DOI https://doi.org/10.24297/ijct.v23i.9472

LLVM-IR Instruction Latency Estimation Using Deep Neural Networks for a Software–Hardware Interface for Multi-Many-Cores

Hiro Mikami, Seira Iwai, and Masato Edahiro

Graduate School of Informatics, Nagoya University, Japan

pdsl@ertl.jp

Abstract

This study presents a method for estimating the latency of each LLVM-IR instruction to enable effective parallelization in model-based development. In recent embedded systems, such as in-vehicle electronic control, multi-many-core processors are utilized for the hardware, and model-based development for software. In the design of these systems, the degree of parallelism and accuracy of performance estimation in the early design stages of the model-based development can be improved by estimating the performance of the blocks in the models and utilizing the estimate for parallelization. Research is therefore being performed on a software performance estimation technique that uses IEEE2804-2019 hardware feature description called Software-Hardware Interface for Multi-many-core (SHIM). In SHIM, each LLVM-IR instruction is associated with an execution cycle of the target processor. Several types of assembly instruction sequences are generated for the target processor from a given LLVM-IR instruction; thus, it is not easy to estimate the number of execution cycles. In this study, we propose a method that uses deep neural networks to estimate execution cycles for each LLVM-IR instruction. It can be observed that our method obtains a better estimation of LLVM-IR instruction latency compared with previous methods in experiments using the Raspberry Pi3 Model B+.

Keywords: SHIM, embedded system, multicore, estimation, neural network

1. Introduction

The increasing scale and complexity of embedded systems in recent years have resulted in a corresponding increase in hardware performance requirements. Single-core processors were used to satisfy these requirements. However, their use has become problematic owing to their limited performance and has increased the power consumption and heat generation. Therefore, the use of multi-many-core processors to improve the performance is expected.

In addition, the labor hours and work costs for control system design and development are increasing because of the increased scale and complexity of control systems. This has led to the popularity of model-based development to reduce development time and human error. In model-based development, a control system is represented by an abstract model. The design was applied to the model.

Owing to these factors, parallelization to support multi-many-core model-based developments is required. One approach to parallelization is the code-based approach [7], in which the sequential code generated from the model is parallelized. Because parallelization is not considered during the design of the model in this approach, the degree of parallelism cannot be improved easily. In addition, if the performance target is not satisfied after parallelization, the system must be redesigned from scratch, which results in high rework costs.

Another approach for parallelization is the model-based approach [3]. In this approach, parallelization was considered during the system-design stage to achieve a high degree of parallelism. Moreover, parallel code instead of sequential code is automatically generated from the model.

However, to increase the parallelism, it is necessary to consider the processing time of each block in the load balancing model. To achieve this, it is crucial to devise a method for estimating the performance of the model design.

The performance estimation in model-based parallelization should not require an actual machine or instruction set simulator (ISS), and the features of the actual machine should be considered during the estimation. This is because it is not easy to use the development environment for embedded systems for the target processor in model-level design, especially during the early design phase.

To address these issues, performance estimation methods are being researched based on the IEEE 2804-2019 hardware abstraction description [6] called the *software-hardware interface for multi-many-core (SHIM)* which is an XML file that describes the hardware features as parameters.

In SHIM, to make tools using SHIM target independent, the LLVM-IR instruction set [9] from the LLVM project [8] was used for the processor instructions, instead of the instruction set of the target processor. The use of LLVM-IR has two limitations in terms of performance estimation. 1) Because the LLVM-IR assumes an infinite number of virtual registers, there is no register spilling in the LLVM-IR code. 2) Several sequences of the target code are generated using the LLVM-IR instruction.

Mikami et al. [11], to overcome these issues, proposed A) a method for software performance estimation using SHIM, and B) a method for the performance estimation of each LLVM-IR instruction to be stored in SHIM based on a regression analysis and created open-source software [16]. In the design flow, SHIM is first created for the target processor using Proposal B) in target-dependent environments. The performance of the system is then estimated with the created SHIM using Proposal A) with target-independent tools. Although this method reduced the cost of producing SHIM and the error in the estimated performance with SHIM within a target error of $\pm 20\%$, the error in the estimated latency for each LLVM-IR instruction remained large.

In this study, we propose an estimation method that uses deep neural networks (DNNs) instead of regression analysis, in which we propose a two-step estimation where the execution count of each target assembler instruction is estimated first. The LLVM-IR latencies are then estimated. In the experimental results obtained using the same environment as [11], the proposed method obtained a much better estimation of LLVM-IR.

The remainder of this paper is organized as follows. Section 2 provides a brief background of related work and previous studies on performance estimation using SHIM. Section 3 describes our proposed method for LLVM-IR instruction latency estimation using DNN, and Section 4 presents an evaluation of the LLVM-IR latency and software performance estimation with SHIM described in Section 3 on a Raspberry Pi3 Model B+.

2. Performance Estimation with SHIM

2.1. SHIM [6]

The widespread use of multi-many-core processors has led to the emergence of new architectures with various characteristics from which the user must select the optimal processor in specific applications. This has led to an increased burden on vendors of tools for analysis, optimization, and automatic parallelization, and real-time operating systems, which are forced to handle a wide variety of processors. System developers require tools with high portability and quality to efficiently use a wide variety of multicore devices. However, tool development requires reading voluminous manuals and an understanding the characteristics of each processor.

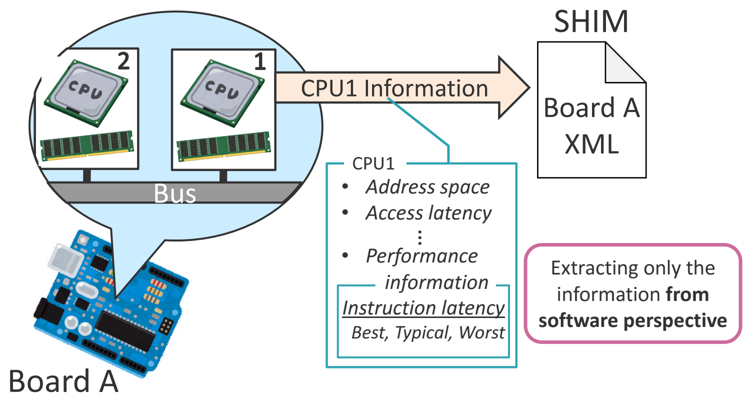
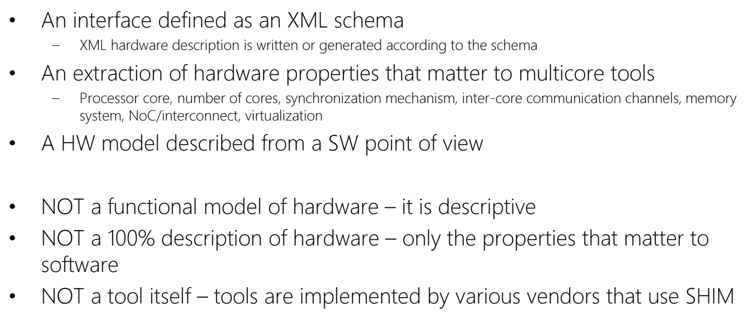


Figure 1: SHIM schematic

To address this issue, the Multicore Association [2] developed an international standard called SHIM (Software-Hardware Interface for Multi-many-core) to support the development of tools for multicore devices. SHIM2.0 was standardized by the IEEE in 2019 [6]. Figures 1 to 5 show a schematic diagram of SHIM, what SHIM is, what SHIM can do, the relationship between SHIM and tools, and its use cases. The class diagram representation of the SHIM XML schema from the top-level view [6] is presented in Appendix A.



- An interface defined as an XML schema
  - XML hardware description is written or generated according to the schema
- An extraction of hardware properties that matter to multicore tools
  - Processor core, number of cores, synchronization mechanism, inter-core communication channels, memory system, NoC/interconnect, virtualization
- A HW model described from a SW point of view

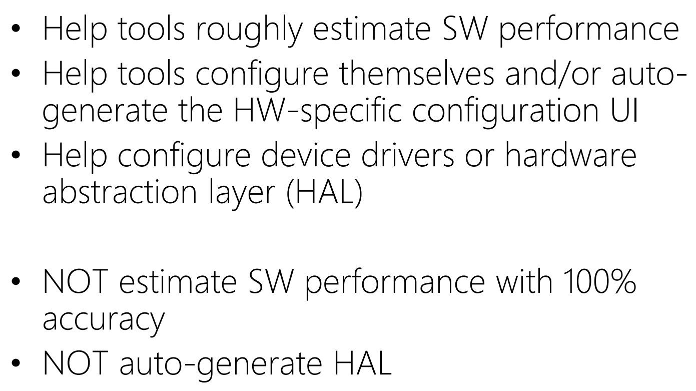
- NOT a functional model of hardware – it is descriptive
- NOT a 100% description of hardware – only the properties that matter to software
- NOT a tool itself – tools are implemented by various vendors that use SHIM

Figure 2: What SHIM is

One characteristic of SHIM is that only important hardware information for software development is extracted rather than a strict description of the hardware details. Therefore, the target accuracy for the performance estimation using SHIM was approximately $\pm 20\%$. The basic principle underlying SHIM is an understanding of the hardware features that affect software at the architectural design level. Therefore, once a program is designed for the specific hardware described by SHIM, software architecture should not require any changes during the later stages of system development. Compared with SHIM1.0, SHIM2.0 offers various improvements, such as pipeline memory architecture and heterogeneity support. SHIM is expected to be used in performance estimation and automatic parallelization and automatic configuration tools in operating systems and middleware.

