Distributed Processing for Automotive Data Stream Management System on Mixed Single- and Multi-core Processors

Jaeyong Rho
Ritsumeikan University, Japan

Takuya Azumi
Osaka University, Japan

Hiroshi Oyama
OKUMA Corporation, Japan

Kenya Sato
Doshisha University, Japan

Nobuhiko Nishio
Ritsumeikan University, Japan

ABSTRACT

Modern automotive systems incorporate a range of data from on-board sensors and outside the vehicle. This results in complex data processing and rising software development costs. To address these issues, we investigated the adaptation of a general-purpose data stream management system (DSMS) use in automotive applications. Existing DSMSs cannot be applied directly in automotive systems, since they are not designed for stream processing in a distributed environment with an architecture of mixed single- and multi-core processors. This makes it difficult to optimize the placement of communicating entities on multiple processors when the parallel processing of massive amounts of streamed data is required. In this study, we investigated distributed and parallel stream processing on mixed single- and multi-core processors for an automotive DSMS. To extend the automotive DSMS in a distributed environment and facilitate the testing of the real-time constraints of stream processing on the various placements of entities, we designed a framework to automatically generate execution files on multiple processors. Our experimental results validated an architecture of mixed single- and multi-core processors and demonstrated the effectiveness of the framework.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; D.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Design

.

Keywords

Data Stream Management System, Distributed environment

1. INTRODUCTION

In recent years, the number of automotive systems using on-board sensors, such as cameras, accelerometers, lasers, and radar, has increased. Advanced driver assistance systems, such as collision warning systems help prevent accidents by alerting the driver to oncoming vehicles or pedestrians, which requires the monitoring of the environment. Autonomous driving systems such as Google driverless car are also able to determine when to change lanes and can take appropriate actions to avoid obstacles in urban environments. These automotive systems acquire both own-vehicle information and surrounding-vehicle information through numerous sensors embedded in the vehicle.

The use of data acquired through external communication, such as vehicle-to-vehicle (V2V) and infrastructure-to-vehicle (I2V) communications, can significantly enhance the safety of automotive systems [11]. For example, automotive systems based on V2V communication periodically broadcast data from the on-board sensors to the surrounding vehicles. The exchange of data on factors such as position and speed with surrounding vehicles provides a more comprehensive model of the environment. This allows for more precise decision-making by the automotive systems, especially at intersections. Data collected from outside the vehicle can therefore enable the detection of potentially dangerous situations, which on-board sensors alone cannot do.

Research has been conducted on adaptation of data stream management systems (DSMS) for automotive embedded systems. This is aimed at reducing the complexity of data processing and lowering the software development costs while increasing the amount of data available to the vehicle [21, 9, 12]. In current systems, automotive data from on-board sensors and from outside the vehicle are processed and managed individually in separate electronic control units (ECUs). The duplication of data processing over multiple applications and associated software development costs increase with the amount of data in the automotive system. To address these issues, researchers have focused on DSMSs for automotive system applications that can process streamed data at low latency using shared data processing over multiple applications.

Existing DSMSs [3, 4, 10, 5] are designed to run on general-purpose computer, and mainly target applications in net-

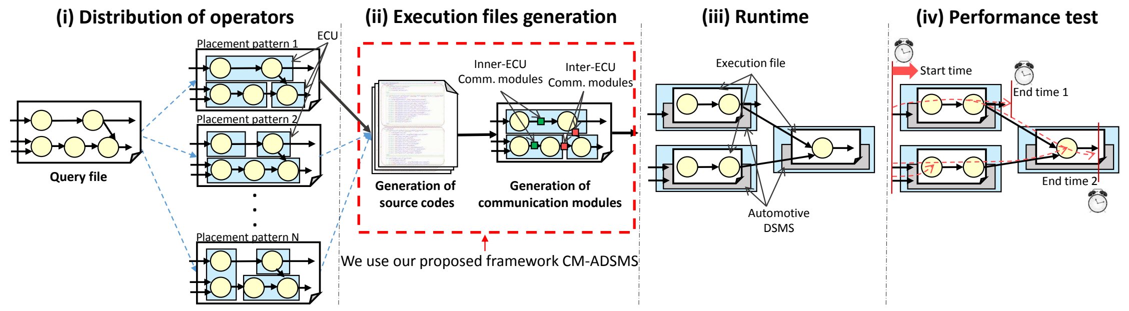

Figure 1: System model for an automotive DSMS development using our framework.

work monitoring and financial analysis, which require abundant resources for stream processing. Such systems are challenging to apply in an automotive context, with strict resources and real-time constraints, which existing DSMSs were not designed to handle.

The European Telecommunications Standards Institute (ETSI) specification for a collision warning system using V2V communication requires that the vehicle is able to receive and process at least 1,000 data points per second [2]. By processing V2V data, i.e., the position, speed, and direction of the surrounding vehicles, the automotive DSMS can compute the probability of collision with each surrounding vehicle. If this processing could be carried out with a larger number of surrounding vehicles, the automotive DSMS could detect an imminent collision at higher precision. However, processing all these data sufficiently and quickly to prevent a collision is difficult without increasing the number of processors or the frequency of a single-core processor. This is undesirable in automotive applications. The use of a multi-core processor is therefore regarded as an effective solution to parallel stream processing for an automotive DSMS with modest power consumption and cost.

