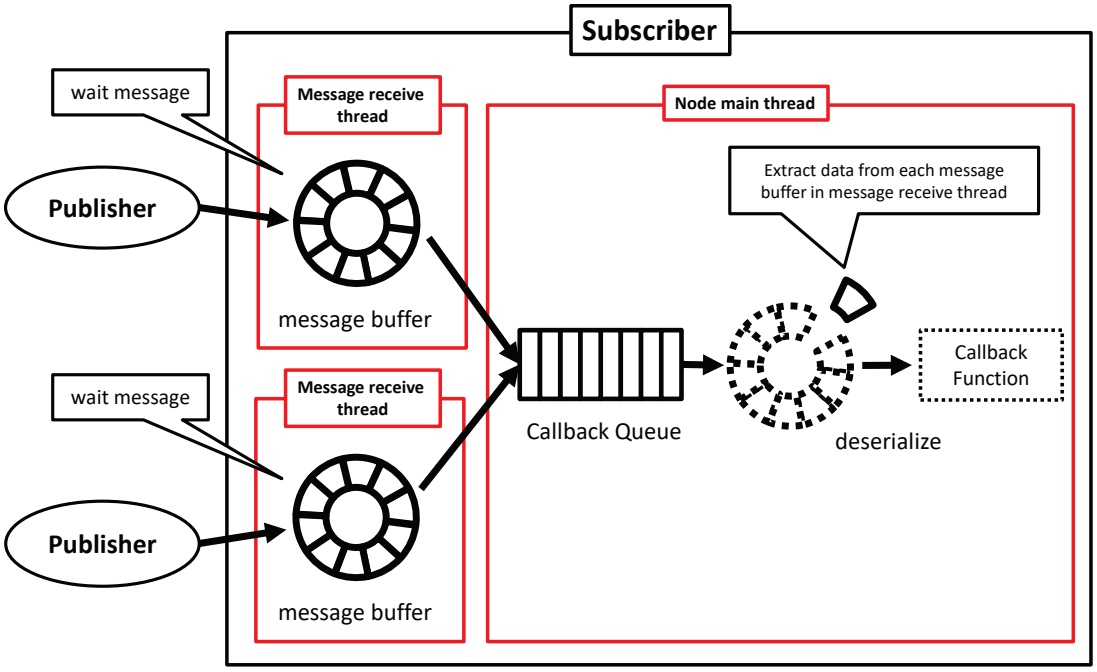

Fig. 3.7 Structure of subscriber node with callback queue.

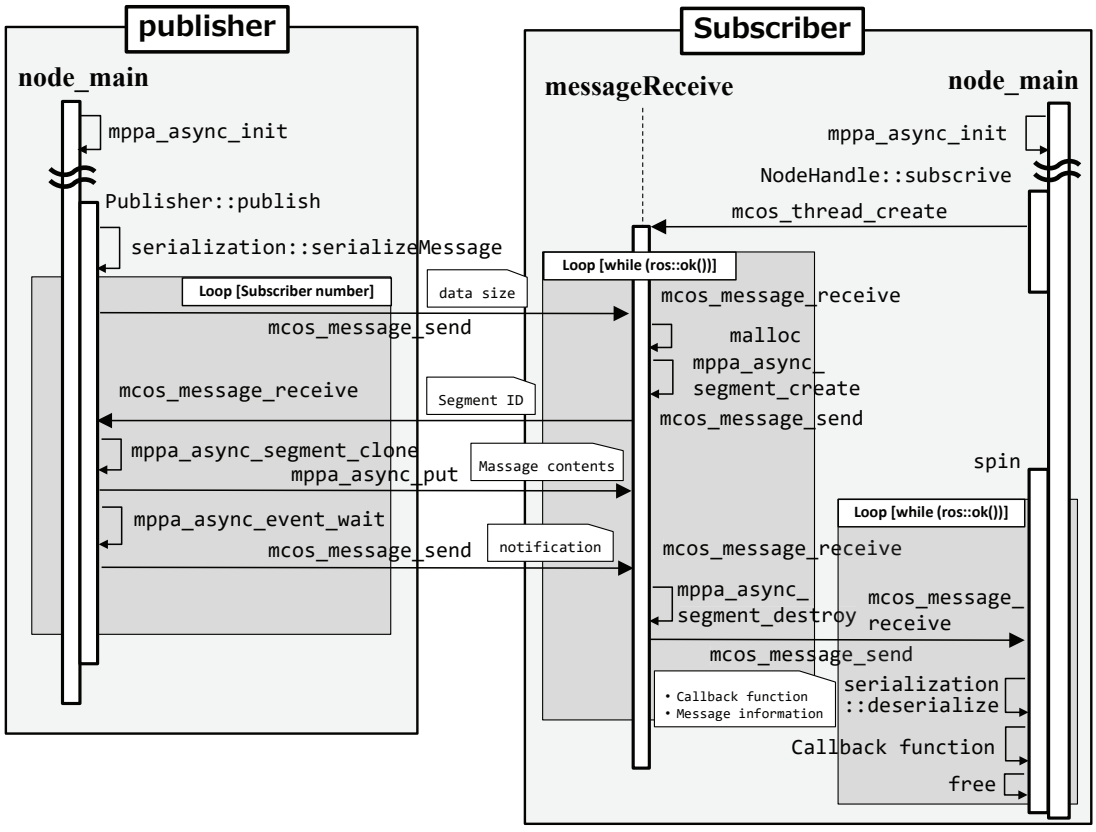

Fig. 3.8 Design of publish/subscribe sequence of Fig. 3.7.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");

  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("/chatter", 1);

  ros::Rate loop_rate(1);  // Set 1 Hz

  while (ros::ok()) {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world";
    mcos_debug_printf("%s\n", ss.str().c_str());
    msg.data = ss.str();

    chatter_pub.publish(msg);
    loop_rate.sleep();
  }

  return 0;
}
```

Fig. 3.9 An example of a C++ source code (`.cpp`): a publisher node.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

void chatterCallback(const std_msgs::String::ConstPtr& msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");

  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("/chatter", 1, chatterCallback);

  ros::spin();
  return 0;
}
```

Fig. 3.10 An example of a C++ source code (`.cpp`): a subscribe node.

```
string data
```

Fig. 3.11 An example of a message file (`.msg`): string message type.

Chapter 4

Evaluations

First, this chapter involves examining two types of evaluations: a D-NoC data transfer evaluation in which latency characteristics of interfaces and memory type are explored and a matrix calculation evaluation that demonstrates the parallelization potential of the MPPA-256 and its memory access characteristics. Subsequently, I conduct a practical self-driving application to examine the practicality of NUMA many cores. The following evaluations are all conducted on real hardware boards with eMCOS.

4.1 D-NoC Data Transfer

4.1.1 Situations and Assumptions

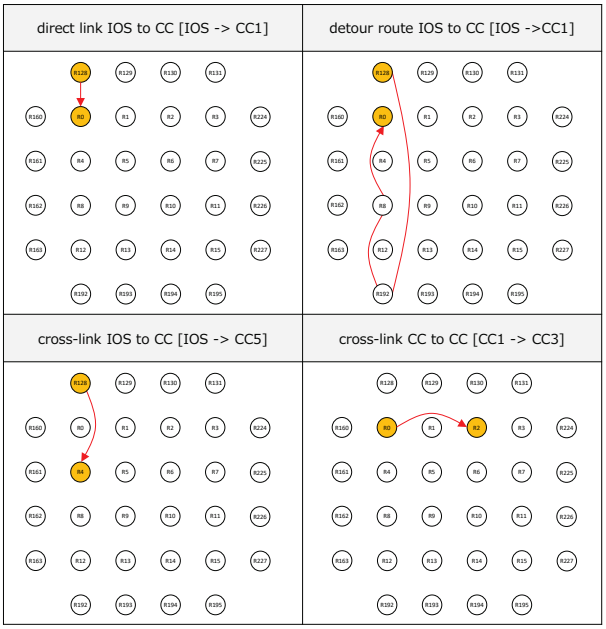

Fig. 4.1 Four D-NoC routes used in the evaluation.

This evaluation involves clarifying end-to-end latency by considering the relation among interfaces (Tx or UC), routing on NoC, and memory type (DDR or SMEM). This is achieved by preparing four routes as shown in Fig. 4.1. The routes on the D-NoC map (Fig. 2.3) contain various connections between routers, a direct link, a cross-link, and a flying link. With respect to the case of routes from the IOS routers to the CC routers, transmitted data are allocated in the DDR or IO SMEM. The CC includes only the SMEM as shown in Fig. 2.2. The transferred data correspond to 100 B, 1 KB, 10 KB, 100 KB, and 1 MB. The buffers are sequentially allocated in DDR or SRAM (IO SMEM or CC SMEM). The capacity of the CC SMEM is 2 MB, and thus it is assumed that the appropriate communication buffer size is 1 MB. Given this assumption, the other memory area corresponds to the application, libraries, and an operating system. End-to-end latencies are measured 1,000 times in numerous situations as shown in Figs. 4.2, 4.3, and 4.4, and boxplots are obtained, as depicted in Figs. 4.5, 4.6, and 4.7. In the evaluation setting, I minimize traffic conflicts to focus on the relation between end-to-end latency and routing on NoC.