2.2. Relationship between SHIM and LLVM-IR

The LLVM-IR instruction set [9] was used in SHIM for processor instructions rather than an instruction set for the target processor. The use of LLVM-IR enhances the versatility of SHIM because LLVM-IR supports a wide variety of programming languages and target hardware. Each LLVM-IR instruction is associated with the corresponding performance information in SHIM. This performance information is used in conjunction with the LLVM toolchain to



- Help tools roughly estimate SW performance
- Help tools configure themselves and/or auto-generate the HW-specific configuration UI
- Help configure device drivers or hardware abstraction layer (HAL)

- NOT estimate SW performance with 100% accuracy
- NOT auto-generate HAL

Figure 3: What SHIM can

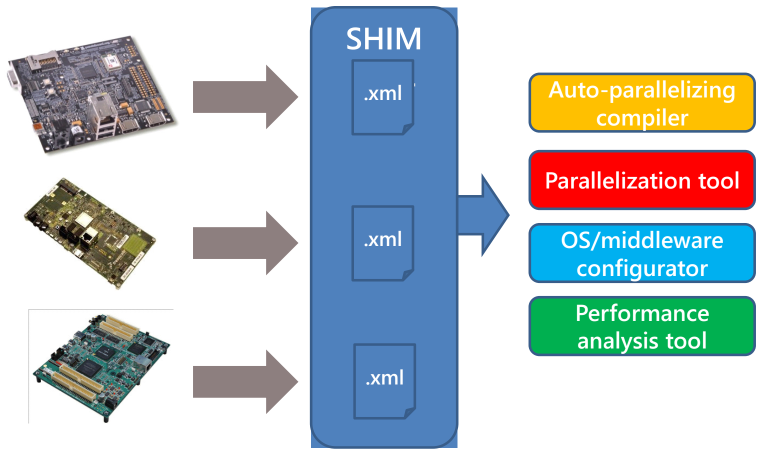

SHIM
.xml
.xml
.xml
Auto-parallelizing compiler
Parallelization tool
OS/middleware configurator
Performance analysis tool

Figure 4: SHIM and tools

estimate the software performance for an arbitrary processor without the use of processor-specific tools.

2.3. Related Work

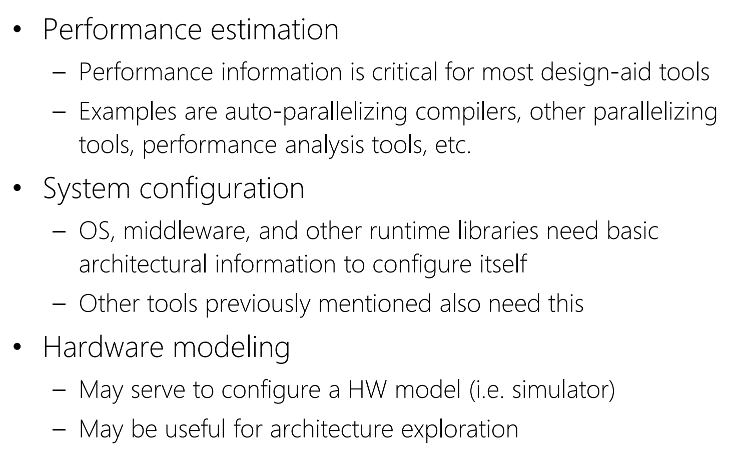
Several studies [5, 12, 13, 14, 17] have focused on performance estimation at the target instruction level, which does not rely on SHIM. In general, the number of loop iterations and branch directions cannot be estimated easily using static analysis, whereas adynamic analysis requires an actual machine or ISS, which is not desirable for model-level designs. In this study, we do not discuss these issues but consider only SHIM-specific issues.

The possibility of performance estimation using SHIM was demonstrated in a preliminary study by using the JPEG decoder program [4]. In this study, the estimation results of the SHIM and LLVM profilers were compared with the simulation results from the cycle-accurate simulator of an actual processor. The estimation errors were maintained at less than ±20%.

In this study, the following equation was used to evaluate the SHIM performance:

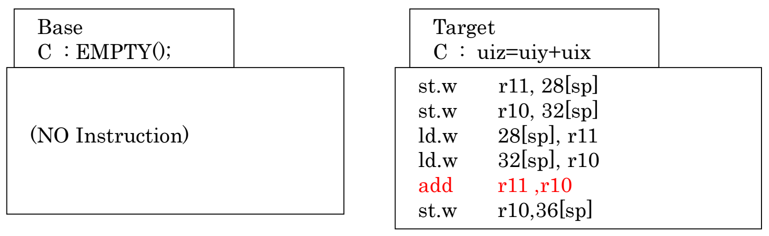
$$Execution\ Cycles = \sum_i (EC(IR_i) \times Latency_i), \tag{1}$$

where $EC(IR_i)$ is the execution count of instruction IR_i in LLVM-IR obtained using the LLVM profiler and $Latency_i$ denotes the execution cycles for instruction IR_i stored in the SHIM.



- Performance estimation
  - Performance information is critical for most design-aid tools
  - Examples are auto-parallelizing compilers, other parallelizing tools, performance analysis tools, etc.
- System configuration
  - OS, middleware, and other runtime libraries need basic architectural information to configure itself
  - Other tools previously mentioned also need this
- Hardware modeling
  - May serve to configure a HW model (i.e. simulator)
  - May be useful for architecture exploration

Figure 5: SHIM use cases



```
Base
C ：EMPTY();

(NO Instruction)
```

```
Target
C ： uiz=uiy+uix

st.w    r11, 28[sp]
st.w    r10, 32[sp]
ld.w    28[sp], r11
ld.w    32[sp], r10
add     r11 ,r10
st.w    r10,36[sp]
```

Figure 6: Measurement of worst-case "add" cycle counts for SHIM

The execution cycles in SHIM were obtained using two methods: 1) extracting from the user manual of the target processor or 2) calculating the difference in clock cycles between a program WITH the target instruction (Target Program) and that WITHOUT it (Base Program) in an evaluation environment of the target processor (Fig. 6). For both methods, obtaining the cycle counts for all instructions requires a long time. In addition, these methods required us to read numerous documents or to write several assembly programs.

To apply this method [4] for performance estimation in general software, two issues related to LLVM-IR should be considered.

Issue1 Incurrence of register spilling penalties

Issue2 Generation of several types of target instruction sequences from a single LLVM-IR instruction

Issue1 is related to the fact that whereas LLVM-IR assumes an infinite number of virtual registers, the finite number of registers in actual hardware results in register spilling. Owing to the infinite virtual registers in LLVM-IR, there is no register spill, and the data in the register will never be lost once it is placed in the register. However, in actual hardware, the number of registers is finite, and it is possible that the data placed in a register will be lost because of register spilling. Memory accesses generated in such cases are not reflected in the LLVM-IR programs. Such memory access may cause cache misses. Handling such events is an important consideration in performance estimation using SHIM.

Issue2 is related to the fact that because LLVM-IR is an intermediate representation in compilers, multiple types of assembly codes for the target processor may be generated from a single LLVM-IR instruction. Figure 7 shows an

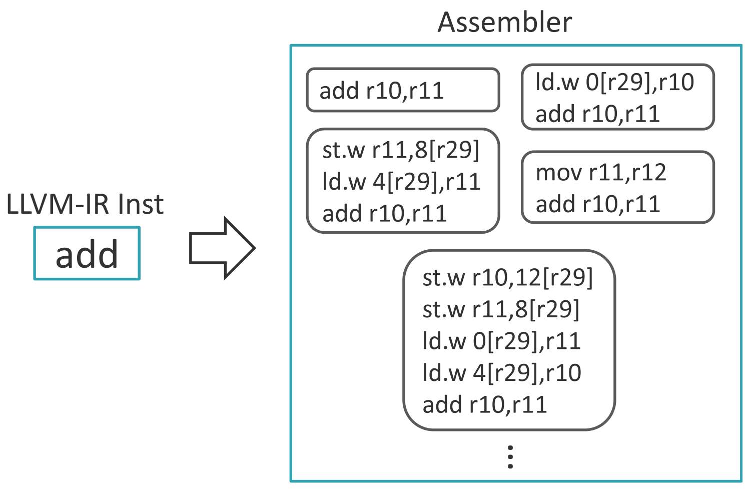


Figure 7: Example of conversion from LLVM-IR instructions to assembly code

example of the conversion from an add instruction in LLVM-IR to the assembly code for RH850/E1M-S2 [15] from Renesas Electronics Corporation. The converted assembler code can be a simple add instruction or an add instruction using several types of register replacement processes. As different target codes can be generated from a single LLVM-IR instruction, the probability of occurrence for each target code must be considered to calculate the latency accurately. Considering such occurrence probabilities becomes more difficult with increasing hardware-architecture complexity and the dependence of code generation on the optimizer.

Mikami et al. [11] proposed A) a method for estimating software performance using SHIM in the early stages of model-based development, and B) a method for estimating the execution cycle of each LLVM-IR instruction stored in SHIM for a processor using regression analysis when SHIM is created for the processor.

In Proposal A), the authors improved Eq. (1), using the following equation:

$$\begin{aligned} Execution\ Cycles \quad = \quad & \sum_i (EC(IR_i) \times \underline{Latency_i}) \\ + \quad & Cache\ Access\ Times \times \underline{Latency^C} \\ + \quad & Memory\ Load\ Times \times \underline{Latency^L} \\ + \quad & Memory\ Store\ Times \times \underline{Latency^S} \\ + \quad & \underline{Overhead} \end{aligned} \tag{2}$$

Issue1 was overcome in Eq. (2) by adding the number of cache accesses, memory loads/stores, and measurement overhead as parameters of the existing terms in Eq. (1).

To estimate the software performance, as shown in Fig. 8, $EC(IR_i)$ was measured using llvm-cov [10], $Cache\ Access\ Times$, $Memory\ Load\ Times$, and $Memory\ Store\ Times$ were calculated using the results from llvm-cov and the given cache miss, and the spilling ratios were registered. $Execution\ Cycles$ can be estimated using $Latencies$ in SHIM and the system $Overhead$, if any.

