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 09, 2018

Division of Mathematical Science for Social System,

Department of Systems Innovation,
Graduate School of Engineering Science, Osaka University

Abstract

Add ROS-lite

This paper presents evaluations of parallel computing on embedded many-core platforms. In embedded systems, high processing requirements and low power consumption require heterogeneous computing platforms including many-core platforms. Considering nonuniform memory access (NUMA) for scalability, many-core applications must be designed based on scalable data allocation and scalable parallel computing. As a reference embedded many-core computing platform, we use the Massively Parallel Processor Arrays 256 (MPPA-256) developed by Kalray, one of the commercial off-the-shelf (COTS) multi-/many-core platforms. In this paper, we conduct evaluations of data transfer on network-on-chip (NoC) implementations, microbenchmarks, and parallelization of a practical application on NoC-based embedded many-core platforms. We investigate currently achievable data transfer latencies between distributed memories on NoC systems, memory access characteristics with NUMA, and parallelization scalability of many cores. As a practical application, we parallelize a portion of the self-driving algorithms on many-core processors. We believe that this performance level is acceptable for the real-world production of autonomous vehicles. By highlighting many-core computing capabilities, we explore scalable parallel computing on NoC-based embedded many-core platforms.

Contents

Chapter 1

Introduction

The evolution of a next-generation computing platform oriented toward having multi-/many-core platforms is necessary to satisfy the demand for increasing computation in conjunction with reasonable power consumption in several domains such as automobiles. This trend is also adapted to embedded systems because of high processing requirements. For example, self-driving systems involve various applications and are sometimes characterized by demands for high-performance computing. Considering the requirements of high processing, predictability, and energy efficiency for self-driving systems, such systems require a heterogeneous computing system such as multi-/many-core platforms and graphical processing units (GPU). (This is discussed in Chapter 5.) In embedded systems, multi-/many-core architectures are an important trend as the architecture integrates cores to realize high-performance and general-purpose computing with low power consumption [1], [2]. Extant studies have examined several applications of multi/many-core platforms [1], [3], [4], [2], [5], [6].

Based on the above background, multi/many-core platforms are fabricated and released as commercial off-the-shelf (COTS) multicore components. (e.g., the Massively Parallel Processor Arrays (MPPA) 256 developed by Kalray [7], Tile-Gx developed by Tilera [8] [9], Tile-64 developed by Tilera [10], Xeon Phi developed by Intel [11], [12], and Single-chip Cloud Computer (SCC) developed by Intel [13]) Several platforms such as MPPA-256 and Tile-64 target embedded systems, and the research focused on multi-/many-core platforms has received increasing attention [14], [15], [16]. We discuss comparisons of many-core platforms in Chapter 5.

In embedded many-core platforms, nonuniform memory access (NUMA) and distributed memories connected with network-on-chip (NoC) components are an important approach for core scalability and power consumption. Several COTS platforms include NoC technology and a cluster of many-core architectures in which cores are allocated closely. For instance, MPPA-256 and Tile-Gx72 have NoC components to share distributed memories instead of shared buses. Furthermore, MPPA-256 contains 16 clusters in which 16 cores are included and contains 256 general-purpose cores in total. Although the cores do not guarantee cache coherency, this significantly exceeds the number of cores in other COTS such as Tile-64 and Tile-Gx72. The clusters of cores are capable of running separate independent applications with respect to the desired power envelope of embedded

applications.

Despite the appearance of embedded many-core platforms, several difficulties persist in the adaptation of these platforms in embedded domain [1], [3]. In NoC-based embedded many-core platforms, researches on NoC-based many-core platforms do not fully reveal data transfer between distributed memories with NoC technology, memory access characteristics in NUMA, and parallelization potential in practical applications for application developers. Additionally, reuse of existing software application is difficult because application developers have to write platform oriented codes for NoC data transfer between clusters. For example, writing software for robotics system including self-driving system is difficult, particularly as the scale and scope of embedded software application continues to grow.

To meet these challenges, we conduct evaluations to reveal the practicality of NoC-based many-core platforms and propose software framework called ROS-lite similar to Robot Operating System (ROS) [17], [18] for reuse of existing applications and efficient development on heterogeneous computing platforms. ROS, an open-source robot operating system, is structured communications layer above the host operating systems of a heterogeneous compute cluster and has been being rapidly developed and widely used in the robotics community [19]. (Please refer to Apendix A: Robot Operating System (ROS).) This work explores scalable parallel computing and data allocation with quantitative evaluations and a practical application. As a reference embedded many-core platform, we have implemented evaluation program, practical applications, and ROS-lite on MPPA-256 [7] which adopts NUMA using NoC technology and realizes numerous cores with low power consumption.

Contributions: This work focuses on examining embedded many-core computing based on NoC technology, such as MPPA-256. We conduct evaluations of data transfer methods on NoC components and microbenchmarks with matrix calculation to clarify latency characteristics of data transfer, parallel computing, and the influence of data allocation. Subsequently, we parallelize a localization algorithm that is the core of the self-driving system. Finally, this paper for the first time presents a light weight ROS architecture supporting NoC called ROS-lite on embedded many-core platforms. We reveal the advantages and disadvantages of embedded many-core computing based on NoC technology in a quantitative manner by introducing the following contributions:

- On DMA-capable NoC components, the evaluations of the data transfer quantitatively characterize the end-to-end latencies, which depend on the routing and DMA configurations, and the memory access speed, which varies according to where data are allocated and what accesses the memory.
- In many-core platforms based on NoC technology, the scalability of parallelization is observed in evaluations of matrix calculations in several situations and a real complex application as a part of a self-driving system.
- Under limited memory, our proposed software framework provides structured communications layer and efficient development on heterogeneous computing platforms including NoC.

To the best of our knowledge, this is the first work that examines data transfer and

data allocation matters for many-core computing beyond an intuitive expectation to allow system designers to choose appropriate data transfer methods. Additionally, the speed-up result of a self-driving application indicates a practical potential for NoC-based embedded many-core computing.

Organization: Add ROS-lite section

The remainder of this work is organized as follows. The system model considered in this work is discussed in Chapter 2 in which the hardware model, namely Kalray MPPA-256 Bostan, and the system model are presented. Chapter 4 explains evaluation setup and approach, and illustrates experimental evaluations. Chapter 5 examines the related work that focuses on multi-/many-core systems. Chapter 6 presents the conclusions and directions for future research.

Chapter 2

System Model

This section 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.

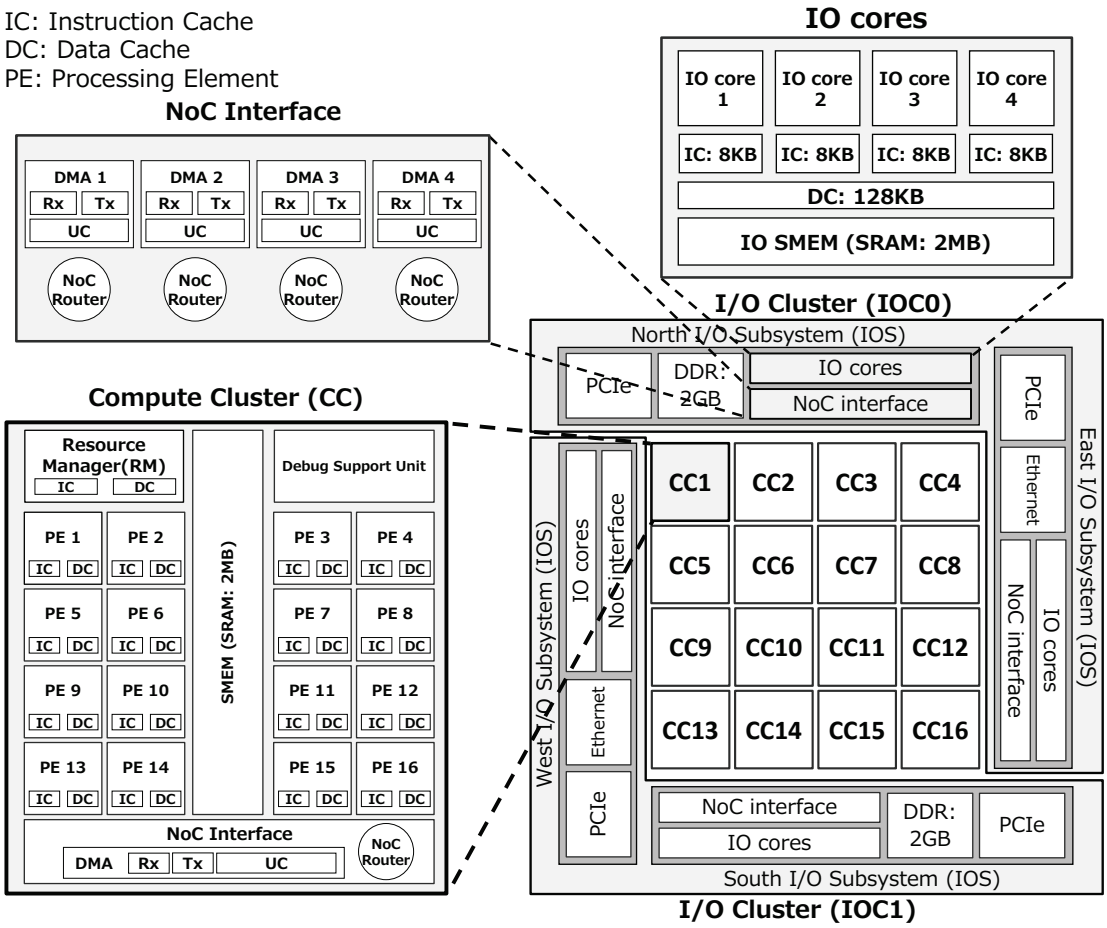

Fig. 2.1 An overview of the architecture of the Kalray MPPA-256 Bostan.

