Performance Estimation for Embedded Many-core Processor with Software/Hardware Performance Description

Yutaro Kobayashi[1,a)] Hiroshi Fujimoto[2] Takuya Azumi[1]

Abstract: Embedded systems, such as automotive systems, are becoming larger and more complex, requiring high computing power and low power consumption. To meet these requirements, multi-/many-core processors and MATLAB/Simulink are increasingly used. Moreover, to support multi-/many-core processors, model-based parallelization tools have been developed. However, the problem of model-based parallelization tools estimated time for each Simulink block has a large error compared to the execution time. Moreover, it is that only hardware information is used to estimate the execution time of parallelized C code. Therefore, the estimation method is proposed to improve the execution time of each Simulink block in comparison with existing methods. Also, a new estimation method is proposed that uses both software and hardware information to estimate the overall execution time. The execution time of the Simulink model is estimated by the conventional method, and the proposed method is measured and compared with the actual execution time to evaluate the proposed method. The experimental results show that the execution time of the parallelized model can be reduced by improving the estimation execution time of each block. It was also found that the use of hardware and software information improved the estimation of the execution time of the parallelized model.

Keywords: Embedded Systems, Model-Based Development, Many-core processors

1. Introduction

In recent years, embedded systems, such as self-driving systems [1, 2], have become larger and more complex [3]. Therefore, processors must have high computing power and low power consumption. Traditionally, computing power has been improved by increasing the frequency of a single core. However, the performance improvement of a single core is limited, and many-core processors have been developed [4]. Many-core processors have high computing power, and the overall power consumption can be reduced [5,6]. Parallelization is important for maximizing performance. However, manual parallelization is difficult and time-consuming. Model-Based Development (MBD), such as MATLAB/Simulink [7], can reduce the software development time.

MBD is a software development methodology based on models. MBD using MATLAB/Simulink has the potential to be applied in the development of self-driving systems [8]. MATLAB/Simulink has an add-on called Embedded Coder [9] that can automatically generate C code from the model. However, MATLAB/Simulink can not generate parallelized C code. Therefore, Model-Based Parallelizer (MBP) [10] has been developed by the Embedded Multicore Consortium to automatically generate parallelized code. MBP estimates the execution time of each Simulink block from the generated C code, and places each Simulink block on the most appropriate core. In addition, other information called Software-Hardware Interface for Multi-/Many-Core (SHIM) [11] is used to perform the estimation.

SHIM was developed to allow tools to understand information about complex processors, such as multi-/many-core processors. As a result, SHIM has a variety of software and hardware information, such as the number of cores. However, MBP has a problem: the estimates used to perform core assignment take into account only information about SHIM hardware, such as the number of cores. In other words, software information is not used in the estimation. Parallelization brings in software information such as communication time. Therefore, if this information is not taken into account, the accuracy of the estimated time will be degraded. This makes it impossible to achieve SHIM's goal of an error of less than 20%, and undermines the reliability of the tool. The reasons for this include the fact that development using MBD in self-driving is still in the infancy, and research on supporting parallelization with MBD is new. Thus, parallelization in MBD has not been optimized and still needs to be studied.

This paper proposes an estimation method that improves the execution time of each Simulink block compared to existing methods. This is because, the estimated execution time per block needs to be improved to optimize the parallelization in MBD. In addition, an overall execution time estimation method using both hardware information and software information for a many-core processor is proposed. This paper uses eMBP [12] with MATLAB/Simulink, Kalray MPPA3-80 Coolidge processor [13], and eMCOS [14] as a real-time operating system (RTOS) .

The main contributions of this paper are as follows:

- This paper investigates and improves the causes of errors in the estimation execution time of a Simulink model.

[1] Graduate School of Science and Engineering, Saitama University, Japan
[2] Technology Headquarters eSOL Co., Ltd
[a)] y.kobayashi.858@ms.saitama-u.ac.jp

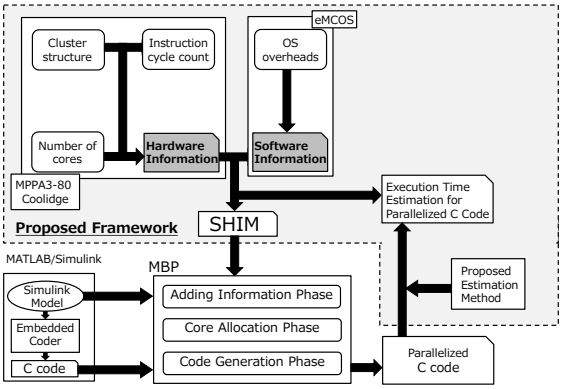

Fig. 1 System model with the proposed method

- This paper reduces the execution time of a parallelized Simulink model using the improved estimation time.
- This paper measures the inter-core communication time as software information and proposes a new estimation method.

The remainder of this paper is organized as follows. Section 2 describes the system model of MPPA3-80 Coolidge, eMBP, and SHIM. Section 3 describes our proposed approach, while Section 4 presents our experimental results. Section 5 discusses related work, and Section 6 concludes this paper.