They also proposed a method to estimate the performance values to be stored in SHIM using regression analysis as a solution to overcome Issue2. The flow and regression formulae for this method are shown in Fig. 9.

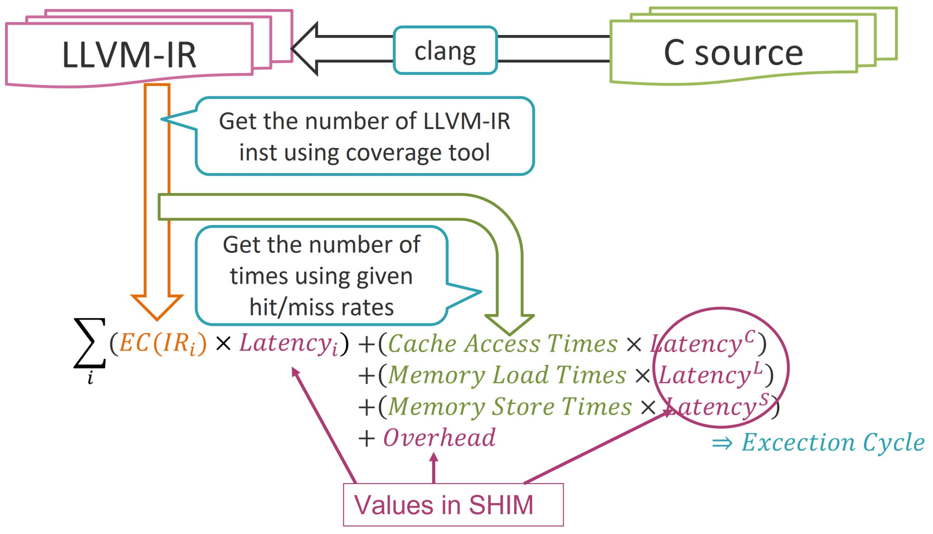

LLVM-IR
clang
C source
Get the number of LLVM-IR inst using coverage tool
Get the number of times using given hit/miss rates
$\sum_i (EC(IR_i) \times Latency_i) + (Cache\ Access\ Times \times Latency^C) + (Memory\ Load\ Times \times Latency^L) + (Memory\ Store\ Times \times Latency^S) + Overhead$
$\Rightarrow Excection\ Cycle$
Values in SHIM

Figure 8: Performance estimation flow

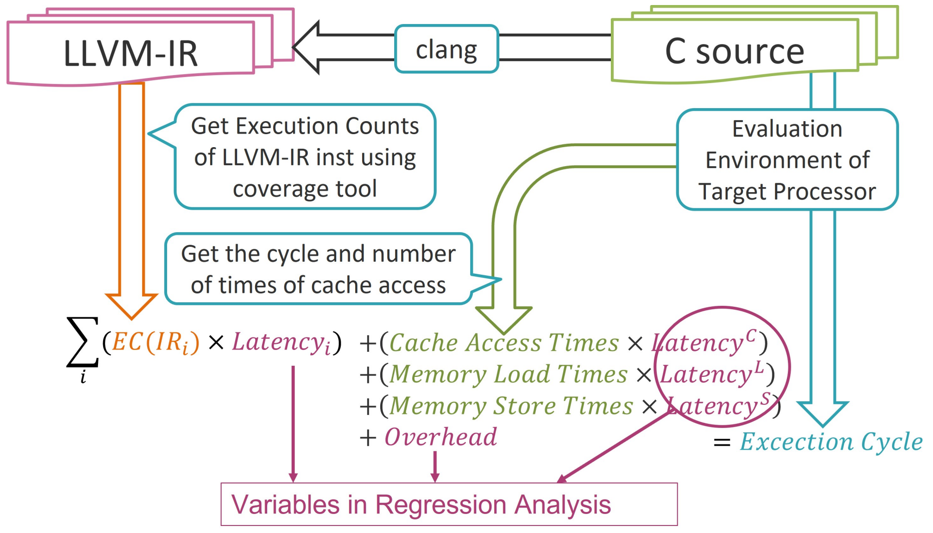

LLVM-IR
clang
C source
Get Execution Counts of LLVM-IR inst using coverage tool
Evaluation Environment of Target Processor
Get the cycle and number of times of cache access
$\sum_i (EC(IR_i) \times Latency_i) + (Cache\ Access\ Times \times Latency^C) + (Memory\ Load\ Times \times Latency^L) + (Memory\ Store\ Times \times Latency^S) + Overhead$
$= Excection\ Cycle$
Variables in Regression Analysis

Figure 9: SHIM value measurement flow

Issue2 is overcome in this method by estimating the latency using a number of measurement programs.

With these proposals, they demonstrated the execution cycles of software within a target error of ±20% using a Raspberry Pi3 Model B+. However, the value of each LLVM-IR instruction latency was not realistic. In this study, we propose an estimation method for a more realistic instruction latency.

2.4. Definition of Accuracy

In this section, we consider *realistic* LLVM-IR instruction latency. Here, we set the *reference latency* from the actual hardware, and using the reference latency, the validity of the estimated latency was determined in later sections. In this study, the reference latency values were determined using ARM CoreSight ETMR5 Technical Reference Manual of Cortex-R5 [1]. As the latency values in the manual have ranges, the reference latency also has a range (Table 1).

With this reference latency, the requirements for SHIM performance values (LLVM-IR instruction latency values), are given as follows:

Table 1: Reference Latency

	arithmetic	mul	div	float	fmul	fdiv	load	store	callret	others1	others2
Min	1.0	1.0	4.0	1.0	2.0	20.0	1.0	1.0	1.0	1.0	1.0
Max	1.0	3.0	32.0	4.0	10.0	36.0	1.5	1.5	5.0	3.0	3.0

- More than zero
- Near around reference latency

Latency values that satisfy these requirements are realistic. In this study, we call these *accurate* latency values.

3. Proposed LLVM-IR Instruction Latency Estimation

In this section, an LLVM-IR instruction latency estimation method is proposed, which contains two estimation steps using deep neural networks (*DNNs*).

3.1. Two Step DNN Estimation

The proposed method uses a two-step DNN approach as shown in Fig. 10, where the first step inputs execution counts (*ECs*) of LLVM-IR instructions and estimates ECs of the target assembler instructions, whereas the second inputs the ECs of the target assembler instructions and estimates the latencies of LLVM-IR instructions. The input data of the first step are prepared in a manner similar to the existing method shown in Fig. 9.

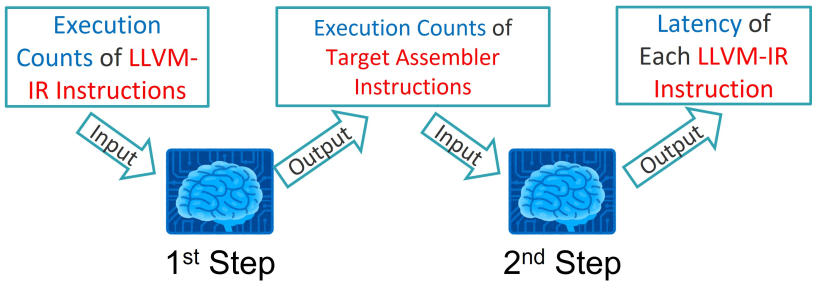

Execution Counts of LLVM-IR Instructions
Input
Output
1st Step
Execution Counts of Target Assembler Instructions
Input
Output
2nd Step
Latency of Each LLVM-IR Instruction

Figure 10: Two Step Estimation

Here, we propose *instruction variables (IVs)* where a group of instructions with similar latencies is represented by variable. In addition, LLVM-IR and target assembler instructions are associated with the IVs. Table 2 presents the *instruction relation table*, which represents an example of the relationship among IVs, LLVM-IR, and target assembler instructions.

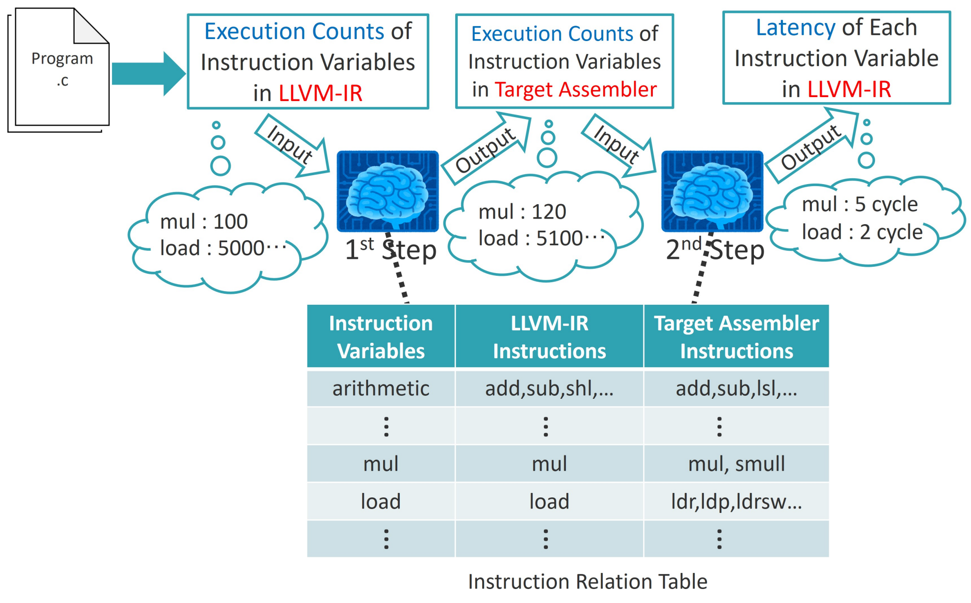
Table 2: Instruction Relation Table

Instruction Variable	LLVM-IR	Assembler
arithmetic	add, sub, shl,...	add, sub, lsl,...
...	...	...
mul	mul	mul, smull
load	load	ldr, ldp, ldrsw,..
...	...	...