4.1.2 Influences of Routing and Memory Type

Data transfer latencies between an IOS and a CC are influenced little by routing. This involves preparing two interfaces (Tx and UC), three routes (direct link, cross-link, and detour route), and two memory locations in which the transferred data are allocated. As shown in Figs. 4.2, 4.3, 4.4, and 4.5, end-to-end latency scales exhibit a linear relation with data size, and there are no significant differences between the three routes with respect to data transfer latency. This result is important in a torus topology NoC because the number of minimum steps exceeds that in a mesh topology. It is observed that queuing in NoC routers and hardware distance on a NoC are not dominant factors for latency. The router latency, the time taken in transmitting and receiving transactions in an RM, exceeds those of other transactions. Additionally, it is briefly recognized that the speed of the UC exceeds that of the Tx. The data are arranged as shown in Figs. 4.6 and 4.7 to facilitate a precise analysis with respect to the interface and memory location. In those figures, only the cross-link from the IOS to CC5 is accepted because routes do not influence latency. To facilitate intuitive recognition, two kinds of figures are arranged: a logarithmic and a linear axis.

In the Tx interface, DDR causes a large increase in latency. The time taken by the DDR is twice that of the IO SMEM as shown in Fig. 4.6. This is due to the memory access speed characteristics of DRAM and SRAM. In the Tx interface, it is necessary for an IO core on an IOS to operate the DMA in the IOS NoC interface. This is attributed to the fact that the core is involved in processing. The speed of the data transfer latency between CCs exceeds that between an IOS and a CC. This result indicates that the MPPA-256 is optimized for communication between the CCs.

With respect to the UC interface, the latency is not significantly affected by the location at which the transferred buffer is allocated (i.e., DDR or SMEM). Similar latency characteristics are observed in Fig. 4.6. In the case of the UC interface, an IO core on the IOS does not involve a DMA transaction. A UC in the NoC interface executes a

programmed thread sending data. This evaluation result suggests that the slow access speed of the DDR is not significant in the case of the UC. In a manner similar to that of the Tx interface, the speed of the data transfer latency between CCs exceeds that between an IOS and a CC.

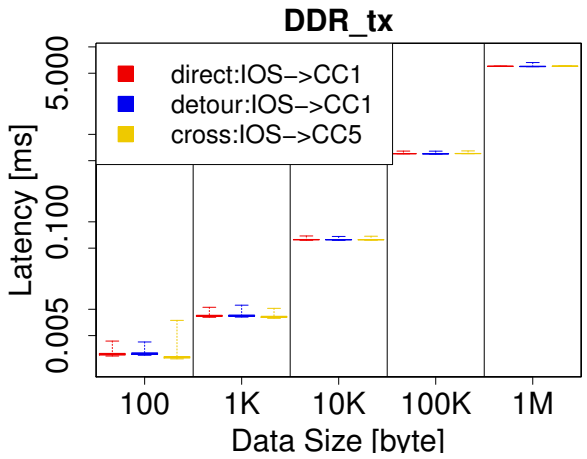

Fig. 4.2 Data transfer with Tx from IO DDR to CC.

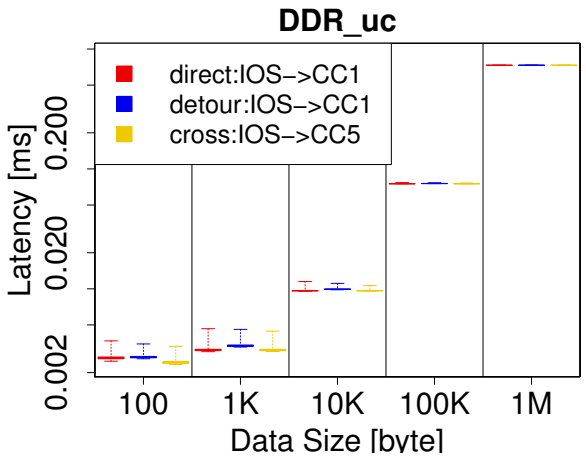

Fig. 4.3 Data transfer with UC from IO DDR to CC.

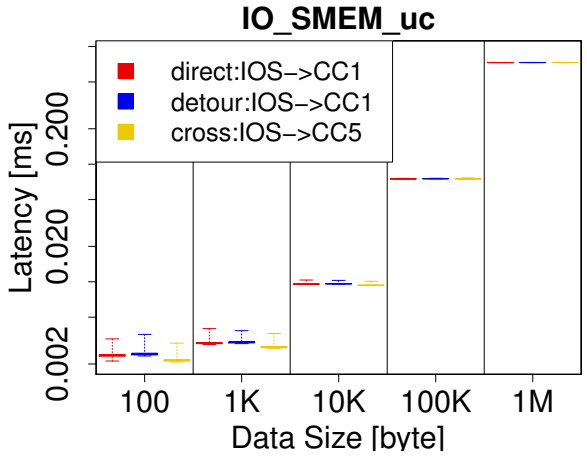

Fig. 4.4 Data transfer with UC from IO SMEM to CC.

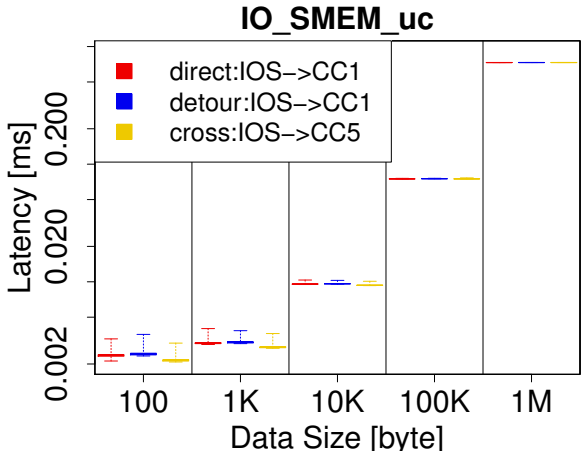

Fig. 4.5 Data transfer with UC from IO SMEM to CC.

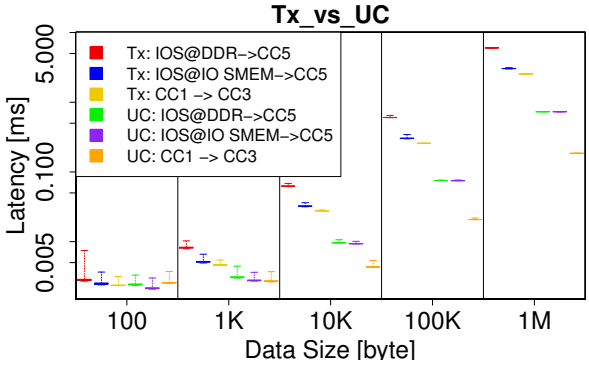

Fig. 4.6 Data transfer with Tx/UC (logarithmic axis).

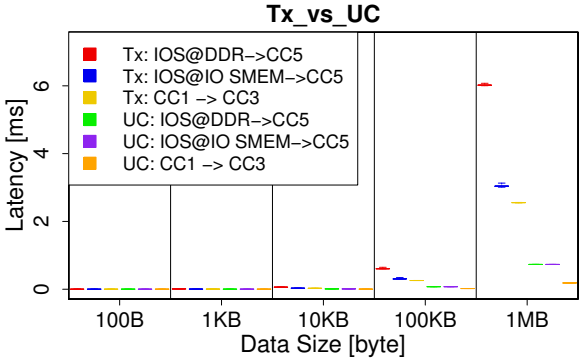

Fig. 4.7 Data transfer with Tx/UC (linear axis).

4.2 Matrix Calculation

4.2.1 Situations and Assumptions

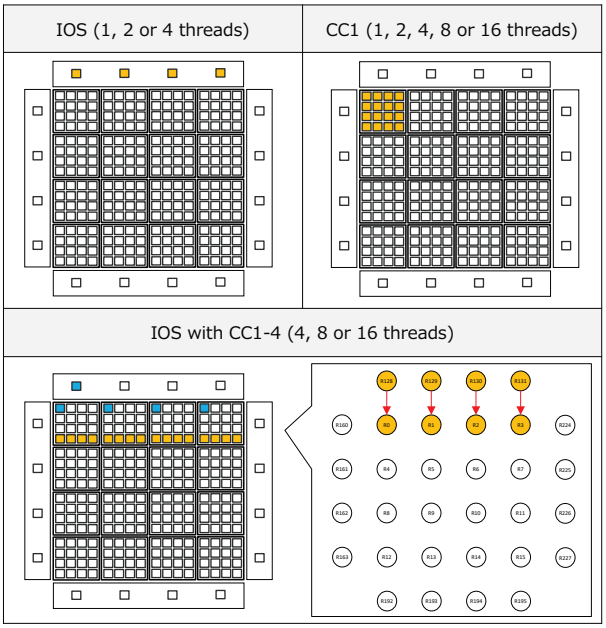

Fig. 4.8 Matrix calculation situations.