2. System Model

This section describes the system model, which illustrates in **Fig. 1**. Section 2.1 describes the many-core processor architecture, which is the target platform in this paper. Then, Sections 2.2, 2.3, and 2.4 describe SHIM, LLVM IR, and eMBP, respectively, for automatically generating parallelized C code. Finally, Section 2.5 describes eMCOS, an RTOS.

2.1 Many-core architecture

Karlay MPPA3-80 Coolidge is a third-generation processor that follows Kalray MPPA2-256 Bostan [4]. The architecture of MPPA3-80 Coolidge is illustrated in **Fig. 2** and further discussed in [15]. MPPA3-80 Coolidge has 80 cores, which are less than MPPA2-256 Bostan. Nevertheless, due to the increase in frequency, the computing power has increased. MPPA3-80 Coolidge features five compute clusters (CCs) consisting of 16 cores each. In addition, it uses a NoC structure with multiple paths to prevent path contention between CCs.

2.1.1 CCs

In MPPA3-80 Coolidge, the 16 internal nodes of the NoC correspond to the CCs and consist of the processing engine (PE), SMEM, NoC interface. Cluster local memory (SMEM) is 4 MB and is shared by 16 PEs and RMs. Each PE has a Kalray-1 core that implements a 64-bit, six-stage, Very Long Instruction Word architecture with a frequency of 1.2 GHz.

2.2 SHIM

SHIM is an IEEE standard interface for describing hardware for software tools [11]. SHIM is defined by an XML schema and has hardware and software information for multi-/many-core processor architecture. By using SHIM to represent hardware, software verification can be performed without actual hardware. Hardware information is stored in XML format as SHIM XML. A SHIM XML consists of a tree structure with three superordinate components of: ComponentSet, AddressSpaceSet, and CommunicationSet. It includes the number of cores, memory access, and communication methods. SHIM allows for efficient use of multi-/many-core processors without understanding hundreds of pages of specifications. The estimation of execution time for SHIM is intended to be used in upstream processes such as substrate selection, and the goal is to keep the error within 20% [16].

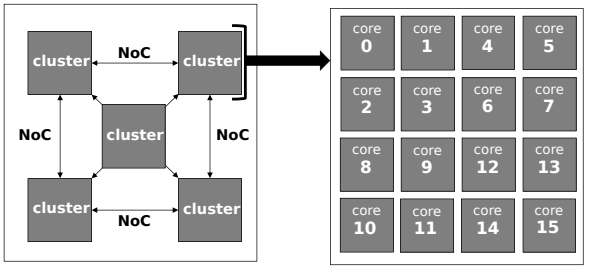

Fig. 2 Overview of Kalray MPPA3-80 Coolidge architecture

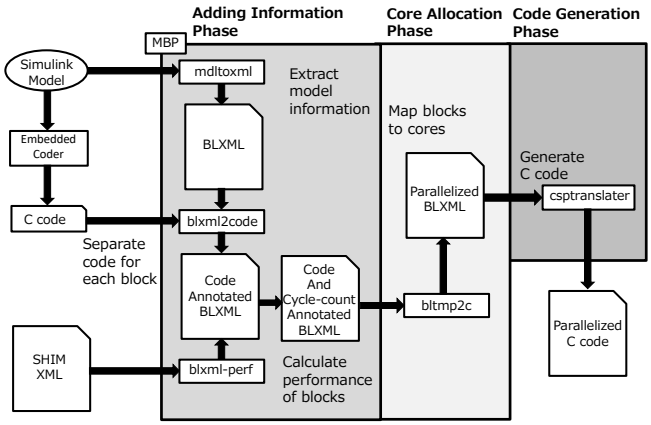

Fig. 3 System model of Model Based Parallelizer (MBP)

2.3 LLVM IR

LLVM IR is compiled with LLVM [17], a compiler base capable of supporting any programming language, and a unique intermediate representation that is independent of language and architecture. The value of LLVM IR is used in SHIM for machine-independent descriptions, because LLVM IR can use the same intermediate representation regardless of machine or language combination.

2.4 eMBP

eMBP is a model-based parallelization tool developed by eSOL based on MBP. The eMBP architecture is presented in **Fig. 3**. eMBP can reduce development costs and time because it automatically generates parallelized C code. An eMBP process consists of three major phases: Adding Information phase, Core Allocation phase, and Code Generation phase.

2.4.1 Adding Information phase

The purpose of this phase is to generate C code and cycle-count-annotated Block-Level XML (BLXML). First, eMBP extracts the block information from the Simulink model. Next, eMBP divides the C code generated using Embedded Coder into code of blocks and generates code-annotated BLXML. The block indicates the smallest unit block that constitutes a Simulink model. The dependencies between the blocks are retained. Finally, eMBP extracts the hardware information from the SHIM

```
1   #define INSTRUCTION_NAME "add"
2   continuity_count = 1;
3   base_cnt = 0;
4   memset(base, 0x00, sizeof (base));
5   start, end =0;
6   /*Measuring Overhead*/
7   AGGREGATION_METHOD(base ,TRY, ITERATION, buff);
8   empty = buff[1];
9   base_cnt = 0;
10  memset(base, 0x00, sizeof (base));
11  for (j = 0; j < TRY; j++) {
12      start = clock();
13      for (i = 0; i < ITERATION; i++) {
14          INSTR_BODY_2(uix, uiy, uiz = uix + uiy); /*add*/
15      }
16      end = clock();
17      base[base_cnt++] = (end − start);
18  }
```

Fig. 4 Part of LLVM intermediate representation measurement code

XML and uses the processor latency to estimate the performance of each Simulink block. This information is used to generate cycle-count-annotated BLXML. The performance of each Simulink block is calculated as typical (most frequently occurring cycles).