The instruction relation table used in the experiments is presented in Table 6 in Appendix B, where an ARM Cortex-A53 was utilized as the target device.

In our method, the execution counts and latencies were considered for each IV. $EC(IV_i)$ and $EC(Total)$ denote the execution counts for IV_i and total execution counts, respectively. Therefore, $EC(Total) = \sum_i EC(IV_i)$.

Our two-step estimation method is as follows (Fig. 11). In the first step, called *execution count estimation in target assembler (ECT)*, the DNN inputs a set of $EC(IV_*)$ into LLVM-IR and estimates $EC(IV_i)$ in the target assembler for each IV_i. In the second step, called the *LLVM-IR instruction latency estimation (LIL)*, the DNN inputs a set of $EC(IV_*)$ into the target assembler and estimates the instruction latencies of LLVM-IR.


Program .c
Execution Counts of Instruction Variables in LLVM-IR
Execution Counts of Instruction Variables in Target Assembler
Latency of Each Instruction Variable in LLVM-IR
Input
Output
Input
Output
mul : 100 load : 5000···
1st Step
mul : 120 load : 5100···
2nd Step
mul : 5 cycle load : 2 cycle

Instruction Variables	LLVM-IR Instructions	Target Assembler Instructions
arithmetic	add,sub,shl,...	add,sub,lsl,...
⋮	⋮	⋮
mul	mul	mul, smull
load	load	ldr,ldp,ldrsw...
⋮	⋮	⋮

Instruction Relation Table

Figure 11: Two Step Estimation using Instruction Relation Table

Use of target assembler instructions between two DNNs solves the issues of SHIM performance estimation described above. In Issue1, the target assembler instructions are generated by considering the number of registers that the target device has. In Issue2, DNN can deal with several types of target assembler instructions generated using a sequence of LLVM-IR instructions.

These steps are described in detail in the following sections.

3.2. Execution Count Estimation in Target Assembler (ECT)

In ECT, the DNN inputs a set of $EC(IV_*)$ into LLVM-IR and estimate $EC(IV_i)$ in the target assembler for each IV_i. To address this problem, we propose a method that includes the following two types of DNN estimations, where the *occurrence ratio* $(OR(IV_i))$ for instruction variable IV_i is defined as the ratio of $EC(IV_i)$ to $EC(Total)$.

1. Total Count: Estimate $EC(Total)$ in the target assembler
2. Occurrence Ratio: Estimate $OR(IV_i)$ in the target assembler

Using the total count and occurrence ratio, the *summarized algorithm* proposed later estimates the ECT by considering the characteristics of each instruction variable.

Table 3: Modified Instruction Variables (MIVs)

Instruction Variable	LLVM-IR Instruction
single	add, sub, ashr, trunc, shl, lshr, xor, icmp, fcmp, load, store
few	fadd, fsub, fptrunc, call, ret, sitofp, fptosi, sext, zext, bitcast, fptoui, uitofp, ptrtoint, inttoptr, fpext
more	mul, sdiv, udiv, srem, urem, fmul, fdiv, frem
others	Other Instructions

It should be noted that to increase the amount of training data, we divide the training programs into basic blocks, and the training processes are performed on a set of basic blocks.

3.2.1. Total Count

For the total count estimation, we propose a DNN that inputs a set of $EC(MIV_*)$ in LLVM-IR, which uses *modified* instruction variables MIVs, and estimates the T/L *ratio*, which is the ratio of $EC(Total)$ in the target assembler to that in LLVM-IR. By multiplying the T/L ratio and $EC(Total)$ in LLVM-IR, the total count ($EC(Total)$ in the target assembler) is calculated. To estimate the total count effectively, we propose *modified* instruction variables where each LLVM-IR instruction is classified into four categories based on the number of target assembler instructions generated (Table 3).

In this classification, MIV *single* includes the instructions from which a single-target assembler instruction is generated. One or more assembler instructions are generated from the MIV *few*, whereas multiple-target assembler instructions are generated from MIV *more*.

The structure of the proposed DNN is illustrated as follows (Fig. 12):

Library	Keras, Tensorflow
Tool	Google Colaboratory
Model	Functions API
# of Neurons in Input Layer	4 (# of modified instruction variables)
# of Neurons in Output Layer	1 (T/L ratio)
# of Inner Layers	24
# of Neurons in Inner Layers	8
Activating Function	ReLU Function (Linear Function for Output Layer)
Place of Dropout	Every 6 Layers
Ratio of Dropout	0.1
Learning Rate	0.0001
# of Epochs	500

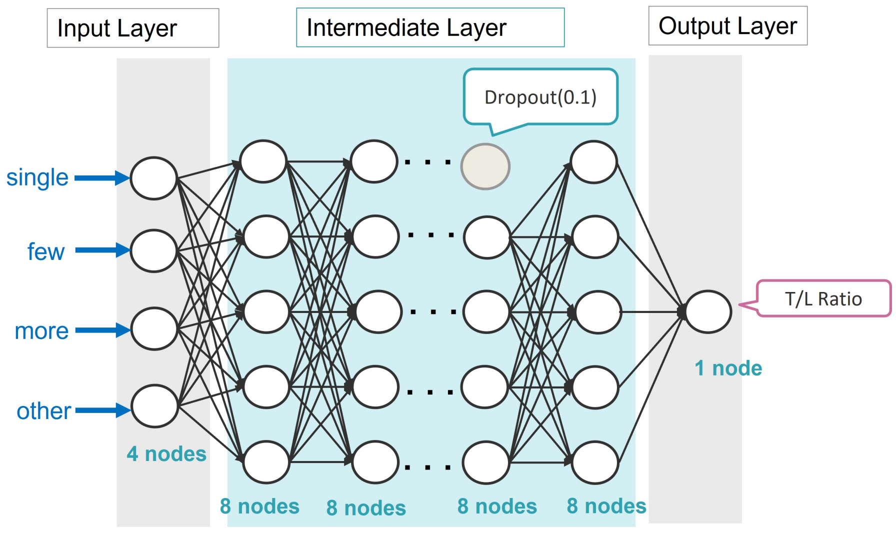


Figure 12: DNN Structure for Total Count Estimation

3.2.2. occurrence Ratio

DNN estimating $OR(IV_i)$ for an instruction variable IV_i in target assembler, inputs a set of $OR(IV_*)$ into LLVM-IR (11 variables in Table 6), and estimate $OR(IV_i)$ in the target assembler for a specified instruction variable IV_i. In our method, ORs were calculated only for the arithmetic, load, and store IVs. For the other IVs, we devised other techniques as described later because the amount of training data was small or the occurrence count was almost equal between LLVM-IR and the target assembler codes.

The structure of the proposed DNN for the arithmetic instruction variable is as follows. (Fig. 13).

Library	Keras, Tensorflow
Tool	Google Colaboratory
Model	Functions API
# of Neurons in Input Layer	11 (# of instruction variables)
# of Neurons in Output Layer	1 (occurrence ratio)
# of Inner Layers	3
# of Neurons in Inner Layers	8, 8, 5
Activating Function	ReLU Function (Linear Function for Output Layer)
Place of Dropout	Between 2nd and 3rd Layers
Ratio of Dropout	0.2
Learning Rate	0.001
# of Epochs	1000

The structure of the proposed DNN for the load instruction variable is as follows: (Fig. 14):

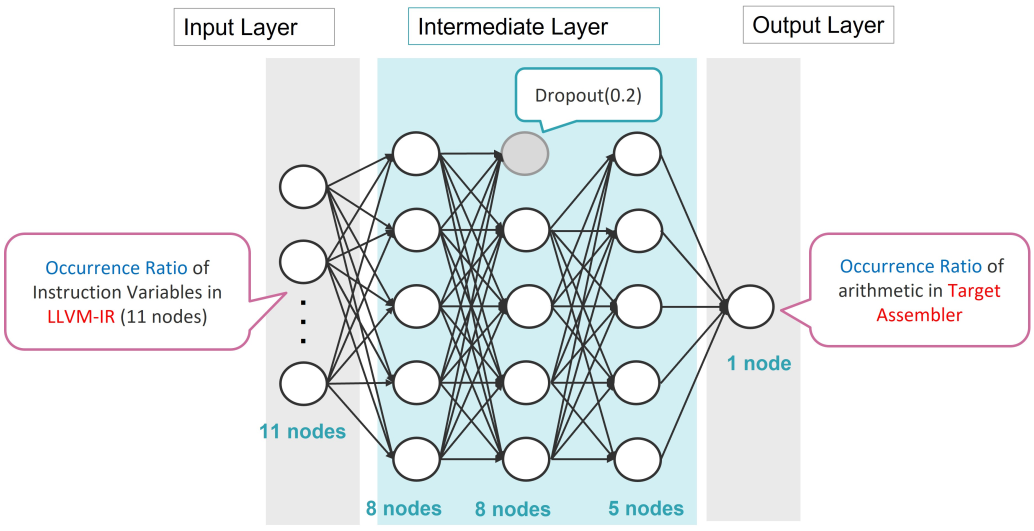

Input Layer
Intermediate Layer
Output Layer
Dropout(0.2)
Occurrence Ratio of Instruction Variables in LLVM-IR (11 nodes)
Occurrence Ratio of arithmetic in Target Assembler
11 nodes
8 nodes
8 nodes
5 nodes
1 node

Figure 13: DNN Structure in Occurrence Ratio Estimation for Arithmetic

Library	Keras, Tensorflow
Tool	Google Colaboratory
Model	Functions API
# of Neurons in Input Layer	11 (# of instruction variables)
# of Neurons in Output Layer	1 (occurrence ratio)
# of Inner Layers	4
# of Neurons in Inner Layers	16
Activating Function	ReLU Function (Linear Function for Output Layer)
Learning Rate	0.001
# of Epochs	1000