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.

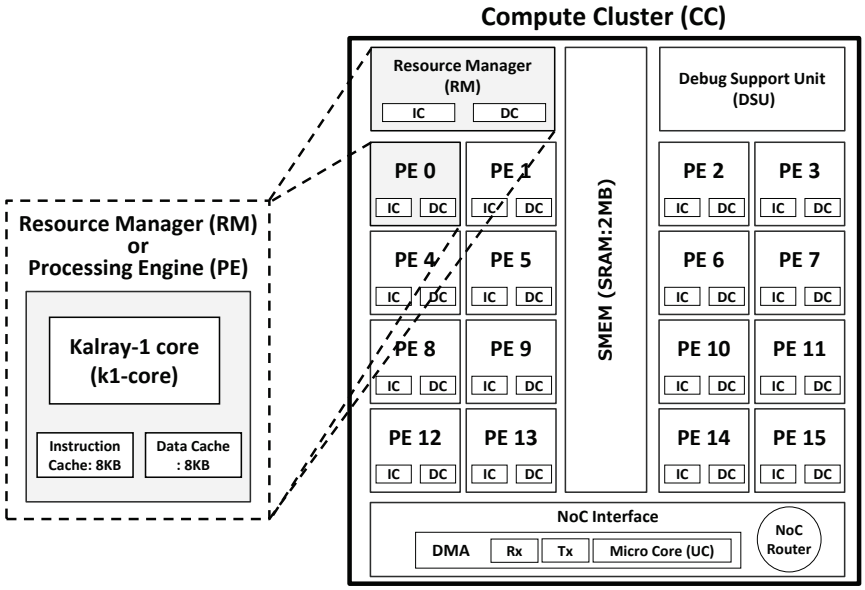

Fig. 2.2 Compute Cluster architecture.

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 [20], which involves a low average number of hops when compared to mesh topology [21], [22]. 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).

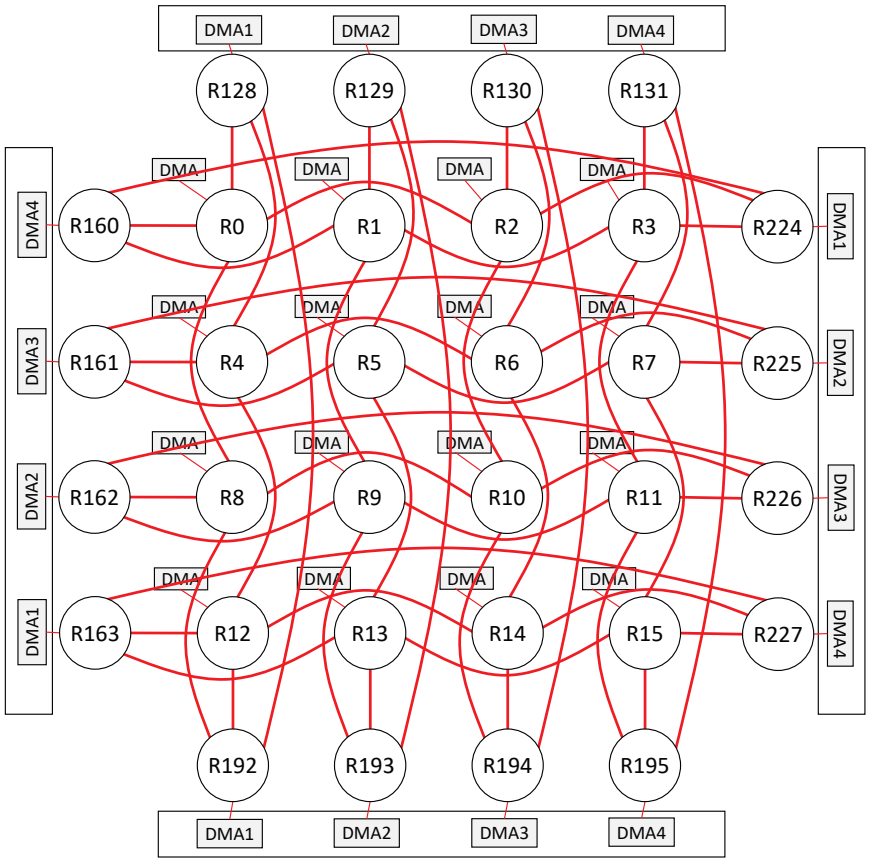

Fig. 2.3 NoC connections (both D-NoC and C-NoC).

2.2 Software Model

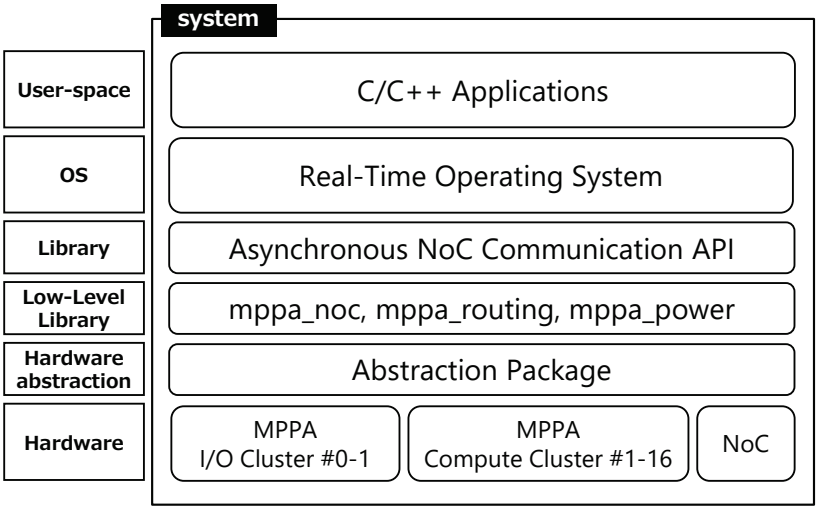

Fig. 2.4 The software stack of Kalray MPPA-256.

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 [23], [24]. 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

In this section, data transfer methods in MPPA-256 are explained. For scalability purposes, MPPA-256 accepts a clustered architecture that avoids frequent memory contention between numerous cores, and it also accepts clustered architectures in which each cluster contains its own memory. Sixteen cores are packed as a cluster sharing 2 MB memory (SMEM), as shown in Fig. 2.2. This avoids frequent memory contention due to having numerous cores and helps in increasing the number of cores. However, the architecture constrains memory that can be directly accessed by the cores. Communicating with cores outside the cluster requires transferring data between clusters through the D-NoC with NoC interfaces.

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

An Rx interface exists on the receiving side to receive data with DMA. It is necessary to allocate a D-NoC Rx resource and configure the Rx to wait to receive the data. A

DMA in a NoC interface contains 256 D-NoC Rx resources.

Two interfaces, the Tx interface and the UC interface as explained in Sections 2.1.1 and 2.1.2, are present with respect to the sending side for users to send data between clusters. The UC is a network processor that is 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 to the direct activation of the Tx interface. However, a DMA in a NoC interface contains only eight D-NoC UC resources. Both interfaces use a DMA engine to access memory and copy data. Regardless of whether a UC interface is used, it is necessary to allocate a D-NoC Tx resource and configure the Tx to send data. Additionally, it is necessary to allocate and configure D-NoC UC resources, if a UC interface is used.