2.4.2 Core Allocation phase

The purpose of this phase is to determine which blocks are assigned to the different cores. Using the C code and cycle-count annotated BLXML generated in the Adding Information Phase, eMBP generates mapping information as parallelized BLXML. In other words, the estimated execution time for each Simulink block estimated is used to allocate the blocks to the cores. eMBP implements a double-hierarchical clustering scheme to ensure proper core allocation [10].

2.4.3 Code Generation phase

The purpose of this phase is to generate parallelized C code. eMBP generates parallelized C code by reconstructing the C code for each assigned core according to the parallelized BLXML, which includes core assignments. The parallelized C code includes the eMBP's communication API (Application Programming Interface), which enables communications between cores and clusters.

In this paper, the estimation of the execution time for each Simulink block used in Core Allocation phase in Section 3.3 and 4.3 is improved. In addition, the execution time of the parallelized C code generated in Code Generation phase is reduced.

2.5 eMCOS

eMCOS is a real-time embedded operating system developed by eSOL, a Japanese RTOS supplier, and was the first commercially available many-core processor RTOS designed for use in embedded systems [14]. eMCOS is also the only operating system that supports many-core processors with a cluster structure.

eMCOS has a message communication function that can send and receive arbitrary data (messages) between threads and perform synchronization control between threads. The message communication function supports the following four types of messages: Regular message, Fast message, Session message, and Request reply message. Especially, Fast message can be used in inter-core communication.

3. Proposal of a Method for Estimation of the Execution Time of Simulink Models

This section presents our proposed approach for generating parallelized code for Simulink models in eMBP and estimating the execution time on multi-/many-core processors. SHIM of the target processor requires to generating parallelized code for Simulink models using eMBP. Therefore, we measure the execution time of each instruction in LLVM of MPPA3-80 Coolidge required for SHIM. Furthermore, we explain how to measure the execution time of the Simulink model on MPPA3-80 Coolidge. Then, an improvement method for the current estimation error and an estimation method using FPU for hardware information are proposed. Next, efficient parallelization through improved execution time estimation is explained. Finally, the inter-core communication time of software information is measured, and an estimation method using this information is proposed. The proposed estimation method can be applied by preparing the SHIM of the target processor, since SHIM can have information of any processor.

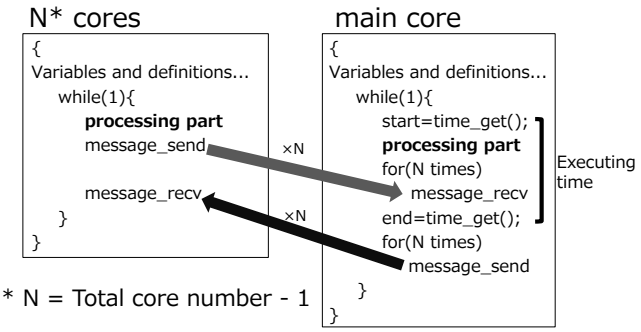

Fig. 5 Method of measuring the execution time

Table 1 Information of use the Simulink model

Model Name	Number of Edges	Number of Nodes
e1000	1,509	1,031
e3000	4,522	3,099
e5000	7,564	5,171

3.1 The calculation method of LLVM IR

This section describes how to measure the execution time of each instruction in LLVM with MPPA3-80 Coolidge. The LLVM IR is the basic information needed for SHIM. The value of LLVM IR depends on the performance of the target processor and is measured for each processor. SHIM is available to the public [18]. Therefore, the source code to measure the execution time of each instruction in LLVM has been published. However, the published source code includes an inline assembler. MPPA3-80 Coolidge does not support the inline assembler. Therefore, the part that measures the time with the inline assembler is rewritten as a clock function. Part of the rewritten code is presented in **Fig. 4**. To set the unit to the number of cycles, the following calculation is performed.

$$second_{instruction} = (time_{end} - time_{start})/second_{clock}$$
$$cycle_{instruction} = second_{instruction} * (1/clock_{frequency})$$

The measurement start time is $time_{start}$, while the measurement end time is $time_{end}$. Therefore, the measured time can be expressed as $time_{end} - time_{start}$. The unit of the measured time is clock, and it needs to be converted to the number of cycles. Thus, the measured time is divided by the number of

```
%3 = alloca double*, align 8
%4 = alloca double, align 8
store double* %1, double** %3, align 8
store double %0, double* %4, align 8
%5 = load double, double* %4, align 8
%6 = load double, double* %4, align 8
%7 = fadd double %5, %6
%8 = load double*, double** %3, align 8
store double %7, double* %8, align 8
ret void
```

Fig. 6 LLVM IR used for estimated execution time

```
%9 = load double, double* %3, align 8
%10 = load double, double* %3, align 8
%11 = fadd double %9, %10
store double %11, double* %2, align 8
```

Fig. 7 LLVM IR used for C code

clocks per second ($second_{clock}$) to obtain the number of seconds per instruction ($second_{instruction}$). Next, to convert the units from seconds to cycles, multiply by the number of seconds per cycle ($1/clock_{frequency}$). This conversion produces the number of cycles per instruction ($cycle_{instruction}$). For MPPA3-80 Coolidge, $second_{clock} = 1,250,000$ and $clock_{frequency} = 1.2$ GHz.