In the evaluation, the matrix calculation time and parallelization potential of MPPA-256 are clarified. Matrix calculations are conducted in an IOS and CCs. Three computing situations are considered as shown in Fig. 4.8. The first situation involves computing in the IOS where four cores are available. To analyze memory access characteristics, a matrix buffer is allocated in the IO DDR and SMEM. The second situation involves computing in a CC in which 16 cores are available. The third situation involves offload-computing using an IOS and four CCs. Parallelized processing is executed with four CCs and SMEMs. A few cores in the IOS and CC manage the parallelized transaction. The method can handle large amount of data in which one cluster is not sufficient, because buffer capacity is not limited to 2 MB in the SMEM. Parallelized processing and the total capacity of the SMEM are superior to single IOS or CC computations. With respect to the IOS, the application can handle large capacity data only in the DDR. However, in this method, distributed memories are used to deal with large capacity data in the SMEM. Thus, it is necessary for IOS and CC cores to access matrix buffers without cache to avoid cache coherency difficulties. To facilitate faster data transfer, a portion of the matrix buffer is transmitted in parallel as shown in Fig. 4.8.

Matrix calculation time is analyzed with parallelization and memory allocation. Additionally, the influences of a cache are analyzed because cache coherency is an important

issue in many-core systems. There are several cases in which applications must access specific memory space without a cache because MPPA does not guarantee cache coherency between PEs. With respect to the given assumptions, the maximum total buffer size is 1 MB, and thus three matrix buffers are prepared, each of size 314 KB. Matrix A and Matrix B are multiplied, and the result is stored in Matrix C. The total for the three matrices is set as approximately 1 MB. I assume that the remainder of the SMEM (1 MB) is occupied with system software and applications in the CC.

4.2.2 Influences of Cache and Memory Type

First, matrix calculation time with the cache in the IOS and CC is depicted in Fig. 4.9. There are almost no differences between the IO DDR, IO SMEM, and CC SMEM due to the cache. A 128 KB data cache in the IOS works well and compensates for the DDR delay. Additionally, it is observed that calculation time scales exhibit a linear relation with the number of threads. This corresponds to ideal behavior with respect to parallelization.

Second, matrix calculation time without a cache in the IOS and CC is shown in Fig. 4.10. The absence of a cache, results in a fourfold increase in the DDR and a large difference arises with respect to the SMEM. Another notable result is that calculation speed in the CC SMEM exceeds that of the IO SMEM. This characteristic is hidden in the calculation with the cache. The computing cores physically involve the same cores in the IOS and CC, and thus it is considered that the characteristics and physical arrangement of the SMEM exert a significant effect. This is an interesting result since there is a large difference that cannot be ignored. It is also observed that calculation time exhibits a linear relation with the number of threads. Furthermore, in the CC SMEM, the calculation speed without the cache exceeds that with cache. This result is contrary to intuition, and a cache line problem is conceivable. When a small data cache (8 KB) in a PE of the CC does not function adequately and an application always misses the cache, memory access will pay the time for a noncached data access and the cost to refill the cache line. As a result, the memory access speed without the cache exceeds that with it.

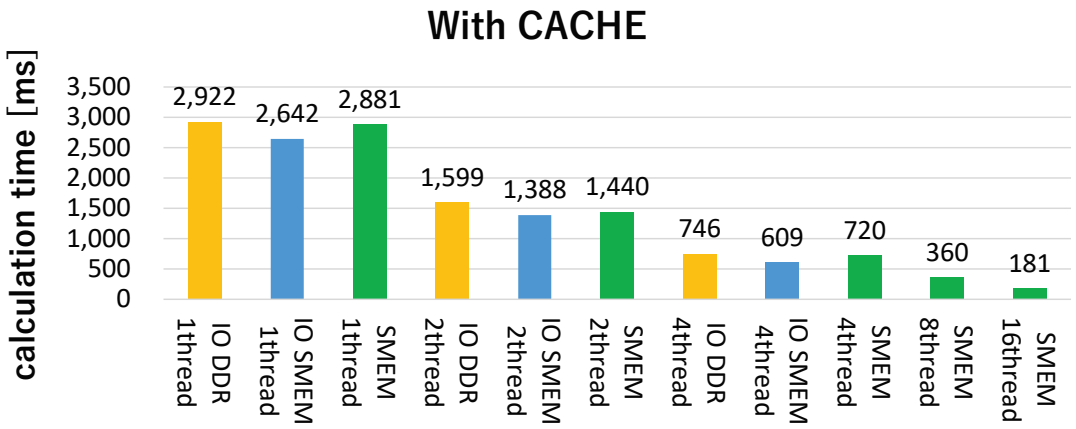

Fig. 4.9 Matrix calculations in IOS and CC with cache.

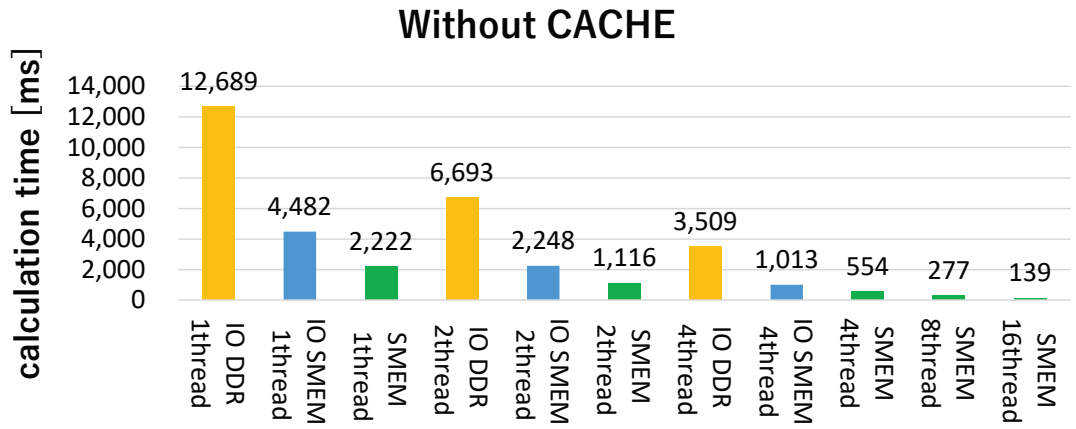

Fig. 4.10 Matrix calculations in IOS and CC without cache.

4.2.3 Four CCs' Parallelization

Finally, matrix calculation with offload-computing in an IOS and CCs is shown in Figs. 4.11 and 4.12. In this case, it is assumed with respect to the calculation of large matrices that the total capacity exceeds 1 MB. The offloading result is compared with the IO DDR (cached) owing to the aforementioned assumption. The aggregate calculation is obtained by offloading on the four CCs to perform a multiplication of a tile of Mat A and a tile of the transpose of Mat B. This produces an overhead irrespective of the number of threads as shown in Figs. 4.11 and 4.12.

However, the speed involved in offloading the result exceeds that of the IO DDR (cached). The result indicates several important facts. First, D-NoC data transfer produces little overhead latency. Second, the speed of DMA memory access to DDR exceeds

that of the IO core's memory access, even if target memory is allocated on the DDR. In the offloading case, a DMA accesses matrix buffers on the DDR and transfers the buffers from the IO DDR to each CC SMEM. Subsequently, PEs in the CC access matrix buffer the calculation without a cache. The overhead of data transfer and DMA memory access is small, and thus parallel data transmission and distributed memory are practical in the case of MPPA-256. The impact of offloading increases when the matrix is large as shown in Fig. 4.12. Only a portion of the matrix is allocated in CCs, and thus it is possible to handle larger matrix buffers.

Additionally, 640 KB matrices are prepared, and matrix calculation is evaluated with offload-computing. The speed of the offloading result exceeds that of the IO DDR result with respect to the 314 KB matrices in Fig. 4.11. In these offloading evaluations, each CC concurrently transmits calculation results to the IOS. When Matrix C in which calculation results are stored is allocated in the DDR, the NoC router's FIFOs sometimes overflow and cause an error due to memory access delay of the DDR. The transmission protocol would ideally be expected to prevent this error, and flow control is intended as future work for MPPA-256. Currently, to avoid this error, Matrix C must be allocated in the IO SMEM. Note that the above evaluation results when Matrix C is allocated in the DDR.

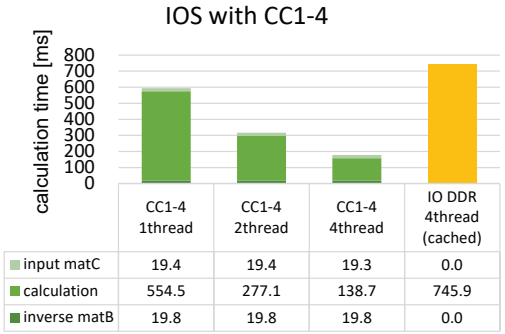

Fig. 4.11 Matrix calculations with offload computing (314 KB matrix x 3).

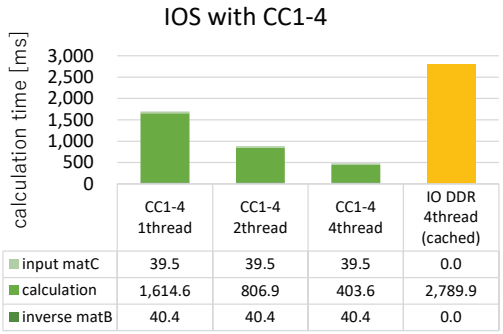

Fig. 4.12 Matrix calculations with offload computing (640 KB matrix x 3).