Application developers have several choices to transfer data on D-NoC between clusters. Kalray provides mppa_noc and mppa_async libraries and eMCOS provides two kinds of message APIs. Table 2.1 compares features of each method. Specifications are different for each method, and it is necessary to use properly according to the purpose. In an internal implementation, all methods are based on mppa_noc. mppa_noc is low-level NoC communication library and provides users interfaces to handle DMA resources such as an Rx, Tx and UC interfaces for NoC data transfer. Each method which application developers are able to use is described in detail as 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**:
- **mppa_async**:
- **eMCOS message**:
- **eMCOS session message**:

Chapter 3

Proposed Framework

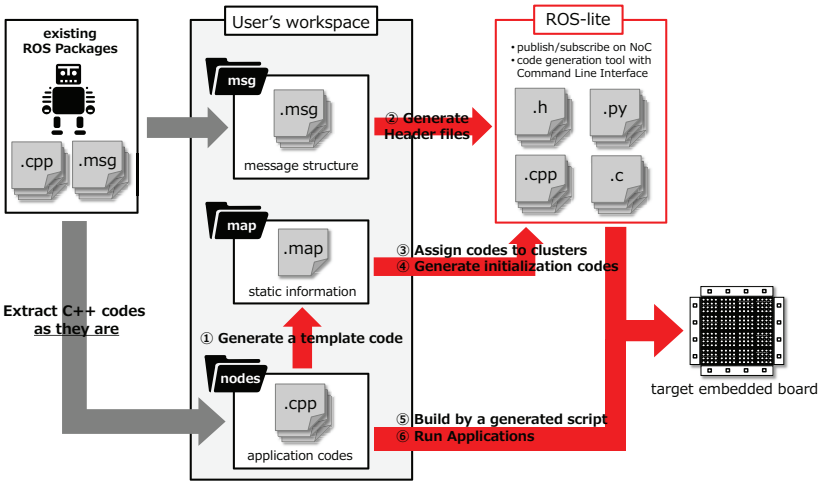

Fig. 3.1 Development flow of ROS-lite framework.

The proposed framework, ROS-lite, is structured communications layer for efficient development on many-core platforms. Although existing ROS has been being rapidly developed and widely used, the ROS does not support NoC communication and embedded requirement such as RTOS and limited memory on embedded platforms. ROS-lite is a solution for these challenges and provides a development environment in which ROS nodes run on each core on many-core platforms and communicate with each other. By enabling and extending numerous existing ROS applications, transporting and new development on embedded platforms are made much more efficient. (For more information of ROS basic, please refer to Apendix A: Robot Operating System (ROS).) Additionally, being based on existing ROS framework, ROS-lite has potential to communicate external ROS nodes on other platform. Design of ROS-lite can enhance efficient development on heterogeneous computing platforms. In this paper, we target eMCOS as RTOS and MPPA-256 as an embedded many-core platform.

3.1 ROS-lite Toolchain

This section describes a system model and a proposed framework used to run ROS nodes on an embedded many-core platform. The system architecture of ROS-lite is shown in Fig. 3.1. ROS-lite can run application codes written as ROS nodes. We assume that application codes following ROS manner are written in C++. Application developers can extract source codes (application codes and message structure files) from existing ROS packages them as they are. For task mapping, developers edit an map file (`.map`) and assign ROS nodes to the clusters on many-core platforms. All developers have to do for modification of task mapping is editing map file. ROS nodes as processes on cores are described as publish/subscribe model, following message structures defined in message files (`.msg`). ROS nodes can communicate with each other via either innner-cluster or inter-cluster communications. While these cores do communications via shared memory in innner-cluster, the cores do communications via NoC in inter-cluster.

Our framework provides code generation and build system by command line interface for efficient development. First, in ① of Fig. 3.1, a template of a map file (`.map`) is generated from application source codes. The map file (`.map`) contains node name, assigned cluster, information of topics published and subscribe to. These descriptions except for assigned cluster number is able to be interpreted from ROS nodes so that we provide code geraration for template codes. Application developers can focus on the number of assigned cluster. Second, in ② of Fig. 3.1, header files defining message structure are generated from message files (`.msg`) as an original ROS does. Note that generated header file has been made lighter because the part required by ROS-lite is generated. This code generation part is based on an original ROS's script and message files (`.msg`) can be described in the same manner. Developers do not have to modify message files (`.msg`) at all. Third, in ③ and ④ of Fig. 3.1, initialization codes of launching processes of ROS nodes are generated from a map file (`.map`). ROS nodes are automatically launched as processes scheduled by RTOS on user-assigned clusters on embedded platforms. Application developers do not have to write codes except for ROS nodes. Finally, in ⑤ of Fig. 3.1, a build script is generated from a user-defined map file (`.map`). Source codes of ROS nodes are separately built in each user-assigned cluster because the executable files are loaded into separate memories for each cluster. This building process is conducted witd generated build script. There is no need to modify build script each time developers change task mapping by modification of the map file (`.map`). All codes of ROS nodes are allocated in a same directory and are automatically built in each user-assigned cluster.

We give a simple example for explaination of ROS-lite framework. One node publishes a *chatter* topic and two nodes subscribe to the topic as shown in 3.2. Messages in the *chatter* topic are defined as string type named *data*. The publisher node is launched in *cluster 1* and two subscriber node in *cluster 2*, as described in assigned cluster number in the map file (`.map`) of Fig. 3.3. Application developers can change node mapping by modification of the field of cluster number. Information of topics in map file (`.map`) is used for initialization of the relationship between the topics and the nodes so that ROS-lite

conducts the process of matching the nodes to communicate by topic names. These field of the node name and topic information are generated from source codes of ROS nodes, and initialization codes of the node relationship for ROS-lite are generated from the map file (`.map`). Note that the field of assigned cluster number has to be filled by developers.

3.2 Design and Implementation

design choice:

assign cluster not core affinity

why? scheduled by eMCOS. mapping is future work and out of scope in this paper.