3.2 Description of the experimental environment and how to measure the execution time

This section describes the Simulink model used for the experiment and how to measure the execution time. Details of using Simulink models are shown in **Table 1**. Using Simulink models are random models. Random models mainly consist of an integration block, an additional block, and a math-function block. In addition, each block for the Simulink model is repeated 100 times in order to make the process heavy. This is because a large Simulink model is required for parallelization with MPPA3-80 Coolidge. Note that the Simulink models are created with reference to [19] to avoid bugs. The math-function block is mainly used for non-quadratic calculations, such as log and power calculations. In addition, all blocks used in this model are the blocks targeted by eMBP.

Next, a measurement method of the model execution time is described. eMCOS has a function to measure time, and the unit is the number of cycles. This function is used to measure the overall execution time (exe_{time}) of the model.

- One core: In the case of one core, the eMCOS function to measure time is used before (exe_{start}) and after (exe_{end}) the task part of the program. In addition, it measures of the function 10,000 times and determines the average overhead time ($over_{tg}$) to eliminate the overhead for the function to measure time. Next, the following formula is used to measure exe_{time}:

  $$exe_{time} = (exe_{end} - exe_{start}) - over_{tg}$$

  In this way, the overhead of the time measurement function is subtracted from the measured execution time to obtain the exact execution time.
- Two or more cores: In the case of two or more cores, it requires synchronization. After $core_{main}$ measures the processing time, $core_{main}$ sends messages to each core. Each core does not perform the next process until it receives messages. Thus, synchronization is achieved. The time measurement is shown in **Fig. 5**. N is the total number of cores minus one. In addition, to measure the average time, it takes 10,000 executions of sending and receiving messages ($over_{mess}$). Next, the following formula is used to calculate exe_{time}:

  $$exe_{time} = (exe_{end} - exe_{start}) - over_{tg} - over_{mess} * N$$

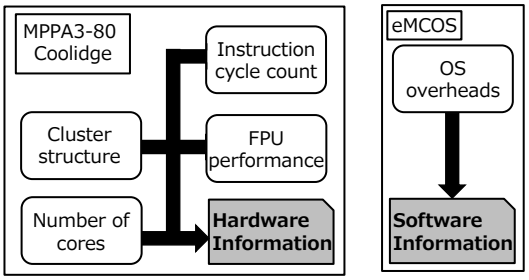

Fig. 8 Detailed hardware and software information

Thus, the overhead of the communication API is subtracted from the measured execution time to compensate for the communication latency and get an accurate execution time.

3.3 Estimation error improved with different instructions

The errors in the execution time estimation are reduced by investigating the causes of the errors and using proposed methods.

To investigate the cause of this error, the LLVM IR code used for estimation (**Fig. 6**) and the same processing part of the LLVM IR code in the actual C code (**Fig. 7**) are compared. As a result, the alloca instruction is executed twice, the store instruction twice, and the load instruction once more than when the actual C code. The error is caused by the difference in whether the process was a function or not. eMBP treats a Simulink block as a function because eMBP takes into account the input and output of the block. The reason for the increase in the number of these instructions is to prepare the function arguments, assign numbers, and assign the return value.

Next, the error caused by the different number of instructions is improved. The basic flow is to subtract the number of cycles of instruction increased by the function from the performance per block calculated in Section 4.1. The number of instructions depends on the number of inputs to the block. In this paper, $alloca_c$, $store_c$, and $load_c$ are the number of cycles of the alloca, store, and load instructions, respectively, and In are the number of inputs to the block. However, if the inputs are equal (e.g., $a = b + b$), the number of cycles when $In = 1$ is subtracted.

$$(alloca_c + store_c) * (In + 1) + load_c * 1$$

3.4 Measurement of FPU performance required for estimation and proposed the estimation method

This section proposes a measurement method for FPU of hardware information, and an estimation method using that information. The details of the measured information are presented in **Fig. 8**. The float (single) type works for a 32-bit, the double type works for a 64-bit. The measurement method is the same as that used to measure the LLVM. Instruction cycle numbers at 64 bits and 32 bits are obtained by setting the variables of the target instructions to double and float types, respectively.

Next, an estimation method is proposed that finds the portions

of the parallelized C code where 32-bit and 64-bit instructions are used and uses this information. The steps are as follows. First, Adding Information Phase of eMBP using SHIM measured in 32 bits to obtain the necessary files. Second, the files are used to obtain the block name containing the specific 64-bit instructions (instructions with a large difference in the number of cycles between 32 bits and 64 bits). Third, the specific instruction performance of the retrieved block name to 64-bit performance is rewritten. Finally, the remaining phases of eMBP are perform, and estimation is performed that takes both 32-bit and 64-bit performance into account.