4.3 Practical Application

This work adopts a portion of a self-driving system and this section demonstrates the parallelization potential of the MPPA-256. I selected an algorithm for vehicle self-localization written in C++ in Autoware, open-source software for urban self-driving [26], and a parallelized part of it. (Please refer to Appendix B: Autoware.) The self-localization adopts the normal-distribution transform matching algorithm [27] implemented in the Point Cloud Library [28]. A diagram depicting self-localization is shown in Fig. 4.13.

The self-localization algorithm is composed primarily of the *computeTransform* function which searches for several nearest neighbor points for each scan query and calculates a matching transformation. This evaluation parallelized a part of *computeTransform* onto 16 CCs and the remainder of the algorithm was executed in parallel on the IOS 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. Redesigning this algorithm is reserved for future work, and there is room for improvement through the parallelization potential of the MPPA-256.

As shown in Fig. 4.14, the evaluation of the parallelized self-localization algorithm indicates the average execution time for each convergence and demonstrates that the parallelization accelerates the *computeTransform* process. The query can be assumed to be 10 Hz in many automated driving systems. Thus, this tuning successfully meets the deadline. 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. A demonstration video of the adaptation of the parallelized self-localization algorithm to Autoware on eMCOS can be seen at: https://youtu.be/wZyqF90c5b8

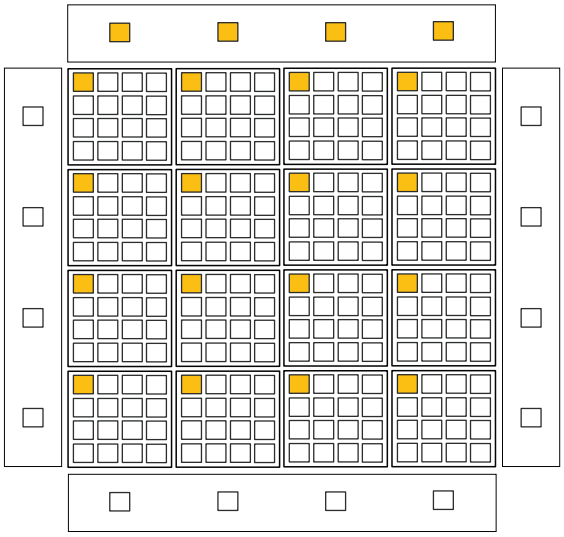

Fig. 4.13 A situation of vehicle self-localization execution.

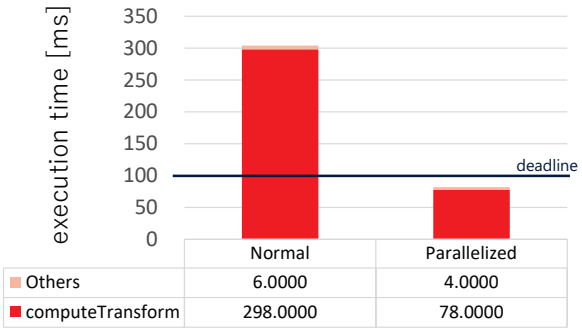

Fig. 4.14 Vehicle self-localization of the partially parallelized self-driving application.

4.4 Lessons Learned

Thus far, in Chapter 4, I have quantitatively clarified the characteristic of data transfer and parallel computing on NoC-based embedded many-core platforms. I can obtain insight and guidelines for users and developers of NoC-based embedded many-core platforms through MPPA-256.

From evaluations of D-NoC data transfer, you can learn two lessons: the influences of NoC routing and DMA. First, data transfer latencies between clusters are hardly influenced by routing for users as shown in Figs. 4.2, 4.3, 4.4, and 4.5. Software transactions of transmitting and receiving in routers and RM are dominant factors for latencies. This is understandable owing to the minimized traffic conflicts of the evaluation setting. However,

when the influence of routing comes to intensive concurrent traffic over a large portion of the network, different routes might still have nontrivial impacts on the end-to-end data transfer delay, especially for a detour route of a longer length. Second, in the IOS, the latency of a UC is not significantly affected by the memory location, the DDR or SMEM, at which the transferred buffer is allocated from the evaluation of Fig. 4.6. The merit of the UC is profitable for users because this is a common situation for transferring data from the DDR to the CC SMEM using the UC, especially for large amount of data.

From microbenchmarks of matrix calculation, you can learn three lessons of memory access: characteristics of the SMEM, influences of a cache, and data flow control on D-NoC. First, the memory access speed of the CC SMEM exceeds that of the IO SMEM as shown in Fig. 4.10. There is a significant difference, which is a major reason to work actively in the CC. Second, I indicate quantitatively that the calculation speed without the cache in the CC SMEM in Fig. 4.10 exceeds that with the cache in Fig. 4.9. It is generally known that there is cache overhead when a miss hit occurs frequently, but it is notable that there is a substantial difference that cannot be ignored by the influences of a cache line. Third, when data are transferred in parallel from multiple CCs to the IO DDR, NoC routers' FIFO sometimes overflows owing to memory access delays of the DDR. This would ideally be expected to be prevented by the transmission protocol and flow control.

From the practical application viewpoint, since I parallelize the vehicle self-localization algorithm of the self-driving system, NoC-based embedded many-core systems can be used practically in real environments. I applied for parallel data transfer from an IOS to CCs and parallel computing on the IOS and CCs by using the CC SMEM as scratch pad memory. I also conducted demonstration experiments with real environments and confirmed the practicality of NoC-based embedded many-core systems.

Chapter 5

Related Work

This chapter compares many-core platforms and discusses previous work related to multi-/many-core platforms. First, comparison of many-core platforms to other platforms is discussed. Second, the Kalray MPPA-256 which this work focuses on is compared to other COTS multi-/many-core components, and I summarize the features of MPPA-256. Finally, discussions of previous work and comparisons with them are described.

Table 5.1 summarizes the features of many-core platforms with those of other platforms. For instance, the GPU is a powerful device to enhance computing performance and has great potential in specific areas (for e.g., image processing, and learning). However, the GPU is mainly used for a specific purpose and its predictability is not suitable for real-time systems. It is difficult to use a GPU for many kinds of applications and to guarantee its reliability due to the GPU architecture. Many-core processors based on CPU are significantly superior to GPU with respect to software programmability and timing predictability. Additionally, it is commonly known that many-core platforms such as MPPA-256 involve a reasonable power consumption [15]. In contrast, the GPU consumes a significant amount of power and generates considerable heat. This is a critical problem for embedded systems. FPGAs are also high-performance devices when compared to CPUs. They are efficient in terms of power consumption. FPGAs guarantee predictability and efficient processing. However, FPGAs are difficult for software developers and are not a substitute for CPU since their software model is significantly different from that of CPU. Many-core platforms can potentially replace single/multi core CPU as they possess ease of programming and scalability.

Based on the aforementioned background, many COTS multi-/many-core components are developed and released by several vendors. (e.g., MPPA-256 by Kalray, [8], Tile-Gx by Tilera [9], [10], Tile64 by Tilera [11], and Xeon Phi by Intel [12], [13], Single-chip Cloud Computer (SCC) by Intel [14]). The present work focuses on the Kalray MPPA-256, which is designed for embedded applications. Kalray [8] presented clustered many-core architectures on the NoC that pack 256 general-purpose cores with high energy efficiency.

MPPA-256 is superior to other COTS multi-/many-core components in terms of the scalability of the number of cores and the power efficiency, as shown in Table 5.2. In terms of scalability, MPPA-256 uses 256 cores, whereas other COTS multi-/many-core components have 64 cores or the number of cores around it. This scalability of cores is

attributed to the NUMA memory architecture; each cluster of 16 cores contains its own local shared memory. The precise hardware model is described in Section 2.1. When all cores share the global DDR memory as in other platforms excluding MPPA-256, specific bus/network routes receive extremely large loads and memory access contention frequently occurs. Local shared memory reduces the above problems and helps the scalability of the number of cores. However, the NUMA memory architecture restricts the capacity of the memory and requires a data copy from the DDR with NoC. This restriction makes the use of existing applications difficult especially in the case of applications that require more memory. As a result, owing to the NUMA memory architecture, the portability of code porting to MPPA-256 is inferior to that of other COTS platforms, as shown in Table 5.2.

In terms of power efficiency, MPPA-256 realizes superior energy efficiency despite its large number of cores [15]. The total clock frequency per watt is the highest of the current COTS multi-/many-core components. The power consumption of the MPPA processor ranges between 16 W at 600 MHz and 24 W at 800 MHz. You must distinguish the COTS multi-/many-core components according to their requirements with reference to Table 5.2. MPPA-256 is typically accepted with respect to many-core platforms and the model has been used in previous work [1], [29], [2], [17].

Previous work has examined real-time applications on many-core platforms, including MPPA-256. Multiple opportunities and challenges of multi-/many-core platforms are discussed in [3]. The shift to multi-/many-core platforms in real-time and embedded systems is also described.