No master as searver. static initialization with map file

why? avoid single point failure

why? for analysis of real-time requirement in future work

why? The dynamic node set change at runtime is rare in embedded system

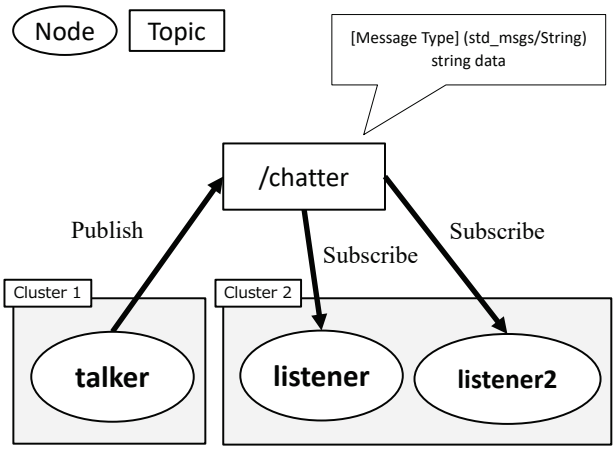

Fig. 3.2 The publish/subscribe model in ROS-lite.

```
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.

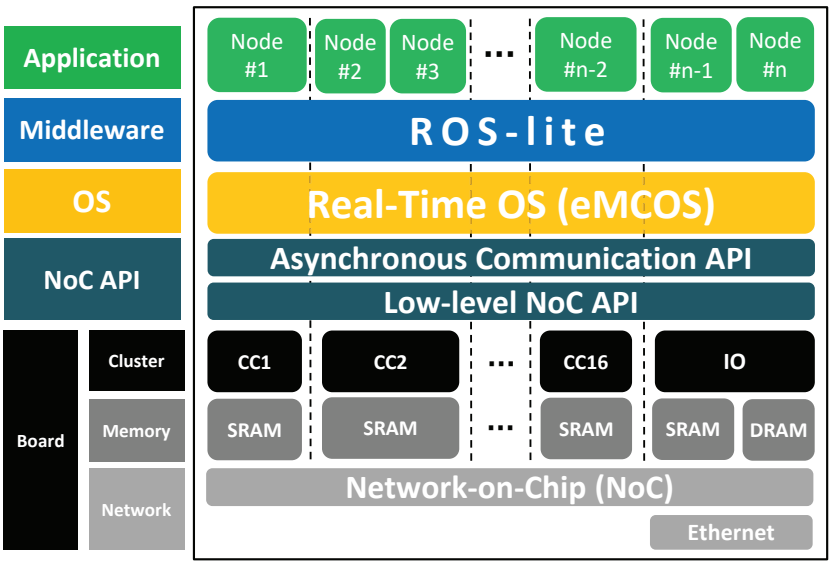

Fig. 3.4 System stack of ROS-lite on a many-core platform.

Chapter 4

Evaluations

First, this section 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, we 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

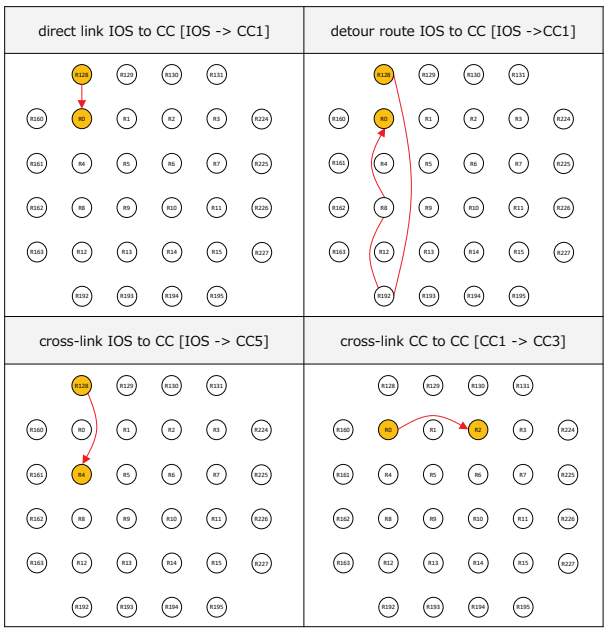

Fig. 4.1 Four D-NoC routes used in the evaluation.

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, we 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.

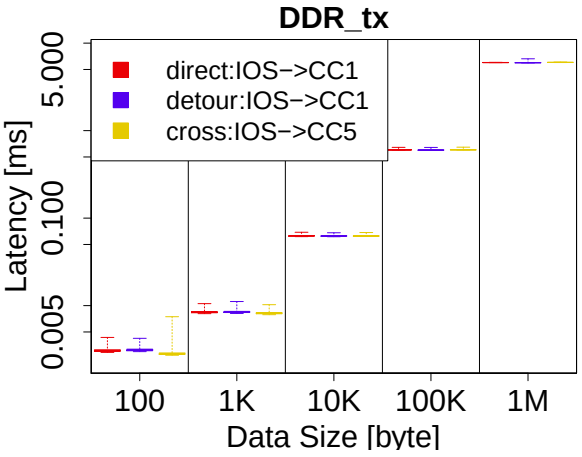

Fig. 4.2 Data transfer with Tx from IO DDR to CC.

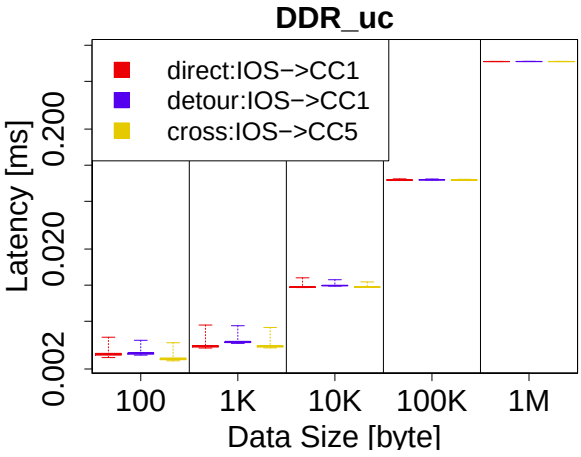

Fig. 4.3 Data transfer with UC from IO DDR to CC.

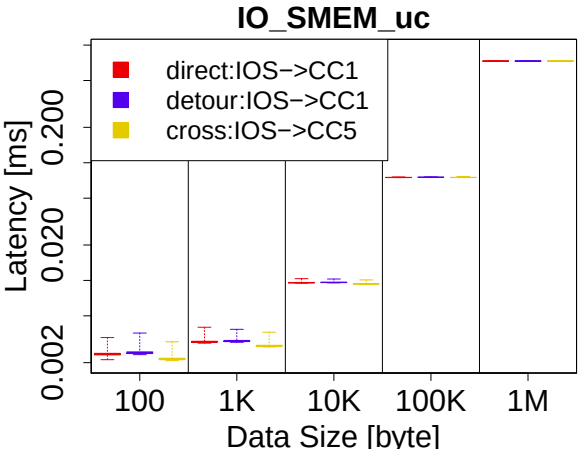

Fig. 4.4 Data transfer with UC from IO SMEM to CC.

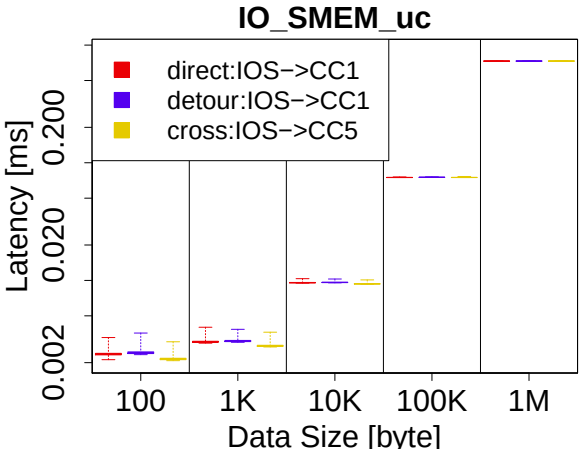

Fig. 4.5 Data transfer with UC from IO SMEM to CC.

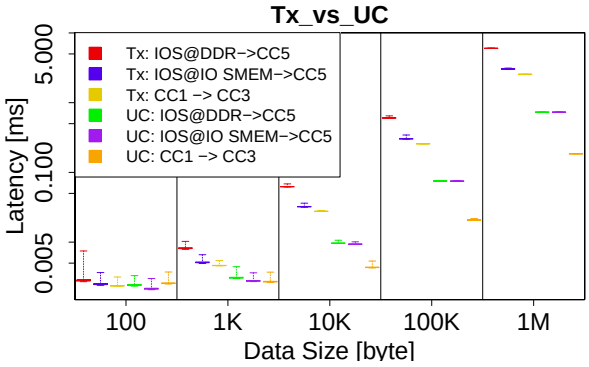

Fig. 4.6 Data transfer with Tx/UC (logarithmic axis).

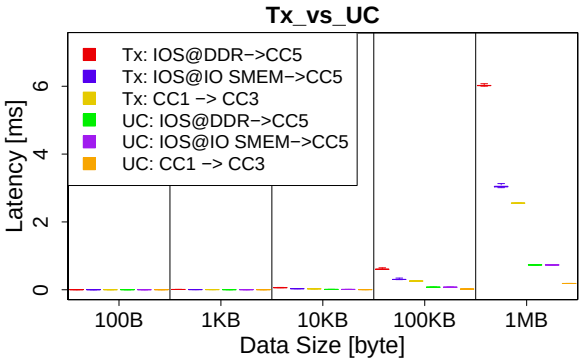

Fig. 4.7 Data transfer with Tx/UC (linear axis).

4.2 Matrix Calculation

4.2.1 Situations and Assumptions

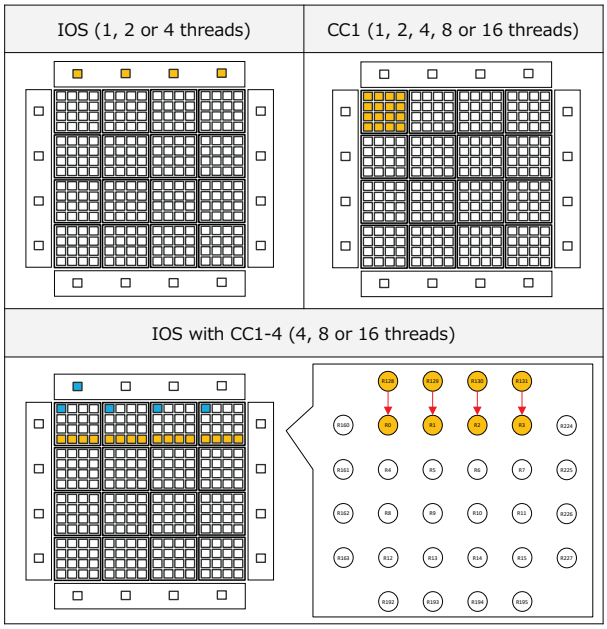

Fig. 4.8 Matrix calculation situations.

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. We 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.

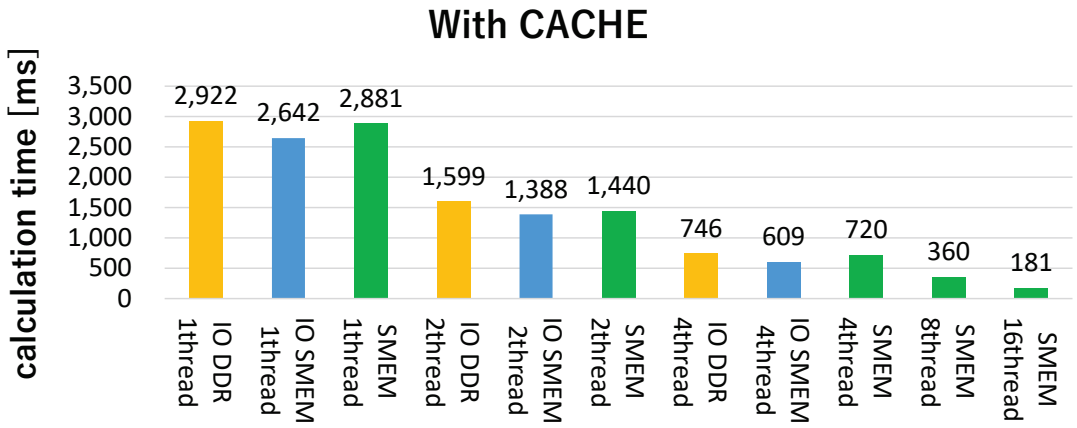

Fig. 4.9 Matrix calculations in IOS and CC with cache.

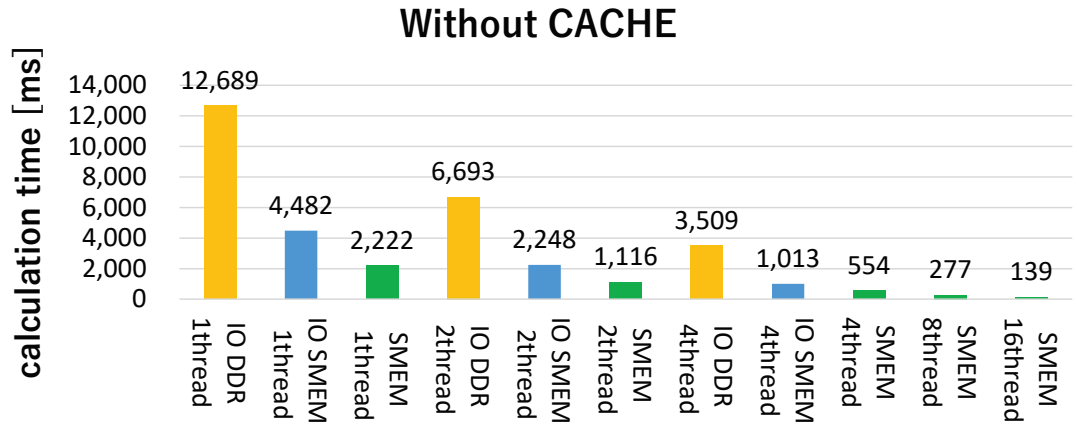

Fig. 4.10 Matrix calculations in IOS and CC without cache.

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 are 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.

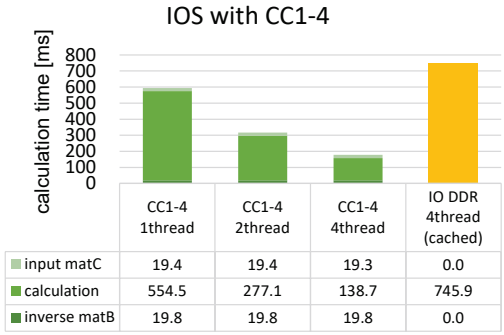

Fig. 4.11 Matrix calculations with offload computing (314 KB matrix x 3).

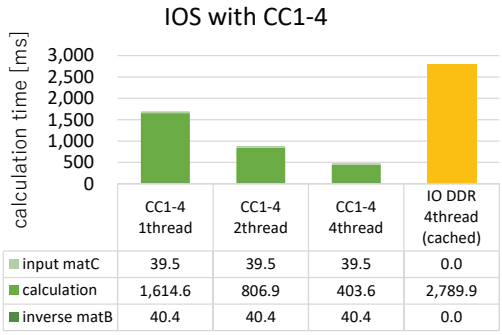

Fig. 4.12 Matrix calculations with offload computing (640 KB matrix x 3).

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. We selected an algorithm for vehicle self-localization written in C++ in Autoware, open-source software for urban self-driving [25], and a parallelized part of it. (Please refer to Apendix B: Autoware.) The self-localization adopts the normal-distribution transform matching algorithm [26] implemented in the Point Cloud Library [27]. 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

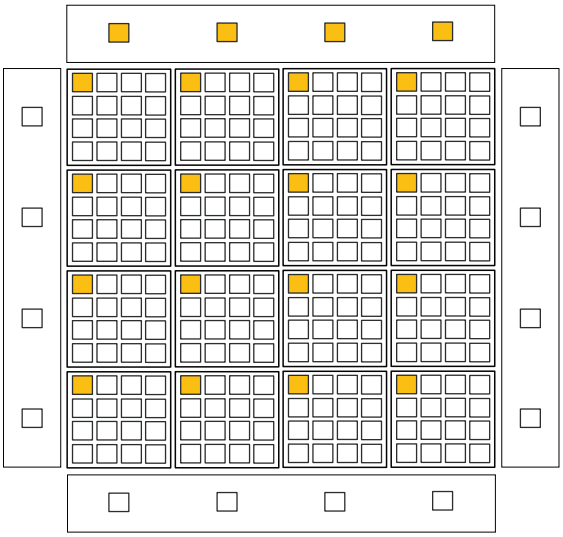

Fig. 4.13 A situation of vehicle self-localization execution.

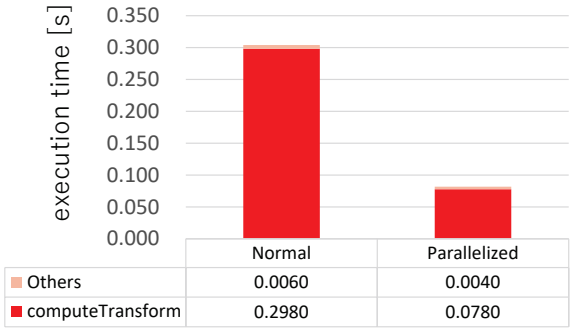

Fig. 4.14 Vehicle self-localization of the partially parallelized self-driving application.

4.4 Lessons Learned

Thus far, in Chapter 4, we have quantitatively clarified the characteristic of data transfer and parallel computing on NoC-based embedded many-core platforms. We 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, we 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, we 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, we 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 we parallelize the vehicle self-localization algorithm of the self-driving system, NoC-based embedded many-core systems can be used practically in real environments. We 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. We also conducted demonstration experiments with real environments and confirmed the practicality of NoC-based embedded many-core systems.

Chapter 5

Related Work

Add comparison to eMCOS, Facrored OS, and Barrelfish

Add ROS-lite comparison table and descriptions

This section 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 we 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 [14]. 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, [7], Tile-Gx by Tilera [8], [9], Tile64 by Tilera [10], and Xeon Phi by Intel [11], [12], Single-chip Cloud Computer (SCC) by Intel [13]). The present work focuses on the Kalray MPPA-256, which is designed for embedded applications. Kalray [7] 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 [14]. 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. We 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], [28], [2], [16].

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 [28], [29], [2]. Airbus [2] proposes a method of directed acyclic graph (DAG) scheduling for hard real-time applications using MPPA-256. In [29], a mapping framework is proposed on the basis of AUTOSAR which is applied as a standard architecture to develop automotive embedded software systems [30]. AUTOSAR task scheduling considering contention in shared resources is presented in [1].

By examining the above mapping algorithms of real-time applications, previous work [31], [15], [14], [16] 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 [14]. 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 [31] and [15]. 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. [16] 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 development	costs	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 [7]	✓	✓	L
Tilera Tile series [10]		L	✓
Intel Xeon Phi [11] [12]			✓
Intel SCC [13]			✓

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 [14]	L				
Network-on-Chip Service Guarantees [15]		✓			
Predictable composition of memory accesses [16]		✓	✓		
this paper	✓	✓	✓	✓	✓

Chapter 6

Conclusions

In this work, we conducted quantitative evaluations of data transfer methods on NoC technology, microbenchmarks with matrix calculation, and a practical application with NUMA many-core platforms such as MPPA-256. 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. Last, parallelization of a real application proved the practicality of NoC-based embedded NUMA many-core platforms.

In future work, we will use benchmark applications such as that in [32] for further analysis. In addition, considering above the results, we will port real-time systems such as that in [33] on CCs, and we propose the parallelization of memory intensive algorithms, such as the nearest neighbor search in Section 4.3.