3.5 Parallelization improvement by better the estimation execution time

This section explains why improving the estimated execution time will allow for efficient parallelization and reduce the overall execution time. The mechanism for parallelizing MBP is to estimate the execution time of each Simulink block, as described in Section 2.4.1. Then, tasks are assigned to cores based on the estimated time. Therefore, if the estimated execution time of each Simulink block has a large error, it may happen that tasks are not allocated evenly even if the parallelization is appropriate in the estimate. Furthermore, as explained in Section 3.3, MBP estimates the estimation execution time to be long. Therefore, improving the estimated execution time makes it shorter than the existing one. In addition, the short estimated execution time of each task allows communication to take place at a more optimal time than existing. As a result, the waiting time due to synchronization can be reduced. Thus, improving the estimated time for each Simulink block can improve parallelization and reduce execution time.

3.6 Measurement of the inter-core communication time required for estimation and proposed the estimation method

A measurement method of the time of synchronous communication between cores is described in this section. In addition, an estimation method that uses the measured time to improve the execution time for multiple cores is proposed. The functions used in parallelization include the communication functions mbp_channel_send/recv which use eMCOS API.

- mbp_channel_send/recv: mbp_channel_send/recv is a function that sends and receives messages. This function determines which core to send a message. eMCOS API is used for this function. The type of message is a regular message for inter-core communication and a fast message for inter-cluster communication.

In addition, two methods are available: synchronous communication and asynchronous communication. However, the proposed method measures the time of synchronous communication for the fast message because eMBP uses synchronous and inter-core communication. The measurement method uses the mbp_channel_send/recv function. The measurement method of the inter-core communication time for each communication method is as follows.

- Synchronous communication: The inter-core communication time varies depending on the timing of the other cores.

Table 2 Part of LLVM IR In MPPA3-80 Coolidge

LLVM IR	typical cycle
fadd	9.60
fsub	9.60
fmul	9.60
fdiv	24.00
alloca	10.56
load	4.80
store	4.80
call	5.37
ret	8.24

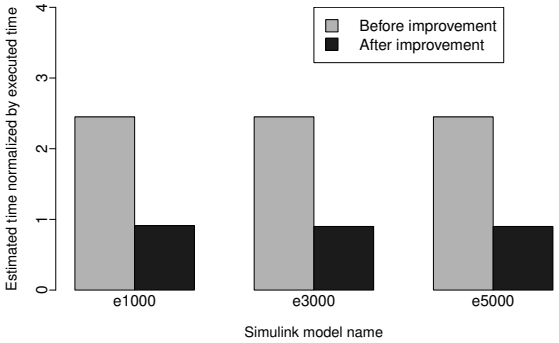

Fig. 9 The estimated time to the execution time with one core in MPPA3-80 Coolidge

In this experiment, additional processing (e.g., for statement) is added before mbp_channel_send/recv to ensure that the receiving side is ready when the sending side is measured, and the message has already been sent when the receiving side is measured.

The proposed measurement method inserts the function to measure the time before and after messages send, and receive to obtain exe_{start} and exe_{end}, and subtract $over_{tg}$.

Then, an estimation method of the execution time is described. First, the number of communication channels used by $core_{main}$ is measured. $core_{main}$ is the core where the task is executed first. The way to find is to count the portion of the function that is creating the communication channel from the parallelized C code. Next, the communication time obtained by the following formula is added to the original estimated time. In this paper, add_time is the time to be added to the estimate, sendtime_size represents the time to send a fast message, recvtime_size represents the time to receive a fast message, block_size represents the average of estimation time of each Simulink block, com_time represents the total time required for communication, and com_num represents the number of communication channels. Synchronous communication causes the waiting time if the timing of sending and receiving is different. The estimation method considers the waiting time as one block in the Simulink model and expresses it as block_size.

$$com_{time} = (sendtime_{size} + recvtime_{size} + block\_size)$$
$$add_{time} = com_{time} * com_{num}$$

4. Evaluation

In this section, the proposed method is evaluated. First, the execution time of each instruction of LLVM is measured, and the execution time of the Simulink model is estimated. In addition, the Simulink model execution time is measured, the error from the estimated execution time is calculated, and the cause of the difference is discussed. Next, the performance of the FPU as hardware information is measured. Based on this information,

Table 3 LLVM IR of FPU32, FPU64

LLVM IR	32-bit	64-bit
fadd	9.64	9.64
fsub	9.64	9.64
fmul	9.64	9.64
fdiv	24.10	317.11

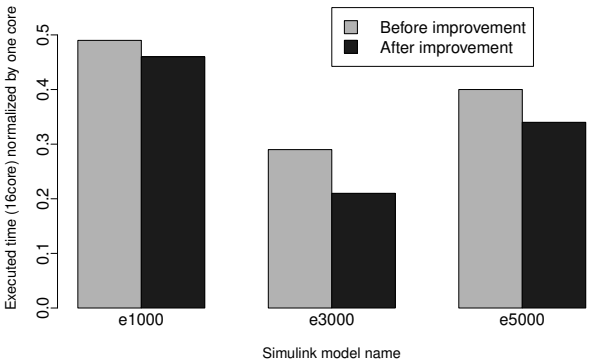

Fig. 10 The execution time with 16 cores in MPPA3-80 Coolidge

the execution time of each Simulink block is estimated, and it has made improvements. Finally, this communication time is measured for each size of the transmitted data as a necessary information to improve the execution time estimation after parallelization on many-core processors. Then, we improve the estimation by taking into account the communication.