Vehicle software systems comprise multiple ECUs, requiring an automotive DSMS to be deployed on multiple ECUs when acquiring data from sensors and running application software. Existing general-purpose DSMSs are mainly implemented on an architecture comprising only single-core processors or one single multi-core processor. Accordingly, it is impossible to determine which processor architecture would enable the optimum collision probability computation in an automotive DSMS.

The developers of automotive DSMSs run development phase tests on a range of processor architectures and DSMS operators corresponding to communicating entities to ensure that the stream processing can function reliably and the performance is maximized. Stream processing is realized by executing operators that compute using tuples and appending the resulting tuples into one or more subsequent streams. When an operator needs to use data generated by a precedent operator, the communication required between the two operators is either an intra-ECU communication or an inter-ECU communication. A developer must consider the placement of the two consecutive operators, because the communication method needed will depend on the operator placement. Thus, writing the codes while considering the operator placement patterns on multiple ECUs requires substantial manual input.

In this study, we adapted an automotive DSMS in a distributed environment of mixed single- and multi-core processors. We compared the stream processing performance in terms of average execution time with an architecture comprised only single-core processors. In addition, we developed CM-ADSMS, a framework that generates a proper communication module between the two operators for the automotive DSMS using TOPPERS Embedded Component System (TECS) [6, 7]. TECS provides auto-generation functionality, and a module that can easily be inserted between the two operators. This can facilitate the implementation of a communication module between the two operators, considering the location of the operators on the ECUs.

Contribution:

• We investigated the application of multi-core processors to an automotive DSMS in which stream processing was distributed accross multiple ECUs with single-core processors. This can help tackle the challenge of operator placement in a distributed architecture of mixed single- and multi-core processors.

• We adapted the automotive DSMS to a distributed environment using CM-ADSMS to automatically generate communication modules and execution files. This allowed the automotive DSMS to test real-time constraints on multiple ECUs with various operator placement patterns.

Organization: The remainder of this study is organized as follows. Section 2 presents the basic technologies of our system. The distributed automotive DSMS is described in Section 3. Section 4 evaluates the effectiveness of the proposed architecture with a mix of single- and multi-core processors. Related work is discussed in Section 5, and our conclusions are reported in Section 6.

2. SYSTEM MODEL

This paper targets a data stream management system for automotive systems (Automotive DSMS). A development of the automotive DSMS using our framework shown in **Figure** 1 takes the following steps: (i) A large number of communicating entities (called operators) in a query are distributed on multiple ECUs. The automotive DSMS developers need to decide the operator placement on ECUs, that enables to meet real-time constraints (e.g., meeting all stream processing with their end-to-end deadlines, processing at least 500 stream data from outside the vehicle in one second, and so on), and the operator distribution information is written in the query file. (ii) Using our framework described in Section 3, the query file is transformed into source codes and inserts adequate communication modules (e.g., Inner-ECU or Inter-ECU comm. module) between two operators automatically according to a relationship between them. One or more than one execution file for each ECU are outputed, and (iii) execution files generated in Step (ii) are installed on ECUs that execute stream processing. (iv) The automotive DSMS developers perform stream processing test in terms of

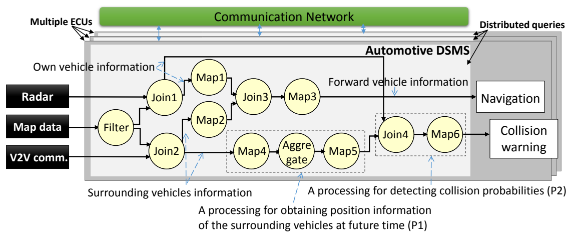
Communication Network
Multiple ECUs
Distributed queries
Automotive DSMS
Own vehicle information
Forward vehicle information
Radar
Map data
V2V comm.
Filter
Join1
Join2
Map1
Map2
Join3
Map3
Map4
Aggre gate
Map5
Join4
Map6
Navigation
Collision warning
Surrounding vehicles information
A processing for detecting collision probabilities (P2)
A processing for obtaining position information of the surrounding vehicles at future time (P1)

Figure 2: Adapting DSMS to automotive systems and sample query for stream processing.

real-time constraints. This development cycle (i)∼(iv) may be repeated until the optimum operator placement is found.

2.1 Automotive DSMS

Here, the automotive DSMS using an DSMS for automotive embedded systems (eDSMS) [20, 12] is described.

2.1.1 *Basic Concept of Stream Processing*

A data stream management system (DSMS), a stream processing system, processes **stream data** which are generated continuously in large volumes in real-time using a query. The query is registered with the system and executes continuously as new stream data arrives. A **query** is composed of one or more operators, and two consecutive operators are connected with a **stream queue**. An **operator** is a communicating entity which computes using a tuple and outputs the result tuples to the next stream queue. A **tuple** is a set of data values. For instance, a circle colored in yellow and an arrow in a query file as shown in **Figure**s 1 and 2 indicates an operator and a stream queue, respectively.

In stream processing, a precedent operator executes and produces one or more result tuples using input tuples. The result tuples are delivered into a subsequent operator or an application through a stream queue. Operators, such as Map, Filter, Join, and Aggregate, used for automotive DSMS are based on operators of Borealis which is a general-purpose DSMS [3]. The functionalities of operators and other details are described in [21, 12].