Apendix A: Robot Operating System (ROS)

As mentioned earlier, numerous robotics software such Autoware [25] is based on ROS [17], [18], which is a component-based middleware framework developed for robot applications. (For more information, please refer to Apendix B: Autoware.) ROS is designed to enhance modularity of robot applications at a fine-grained scale and is made suitable for distributed systems, while respecting efficient development. Since autonomous vehicles require many software packages, ROS is a strong basis to develop Autoware.

In ROS, the system is abstracted by *nodes* and *topics*. The *nodes* represent individual component modules, whereas the *topics* hold input and output data between *nodes* as shown in Figure 1.1. ROS *nodes* are usually standard C++ programs. They can use any other software libraries installed in the system.

Communication among *nodes* is based on a publish/subscribe model. This is a strong abstraction model for compositional development. In this model, *nodes* communicate by passing *messages* via a *topic*. A *message* has a simple data structure (much like C structs) defined by .msg files. *Nodes* identify the content of the *message* by the *topic* name. As a *node* publishes a *message* to a *topic*, another *node* subscribes to the *topic* and utilizes the *message*. For example, in Figure 1.1, the "Camera Driver" *node* sends *messages* to the "Images" *topic*. The *messages* in the *topic* are received by the "Traffic light Recognition" *node* and "Pedestrian Detection" *node*. *Topic* is also managed by first-in, first-out queues when accessed by multiple *nodes* simultaneously. Meanwhile, ROS *nodes* can launch several threads implicitly. Real-time issues, however, must be addressed.

ROS also provides an integrated visualization tool, called RViz, and a data-driven simulation tool, called ROSBAG. Figure 1.4 (a) shows examples of visualization for perception tasks in Autoware with RViz. The RViz viewer is useful for checking the status of tasks. The ROSBAG is a set of tools for recording from and playing back to ROS *topics*. It provides us development environments to test self-driving algorithms without hardware devices such as a vehicle and sensors devices. This is also useful for efficient development.

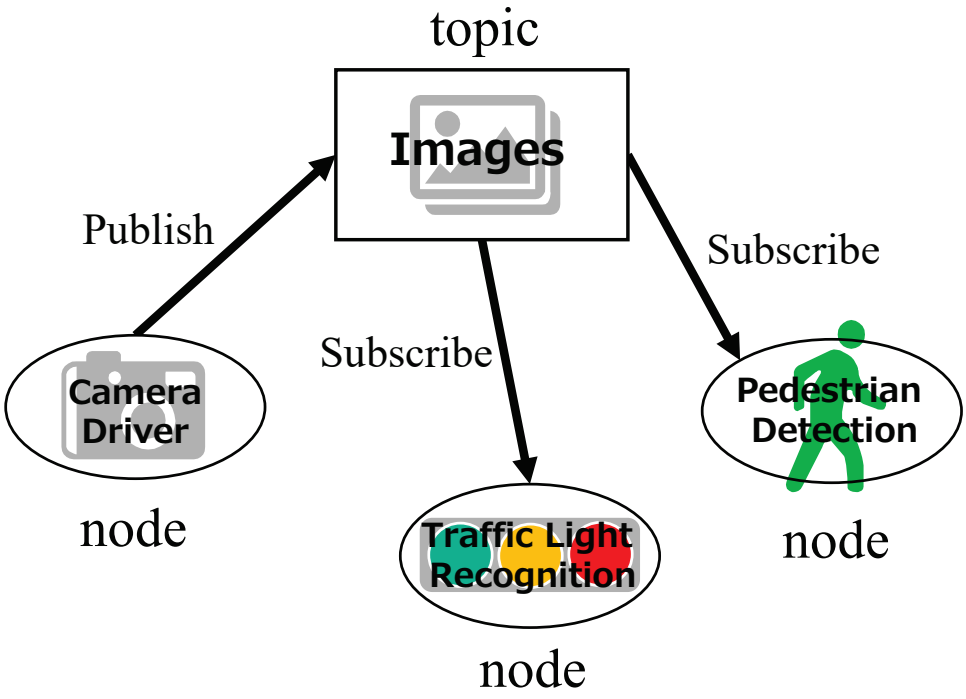

Fig. 1.1 The publish/subscribe model in ROS.

Apendix B: Autoware

1.1 System Model

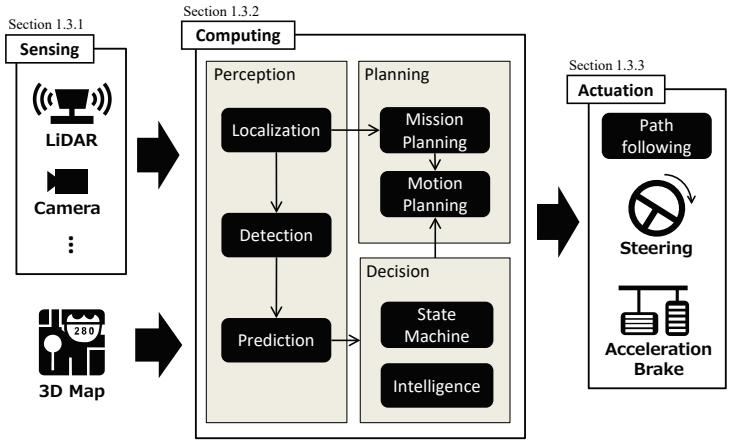

Fig. 1.2 Basic control and data flow of autonomous vehicles.

Autonomous vehicles as cyber-physycal systems can be abstracted by sensing, computing, and actuation, as shown in Figure 1.2. Preferred sensing devices for urban-area self-driving, in particular, include laser scanners (LiDAR) and cameras. Actuation corresponds to manipulation of steering and stroking whose twisted control commands are often generated by the path following module. The computing intelligence is the major part of self-driving technology. For instance, scene recognition requires localization, detection, and prediction modules, while path planning falls into mission-based and motion-based modules. Each module employs corresponding algorithms. Those implemented in Autoware, in particular, are introduced in Section 1.3.

Figure 1.2 shows a basic control and data flow of autonomous vehicles. Sensing devices acquire environmental information as input data for the computing intelligence. 3D maps are also becoming more and more commonplace for self-driving technology, especially for urban areas, to complement sensing and computing capabilities of autonomous vehicles by high-definition and high-resolution geographical information. Leveraging these pieces of information, for example, the accuracy of localization and detection can be improved highly without causing their algorithms to become more complex. Typical outputs of the computing intelligence are angular and linear velocity values, which correspond to the

commands for steering and stroking, respectively.

1.2 System Stack

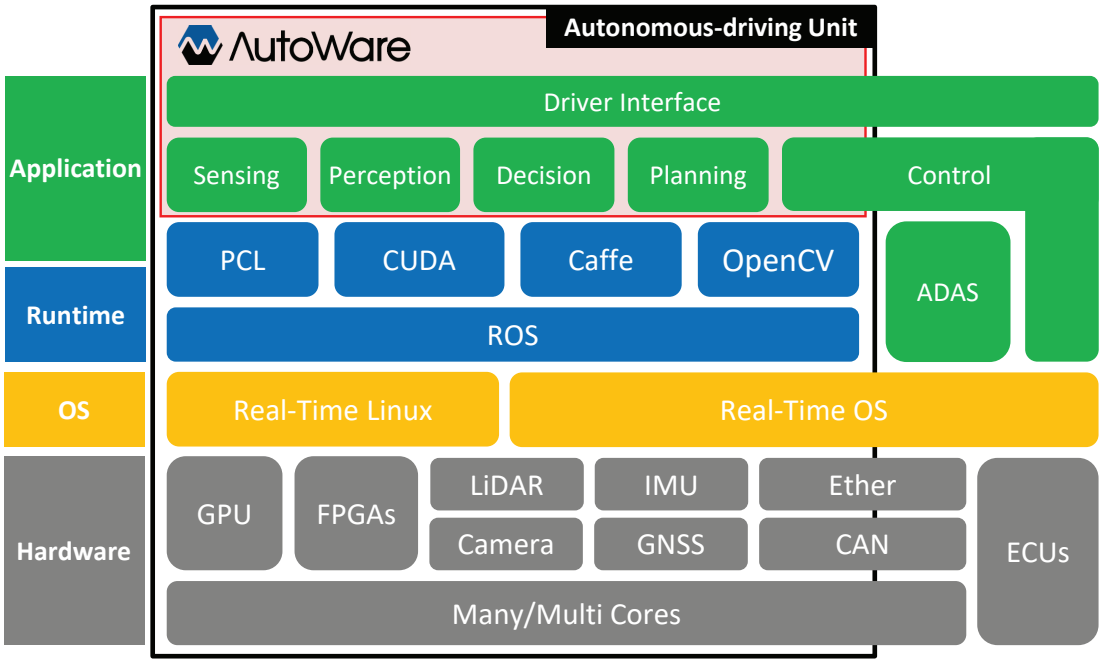

Fig. 1.3 Complete system stack of autonomous vehicles using Autoware.

We build the complete system stack of autonomous vehicles on top of integrated platforms using open-source software, as shown in Figure 1.3. The primary target of self-driving technology presented in this paper is set for urban areas rather than freeways and highways. We develop Autoware which is a popular open-source software project developed for urban-area autonomous vehicles. Autoware is based on Robot Operating System (ROS) and other well-established open-source software libraries, as shown in Figure 1.3. We introduce a brief overview of ROS separately in Apendix A: Robot Operating System (ROS). Point Cloud Library (PCL) [27] is mainly used to manage LiDAR scans and 3D maps. It is also used for data filtering and visualization. CUDA [34] is a programming framework for general-purpose computing on GPUs (GPGPU). Due to compute-intensive tasks in autonomous vehicles, GPUs and CUDA are promising solutions for self-driving technology, though this paper does not focus on them. Caffe [35], [36] is a native deep learning framework made with expression, speed, and modularity in mind. OpenCV [37] is a popular computer vision library for image processing.

Autoware provides 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 by installing Autoware. In several countries, Autoware has already succeeded in demonstration of autonomous vehicles driving for a long distance

of public roads in urban areas. Recently, automotive manufacturers and suppliers have also started adopting Autoware as a baseline of their prototype autonomous vehicles.