4.1 Measurement results of LLVM IR

The number of LLVM IR cycles in the SHIM basic information for MPPA3-80 Coolidge is calculated, and the results are presented in **Table 2**. The columns display the typical values, and the units are the number of cycles. eMBP used the typical values for estimation and core assignment.

4.2 Estimation and measurement results using existing methods

The estimated execution time on eMBP with one core and the actual execution time on MPPA3-80 Coolidge is measured for comparison. The execution times were calculated as the average of 1,000 runs. The bar graph (Before improvement) on the left side of **Fig. 9** shows the estimated execution time divided by the actual execution time. The results show that the execution time estimated by eMBP and the actual execution time on the MPPA3-80 Coolidge differs significantly. eMBP uses the estimated time at one core (in other words, the estimated time for each Simulink block) to perform parallelization. Therefore, large errors affect the parallelization.

Then, the performance differences by instruction type are measured. FPU performances for 64 bits and 32 bits are presented in **Table 3**. The result shows that the fadd, fsub, and fmul (addition, subtraction, and multiplication of floating-point numbers, respectively) instructions had no performance difference between 32 bits and 64 bits, and only the fdiv (division of floating-point numbers) instruction differed. The existing measurement method calculates the estimated time of a Simulink block with a 32-bit type fdiv instruction. Therefore, the estimation of the execution time of Simulink blocks containing fdiv instructions is improved to the correct estimated time by determining whether it is of type double or float.

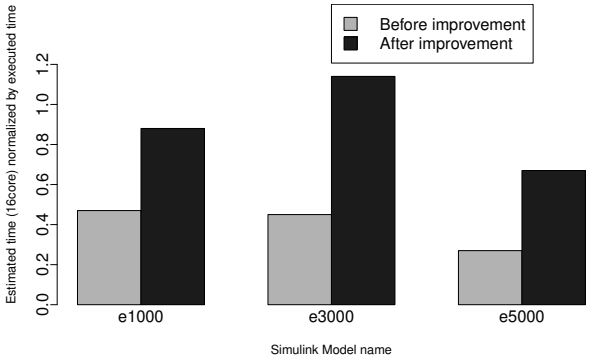

Fig. 11 The estimation time with 16 cores in MPPA3-80 Coolidge

4.3 Improved estimation of Simulink blocks and associated parallelization efficiency

In this section, the estimation of the execution time of Simulink blocks is improved with the proposed methods in Section 3.3. Furthermore, the execution time of parallelization has been reduced by improving the estimates. Since the estimated execution time for each Simulink block was used, the estimated execution time on one core is measured. The execution time was calculated as the average value of 1,000 runs. The bar graph (After improvement) on the right side of Fig. 9 shows the estimated execution time calculated using our estimation method divided by the actual execution time. Therefore, the closer the value is to one, the better the accuracy of the estimate. The results show that the improvement estimation method able to keep the estimated execution time of the target Simulink model within the target error of 20%.

The execution time of parallelized code is reduced by 4% to 8%. **Fig. 10** shows the result, where the left side is the time before the improvement, and the right side is the time after the improvement, which is the execution time with 16 cores divided by the execution time with one-core. Therefore, the smaller the value, the shorter the execution time due to parallelization. The results show that both are faster than the execution time with one core. In addition, it shows that the execution time after the improvement is even shorter than before the improvement. This is due to the improved estimation of each Simulink block, which enables parallelization with finer granularity. Based on these results, the proposed improvement methods of the estimation time of each Simulink block and parallelization.

The estimation execution time when parallelized is shown in **Fig. 11**. The bar graph on the left side of Fig. 11 (before improvement) shows the estimated execution time with 16 cores. The vertical axis of this graph is the execution time on 16 cores divided by the execution time on one core. The results show that the estimated time is shorter than the execution time for all models. This is due to the fact that the communication time is not taken into account in the estimation of the parallelized model. Therefore, in the next section, the estimated time taking into account the communication time is measured.

4.4 Estimated execution time improvement using inter-core communication time

The time of synchronous communication between the cores is

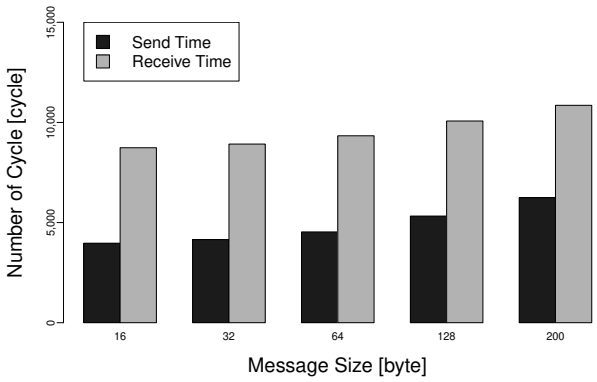

Fig. 12 The synchronous inter-core communication time

measured. The measured time is used to improve the estimate of the execution time when multiple cores.