The structure of the proposed DNN for the store instruction variable is as follows: (Fig. 15):

Library	Keras, Tensorflow
Tool	Google Colaboratory
Model	Functions API
# of Neurons in Input Layer	11 (# of instruction variables)
# of Neurons in Output Layer	1 (occurrence ratio)
# of Inner Layers	1
# of Neurons in Inner Layers	4

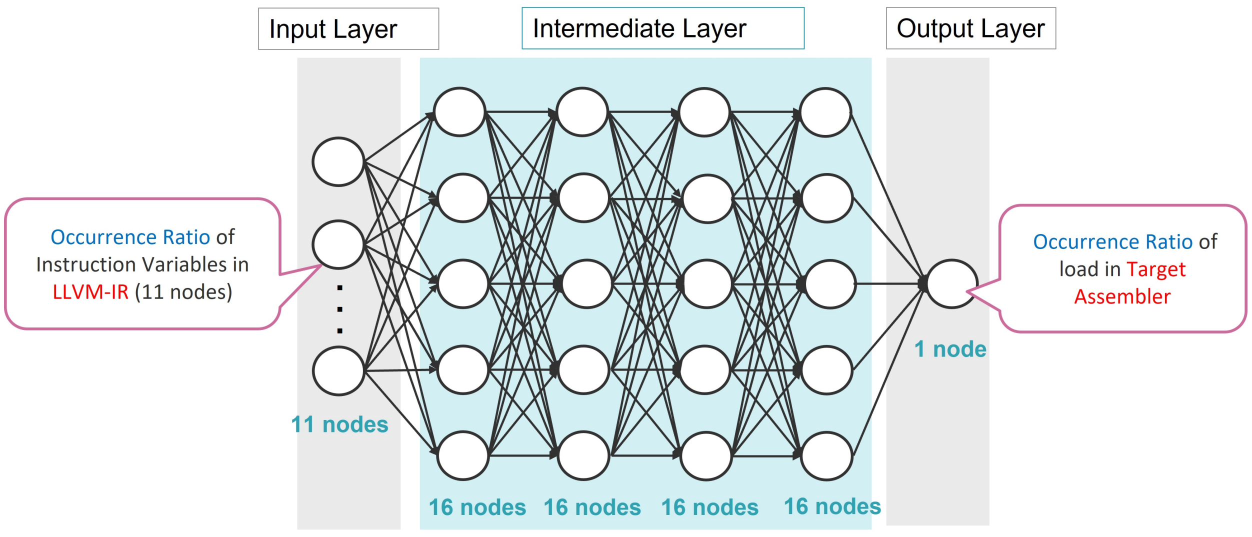

Input Layer
Intermediate Layer
Output Layer
Occurrence Ratio of Instruction Variables in LLVM-IR (11 nodes)
11 nodes
16 nodes
16 nodes
16 nodes
16 nodes
1 node
Occurrence Ratio of load in Target Assembler

Figure 14: DNN Structure in Occurence Ratio Estimation for load

Activating Function	ReLU Function (Linear Function for Output Layer)
Learning Rate	0.001
# of Epochs	1000

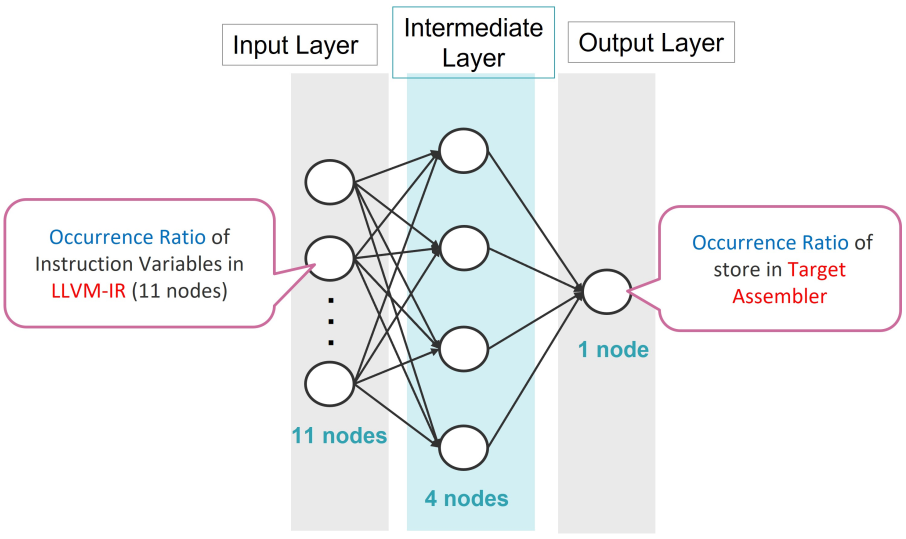

Input Layer
Intermediate Layer
Output Layer
Occurrence Ratio of Instruction Variables in LLVM-IR (11 nodes)
11 nodes
4 nodes
1 node
Occurrence Ratio of store in Target Assembler

Figure 15: DNN Structure in Occurrence Ratio Estimation for store

3.2.3. Summarized Algorithm

In this section, we propose a summarized algorithm to estimate ECT considering total count, occurrence ratio, and characteristics of each instruction variable.

The summarized algorithm estimates $EC(IV_i)$ for each IV_i in the target assembler as follows:

1. For IVs *arithmetic*, *load*, and *store*, $EC(IV_i)$ in target assembler is estimated by multiplying the two estimated values: *Total Count* $EC(Total)$ and *Occurrence Ratio* $OR(IV_i)$.

2. For IVs *fmul*, *fdiv*, and *callret*, $EC(IV_i)$ in target assembler is set to that in LLVM-IR.
3. For IVs *mul*, *div*, *float*, and *others1*, $EC(IV_i)$ in target assembler is estimated by multiplying estimated *Total Count* $EC(Total)$ in target assembler and actual occurrence ratio $OR(IV_i)$ in LLVM-IR.
4. $EC(IV_i)$ for IV *others2* is calculated by subtracting 1, 2, and 3 from *Total Count* $EC(Total)$.

First, for IVs *arithmetic*, *load*, and *store*, because occurrence counts are relatively large and DNN estimation is effective, the summarized algorithm estimates $EC(IV_i)$ by multiplying the two estimated values: *total count* $EC(Total)$, and *occurrence ratio* $OR(IV_i)$. Next, for IVs *fmul*, *fdiv*, and *callret*, the occurrence counts in the target assembler are almost equal to those in LLVM-IR. The summarized algorithm uses the occurrence counts in LLVM-IR. For IVs *mul*, *div*, *float*, and *others1*, because occurrence counts are relatively large and the occurrence ratio in LLVM-IR was considered a good estimate. The summarized algorithm estimates the execution count by multiplying the estimated *total count* $EC(Total)$ in the target assembler and the actual occurrence ratio $OR(IV_i)$ in LLVM-IR.

3.3. LLVM-IR Instruction Latency Estimation (LIL)

Second, we propose another DNN that inputs the results of the first step, that is a set of $EC(IV_*)$ in the target assembler and outputs LLVM-IR instruction latency.

Because this is a regression problem, supervised learning is generally used. However, in our case, we could not prepare the training data for supervised learning because each training data point should include correct outputs, that is a set of correct LLVM-IR instruction latencies that are unknown.

To address this problem, we propose a loss function that enables supervised learning. Although measuring LLVM-IR instruction latency is difficult in an evaluation environment with a target device, it may not be difficult to obtain the actual execution cycles for programs in the environment. Therefore, a loss function can be used to exploit the error between the actual and estimated execution cycles using a set of LLVM-IR instruction latency outputs from DNN.

Based on these observations, *loss* is defined as follows:

$$loss = \frac{1}{n}\sum_j \left(\frac{cycle_j - p\_cycle_j}{cycle_j}\right)^2 \tag{3}$$

$$p\_cycle_j = \sum_i EC(IV_i)_j \times \hat{y}_{i,j}, \tag{4}$$

where n is the number of programs and $cycle_j$ is the number of execution cycles of program j measured on the target device. p_cycle_j is the execution cycle of program j estimated using Eq. (4), where $EC(IV_i)_j$ is $EC(IV_i)$ for program j obtained using a coverage tool for LLVM-IR such as llvm-cov and $\hat{y}_{i,j}$ is the latency of the IV_i output from DNN for program j. The *loss* function is considered the mean square error.

The DNN structure for LIL estimation is shown as follows (Fig. 16):

Library	Keras, Tensorflow
Tool	Google Colaboratory
Model	Subclassing API
# of Neurons in Input Layer	11 (# of instruction variables)

# of Neurons in Output Layer	11 (# of instruction variables)
# of Inner Layers	4
# of Neurons in Inner Layers	64
Activating Function	ReLU Function (Linear Function for Output Layer)
Place of Dropout	Between 3rd and 4th Layers
Ratio of Dropout	0.2
Learning Rate	0.001
# of Epochs	1000

In our implementation, the input values are normalized with an average of zero and a variance of one, whereas the outputs are LLVM-IR instruction latencies for the instruction variables. The Subclassing API in TensorFlow 2.0 is utilized because its high programmability is appropriate for realizing a loss function. To learn larger latency of IVs such as $fmul$ and $fdiv$, a larger number of neurons was used in our network. The use of dropout prevents overlearning, and the ReLU function is effective for estimating the latency that has a positive value.