2.1.2 *Data Integration Architecture Based on DSMS*

In current architecture of automotive systems, data retrieved from on-board sensors and outside the vehicle is processed individually in each application program embedded in software on ECU. It is because that automotive suppliers provide a product as a set of software and the related sensor. Therefore, similar data processing can be duplicated over multiple applications located in different ECUs. Furthermore, in the case that if system properties (e.g., a type of sensor device) are needed to be changed, large parts of the application programs are required to be modified. To tackle these issues, the automotive DSMS has been developed on the basis of the data integration architecture [21, 12] shown in Figure 2. Applications can be separated from sensor device part because data processing using a variety of sensors are defined in the automotive DSMS instead. Data processed in the automotive DSMS can be accessible in a location-transparent manner from multiple applications. In addition, changing system properties becomes easier than the current architecture since application programs are independent from specific sensors; thus, the cost of automotive software development can be reduced.

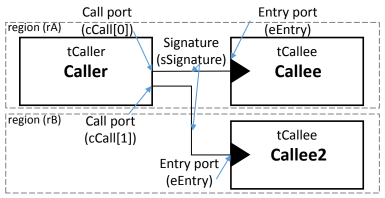

Figure 3: Example of cells in TECS.

```
 1 [to_through(rB,Plugin,"Args"),linkunit]
 2  region rA {
 3    cell tCaller Caller {
 4      cCall[0] = Callee.eEntry;
 5      cCall[1] = rB::Callee2.eEntry; };
 6    cell tCallee Callee { };
 7  };
 8  region rB {
 9   cell tCallee Callee2 { };
10  };
```

Figure 4: Example of a build description.

2.2 TECS

We use TECS (TOPPERS Embedded Component System) [6, 7] for the development of a framework of CM-ADSMS described in Section 3.

2.2.1 *Basic Concept*

A *cell* is a component instance in TECS, and it has one or more than one *entry port* and *call port* in respectively. The *entry port* of a *cell* is an interface to provide its function to other *cell*s. The *call port* of a *cell* is an interface to use a function defined in other *cell*s. The *signature*, a set of function declarations, is allocated to the *entry port* and *call port*. A *celltype* is the definition of a *cell*. For instance, the *Caller cell* is the *cell* of the *tCaller celltype*, and the *Callee cell* is the *cell* of the *tCallee celltype* in **Figure** 3. **Figure** 4 shows an example of the *Caller cell* connected to the *Callee cell* located in the same region with the *Caller cell*, and the *Callee2 cell* located in a different region as the *Caller cell*. The region corresponds to a partition where *cells* are allocated in TECS. For instance, an *rA* and an *rB* are the names of the regions and are defined with the "*region*" keyword. To use the function defined in the *Callee cell* from the *Caller cell*, *cCall[0]* of the *Caller cell* needs to be connected to *eEntry* of the *Callee cell* (Line 4).

2.2.2 *Through Plugin*

TECS *through* plugin [6] is used for inserting a *cell* between two *cells*. In addition, in the case that two *cells* are allocated to different regions, TECS provides *to_through* plugin which is one kind of *through* plugin. We can adapt the *to_through* plugin to the TECS generator by writing the "*to_through*" keyword in a TECS component description file (.cdl).

For instance, **Figure** 4 shows an example of a case where *cCall[1]* of the *Caller cell* in region *rA* is connected to *eEntry* of the *Callee2 cell* in region *rB*. A component, such as a component to print log messages, between the *Caller cell* and the *Callee2 cell* can be inserted by writing the "*to_through*" keyword before the description of a region *rA* (Line 1). "Plugin" in Line 1 indicates a type of plugin which defines the inserted *cell*, and we can set the argument for the plugin if it is needed ("Arg" in Line 1).

3. DISTRIBUTED AUTOMOTIVE DSMS

This section describes the data stream processing required for parallel execution in the automotive DSMS and three architectures for parallel processing. We present a proposed framework for an automotive DSMS, the CM-ADSMS, and the development flow using CM-ADSMS.

3.1 Example of Parallel Stream Processing

Figure 2 presents a data processing query shared by a collision warning system and a navigation system. To avoid a crash, it is necessary to predict the positions of both the vehicle and the surrounding vehicles before the time to collision (TTC). In this query, the vehicle calculates its own position using data from its on-board sensors and a map database, and the position of surrounding vehicles is calculated using data from V2V communication and a map database. By comparing its position and the position of the surrounding vehicles at a future time (*Join4*), the automotive DSMS can compute the probability of collision with each surrounding vehicle. Only the information about a vehicle with a high collision probability is provided to the collision warning system using *Map6*.

To avoid collision with surrounding vehicles, the collision probabilities must be processed from all the streamed data received from the V2V communication, before the TTC. However, following the ETSI specification [2] for a collision warning system utilizing V2V communication, the vehicle must be able to receive and process at least 1,000 data points per second. This makes it difficult to complete the calculation of collision probability within the time constraints without increasing the number of processors. The prediction of surrounding vehicle positions at a future time (*Map4, Aggregate, Map5* in **Figure** 2) must therefore be executed in parallel in the automotive DSMS.

3.2 Architecture for Parallel Processing

Automotive systems require stream processing in which the query is distributed and executed across multiple ECUs, because the on-board sensors and application software run on several ECUs connected with a network bus. Operators and stream queues in the query must be allocated to multiple ECUs to satisfy the requirements for parallel processing and end-to-end timing constraints while considering the limited resources available.