In this section, we also briefly introduce hardware pieces of autonomous vehicles assumed in this paper. To apply our system stack, we need to make a modification to commercial vehicles so that they can be controlled by external computers through some secure gateway, and also install sensors on the vehicles. The interface among the vehicles, computers, and sensors could be the CAN bus, Ethernet, and/or USB 3.0. The specification of sensors and computers, however, highly depends on the functional requirement of each autonomous vehicle. Autoware supports various sensing devices, but our prototype system specifically uses Velodyne HDL-32e LiDAR scanners and PointGrey Grasshopper3 cameras. Autoware also supports various computers. Many users often run Autoware on desktop and laptop computers. In this paper, we specifically use NVIDIA DRIVE PX2, though our previous work partly ported Autoware to another embedded computing platform, called Kalray Massively Parallel Processor Array (MPPA) [38].

1.3 Algorithms

In this section, we present the detail of Autoware in accordance with Figure 1.2, with a particular emphasis on the modules that we develop and extend for DRIVE PX2. The remaining pieces of Autoware can be found and studied through its project repositry [25]. Note that Autoware is designed for urban-area autonomous vehicles, so we may need alternative projects for those intended to drive on freeways and highways. Discussions on the accuracy and optimization of each algorithm are not within the scope of this section.

1.3.1 Sensing

Our system recognizes road environments, using various sensors, such LiDAR scanners, cameras, radars, and GPS/IMU. We use LiDAR scanners and cameras as primary sensing capabilities. LiDAR scanners measure the distance to a target by illuminating that target with a pulsed laser light and measuring the reflected pulses with the scan. Point-cloud data measured by LiDAR can be used to make digital 3D-representations of the environments. Cameras are often used to recognize traffic lights and object classes, which are sometimes superior to LiDAR scanners in terms of feature quantity of objects and sampling rates.

Raw point-cloud data may need to be filtered and preprocessed before proceeding to the computing stage. For instance, we apply the Normal Distributions Transform (NDT) algorithm [39] to 3D point-cloud data processing. Replacing the original points with one point of those centroids with cubic lattice grid called a voxel, we can downsample data as approximate points in the grid [26]. This filter is applied before performing localization, detection, and mapping.

In addition, radars and GPS/IMU can be used to further complement localization, detection, and mapping. Radars are already deployed in commercial vehicles, which are often used for the purpose of ADAS safety applications. GPS/IMU can be also coupled with gyro sensors and odometers to fix the positioning information.

1.3.2 Computing

The computing intelligence is a major component of self-driving technology. Receiving sensor data and 3D maps as inputs, it computes the final path trajectory as outputs to actuation components. In this section, we explain its detail from the viewpoint of perception, decision, and planning.

Perception

The purpose of perception is to estimate of the position of the ego-vehicle in the 3D map, and also to recognize surrounding scenes including moving objects and traffic signals.

Localization: Localization is one of the most basic and important tasks of autonomous vehicles. Especially in urban areas, the accuracy of localization dominates the reliability of autonomous vehicles. In Autoware, localization is based on scan matching with 3D maps and LiDAR scanners, which achieves a few centimeters order of accuracy for position and rotation. We use the NDT [39] algorithm to solve the localization problem. The computational cost of the NDT algorithm does not suffer from the map size (the number of points). Thus, we can use high-definition and high-resolution 3D map data in real-time. To be precise, we use the 3D version of NDT to perform scan matching over filtered 3D point-cloud data and 3D map data by using PCL, as shown in Figure 1.4 (b) [26]. As a result, localization meets an order of centimeters accuracy.

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

We primarily use the NDT algorithm for both localization and mapping, though Autoware also supports another well-known algorithm called Iterative Closest Point (ICP). This is a feature of Autoware that users can choose preferred algorithms for their system.

Detection: Once localized, we next have to detect surrounding objects, such as vehicles, pedestrians, and traffic signals, to avoid accidents and violation of traffic rules. We first focus on moving objects (vehicles and pedestrians), while Autoware is also capable of recognizing traffic signals and lights as explained later in this section. Autoware supports deep learning [40], [41] and pattern recognition [42] for detection of vehicles and pedestrians, using libraries such as Caffe and OpenCV. In particular, as shown in Figure 1.4 (c), we primarily use the Single Shot MultiBox Detector (SSD) algorithm [40] and the You only look once (Yolo2) algorithm [41], which are based on deep learning. They are unified frameworks for object detection with a single neural network, realizing fast object detection in real-time. On the other hand, the pattern recognition algorithm based on Deformable Part Models (DPM) [42] searches and scores the Histogram of Oriented Gradients features of target objects on the image captured by a camera [43].

Apart from image processing, we also use point-cloud data scanned from a 3D LiDAR scanner to detect objects by Euclidean clustering. Point cloud clustering aims to obtain the distance to objects rather than to classify them. The distance information can be

used to range and track the objects classified by image processing.

Assuming that localization is precise and a 3D map is built for the area where the target autonomous vehicle is self-driving, we can improve the accuracy of recognizing traffic signals and traffic lights. Projecting the 3D map onto the image originated on the current position, we know the exact road area on the image. Therefore, we can constrain the region of interest (ROI) for image processing to this road area so that we can save execution time and reduce false positives. To obtain ROI, we conduct calibration between the 3D LiDAR scanner and the camera. The calibration process needs to be done offline in advance. We apply this approach to recognize traffic signals from the camera image as shown in Figures 1.4 (f) and (g). In general, traffic light recognition is known to be one of the most difficult problems for autonomous vehicles. We can, however, achieve accurate traffic light recognition, due to the presence of 3D map information and the result of precise localization.