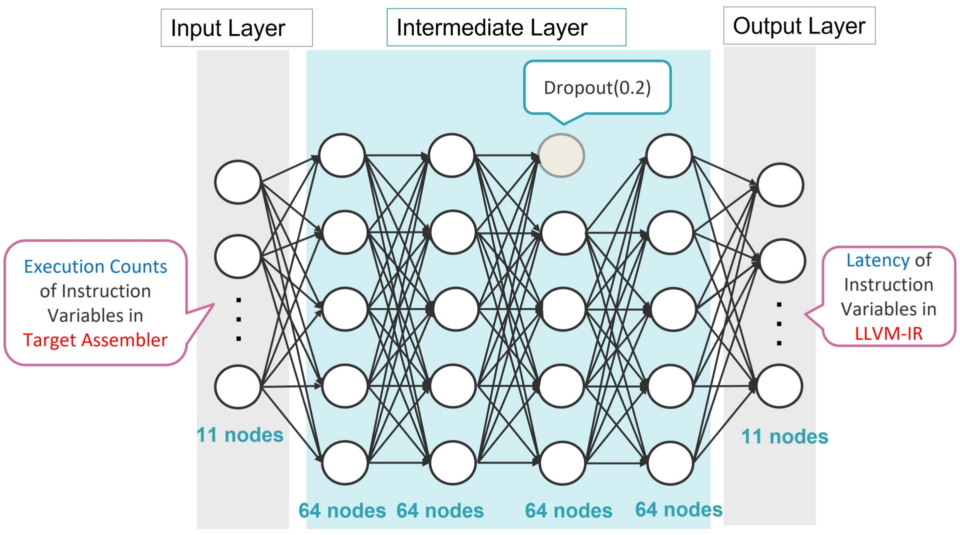

Input Layer
Intermediate Layer
Output Layer
Dropout(0.2)
Execution Counts of Instruction Variables in Target Assembler
Latency of Instruction Variables in LLVM-IR
11 nodes
64 nodes
64 nodes
64 nodes
64 nodes
11 nodes

Figure 16: DNN Structure for LIL Estimation

Notably, with the proposed DNN, the LLVM-IR instruction latency for IV_i is the output as $\hat{y}_{i,j}$ in Eq. (4), which means that the values also depend on the program j. Therefore, we use the last values in the training phase as the estimated latency, and the values are assumed to be written in SHIM.

3.4. Software Performance Evaluation with SHIM

Using the ECT and LIL methods described above, we proposed a software performance estimation method, which estimates the execution cycles of a program using Eq. (5).

$$Execution\ Cycles = \sum_i (EC(IV_i) \times Latency_i), \tag{5}$$

where $EC(IV_i)$ and $Latency_i$ are the execution counts in LLVM-IR and LIL, respectively. Note that Eq. (5) is almost equal to Eq. (1).

4. Experiments

In this section, we evaluate the proposed method introduced in the previous sections for the Raspberry Pi3 Model B+.

4.1. Target Hardware

The target hardware in our experiments was a Raspberry Pi3 Model B+ with the following basic specifications.

CPU	Cortex-A53
Number of cores	4 cores
CPU clock	1.4 GHz
Memory	1 GB
Cache	L1 Cache 32 KB
	L2 Cache 512 KB
OS	Linux ver 4.14.68

4.2. Measurement Programs

We prepared 40 programs in C language for training (Table 4). The programs were designed to increase the variations in LLVM-IR instructions.

4.3. ECT Estimation

In this section, we evaluate the first step of our method, that is, ECT estimation.

4.3.1. Total Count

First, we evaluate the total count estimation in the first step. In these experiments, we used 410 basic blocks for training generated from the 40 programs in Table 4. After the training phase, we estimated $EC(Total)$ in the target assembler for 40 programs and calculated the error ratio using Eq. (6).

$$Error\ Ratio = \frac{(Actual\ Execution\ Count) - (Estimated\ Execution\ count)}{(Actual\ Execution\ Count)} \tag{6}$$

Figure 17 presents a histogram of the error ratios.

An important factor in the estimation of total count is the absence of outliers, in addition to high accuracy. Figure 17 shows that 30 of the 40 programs exhibited an error ratio of ±20%. The mean absolute error ratio was 12.35%. No outliers were observed, and errors of this magnitude were considered to have little effect on latency estimation. For programs that fall outside of the ±20% range, we confirmed that the ratio of $EC(Total)$ in the target assembler to that in LLVM-IR was far from 1.0. Specifically, programs with a factor of less than 0.8 or greater than 1.2 are included in ±20%. Therefore, there is room for improvement in the estimation of the total count for programs with large differences in $EC(Total)$ between LLVM-IR and the assembler.

Table 4: 36 Programs for Training and Their Basic Blocks

Program	# of Basic Blocks		
BubbleSort.c	9	RadixConversion.c	6
CelsiustoFahren.c	7	RadixConversion_F.c	6
EuclideanAlgorithm.c	13	Selectionsort.c	11
Exchange.c	6	SeriesSum.c	2
Exchange_F.c	6	Shellsort.c	23
fadd.c	3	SimpleFilterProcess.c	9
fcmp.c	2	sum.c	3
Fibonacci.c	5	Insertionsort.c	14
Fibonacci_F.c	5	Functions2.c	3
fmul.c	3	Functions3.c	6
FunctionCallTest.c	9	Functions4.c	11
GaussianElimination.c	21	Functions5.c	14
hanoi.c	5	Functions6.c	9
MergeSort.c	21	Functions7.c	9
NapierNumber.c	6	Functions8.c	14
NumberPlace.c	33	Functions9.c	5
PerfectNumber.c	9	Functions10.c	17
PerfectNumber_F.c	9	Functions11.c	14
PrimeNumber.c	12	Functions12.c	19
QuickSort.c	14	Functions13.c	17

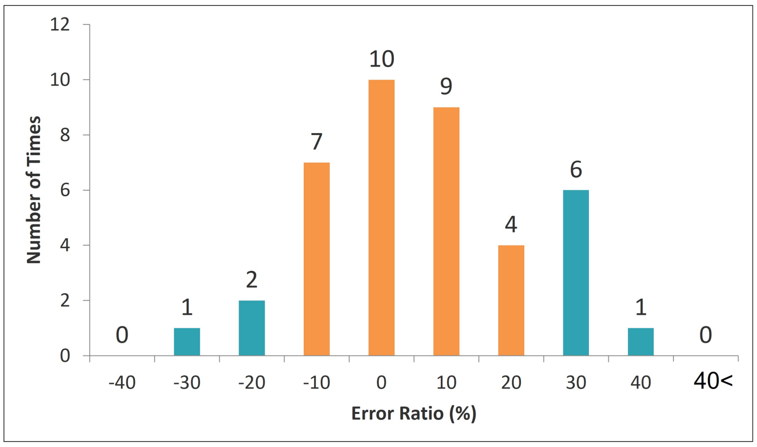


Figure 17: Histogram of Error Ratio for Total Count

4.3.2. Occurrence Ratio

Next, the estimation of the occurrence ratio was evaluated for IVs arithmetic, load, and store. The DNN inputs a set of $OR(IV_*)$s into the basic blocks of LLVM-IR codes from the 40 programs. To evaluate the training, we estimated the occurrence ratios in the target assembler codes from 40 programs and calculated the error ratio using Eq. (7).

$$Error\ Ratio = \frac{(Actual\ Occureance\ Ratio) - (Estimated\ Occureance\ Ratio)}{(Actual\ Occureance\ Ratio)} \tag{7}$$

Figures 18, 19, and 20 show histograms of the error ratios for IVs arithmetic, load, and store, respectively.

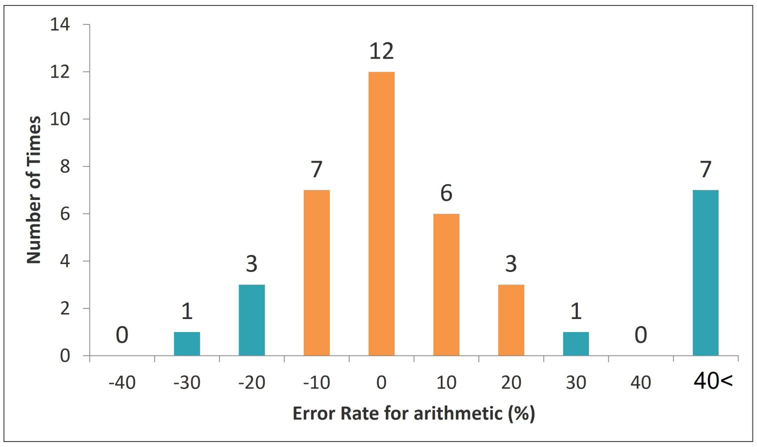


Figure 18: Occurence Ratio for arithmetic

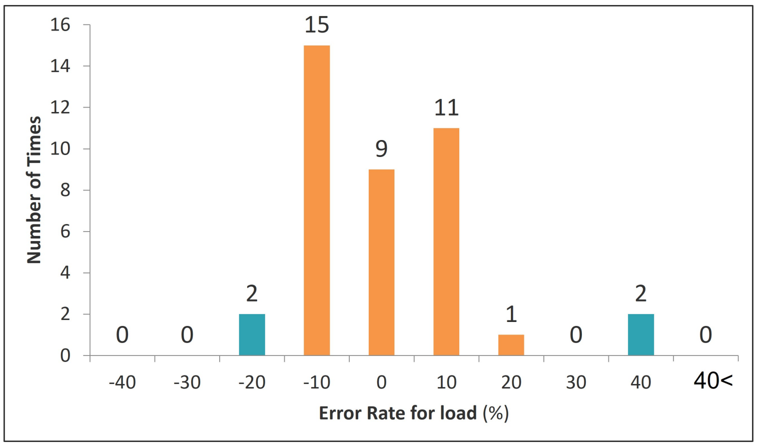


Figure 19: Occurence Ratio for load

In the arithmetic estimation results, 28 of the 40 programs are within an error ratio of ±20%. In particular, the interval 0% ∼ 10% is the largest. However, seven outliers are also observed. It is confirmed for these programs that $EC(arithmetic)$ in the assembler code differs by approximately ±20% from $EC(arithmetic)$ in the LLVM-IR code. The other programs differ by approximately ±8%, indicating that the larger the difference is between $EC(arithmetic)$s of the LLVM-IR and the target assembler, the larger is the error in the estimation results.