The parallel processing of stream data can be achieved using an architecture consisting only of single-core processors, or one that uses a multi-core processor mixed with single-core processors. The query shown in **Figure** 2 must be divided into multiple sub-queries, and each sub-query is allocated to one of the multiple ECUs. Operators, stream queues, and applications must therefore be distributed across the ECUs. To allow the parallel execution of stream processing, **Figure** 5 shows the three possible placement patterns of the operators and applications. Many other operator placement patterns would be possible, but we selected these three patterns to investigate whether network latency between the processors or the latency in processing burst stream data in the single network controller of a multi-core processor had a larger influence on the performance of the parallel stream processing. Each placement pattern was investigated using the example of predicting the position of a surrounding vehicle (P1 in **Figure** 2).

The operators for stream processing to be executed in parallel were allocated to each single-core processor. For example, P1 could be executed in parallel using four single-core processors as shown in **Figure** 5 (A). This is a simple placement pattern, in which the operators required to be executed in parallel are allocated on each single-core processor (i.e., *ECU2 and ECU3*). However, increasing the number of ECUs is undesirable in the automotive field since this can lead to significant cost increases and longer development time.

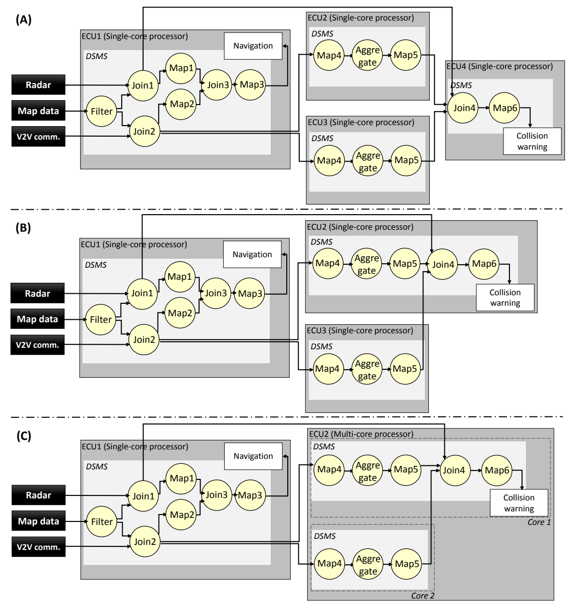
(A)
ECU1 (Single-core processor)
DSMS
Navigation
Radar
Map data
V2V comm.
Filter
Join1
Join2
Map1
Map2
Join3
Map3
ECU2 (Single-core processor)
ECU3 (Single-core processor)
Map4
Aggre gate
Map5
ECU4 (Single-core processor)
Join4
Map6
Collision warning
(B)
(C)
ECU2 (Multi-core processor)
Core 1
Core 2

Figure 5: Three patterns for distributed stream processing.

Second, the operators for combining the tuples from each operator being executed in parallel were allocated to either of the single-core processors. For example, the processing required to compare with the vehicle's own position and with those of the surrounding vehicles (P2 in **Figure** 2) could be co-located with the application on *ECU2*, as shown in **Figure** 5 (B). Only three single-core processors were needed, fewer than those in pattern (A). The processing time for each *ECU2* and *ECU3* was different, because P1 and P2 were co-located in *ECU2*. To execute P1 in parallel when allocated to *ECU2* and *ECU3*, the number of input tuples generated from *Join2* and provided to *Map4* on each *ECU2* and *ECU3* must be considered.

Third, the operators required for the processing to be executed in parallel with the application were co-located in a multi-core processor, as shown in **Figure** 5 (C). For example, P1 had to be allocated to each core of the multi-core processor, while P2 was allocated to either core. Using an architecture that utilizes the multi-core processor allowed the number of ECUs and software development cost to be reduced compared with those in patterns (A) and (B).

3.3 Design of CM-ADSMS

To adapt the automotive DSMS in the distributed environment and to reduce the developers' efforts, we have designed and developed the framework of CM-ADSMS which automatically generates the communication modules between operators and execution files for the automotive DSMS. We used TECS for devoloping the framework because TECS provides auto-insert functionality between two components, and we can improve the reusability of the communication

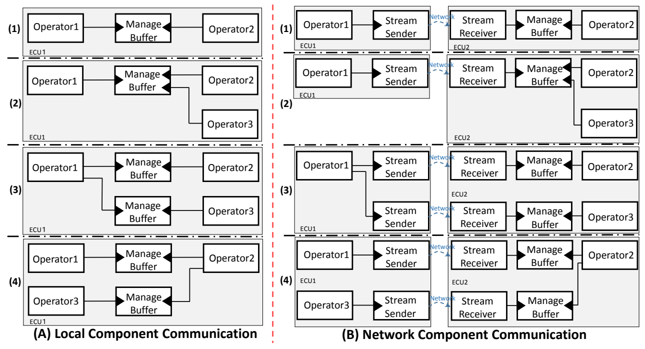

Figure 6: Design of local and network component communication.

modules by defining these modules as components.

3.3.1 *Local Component Communication*

It is the communication for transmitting a tuple between operators that are allocated to the same ECU. The local component communication is realized by using a *ManageBuffer* cell. The *ManageBuffer cell* is a component which manages a stream queue, and the stream queue is comprised of a ring buffer to maintain recent data. The *ManageBuffer cell* provides two functions to enqueue a tuple into a stream queue and to dequeue a tuple from the stream queue.