Based on the above background, several task mapping algorithms for multi/many-core systems have been proposed [29], [30], [2]. Airbus [2] proposes a method of directed acyclic graph (DAG) scheduling for hard real-time applications using MPPA-256. In [30], a mapping framework is proposed on the basis of AUTOSAR which is applied as a standard architecture to develop automotive embedded software systems [31]. AUTOSAR task scheduling considering contention in shared resources is presented in [1].

By examining the above mapping algorithms of real-time applications, previous work [32], [16], [15], [17] has analyzed the potential of MPPA-256 and data transfer with NoC, as shown in Table 5.3. MPPA-256 is introduced and its performance and energy consumption are reported in [15]. However, this report contains few evaluations and does not refer to data transfer with NoC and memory access characteristics. Data transfer with NoC in MPPA-256 is described, and NoC guaranteed services are analyzed in [32] and [16]. While the theoretical analysis is thorough in these works, the practical evaluations are poor and parallel data transfer is not referred to. The authors of Ref. [17] focused on the predictable composition of memory access. An analysis of their work identified the external DDR and the NoC as the principal bottlenecks for both the average performance and the predictability on platforms such as MPPA-256. Although the analysis examined the memory access characteristics of the external DDR and provided notable lessons, a solution for the DDR bottleneck was not examined and practical evaluations were lacking.

Table 5.1 Comparison of Many-core Platform to CPU, GPU, and FPGA

	performance	power/heat	real-time	software	costs development	multiple instruction
CPU		L	✓	✓	✓	L
GPU	✓			L	✓	
FPGA	✓	L	L		L	
Many-core Platform	✓	✓	✓	✓	L	✓

∗In a table, "L" means "limited".

Table 5.2 Comparison of Many-core Platforms

	scalability	power efficiency	code transplant
Kalray MPPA-256 [8]	✓	✓	L
Tilera Tile series [11]		L	✓
Intel Xeon Phi [12] [13]			✓
Intel SCC [14]			✓

Table 5.3 Comparison of Previous Work

	performance analysis	data transfer analysis with NoC	memory access characteristics	real applications	parallel data transfer
Kalray clusters calculate quickly [15]	L				
Network-on-Chip Service Guarantees [16]		✓			
Predictable composition of memory accesses [17]		✓	✓		
this thesis	✓	✓	✓	✓	✓

Chapter 6

Conclusions

This thesis conducted quantitative evaluations of data transfer methods on NoC technology, microbenchmarks with matrix calculation, and a practical application with NoC-based embedded many-core platforms such as MPPA-256. Additionally, I proposed software development framework called ROS-lite for the structured communications layer on NoC components efficient software development. Evaluations indicate latency characteristics on NoC platforms, influences of data allocation, and the scalability of parallelization. Our experimental results will allow system designers to select appropriate system designs. Parallelization of a real application proved the practicality of NoC-based embedded many-core platforms. Last, the ROS-lite runs with few memory consumption and realizes that ROS nodes run on each core on many-core platforms and communicate with each other.

In future work, I will use benchmark applications such as that in [33] for further analysis. In addition, considering above the results, I will extend ROS-lite for real-time systems such as that in [34], and propose the parallelization of memory intensive algorithms, such as the nearest neighbor search in Section 4.3.

Appendix A: Robot Operating System (ROS)

As mentioned earlier, Autoware is based on ROS [18], [19], 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. Since 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 individual component modules, whereas the *topics* mediate inputs and outputs between *nodes*, as illustrated in Figure A.1. ROS *nodes* are usually standard programs written in C++. They can interface with any other software libraries that are installed.

Communication among *nodes* follows a publish/subscribe model. This model is a strong approach for modular development. In this model, *nodes* communicate by passing *messages* via a *topic*. A *message* is structured simply (almost identical to structs in C) and is stored in .msg files. *Nodes* identify the content of the *message* in terms of 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 Figure A.1, 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*. *Topics* are managed using first-in, first-out queues when accessed by multiple *nodes* simultaneously. At the same time, ROS *nodes* can launch several threads implicitly; however, issues with real-time processing must be addressed.

ROS also includes an integrated visualization tool, RViz, and a data-driven simulation tool, ROSBAG. Figure B.3 (a) in Appendix B shows examples of visualizations produced by RViz for perception tasks in Autoware. The RViz viewer is useful for monitoring the status of tasks. ROSBAG is a set of tools for recording and playing back ROS *topics*. It provides development environments for testing self-driving algorithms without hardware, thereby making development processes more efficient.

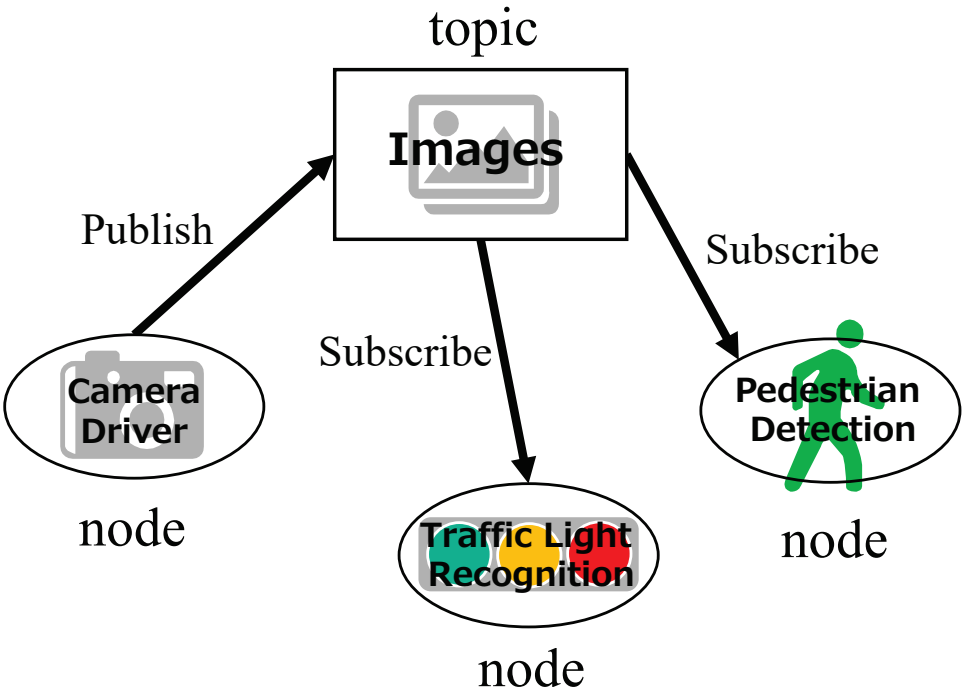

Fig. A.1 The publish/subscribe model in ROS.

Appendix B: Autoware

B.1 System Model

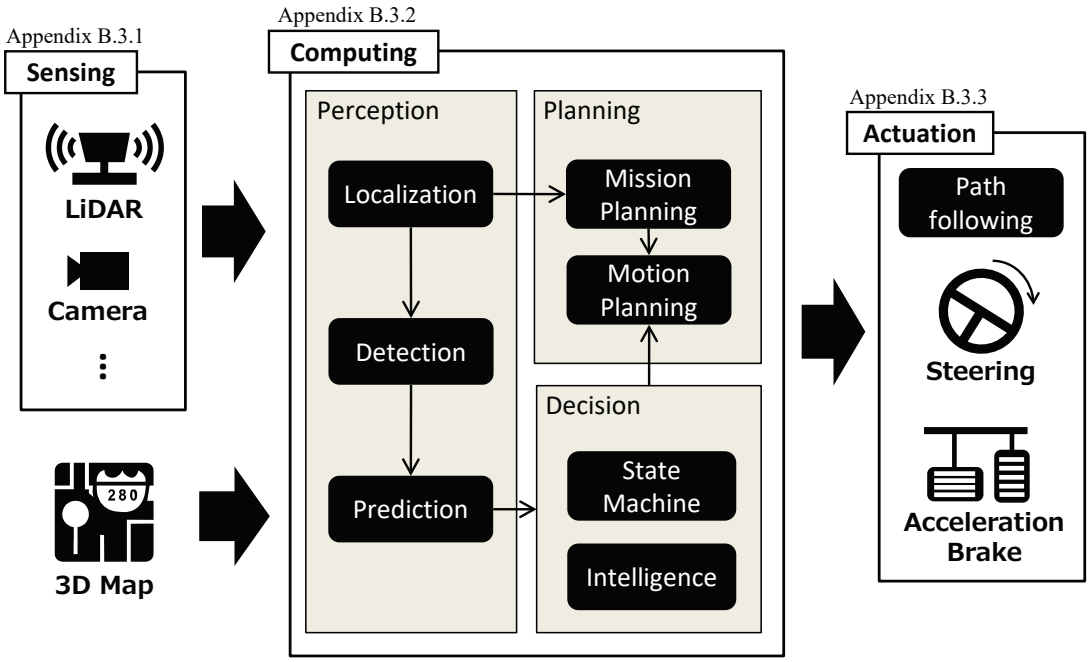

Fig. B.1 Basic control and data flow of autonomous vehicles.