The inter-core communication time is measured by varying the size of the message. The measured values were the average of the middle 80% of the 1,000 measurements of the communication time. The maximum value was 200 bytes, as this is the maximum size that MPPA3-80 Coolidge can send. The communication time between cores is proportional to the size of the message sent, presented in **Fig. 12**.

Next, the execution time estimated by the proposed improvement method is described. The bar graph on the right side of Fig. 11 (after improvement) shows the improved estimated execution time with 16 cores. The vertical axis of this graph is the estimated execution time of 16 cores divided by the execution time of 16 cores. Therefore, the closer the value is to one, the better the accuracy of the estimate.

The proposed method uses the communication time between cores to improve the estimation time. For this purpose, we obtained the number of communication channels for each Simulink model. As a result, the number of communication channels was 41 for e1000, 99 for e3000, and 150 for e5000. In addition, the size of the communication messages for each model is all the same. As a result, the error in the estimation of the execution time when e1000 and e3000 are used is within 20%.

However, the error in the estimated execution time of e5000 is 33%, which is not on target. This is due to the fact that as the scale of the model increases and the communication becomes complex, the time spent waiting for synchronous communication becomes larger than that of a single Simulink block. The estimation of the waiting time for synchronous communication to be an issue for the future. However, the proposed method is effective because it improves the error of the estimated execution time.

4.5 Discussion

In this section, scalability and limitations of the proposed estimation method are discussed. The proposed method can support not only MPPA-80 Coolidge but also other hardware platforms [20]. In particular, the proposed method can be directly used for parallelization using MBP. This is because MBP has improved the part of the process that uses the SHIM information to estimate the Simulink blocks. SHIM can have information on any processor as described in Section 2.2, and therefore the proposed method can be applied to any processor. MPPA3-80 Coolidge is not yet capable of inter-cluster communication. However, the present method can apply to the related work [21]. Thus, the proposed method can apply to processors with a cluster structure. Furthermore, the proposed estimation method can be automated and requires little time for improvement.

Next, regarding the marginal nature of the approach. The estimated time normalized by executing time in Fig. 9 is close to one, which is a sufficient improvement with respect to the estimation by using hardware information. However, the estimation of execution time after parallelization using software information has room for improvement. Although software information was used in this study, the delay due to synchronization of communication was not taken into account, resulting in this result.

5. Related Work

This section describes studies that perform estimation for MATLAB/Simulink, the mainstream platform for MBD, and the research that targets MPPA2-256 Bostan, the previous generation of MPPA3-80 Coolidge.

5.1 Code parallelization tool

MBP is a code parallelization tool for model-based development. Zhong et al. [10] proposed a model-based parallelization approach for parallelizing embedded systems built in the Simulink environment on a multi-core processor. Moreover, MBP extended to support heterogeneous multi-cores [25].

Another study proposed the SLX tool [22], which provides the ability to partition existing source code for a single core and map partition source code to hardware. SLX also allows users to estimate processing performance and power consumption for both heterogeneous and homogeneous multi-core environments based on an abstracted hardware information description.

5.2 Improved core allocation is using eMBP

Remapping Block Method (RBM), which uses eMBP results to remap blocks to the core. In addition, Deciding Execution Order Method (DEOM) is the process of determining the order of execution in which the entire process can be completed. Two methods are proposed by Kojima et al. [21]. These methods can be used to improve the parallelism of blocks and speed up the processing while distributing the load compared to existing methods.

In another study, MAPA and RCAA were proposed by Honda et al. [23]. These algorithms determine core allocation to many-core processors with a cluster structure such as MPPA2-256 Bostan. MAPA uses the result of allocating N cores in eMBP (N is the number of cores used by the user) to determine the cluster allocation according to the communication contention of NoC. RCAA takes 16*N core allocations from eMBP and remaps these allocations to perform cluster allocation (N is the number of clusters to be used). The combination of MAPA and RCAA provides better results than using MAPA, RCAA, or eMBP alone.

In contrast to existing studies, which use the eMBP estimation results and aim to improve performance by mapping. This paper aims to improve the estimation time itself.

5.3 Improved the eMBP estimation time

The estimation method of the execution time considering the overhead in model-based development was proposed by Honda et al. [24]. They proposed an estimation method for the execution

Table 4 Comparison of proposed method and other methods

	MATLAB/ Simulink	Estimation of Simulink block	Cluster Structure	OS Overhead	MPPA3-80 Coolidge Architecture
MBP [10]	✓	✓			
SLX tool [22]	✓	✓			
RBM and DEOM [21]	✓	✓			
MAPA and RCAA [23]	✓	✓	✓		
Honda et al. [24]	✓	✓	✓	✓	
Proposed Method	✓	✓	✓	✓	✓

time of applications developed using MATLAB/Simulink models on many-core platforms in MBD. SHIM was used for the estimation, and various performance information on MPPA2-256 Bostan, which was required information for SHIM, was measured. This estimation method took into account the operating system overhead and the cluster structure, which differed from existing methods. Existing studies used MPPA2-256 Bostan to measure the eMBP estimation time and evaluate the error. On the other hand, this paper uses techniques to reduce the error in the estimation time using the hardware information and the software information.

As described above, many studies have been done on the estimation of execution time in model-based development. However, as far as we know, no research has been done on model-based development using the MPPA3-80 Coolidge.