The local component communication can be designed in four patterns as shown in **Figure** 6. In the relation between one operator and another operator, one *ManageBuffer cell* is placed between two operators (**Figure** 6(A)-(1)). In the case of a result tuple from one operator is shared among two operators, one *ManageBuffer cell* is located between these operators, and the result tuple is copied to be provided for two operators (**Figure** 6(A)-(2)). When an operator such as a filter produces two different tuples according to a condition and each tuple is provided for two subsequent operators, two *ManageBuffer cell*s are required for each stream queue to manage each result tuple (**Figure** 6(A)-(3)). If an operator such as Join combines two input stream data into one stream data, two *ManageBuffer cell*s are placed and an operator which combines two input stream data dequeues from the two *ManageBuffer cell*s (**Figure** 6(A)-(4)).

3.3.2 *Network Component Communication*

It is for transmitting a tuple from an operator to an another operator, where these operators are allocated on the different ECUs. The network component communication is realized by using three components: the *StreamSender cell*, the *StreamReceiver cell*, and the *ManageBuffer cell*. The *StreamSender cell* is the component which provides functions for sending a tuple to a target ECU by each TCP and UDP mode. In the *StreamReceiver cell*, functions for receiving a tuple from a specific port by each TCP and UDP mode are provided. For the network component communication, we implement the components using TCP/IP and UDP/IP, because automotive Ethernet is currently being discussed and researched for a next generation in-vehicle networking standard within the automotive domain [14]. Meanwhile, CAN and FlexRay, which are commonly employed as the in-vehicle network, can also be implemented by switching into a communication component using TECS.

The network component communication can also be designed in four patterns as shown in **Figure** 6. In the relation between one operator and another operator, the *StreamSender cell* needs to be placed on the sender side ECU, and the *StreamReceiver cell* and the *ManageBuffer cell* is placed on the receiver-side ECU. The received tuples from *StreamReceiver cell* are enqueued into a stream queue managed in the *ManageBuffer* (**Figure** 6(B)-(1)). The rest cases are designed in the same manner of the local component communication .

```
1 [to_through(rECU2,NetworkPlugin,"targetAddr=192.168.
227.120,portNo=9000,connectType=TCP),linkunit]
2 region rECU1 {
3   cell tMAPOperator MAPOperator {
4     cReadBuffer  = ManageBuffer1.eReadBuffer;
5     cWriteBuffer = rECU2::ManageBuffer2.eWriteBuffer;
6 };
```

Figure 7: Example of the *to_through* keyword.

3.3.3 *Auto-generation Functionality*

The *ManageBuffer cell* is inserted between every two consecutive operators regardless to whether the local component communication or the network component communication is needed. In the case that an operator sends a tuple through the network communication to a subsequent operator, the *StreamSender cell* and the *StreamReceiver cell* are required to be inserted between them. We developed a *NetworkPlugin* which is one kind of *to_through* plugin in TECS, and it is used in (ii) Step of **Figure** 1. The *NetworkPlugin* automatically inserts the *StreamSender cell* and the *StreamReceiver cell* between two operators placed on the different ECUs. **Figure** 7 shows an example of inserting the *StreamSender cell* and *StreamReceiver cell* automatically using *to_through* keyword (Line 1).

4. EVALUATION

The performance of the distributed stream processing and auto-generation code performance of the proposed framework were evaluated. The performance of the distributed stream processing was conducted by comparing end-to-end execution times for one single-core processor, the distributed architecture comprising only single-core processors, and the distributed architecture with a mix of single- and multi-core processors. We evaluated CM-ADSMS in terms of the volume of source codes that were automatically generated.

4.1 Experimental Setup

To compare the distributed stream processing of the three architectures, we used a simple query with three operators: *Filter, Map, and Join*. A result tuple from the *Filter* operator was used as an input tuple to the *Map* operator, and we assumed that the *Map* operator was required to be executed in parallel. The *Join* operator outputs a tuple when it receives a result tuple from either of the two *Map* operators.

To allow stream processing of the *Map* operator to be executed in parallel, the query operators were placed on multiple ECUs following the two patterns shown in **Figure** 8 described by TECS component description [6]. The two *Map* operators to be executed in parallel were placed on each ECU (*ECU2* and *ECU3*), and the subsequent *Map* operator, i.e., the *Join* operator, was placed on *ECU2* in placement (A). Two *Map* operators were placed on each core in *ECU2*, and the *Join* operator was placed on *core 1* in placement (B). We also placed all operators on one single-core processor and ran experiments to compare end-to-end execution times

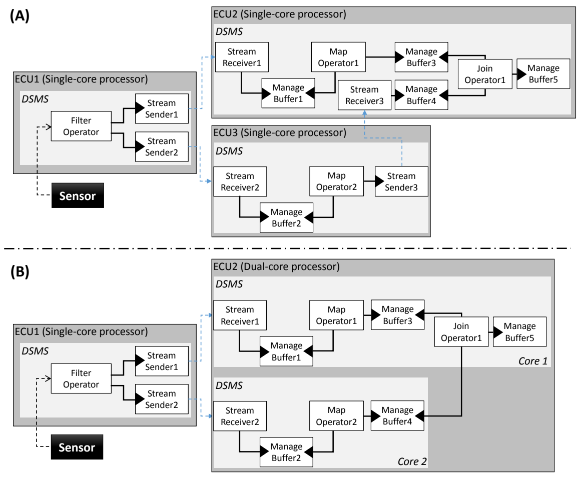

Figure 8: Two architectures for the evaluation.