Autonomous vehicles as cyber-physical systems can be abstracted into sensing, computing, and actuation modules, as shown in Figure B.1. Sensing devices, such as laser scanners (LiDAR) and cameras, are typically used for self-driving in urban areas. Actuation modules handle steering and stroking whose twisted control commands that are typically generated by the path following module. Computation is a major component of self-driving technology. Scene recognition, for instance, requires the localization, detection, and prediction modules, whereas path planning is handled by mission-based and motion-based modules. Each module employs its own set of algorithms. The modules implemented in Autoware are discussed in Section B.3.

Figure B.1 shows the basic control and data flow for an autonomous vehicle. Sensors record environmental information that serves as input data for the artificial intelligence core. 3D maps are becoming a commonplace for self-driving systems, particularly, in urban

areas as a complement to the planning data available from sensors. External data sources can improve the accuracy of localization and detection without increasing the complexity of the vehicle's algorithms. Artificial intelligence cores typically output values for angular and linear velocities, which serve as commands for steering and stroking, respectively.

B.2 System Stack

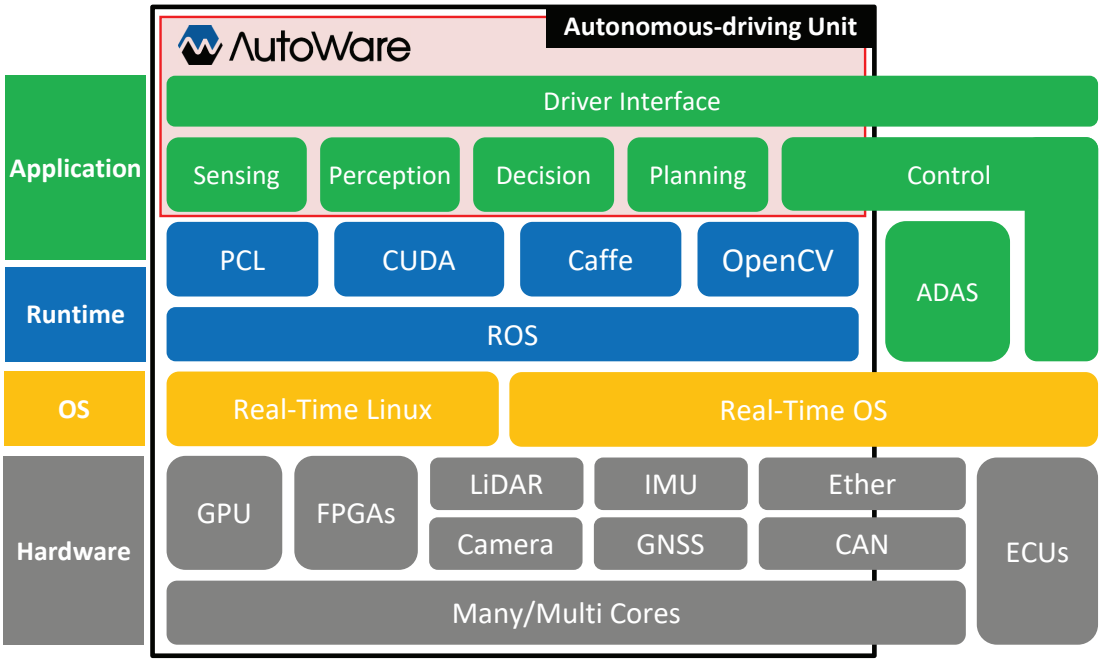

Fig. B.2 Complete system stack of autonomous vehicles using Autoware.

Autonomous is designed for a complete software stack for autonomous vehicles that are implemented with open-source software, as shown in Figure B.2. The self-driving technology discussed herein is intended for vehicles that are driven in urban areas instead in freeways or highways. I contributed to the development of Autoware, a popular open-source software project developed for autonomous vehicles intended for using in urban areas. Autoware is based on Robot Operating System (ROS) and other well-established open-source software libraries, as shown in Figure B.2. Appendix A provides a brief overview of ROS. Point Cloud Library (PCL) [28] is mainly used to manage LiDAR scans and 3D mapping data, in addition to performing data-filtering and visualization functions. CUDA [35] is a programming framework developed by NVIDIA and is used for general-purpose computing on GPUs (GPGPU). To handle the computation-intensive tasks involved in self-driving, GPUs running CUDA are promising solutions for self-driving technology, though this thesis does not focus on them. Caffe [36], [37] is a deep learning framework designed with expression, speed, and modularity in mind. OpenCV [38] is a popular computer vision library for image processing.

Autoware uses such tools to compile a rich set of software packages, including sensing, perception, decision making, planning, and control modules. Many drive-by-wire vehicles can be transformed into autonomous vehicles with an installation of Autoware. In several countries, successful implementations of Autoware have been tested for autonomous vehicles that are driven for long distances in urban areas. Recently, automotive manufacturers and suppliers have begun implementing Autoware as a baseline for prototype autonomous vehicles.

This section briefly introduces the hardware components that are generally installed in autonomous vehicles. Commercially available vehicles require minor modifications for a successful application of our software stack. This will enable us to control the vehicles using external computers through a secure gateway and install sensors on the vehicles. The vehicle, computers, and sensors can be connected by Controller Area Network (CAN) bus, Ethernet, and/or USB 3.0. However, the specifications of sensors and computers required for a particular application depend closely on specific functional requirements of the autonomous vehicle. Autoware supports range sensor, and I chose Velodyne HDL-32e LiDAR scanners and PointGrey Grasshopper3 cameras for our prototype system. Autoware also supports a range of processors. Many users run Autoware on desktops and laptops. The NVIDIA DRIVE PX2 is used herein though our previous study ported Autoware to another embedded computing platform, the Kalray Massively Parallel Processor Array (MPPA) [39].

B.3 Algorithms

This section briefly discusses Autoware modules marked in Figure B.1 emphasizing on the modules that are developed and extended for using in DRIVE PX2. The remaining models of Autoware can be studied on its project repository [26]. Note that Autoware is designed for autonomous vehicles to be used in urban areas, so additional modules may be required for vehicles driven on freeways or highways. Discussions of the accuracy and optimization for each algorithm are beyond the scope of this study.

B.3.1 Sensing

Our system recognizes road environments using LiDAR scanners, cameras, radars, and GPS/IMU data. LiDAR scanners and cameras are used as primary sensors. LiDAR scanners measure the distance by illuminating a target with pulsed lasers and measuring the timing of reflected pulses. Point-cloud data from LiDAR units can be used to create digital 3D representations of the environment. Cameras are often used to recognize traffic lights and objects and are sometimes superior to LiDAR scanners in recognizing object features and providing better sampling rates.

Raw point-cloud data requires filtering and pre-processing. Autoware applies the Normal Distributions Transform (NDT) algorithm [40] for 3D point-cloud pre-processing. The data can be reduced to approximate points in the lattice by replacing the original points with voxels in a computational lattice [27]. This filter is applied before localization,

detection, and mapping.

Radars and GPS/IMU data can be used to refine the localization, detection, and mapping processes. Radars are already available in commercial vehicles and are often used for ADAS safety applications. GPS/IMU data can be coupled with gyroscope sensors and odometers to refine the positioning information.

B.3.2 Computing

The computing center is a major component of self-driving vehicles. Taking sensor data and 3D maps as inputs, it computes the final trajectory as the output to actuation modules. This section explains the computing center in detail from the viewpoints of perception, decision-making, and planning.

Perception

The perceptual module must estimate the position of the ego-vehicle in the 3D map and recognize objects in the surrounding scene including moving objects and traffic signals.

Localization: The localization submodule has a great effect on the reliability of the autonomous system, as it senses the vehicle's position within the environment. Autoware performs localization by scan matching between 3D maps and LiDAR scanners, which allows precision on the order of a few centimeters for position and rotation. Autoware uses the NDT [40] algorithm for localization. The computational cost of the NDT algorithm is not affected by the map size, thereby enabling a real-time application of high-definition and high-resolution 3D data. To be precise, the 3D version of NDT is used for scan matching over filtered 3D point-cloud data and 3D map data with PCL, as shown in Figure B.3 (b) [27]. As a result, Autoware can localize the position of ego-vehicles that are located within a few centimeters.

Localization is also a key technique for 3D mapping. If autonomous vehicles are localized precisely in real-time, 3D maps can be generated and updated continuously by uploading 3D point-cloud data acquired from every 3D LiDAR scan. This approach is often referred to as Simultaneous Localization And Mapping (SLAM).

Although Autoware also supports another well-known algorithm called iterative closest point, the NDT algorithm is used for both localization and mapping herein. Autoware allows users to choose the best algorithm for their system.

Detection: Autoware must next detect surrounding objects, such as vehicles, pedestrians, and traffic signals, to avoid accidents or violations of traffic rules. Autoware recognizes traffic signals and lights; however, this section primarily focus on moving objects such as vehicles and pedestrians. Autoware supports deep learning [41], [42] and pattern recognition [43] for object detection using libraries such as Caffe and OpenCV. As shown in Figure B.3 (c), Autoware primarily use SSD [41] and the You only look once (Yolo2) algorithms [42], which are based on deep learning. They are unified frameworks for object detection that use a single neural network and allow real-time object detection. Autoware also includes a pattern recognition algorithm based on Deformable Part Models (DPM) [43], which searches and scores the histogram of oriented gradients features of target objects in 2D images [44].