Prediction: Because we perform the object-detection algorithm on each frame of the image and point-cloud data, we must associate its results with other frames on a time basis so that we can predict the trajectories of moving objects for mission and motion planning.

We can use two algorithms, Kalman Filter and Particle Filter, to solve this prediction problem. Kalman Filter is used under a linear assumption that our autonomous vehicle is driving at constant velocity while tracking moving objects [44]. Its computational cost is lightweight and suited for real-time processing. Particle Filter, on the other hand, can work for nonlinear tracking scenarios, in which both our autonomous vehicle and tracked vehicles are moving [45]. In our platform, we use both Kalman Filter and Particle Filter, depending on the given scenario. We also apply them for tracking on both the 2D (image) plane and the 3D (point-cloud) plane.

We can combine detected and tracked objects on the image captured by the camera, and clustered and tracked objects obtained by the 3D LiDAR sensor. This is often referred to as sensor fusion with projection and re-projection. We conduct sensor fusion between 3D LiDAR sensors and cameras by beforehand calibration.

We augment scene recognition supported by our platform with sensor fusion of a camera and a 3D LiDAR sensor. To calculate the extrinsic parameters required to make this sensor fusion, we must calibrate the camera and the 3D LiDAR sensor. We can then project the 3D point-cloud information obtained by the 3D LiDAR sensor onto the image captured by the camera so that we can add depth information to the image and filter out the region of interest of object detection. With sensor fusion, Figures 1.4 (d) and (e) represent the results of projection and bounding boxes of clustered 3D point-cloud objects projected onto the image.

The detected and tracked objects on the image can also be reprojected onto the 3D point-cloud coordinates using the same extrinsic parameters. We also use the reprojected object positions to determine the motion plan and, in part, the mission plan.

Decision
After we recognize surrounding road environments such as obstacles and traffic signals, we can predict the trajectories of moving objects and make decisions for mission and motion planning. This prediction and comprehensive decision components are under development in our autonomous-drive platform.

We adopt a state machine and machine learning intelligence for understanding, forecasting of a situation, and decisions according to the road scenarios. Our platform enables the drivers to supervise state of the vehicle and makes comprehensive decisions for following planners. Based on the results of this component, the suitable motion planner must generate trajectories depending on each scenario.

Planning
This component conducts trajectory planning on the basis of the comprehensive decision by above decision component. We breakdown path planning into mission and motion planning. We plan a global trajectory based on current location and destination, and then conduct basic local motion planning to determine the final path trajectory. We can use graph-search algorithms such as hybrid-state A* search [46], and trajectory generation [47] such as lattice-based algorithms [48]. We must choose the suitable planner with the decision component, depending on the road scenario.

Mission planning: Under the traffic rules, our mission planner uses a rule-based mechanism to autonomously assign the path trajectory, such as for lane change, merge, and passing. This mechanism is not completely autonomous in our platform. The high-definition 3D map contains static road information 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 planner is launched.

The basic policy of our mission planner is that we drive on the cruising lane throughout the route provided by a commodity navigation application based on the high-definition 3D map. The lane is changed only when our autonomous vehicle is passing the preceding vehicle or approaching an intersection followed by a turn.

Motion planning: The motion planner is a design knob for self-driving that corresponds to the driving behavior, which is not identical among users and environments. Hence, our platform currently provides only a basic motion-planning strategy so that we are building a high-level intelligence for the comprehensive decision on top of this motion planning component. Under the constraints of the current and goal state of the vehicle, the feasible trajectory must be dynamically generated from the travelable area based on the 3D map information, considering the surrounding objects and traffic rules.

In unstructured environments, such as parking lots, we provide graph-search algorithms, such as A* [49] and hybrid-state A* [46], to find a minimum-cost path to the goal with a cost map, as shown in Figures 1.4 (h) and (i). The graph-search algorithms can solve complex scenarios, although they sacrifice processing speed. In structured environments, such as roads and traffic lanes, on the other hand, the density of vertices and edges

is likely high and not uniform, which constrains the selection of feasible headings. We, therefore, use conformal spatiotemporal lattice-based algorithms to adapt the motion plan to the environment [50]. State-of-the-art research encourages the implementation of these algorithms [51]. Trajectories for obstacle avoidance and lane change must be calculated in real time with fast algorithms such as lattice-based algorithms [52], [50], and selected with an evaluation function, as shown in Figure 1.4 (j).

1.3.3 Actuation

We control our autonomous vehicle to follow the path generated by the motion planner as mentioned above.