In estimating the load, 36 of the 40 programs were within an error ratio of ±20%. As there are more load instructions than arithmetic instructions, more accurate results are obtained than the arithmetic results. Programs that fall between

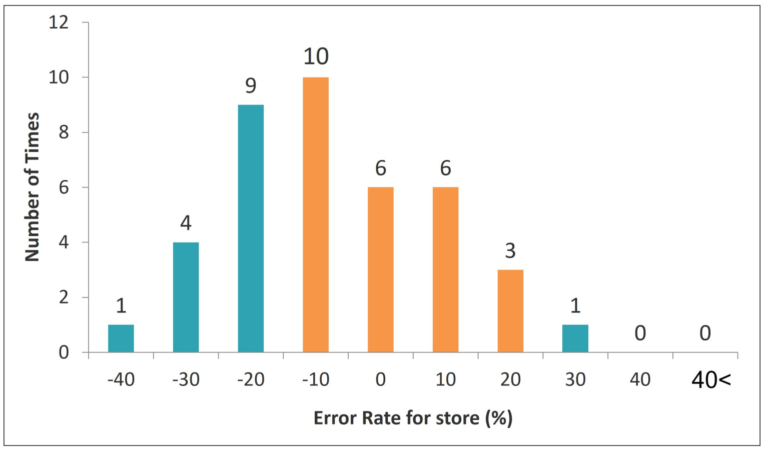


Figure 20: Occurence Ratio for store

30% and 40% are those in which LLVM-IR and assembler differ in the proportion of occurrences as is the case in arithmetic.

For store estimation, 25 of the 40 programs were within an error ratio of ±20%. However, the number of programs that fell within the target range was the lowest. This may be because the number of store instructions is the smallest.

4.3.3. ECT Estimation

ECTs for 40 programs were estimated with the summarized algorithm using the total count and occurrence ratio, and bar graphs that compare the estimated and actual $EC(Total)$ and $EC(IV_*)$ are shown. Figure 21 shows the results for the programs with better estimations, whereas Fig. 22 shows the poorest results.

In Fig. 21, the estimated $EC(Total)$ and $EC(IV_*)$ are similar to the actual values. Because these programs contain more than a thousand instructions, larger programs may provide a better estimation. In SimpleFilterProcess.c and SeriesSum.c, the div instruction appears in the estimated but not in the actual results. Because the div instructions appear in LLVM-IR code for these programs, DNN estimates that it should also appear in the assembler code. It is believed that the div instruction has been replaced with different instructions owing to compiler optimization.

Figure 22 shows the programs for which $EC(Total)$ or $EC(IV_*)$ differed significantly between the estimated and actual values. For Functions9.c, $OR(IV_*)$ is close but $EC(Total)$ differed significantly between the estimated and actual values. For this program, the actual execution counts in the assembler were lower than those in LLVM-IR. It seems that this is the reason why the difference was larger than that in the other programs. In contrast, in fcmp.c, the $EC(Total)$ for the estimated and actual values are close, but the differences in $OR(IV_*)$ are significant. It is observed that the DNN estimates fewer arithmetic instructions and more others1 instructions. fcmp.c has many operations for assigning comparison results (Boolean values) to variables. In LLVM-IR, these operations are implemented as cast operations after comparison, while the assembler does not cast. This seems to be the reason why others1 instructions are smaller in the estimation.

Good results were obtained when total count was estimated to be close to the actual count.

4.4. LIL Estimation

In this section, the second step, LIL estimation, is evaluated.

4.4.1. Experiments and Results

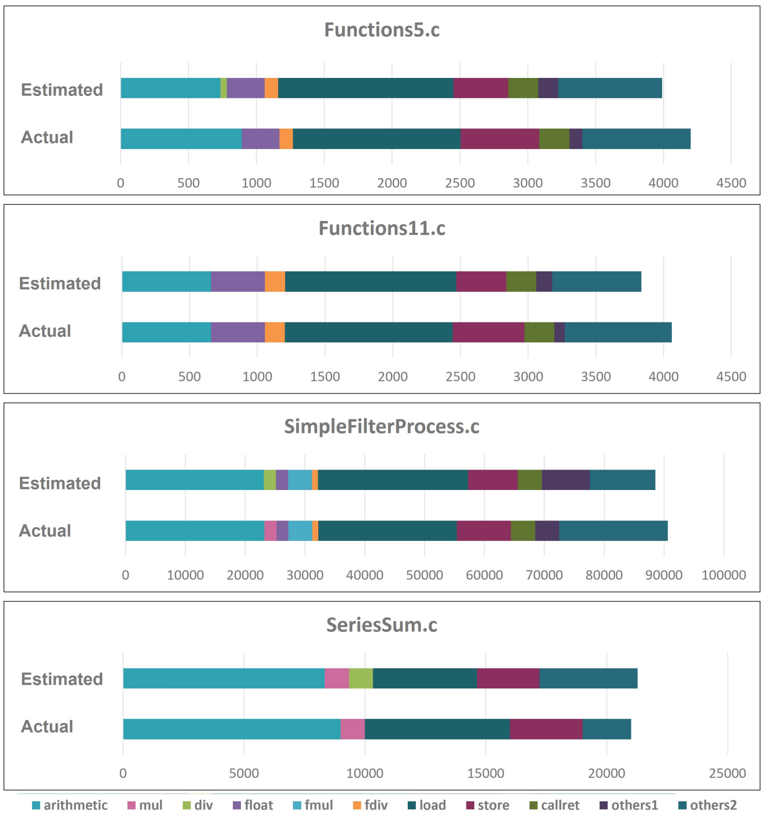


Figure 21: ECT Estimation (Better Results)

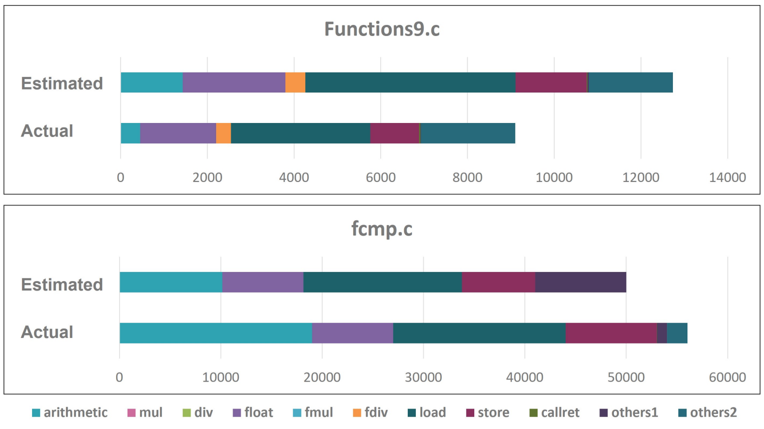


Figure 22: ECT Estimation (Worse Results)

In these experiments, 36 of 40 programs were used for training or regression analysis, and the others, Functions11.c, Functions3.c, Functions4.c, and Selectionsort.c, are utilized for evaluation. We evaluated three methods as follows:

- Regression: Existing method [11]
- Proposed: Proposed two-step DNN method, in which the inputs of the second step (LIL) are the outputs of the first phase (ECT), that is, the estimated $EC(IV_*)$ in target assembler
- LIL (Assembler): Method using only DNN proposed for LIL, which inputs actual $EC(IV_*)$ in the target assembler
- Reference Latency: Reference latency (Table 1)

Table 5 lists the results of LLVM-IR instruction latency estimation, where the rows and columns represent the methods and IVs, respectively. For IVs, arith, oth1, and oth2 represent arithmetic, others1, and others2, respectively. Fig. 23 depicts them, where three marks on the vertical bar for each IV represent maximum, average, and minimum reference latencies from the top of the bar, respectively.

Table 5: LLVM-IR Instruction Latency Estimation Results

	arith	mul	div	float	fmul	fdiv	load	store	callret	oth1	oth2
Regression	1.63	0	-	4.56	17.48	-	2.69	0	3.80	9.49	0.20
Proposed	1.17	0.95	1.41	4.44	10.48	13.28	1.56	1.07	2.82	0.91	1.22
LIL (Assembler)	1.04	1.12	5.54	7.09	23.56	31.36	1.06	1.01	7.18	1.03	1.00
Reference Latency	$1 \sim 1$	$1 \sim 3$	$4 \sim 32$	$1 \sim 4$	$2 \sim 10$	$20 \sim 36$	$1 \sim 1.5$	$1 \sim 1.5$	$1 \sim 5$	$1 \sim 3$	$1 \sim 3$