In addition to 2D image processing, our prototype also uses point-cloud data scanned from a 3D LiDAR scanner to detect objects using Euclidean clustering. Point cloud clustering allows Autoware to measure the distance between objects and the vehicle. This distance information can be used to range and track objects that are classified by 2D image processing algorithms.

Assuming that localization is precise and a 3D map has been constructed for the area wherein the autonomous vehicle is driven, I improve upon Autoware's accuracy at recognizing traffic signals and traffic lights. Autoware determines the exact road area in the image by projecting the 3Dmap onto an image originating at the current position. The region of interest (ROI) for image processing can be constrained to this area, thereby reducing execution time and the occurrence of false positives. To obtain ROIs, I calibrated the 3D LiDAR data to the 2D camera in advance, as this calibration cannot be performed by the prototype while operating. Our ROI approach can recognize traffic signals from 2D images, as shown in Figures B.3 (f) and (g). In general, traffic light recognition is the most difficult problem associated with autonomous vehicles. Autoware can accurately recognize traffic lights using 3D mapping data and precise localization.

Prediction: Because the object-detection algorithm processes each frame of the image and point-cloud data, the results must be associated with other frames in the time series to predict the trajectories of moving objects for mission and motion planning.

Kalman Filter or Particle Filter can be used to solve this inverse problem. Kalman Filter is used if the assumption that the autonomous vehicle is driving at constant velocity while tracking moving objects holds true [45]. The computational cost of this filter is lightweight, thereby making it suitable for real-time processing. Particle Filter, in contrast, can be applied to nonlinear tracking scenarios, which are appropriate for realistic driving [46]. Our platform uses both Kalman and Particle Filter, depending on the given scenario. They are used for tracking in both the 2D image plane and the 3D point-cloud domain.

The detected and tracked objects in the 2D image can be combined with clustered and tracked objects obtained from the 3D LiDAR sensor; this process is referred to as sensor fusion with projection and re-projection. The sensor fusion parameters are determined in the lab while calibrating the LiDAR scanner to the 2D camera.

Autoware support scene recognition with sensor fusion of the camera and 3D LiDAR sensor. You can then project the 3D point-cloud information obtained by the 3D LiDAR sensor onto 2D images, thereby adding depth information to the image and filtering for ROIs. Figures B.3 (d) and (e) show the results of projection and adding bounding boxes to clustered 3D point-cloud objects projected onto the image.

The detected and tracked objects on the image can be re-projected onto the 3D point-cloud coordinates using the same extrinsic parameters. You also use the reprojected object positions to determine the motion plan and, partially, the overall trajectory.

Decision

After Autoware recognizes relevant obstacles and traffic signals in the environment, it can plot the trajectories of other moving objects and make decisions about mission and motion planning. The prediction and comprehensive decision-making modules are under development in our self-driving platform.

Autoware adopt a state machine and machine learning intelligence for understanding, forecasting, and decision-making in response to road scenarios. Our platform enables drivers to supervise the state of the vehicle while making comprehensive decisions for the planning modules to execute.

Planning
This module plans trajectories following the decision-making module's output. Autoware break path planning into mission and motion planning. The platform plans a global trajectory based on the current location and specified destination and conducts basic local motion planning to determine the overall trajectory. Available graph-search algorithms include hybrid-state A* [47], and trajectory generation algorithms include [48] such as lattice-based algorithms [49]. A motion planner suited to the decision-making module must be chosen, depending on the road scenario.

Mission planning: Drawing upon traffic laws, the our mission planner uses a rule-based mechanism to autonomously assign a path trajectory for purposes such as lane changes, merges, and passing. This mechanism is not completely autonomous in our platform. The high-definition 3D map contains static road information used for navigation and global planning. In more complex scenarios, such as parking and recovering from operational mistakes, the driver can supervise the path. In either case, once the path is assigned, the local motion planning module is launched.

The basic goal of the mission planner is to drive in a cruising lane over the route provided by a commercially available navigation application based on the high-definition 3D map. The vehicle changes lanes only when it passes a preceding vehicle or approaches an intersection followed by a turn.

Motion planning: The motion planner adjusts self-driving in response to driving behavior, which is not common across users and environments. Hence, the prototype platform currently provides only a basic motion-planning strategy to allow us to build a high-level intelligence for making comprehensive decisions. Constrained by the current and goal states of the vehicle, a feasible trajectory must be generated dynamically from the travelable area based on the 3D map and with consideration of surrounding objects and traffic rules.

In unstructured environments, such as parking lots, Autoware can use graph-search algorithms, such as A* [50] and hybrid-state A* [47], to find the minimum-cost path to the goal using a cost map, as shown in Figures B.3 (h) and (i). Although graph-search algorithms consume a great deal of processing power, they can analyze complex scenarios. In contrast, in structured environments, such as roads and traffic lanes, vertices are likely to be dense and and irregularly distributed, which constrains the options for feasible headings. Therefore,for urban areas, the prototype uses spatiotemporal lattice-based algorithms to adapt motion plans to the environment [51]. State-of-the-art research encourages the implementation of these algorithms [52]. Trajectories for obstacle avoidance and lane changes must be calculated in real time using fast algorithms [53], [51], and they can be selected with an evaluation function, as shown in Figure B.3 (j).

B.3.3 Actuation

Our autonomous vehicle prototype follows the path generated by the motion planner.

Path following: The pure pursuit algorithm [54] acutuates our prototype. The pure pursuit algorithm breaks down the path into multiple waypoints. During every control cycle, the module searches for the closest waypoint in the direction of travel. The search extends beyond the specified threshold distance to reduce the change in angle that is required when returning onto the path from a deviation. As shown in Figure B.3 (k), the velocity and angle of the next movement are set to values that bring the vehicle to the selected waypoint following a predefined curvature.

The target waypoint is updated accordingly until the goal is reached. The vehicle follows these updated waypoints until it reaches the user-specified goal. If the control of acceleration steps is not aligned with the velocity and angle output of the pure pursuit algorithm because of noise, the vehicle can temporarily deviate from the planned trajectory. Although the current iteration of our prototype uses a simple PID controller to actuate the vehicle, parameters highly depend on the vehicle and the controller is not sufficient to control the vehicle smoothly. Localization errors can also cause gaps between the vehicle state and the outputs of the pure pursuit algorithm. As a result, the vehicle could collide with unexpected obstacles. To cope with this scenario, the path following module ensures a minimum distance between the vehicle and detected obstacles, overriding the planned trajectory. Our motion planner also updates the waypoints accordingly, considering obstacles in the upcoming travel lane.