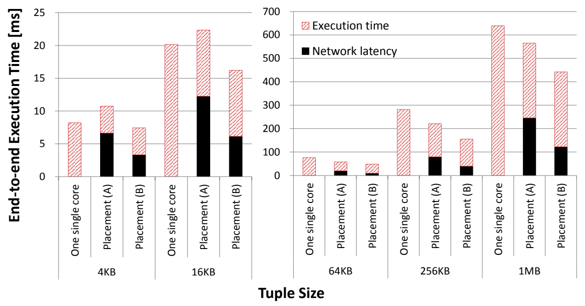

Figure 9: One single-core processor vs. only single-core processors vs. mixed with a dual-core processor.

of the three placements. The execution time for processing one tuple was the time from inputting a tuple to the *FilterOperator* in *ECU1* to outputting a result tuple from the *JoinOperator1* in *ECU2*.

The experiments were performed using three processors: (i) a single-core processor (700 MHz ARM1176JZF-S with 512MB of RAM) and (ii) two dual-core processors (1 GHz ARM Cortex-A7 with 1GB of RAM), running Linux OS (Fedora). In (A), we used a single-core processor on *ECU1* and a dual-core processor on each *ECU2* and *ECU3*. Since the specification of each *ECU2* and *ECU3* must be the same as *ECU2* in (B), the two dual-core processors on *ECU2* and *ECU3* in (A) were run in single-core processor mode. Ethernet was used for communication between the ECUs, since automotive Ethernet is considered as a next generation in-vehicle network. CAN and FlexRay, the most commonly employed in-vehicle networks, can be implemented as the communication mechanism by switching to their components using TECS, as explained in Section 3.

4.2 End-to-end Execution time

We evaluated the average end-to-end execution time with increasing tuple size from 4KB to 1MB in the three operator placements described above. 100 tuples were generated and we calculated their average end-to-end execution times.

As shown in **Figure** 9, the average end-to-end execution times in placement (B) were lower than those in the one single-core processor placement and placement (A) across the range from 4KB to 1MB. This shows that a distributed architecture with mixed single- and dual-core processors can

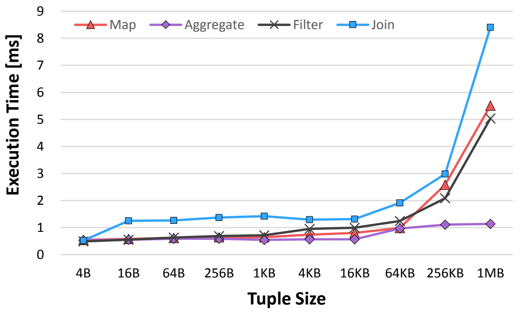

Figure 10: Execution time of operators.

reduce the end-to-end execution time for each tuple, compared with the other two placements.

When comparing parallel stream processing in placements (A) and (B), the average end-to-end execution time in placement (B) was shorter than that in placement (A) over all tuple sizes. For experimental results in the one single-core processor placement and placement (A), and shorter average end-to-end execution times were recorded with the one single-core processor placement than with placement (A) until 16KB. In contrast, placement (A) processed streamed data with shorter end-to-end execution times than that for the one single-core processor placement from 64KB to 1MB of tuple sizes, as shown in **Figure** 9.

Figure 10 gives the execution time of the four operators, i.e., *Map*, *Aggregate*, *Filter*, and *Join*, used in the automotive DSMS. We implemented a simple calculation for each operator. *Map* multiplied a value in the *value* field of a tuple by two, and the *Aggregate* operator calculated an average speed value from tuples with the same ID. The *Filter* operator filtered an input tuple according to its even or odd ID number, while *Join* operator combined the fields of two input tuples and outputs one combined tuple into a stream queue.

We input 100 tuples to each operator to estimate an average execution time for each operator. As shown in **Figure** 10, the execution times across the range from 4 bytes to 16 KB did not vary greatly, while a tuple size of 16 KB, execution time of the *Map* and *Filter* operators increased with tuple size. The execution time of the *Join* operator increased more drastically than that for the other three operators, whereas, the execution time of the *Aggregate* operator did not vary greatly with tuple size. A different trend in the execution time of each operator was therefore observed.

4.3 Performance of the Framework

We evaluated the performance of the CM-ADSMS using *NetworkPlugin*, by which the *StreamSender* and *StreamReceiver* cells required to communicate between the two consecutive operators allocated to the different ECUs were inserted automatically.

The total number of lines of source codes required to output one or more execution files for each ECU are compared. We measured the auto-generated source codes as the increase in the number of pairs of communicating operators that needed data transmission through the network by generating 10 random operator, and allocating them randomly to one ECU with zero communicating operators, two ECUs with one communicating operator, three ECUs with two communicating operators, and so on.

With no communicating operators (horizontal value of zero in **Figure** 11), 2,843 lines of source codes were automatically generated. When the horizontal axis value was

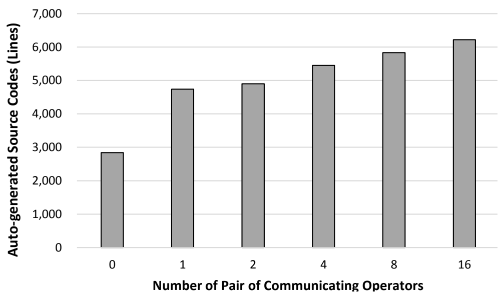

Figure 11: Generated Lines of the Source Codes.

one and operators were distributed on two ECUs, 4,721 lines were automatically generated. This suggests that the source code needed to build a connection between the two operators on different ECUs corresponded to 1,878 lines. This source code must be implemented by automotive DSMS developers when not using an auto-generation framework such as CM-ADSMS. The volume of auto-generated source code increased with the number of communicating operator pairs and ECUs. As a result, a considerable amount of source code could be automatically generated for automotive DSMS developers, without considering the operator placement patterns on the ECUs.