6. Conclusion

This paper proposed the estimation time and execution time of the Simulink model before and after parallelization are improved using the proposed method. The proposed method estimated the execution time for each Simulink block used for parallelization and FPU. In addition, it estimated the execution time after parallelization using communication time. The results showed that the proposed method improved the estimation execution time of each block and reduces the execution time during parallelization. In addition, the estimation execution time after parallelization was improved by considering the communication time. Since this paper proposed an estimation method using SHIM, it can be applied to other processors. Future work includes investigating how to measure the estimated execution time taking into account the latency due to synchronous communication and improving the MBP allocation method. In addition, we would like to compare the results with the latest estimation methods.

Acknowledgments

This work was partially supported by JST PRESTO, Japan (Grant No. JPMJPR1751).

References

[1] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, A. Tomohito, Y. Fujii and T. Azumi: Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems, *Proc. of ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pp. 287–296 (2018).

[2] S. Tokunaga, N. Ota, T. Yoshiharu, K. Miura and T. Azumi: MATLAB/Simulink Benchmark Suite for ROS-based Self-driving System, *Proc. of ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)* (2019).

[3] S. Saidi, R. Ernst, S. Uhrig, H. iling and B. D. de Dinechin: shift to Multicores in Real-Time and Safety-Critical Systems, *Proc. of International Conference on Hardware/Software Codesign and System Syn sis (CODES+ ISSS)*, pp. 220–229 (2015).

[4] T. Azumi, Y. Maruyama and S. Kato: ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform, *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020).

[5] M. Becker, D. Dasari, B. Nikolic, A. Benny, V. Nélis and T. Nolte: Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform, *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 14–24 (2016).

[6] Q. Perret, P. Maurère, É. Noulard, C. Pagetti, P. Sainrat and B. Triquet: Mapping Hard Real-Time Applications on Many-Core Processors, *Proc. of International Conference on Real-Time Networks and Systems (RTNS)*, pp. 235–244 (2016).

[7] MathWorks: MATLAB/Simulink, `http://www.mathworks.com/products/simulink/.`

[8] R. Yoshinaka and T. Azumi: Model-Based Development Considering Self-Driving Systems for Many-Core Processors, *Proc. of IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 337–344 (2020).

[9] MathWorks: Embedded Coder, `https://jp.mathworks.com/products/embedded-coder.html`.

[10] Z. Zhong and M. Edahiro: Model-Based Parallelizer for Embedded Control Systems on Single-ISA Heterogeneous Multicore Processors, *Proc. of International SoC Design Conference (ISOCC)*, pp. 117–118 (2018).

[11] IEEE: SHIM, `https://standards.ieee.org/standard/2804-2019.html`.

[12] eSOL Co., Ltd: eMBP, `https://www.esol.com/embedded/mbp.html`.

[13] B. D. de Dinechin, D. van Amstel, M. Poulhiès and G. Lager: Time-critical computing on a single-chip massively parallel processor, *Proc. of IEEE Conference on Design, Automation and Test in Europe Conference and Exhibition (DATE)*, IEEE, pp. 1–6 (2014).

[14] eSOL Co., Ltd: eMCOS, `https://www.esol.co.jp/embedded/emcos.html`.

[15] B. D. de Dinechin: INVITED Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore, *Proc. of ACM/IEEE Design Automation Conference (DAC)*, pp. 1–4 (2019).

[16] K. Honda and T. Azumi: Performance estimation for many-core processor in model-based development, *Proc. of Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6 (2019).

[17] C. Lattner and V. Adve: LLVM: a compilation framework for lifelong program analysis & transformation, *Proc. of International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86 (2004).

[18] OpenSHIM: Open SHIM, `https://github.com/openshim`.

[19] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson and C. Csallner: SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink, *Proc. of International Conference on Software Engineering (ICSE)*, pp. 335–346 (2020).

[20] C. Hiroyuki, S. Kazutoshi, I. Tsutomu, M. Seiya, A. Takuya, F. Kenji and K. Shinpei: Towards Heterogeneous Computing Platforms for Autonomous Driving, *Proc. of IEEE International Conference on Embedded Software and Systems (ICESS)*, pp. 1–8 (2019).

[21] S. Kojima, M. Edahiro and T. Azumi: Remapping Method to Minimize Makespan of Simulink Model for Embedded Multi-Core Systems, *Proc. of International Conference on Computers and ir Applications (CATA)* (2018).

[22] Xilinx: SLX, `https://www.silexica.com/`.

[23] K. Honda, S. Kojima, H. Fujimoto, M. Edahiro and T. Azumi: Mapping Method of MATLAB/Simulink Model for Embedded Many-Core Platform, *Proc. of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 182–186 (2020).

[24] K. Honda, H. Fujimoto and T. Azumi: Estimation Method Considering OS Overheads for Embedded Many-Core Platform, *Proc. of IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (2020).

[25] R. Yamamoto, M. Oinuma, M. Kondo, S. Honda and M. Edahiro: Multirate Model Parallelization for MPSoC with FPGA in Model-Based Development: A Case Study, *Proc. of Asia Pacific Conference on Robot IoT System Development and Platform (APRIS)*, pp. 6–13 (2021).