Path following: We use the pure pursuit algorithm [53] to actuate our autonomous vehicle. According to the pure pursuit algorithm, we break down the path into multiple waypoints, which are discrete representations of the path. At every control cycle, we search for the close waypoint in the heading direction. We limit the search to outside of the specified threshold distance so that we can relax the change in angle in the case of returning onto the path from a deviated position. As shown in Figure 1.4 (k), the velocity and angle of the next motion are set to such values that bring the vehicle to the selected waypoint with predefined curvature.

We update the target waypoint accordingly until the goal is reached. The vehicle keeps following updated waypoints and finally reaches the goal. If the control of gas and brake stroking and steering is not aligned with the velocity and angle output of the pure pursuit algorithm because of some noise, the vehicle could temporarily get off the path generated by the motion planner. Although we currently use simple PID controller to handle the vehicle, parameters highly depend on the vehicle and the controller is not sufficient to control the vehicle smoothly. Localization errors could also cause the gap between the vehicle and outputs of the pure pursuit algorithm. As a result, the vehicle could come across unexpected obstacles. To cope with this scenario, our path follower ensures a minimum distance to obstacles, overwriting the given plan. Our motion planner also updates the waypoints, considering obstacles on heading lane area.

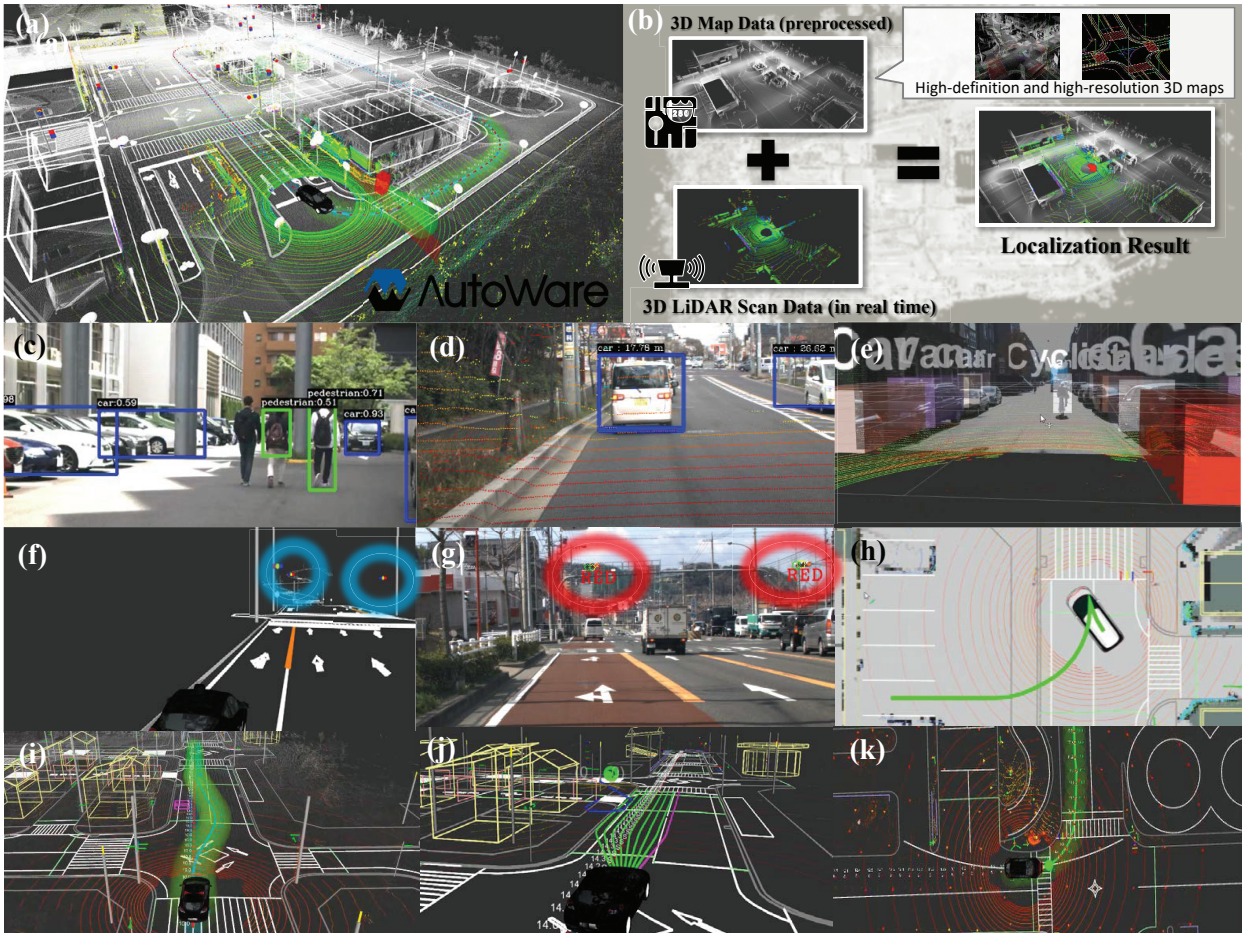

Fig. 1.4 Self-driving software packages in Autoware: (a) RViz visualization with high-definition and high-resolution geographical information, (b) NDT scan matching localization using a 3D map and a 3D LiDAR scan, (c) deep learning based object detection with SSD, (d) projection of the 3D point-cloud data onto the image, (e) sensor fusion between the LiDAR sensor and the camera with calibration, (f) obtainment of the traffic light positions from the 3D map, (g) traffic light recognition with the ROI, (h) trajectory generation for parking lots using the hybrid-state A* search, (i) trajectory generation for object avoidance using the hybrid-state A* search, (j) trajectory generation for object avoidance using the lattice-based algorithm, (k) steering angular velocity calculation for path following using the pure pursuit algorithm.

Acknowledgment

This paper was partly supported by Toyota Motor Company, eSOL, and Kalray. In addition, this research was partially supported by JST, PRESTO.

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, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Proc. of IEEE DATE*, 2014, pp. 1–6.

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

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

[10] 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*. IEEE, 2008, pp. 88–598.

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

[12] G. Chrysos and S. P. Engineer, "Intel Xeon Phi coprocessor (codename knights corner)," in *Proc. of the 24th Hot Chips Symposium*, 2012.

[13] M. Baron, "The single-chip cloud computer," *Microprocessor Report*, vol. 24, p. 4,

2010.
[14] D. Kanter and L. Gwennap, "Kalray clusters calculate quickly," *Microprocessor Report*, 2015.
[15] 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.
[16] 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.
[17] 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 International Conference on Robotics and Automation Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.
[18] "ROS.org," http://www.ros.org/.
[19] S. Cousins, "Exponential growth of ROS [ROS Topics]," *IEEE Robotics & Automation Magazine*, vol. 1, no. 18, pp. 19–20, 2011.
[20] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. of IEEE DAC*, 2001, pp. 684–689.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] "Autoware: Open-source software for urban autonomous driving," https://github.com/CPFL/Autoware.
[26] M. Magnusson, "The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection," Ph.D. dissertation, Örebro universitet, 2009.
[27] "Point Cloud Library (PCL)," http://pointclouds.org/.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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 *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2010, pp. 1–11.
[33] 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.
[34] "Compute Unified Device Architecture (CUDA)," https://developer.nvidia.com/cudazone.
[35] 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.
[36] "Caffe," http://caffe.berkeleyvision.org/.
[37] "OpenCV," http://opencv.org/.
[38] Y. Maruyama, S. Kato, and T. Azumi, "Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," in *Proc. of IEEE International Conference on Computer Design (ICCD)*, 2017.
[39] P. Biber and W. Straßer, "The normal distributions transform: A new approach to laser scan matching," in *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2003, pp. 2743–2748.
[40] 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 European Conference on Computer Vision (ECCV)*, 2016.
[41] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242
[42] 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.
[43] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, 2005, pp. 886–893.
[44] 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.
[45] 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.
[46] 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.
[47] B. Nagy and A. Kelly, "Trajectory generation for car-like robots using cubic curvature polynomials," *Field and Service Robots*, vol. 11, 2001.

[48] 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.

[49] 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.

[50] 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 International Conference on Robotics and Automation (ICRA)*, 2011, pp. 4889–4895.

[51] 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.

[52] 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.

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

Publication List