4.4 Discussion

The results of the first experiment show that the end-to-end execution times of stream processing in the architecture using a dual-core processor can be significantly reduced by distributing the massive amount of streamed data across single- and multi-core processors. The gap in the end-to-end execution times of placements (A) and (B) in **Figure** 9, was incurred by the network latency between the two processors. This network latency was incurred in transmitting a tuple between the *Map2* operator on *ECU2* and the *Join1* operator on *ECU3* in placement (A). In contrast, the data transmission through the network was lower in placement (B), since the network transmission between *ECU2* and *ECU3* was replaced by a core transmission from *core2* to *core1* as shown in **Figure** 8. The latency of the core transmission of streamed data is much smaller than that of network transmission.

We confirmed that the end-to-end execution time for distributed stream processing is influenced largely by the network latency between the processors, rather than the latency in processing burst stream data in the single network controller of the multi-core processor. Thus, the greater parallel processing of massive amounts of streamed data is possible in an architecture of mixed single- and multi-core processors. We also confirmed that developers need to consider operator types when optimizing operators in multiple ECUs for real-time and parallel stream processing.

Our second experiment automatically generated large amounts of source codes, even though the queries used for the evaluation were simple ones. Therefore, we expect that still more significant amounts of source codes will be generated in real development environments, irrespective of the operator placement patterns of the multiple ECUs. This is because the number of operators and ECUs and the complexity of the operator precedence relations are significantly greater in a real development environment.

Our experimental results can help reduce the operator placement search problem by excluding placements poorly suited to parallel stream processing. By Assuming that m operators are required to be placed on N processors whose specifications are different from each other, the total number of searches for the best operator placement is N^m. A large number of real-time stream processing test can be reduced by the experimental results, and can be conducted for a range on placements using CM-ADSMS.

5. RELATED WORK

There are many prototypes of the DSMSs for general-purpose systems such as Borealis [3], TelegraphCQ [10], and STREAM [5]. Queries registered in these systems can be changed to other queries or modified during the execution time according to dynamic conditions. To employ these dynamic changes during the execution time, many modules are required to be embedded in the execution environment, and large amount of memory is required. However, the existing DSMSs do not meet the requirement of the automotive systems since the automotive systems generally have low-speed and low-capacity memories as well as low CPU performance due to strict cost limitation requirements.

Based on these general-purpose DSMSs, many operator scheduling algorithms have been researched [8, 15, 18]. Babcock et al. [8] introduced the chain scheduling algorithm, which targets applications including networking systems and sensor networks. The purpose of this algorithm is to minimize run-time system memory usage. Tick scheduling strategy [15] is used to minimize deadline miss ratio by reducing scheduling overheads and to maximize throughput. Preemptive rate-based scheduling strategy [18] has been proposed to maximize throughput over stream data. However, these existing algorithms do not explicitly target a distributed environment consisting of multiple processors. Moreover, these algorithms do not consider strict timing constraints required for use in automotive systems.

To satisfy the strict timing constraints of DSMS, several studies have been presented [13, 17, 19]. Li et al. introduced the OP-EDF scheduling algorithm [13], in which a task with shorter absolute deadline is assigned higher priority to minimize deadline miss ratio. Schmidt et al. [17] introduced the RM scheduling algorithm based on the rate monotonic algorithm and a hard real-time scheduling strategy. Queries can be processed in accordance with their timing constraints in RTSTREAM [19]. However, these methods cannot be applicable to a distributed environment since they are assumed to be implemented in the uni-processor environment.

As methods for parallel stream processing in a distributed environment, many mechanisms are presented. In [16], a processing route of each input tuple is dynamically determined by selecting the next operator to process in the execution time. A method to manage the load across the multiple processing ECUs by migrating operators to lightly loaded ECUs is presented in [22]. However, these existing algorithms are assumed an architecture consisting of only single-core processors or only multi-core processors; thus, they cannot be adapted to the automotive systems comprising of mixed with single-core and multi-core processors directly. Furthermore, these algorithms do not consider strict timing constraints required for use in automotive systems.

To employ the general-purpose DSMS to the automotive systems, StreamCars [9] and eDSMS [21, 12] have been developed. These automotive DSMSs integrate data retrieved from a variety of sensors, and they enable to change system properties such as sensors and algorithms used in applications easily. However, StreamCars [9] do not consider the strict timing constraints and cannot be applied to distributed environments. eDSMS [21, 12] does not consider the distributed environments consisting of single-core and multi-core processors. As a result, it cannot be known whether multi-core processors are applicable to the automotive DSMS in distributed environments.

6. CONCLUSIONS

Distributed stream processing on an architecture comprising a mix of single- and multi-core processors was proposed and evaluated. We conducted experiments to compare stream processing in three cases: i) processing all stream data on one single-core processor, ii) dividing the stream data between the multiple single-core processors, and iii) dividing the stream data between the cores in a single multi-core processor. The results suggest that for the range of tuple size between 4KB and 1MB, an architecture of mixed multi-core processors has a shorter average end-to-end execution times for parallel stream processing than a one single-core processor or an architecture comprising only single-core processors. We also confirmed that placing all operators in one single-core processor can reduce the execution time compared with that in a distributed architecture of networked single-core processors until 16KB. At tuple sizes larger than 16KB, a distributed architecture of networked single-core processors processes streamed data faster than one single-core processor. These results can help developers to place operators in distributed processor architecture for parallel stream processing in automotive DSMS applications.