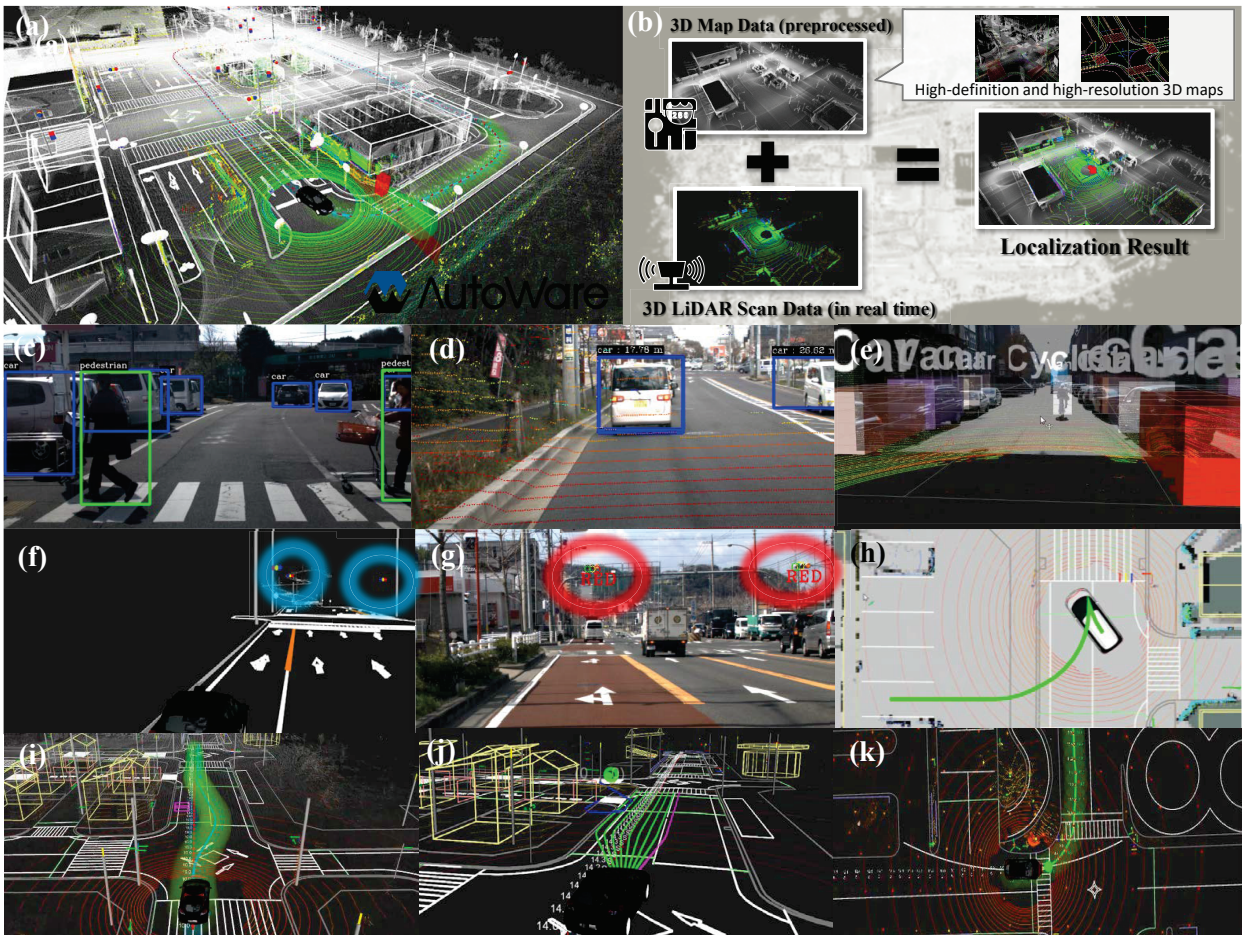

Fig. B.3 Self-driving software packages in Autoware: (a) RViz visualization with high-definition and high-resolution geographical information, (b) NDT scan matching localization using a 3D map and a 3D LiDAR scan, (c) deep learning based object detection with the You only look once (Yolo2) algorithms, (d) projection of the 3D point-cloud data onto the image, (e) calibrated data fusion for LiDAR clouds and images data, (f) traffic light positions extracted from the 3D map, (g) traffic light recognition with the region of interest marked, (h) trajectory planning in a parking lot using hybrid-state A* search, (i) trajectory planning for object avoidance using hybrid-state A* search, (j) trajectory generation for object avoidance using the lattice-based algorithm, and (k) steering angular velocity calculations for path following using the pure pursuit algorithm.

Acknowledgment

I would like to express my deep sense of gratitude to my adviser Professor Toshimitsu Ushio and Assistant Professor Takuya Azumi, Graduate School of Engineering Science, Osaka University, for their invaluable, constructive advice and constant encouragement during this work. Professor Ushio and Assistant Professor Azumi's deep knowledge and their eyes for detail have inspired me much. Without thier encouragement, this thesis would not have materialized. Assistant professor Azumi's deep knowledge and his eye for detail have inspired me much.

I am deeply grateful to Associate Professor Shinpei Kato, Graduate School of Information Science and Technology, the University of Tokyo, Atsushi Matsuo, and Masaki Gondo, eSOL Inc, for their helpful comments on this thesis and warm encouragement. I also would like to thank all members of Ushio Laboratory for thier kind help and cooperation. I am particularly grateful for the assistance given by research assistant Seiya Maeda, my colleage Shunsuke Hori, and Takuro Yamamoto.

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 28th IEEE 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 the ACM 24th RTNS*, 2016, pp. 235–244.

[3] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *Proc. of IEEE CODES+ISSS*, 2015, pp. 220–229.

[4] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in *Proc. of IEEE RTAS*, 2016, pp. 1–11.

[5] M. Becker, K. Sandström, M. Behnam, and T. Nolte, "Mapping real-time tasks onto many-core systems considering message flows," in *Proc. of the WiP Session of the 20th IEEE RTAS*, 2014.

[6] B. Paolo, B. Marko, N. Capodieci, C. Roberto, S. Michal, H. Pemysl, 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, no. Supplement C, pp. 299 – 311, 2017.

[7] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *Proc. of HPEC*, 2013, pp. 1–6.

[8] 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.

[9] C. Ramey, "TILE-Gx100 manycore processor: Acceleration interfaces and architecture," in *Proc. of the 23th IEEE HCS*, 2011, pp. 1–21.

[10] R. Schooler, "Tile processors: Many-core for embedded and cloud computing," in *Proc. of HPEC*, 2010.

[11] 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 IEEE ISSCC*, 2008, pp. 88–598.

[12] G. Chrysos, "Intel® Xeon Phi Coprocessor-the Architecture," *Intel Whitepaper*, 2014.

[13] G. Chrysos and S. P. Engineer, "Intel Xeon Phi coprocessor (codename knights corner)," in *Proc. of the 24th Hot Chips Symposium*, 2012, pp. 1–31.
[14] M. Baron, "The single-chip cloud computer," *Microprocessor Report*, vol. 24, p. 4, 2010.
[15] D. Kanter and L. Gwennap, "Kalray clusters calculate quickly," *Microprocessor Report*, 2015.
[16] B. D. de Dinechin and A. Graillat, "Network-on-Chip Service Guarantees on the Kalray MPPA-256 Bostan Processor," in *Proc. of the 2nd AISTECS*, 2017.
[17] Q. Perret, P. Maurère, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Predictable composition of memory accesses on many-core processors," in *Proc. of the 8th ECRTS*, 2016.
[18] 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.
[19] "ROS.org," http://www.ros.org/.
[20] S. Cousins, "Exponential growth of ROS [ROS Topics]," *IEEE Robotics & Automation Magazine*, vol. 1, no. 18, pp. 19–20, 2011.
[21] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. of IEEE DAC*, 2001, pp. 684–689.
[22] 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 IEEE ISSCC*, 2007, pp. 98–99.
[23] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.
[24] 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.
[25] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *Proc. of the Workshop on Managed Many-Core Systems*, 2008, p. 27.
[26] "Autoware: Open-source software for urban autonomous driving," https://github.com/CPFL/Autoware.
[27] M. Magnusson, "The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection," Ph.D. dissertation, Örebro universitet, 2009.
[28] "Point Cloud Library (PCL)," http://pointclouds.org/.
[29] T. Carle, M. Djemal, D. Potop-Butucaru, R. De Simone, and Z. Zhang, "Static mapping of real-time applications onto massively parallel processor arrays," in *Proc. of the 14th IEEE ACSD*, 2014, pp. 112–121.
[30] H. R. Faragardi, B. Lisper, K. Sandström, and T. Nolte, "A communication-aware solution framework for mapping AUTOSAR runnables on multi-core systems," in

Proc. of IEEE ETFA, 2014, pp. 1–9.
[31] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *Proc. of the 14th ELIV*, vol. 62, 2009.
[32] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed Services of the NoC of a Manycore Processor," in *Proc. of ACM NoCArc*, 2014, pp. 11–16.
[33] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proc. of IEEE IISWC*. IEEE, 2010, pp. 1–11.
[34] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," in *Proc. of the 13th ACM EMSOFT*, 2016, pp. 5:1–5:10.
[35] "Compute Unified Device Architecture (CUDA)," https://developer.nvidia.com/cudazone.
[36] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[37] "Caffe," http://caffe.berkeleyvision.org/.
[38] "OpenCV," http://opencv.org/.
[39] Y. Maruyama, S. Kato, and T. Azumi, "Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," in *Proc. of IEEE ICCD*, 2017.
[40] P. Biber and W. Straßer, "The normal distributions transform: A new approach to laser scan matching," in *Proc. of IEEE/RSJ IROS*, vol. 3, 2003, pp. 2743–2748.
[41] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. of ECCV*, 2016.
[42] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242
[43] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.
[44] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. of IEEE CVPR*, vol. 1, 2005, pp. 886–893.
[45] R. E. Kalman *et al.*, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
[46] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on signal processing*, vol. 50, no. 2, pp. 174–188, 2002.
[47] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Path planning for autonomous vehicles in unknown semi-structured environments," *The International Journal of Robotics Research*, vol. 29, no. 5, pp. 485–501, 2010.
[48] B. Nagy and A. Kelly, "Trajectory generation for car-like robots using cubic curvature polynomials," *Field and Service Robots*, vol. 11, 2001.
[49] H. Darweesh, E. Takeuchi, K. Takeda, Y. Ninomiya, A. Sujiwo, L. Y. Morales, N. Akai, T. Tomizawa, and S. Kato, "Open source integrated planner for autonomous

navigation in highly dynamic environments," *Journal of Robotics and Mechatronics*, vol. 29, no. 4, pp. 668–684, 2017.

[50] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[51] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. of IEEE ICRA*, 2011, pp. 4889–4895.

[52] 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.

[53] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, no. 3, pp. 308–333, 2009.

[54] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, Tech. Rep., 1992.

Publication List

Journal

[1] Y.Maruyama, S.Kato, and T.Azumi, "Scalable Parallel Computing on NoC-based Embedded Many-Core Platform," *IEEE Transactions on Computers*, 2018. (submitted)

International Conference Proceedings

[1] Y.Maruyama, S.Kato, and T.Azumi, "Exploring the Performance of ROS2," *In Proceedings of the 13th ACM International Conference on Embedded Software (EMSOFT2016)*, pp. 5, 2016.

[2] Y.Maruyama, S.Kato, and T.Azumi, "Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," *In Proceedings of the IEEE 35th International Conference on Computer Design (ICCD2017)*, pp. 225-228, 2017.

[3] 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 the ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS2018)*, 2018.

Domestic Conference

[1] Y.Maruyama, S.Kato, and T.Azumi, "Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," *In Proceedings of IPSJ Kansai-Branch Convention*, A-2, 2017. (in Japanese)