We also recommend the use of the CM-DSMS framework to extend automotive DSMS to a distributed environment and reduce the implementation challenges facing automotive DSMS developers. This framework reduces the severity of the operator placement problem and allows the the testing of the real-time constraints on stream processing across various operator placement patterns.

In future work, we plan to apply AUTOSAR [1] to the automotive DSMS and extend CM-ADSMS to the automatic generation of all ECU execution files from a query for a variety of operator placement patterns. The extended CM-ADSMS should help developers perform the testing of a variety of operator placements with much reduced effort.

7. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 15H05305.

8. REFERENCES

[1] AUTOSAR development partnership. available from <http://www.autosar.org/index.php> (accessed 2015-08-15).

[2] Intelligent Transport Systems (ITS); V2X Applications; Part 3: Longitudinal Collision Risk Warning (LCRW) Application Requirements Specification. Nov. 2013.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, Jan. 2005.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[5] D. P. Arvind, A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (demonstration description). 26:2003, Jun. 2003.

[6] T. Azumi, H. Oyama, and H. Takada. Memory Allocator for Efficient Task Communications by Using RPC Channels in an Embedded Component System. In *Proc. of the 12th IASTED International Conference on Software Engineering and Applications*, pages 204–209, Nov. 2008.

[7] T. Azumi, T. Ukai, H. Oyama, and H. Takada. Wheeled Inverted Pendulum with Embedded Component System: A Case Study. In *Proc. of the 13th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 151–155, May 2010.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[9] A. Bolles, H.-J. Appelrath, D. Geesen, M. Grawunder, M. Hannibal, J. Jacobi, F. Koster, and D. Nicklas. StreamCars: A New Flexible Architecture for Driver Assistance Systems. In *Proc. of the Intelligent Vehicles Symposium (IV)*, pages 252–257. IEEE, Jun. 2012.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertan World, Jan. 2003.

[11] K. Golestan, S. Seifzadeh, M. Kamel, F. Karray, and F. Sattar. Vehicle Localization in VANETs Using Data Fusion and V2V Communication. In *Proc. of the 2nd ACM International Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications (DIVANet)*, pages 123–130, Oct. 2012.

[12] S. Katsunuma, S. Honda, K. Sato, Y. Watanabe, Y. Nakamoto, and H. Takada. Real-Time-Aware Embedded DSMS Applicable to Advanced Driver Assistance Systems. In *Proc. of the IEEE 33rd International Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 106–111, Oct. 2014.

[13] X. Li, Z. Jia, L. Ma, R. Zhang, and H. Wang. Earliest Deadline Scheduling for Continuous Queries over Data Streams. In *Proc. of the International Conference on Embedded Software and Systems (ICESS)*, pages 57–64. IEEE Computer Society, 2009.

[14] H.-T. Lim, L. Völker, and D. Herrscher. Challenges in a Future IP/Ethernet-based In-car Network for Real-time Applications. In *Proc. of the 48th Design Automation Conference (DAC)*, DAC, pages 7–12. ACM, Jun. 2011.

[15] Z. Ou, G. Yu, Y. Yu, S. Wu, X. Yang, and Q. Deng. Tick Scheduling: A Deadline Based Optimal Task Scheduling Approach for Real-Time Data Stream Systems. In *Proc. of the 6th International Conference on Advances in Web-Age Information Management*, volume 3739 of *Lecture Notes in Computer Science*, pages 725–730. Springer, Nov. 2005.

[16] A. A. Safaei, A. Sharifrazavian, M. Sharifi, and M. S. Haghjoo. Dynamic Routing of Data Stream Tuples Among Parallel Query Plan Running on Multi-core Processors. *Distributed and Parallel Databases*, 30(2):145–176, Apr. 2012.

[17] S. Schmidt, T. Legler, D. Schaller, and W. Lehner. Real-time Scheduling for Data Stream Management Systems. In *Proc. of the 17th Euromicro Conference on the Real-Time Systems (ECRTS)*, pages 167–176, Jul. 2005.

[18] M. Sharaf, P. Chrysanthis, and A. Labrinidis. Preemptive rate-based Operator Scheduling in a Data Stream Management System. In *Proc. of the 3rd ACS/IEEE International Conference on Computer Systems and Applications*, Apr. 2005.

[19] Y. Wei, S. Son, and J. Stankovic. RTSTREAM: Real-Time Query Processing for Data Streams. In *Proc. of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 141–150, Apr. 2006.

[20] M. Yamada, K. Sato, and H. Takada. Implementation and Evaluation of Data Management Methods for Vehicle Control Systems. In *Proc. of the IEEE Vehicular Technology Conference (VTC Fall)*, pages 1–5, Sep. 2011.

[21] A. Yamaguchi, Y. Nakamoto, k. Sato, I. Yoshiharu, Y. Watanabe, S. Honda, and H. Takada. AEDSMS: Automotive Embedded Data Stream Management System. In *Proc. of the 31st International Conference on Data Engineering (ICDE)*, Apr. 2015.

[22] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *Proc. of the 14th Int'l Conf. Cooperative Information Systems*, volume 4275, pages 54–71, Dec. 2006.