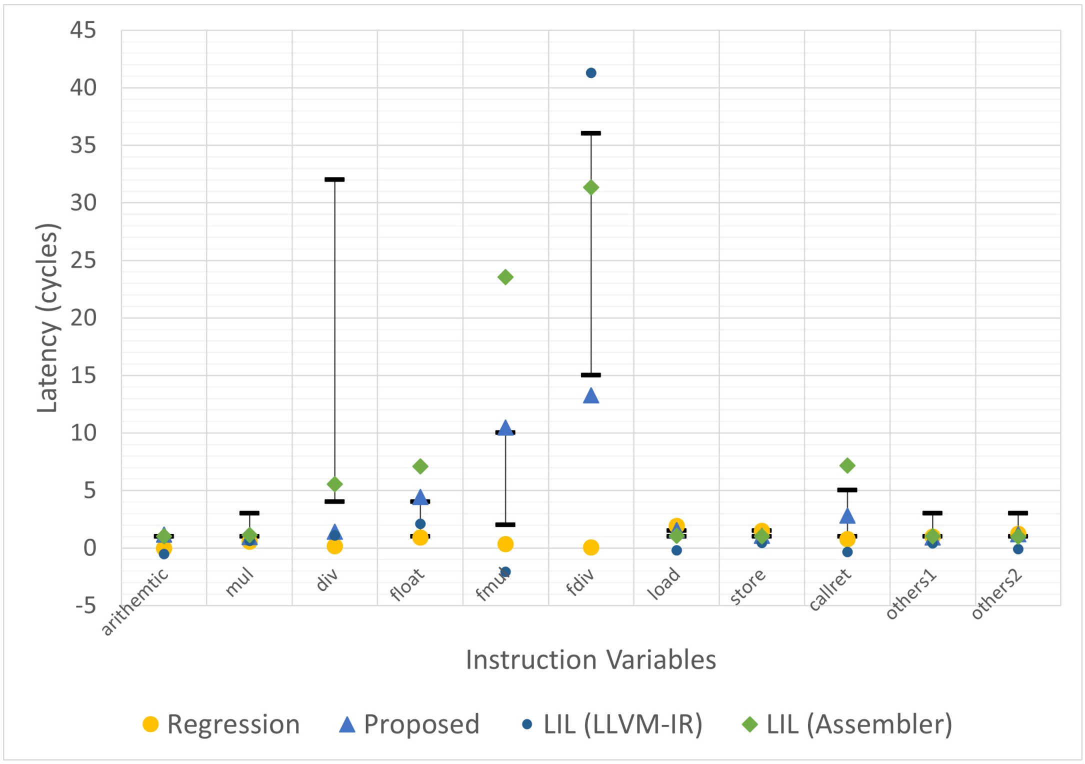


Figure 23: LLVM-IR Instruction Latency Estimation Results

In addition, Fig. 24 shows the results of the software performance evaluation calculated using Eq. (5) described in the previous section, using the LLVM-IR latency estimated using each method.

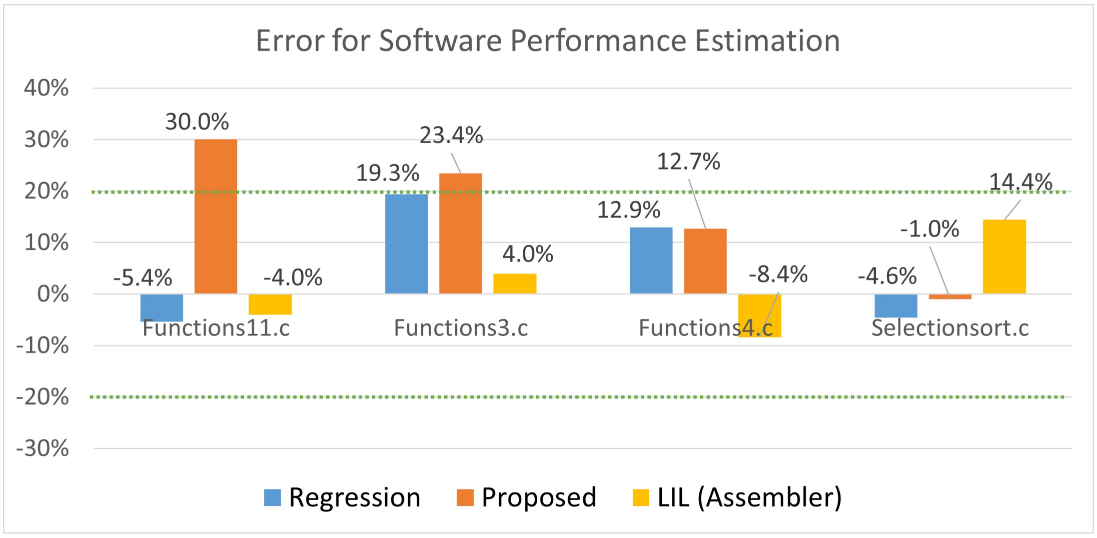


Figure 24: Performance Estimation for Evaluation Software

To compare the proposed method with existing methods [11], we compare the *regression* and *proposed* methods in Table 5 and Fig. 23.

Some of the latency values estimated using the regression method were zero, whereas the values in the proposed method are near the reference latency range. Therefore, our method is considered to be much more accurate than the existing methods, in terms of LLVM-IR instruction latency estimation.

The software performance estimates are shown in Fig. 24. The proposed method estimated errors outside the range of $\pm 20\%$, although the error remains within $\pm 30\%$. To show the reason for the larger error compared to the existing method, we compare the case of *LIL (assembler)* in Fig. 24, where the second step of DNN inputs actual $EC(IV_*)$. Because the estimation error of the *LIL (assembler)* is within $\pm 20\%$ and is significantly better than that of the regression method, the cause of this large error is the first step of DNN. This issue should be addressed in future studies.

Conclusions

In this study, we proposed a method using deep neural networks to estimate LLVM-IR instruction latency for SHIM XML generation used for model-based parallelization and other applications. The proposed method uses a two-step DNN. As a first step, we propose a method to estimate the execution counts of the assembler instructions using the LLVM-IR instructions as the input. To avoid outliers, we first estimate the total execution counts of the assembler instructions and then combine it with the estimated instruction occurrence ratio. For the execution count estimation, we constructed a DNN and achieved a target error ratio of $\pm 20\%$ in 30 of the 40 evaluation programs, with no outliers. The error ratio target was achieved in the instruction occurrence ratio estimation for approximately 73% of all the instructions by devising an estimation method tailored to the characteristics of each instruction group.

In the second step, we propose another DNN to estimate LLVM-IR instruction latency from a set of execution counts for the target assembler instructions in which we devise a new loss function

Using a Raspberry Pi3 Model B+ as the target hardware, we evaluated our method and compared it with existing methods. From the experimental results, it can be observed that our proposed method performs much better than existing methods for LLVM-IR latency estimation, although the software performance estimation using our method was worse than that of the existing method.

Since software performance was much better estimated when the actual assembler execution counts were input into the second step of DNNs, we plan to improve the first step of DNNs in future studies.

Conflicts of Interest

The authors have no conflict of interest about anything in this article

References

[1] ARM Limited. "ARM CoreSight ETM-R5 Technical Reference Manual r0p0," (Referred 2023-06-25). https://developer.arm.com/documentation/ddi0469/b/?lang=en.

[2] EMC, "The Multicore Association Specifications", (Accessed 2023-06-25). https://www.embeddedmulticore.org/the-multicore-association-specifications/.

[3] eSOL Co. Ltd., "eMBP (Model Based Parallelizer)," (Accessed 2023-06-25). https://www.esol.com/embedded/product/embp_overview.html.

[4] Gondo, Masaki, Fumio Arakawa and Masato Edahiro, "Establishing a standard interface between multi-manycore and software tools - SHIM", *COOL Chips XVII*, VI-1, 2014.

[5] Hwang, Yonghyun, Samar Abdi, and Daniel Gajski. "Cycle-approximate retargetable performance estimation at the transaction level." Proceedings of the Conference on Design, Automation and Test in Europe. 2008.

[6] IEEE, "IEEE Standard for Software-Hardware Interface for Multi-Many-Core", IEEE 2804-2019, (Accessed 2023-06-25). https://standards.ieee.org/ieee/2804/7477/.

[7] Kasahara, Hironori, Honda, Hiroki, Mogi, A., Ogura, A., Fujiwara, F., Na rita, Seinosuke. “A multi-grain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor).” International Workshop on Languages and Compilers for Parallel Computing, Springer, Berlin, Heidelberg, pp.283-297, 1991.

[8] LLVM project, "The LLVM Compiler Infrastructure",(Accessed 2023-06-25). https://llvm.org/.

[9] LLVM project, "LLVM Language Reference Manual", (Accessed 2023-06-25). https://llvm.org/docs/LangRef.html.

[10] LLVM project, "llvm-cov", (Accessed 2023-06-25). https://llvm.org/docs/CommandGuide/llvm-cov.html.

[11] Mikami, Hiro, Kei Torigoe, Makoto Inokawa, and Masato Edahiro. "LLVM Instruction Latency Measurement for Software-Hardware Interface for Multi-many-core." International Journal of Computers & Technology, 22 (2022): 50–63. https://doi.org/10.24297/ijct.v22i.9231

[12] Patel, Rajendra, and Arvind Rajawat. "Recent trends in embedded system software performance estimation." Design Automation for Embedded Systems 17.1 (2013): 193-213.

[13] Powell, Daniel Christopher, and Björn Franke. "Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators." Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis. 2009.

[14] Ray, Abhijit, Thambipillai Srikanthan, and Jigang Wu. "Rapid techniques for performance estimation of processors." Journal of Research and Practice in Information Technology 42.2 (2010): 147-165.

[15] Renesas electronics, "RH850/E1M-S2", (Accessed 2023-06-25). https://www.renesas.com/jp/en/products/microcontrollers-microprocessors/rh850-automotive-mcus.

[16] SHIM Working Group, "SHIM Latency Measurement and Insertion", (Accessed 2023-06-25). https://github.com/openshim/shim2/tree/master/shim-measure.

[17] Wijesundera, Deshya, et al. "Framework for rapid performance estimation of embedded soft core processors." ACM Transactions on Reconfigurable Technology and Systems (TRETS) 11.2 (2018): 1-21.

Author Biographies

Hiro Mikami received B.S.T. degree in the Faculty of Science and Technology, Nanzan University in 2021, and received M.I. degree in the Graduate School of Informatics, Nagoya University in 2023.

Seira Iwai is a Master's student in the Graduate School of Informatics, Nagoya University. She received B.I. degree in the School of Informatics, Nagoya University in 2023.

Masato Edahiro is a Professor at the Department of Computing and Software Systems, the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Computer Science from Princeton University in 1999. He joined NEC Corporation in 1985, had worked on EDA and multicore SoCs and their software for mobile phones and automotive vehicles, and moved to Nagoya University in 2011. His research topics include graph and network algorithms and software for multi- and many-core processors. He is a member of IEEE, IPSJ, IEICE, ORSJ.

Appendix A. Top-level class diagram of SHIM XML

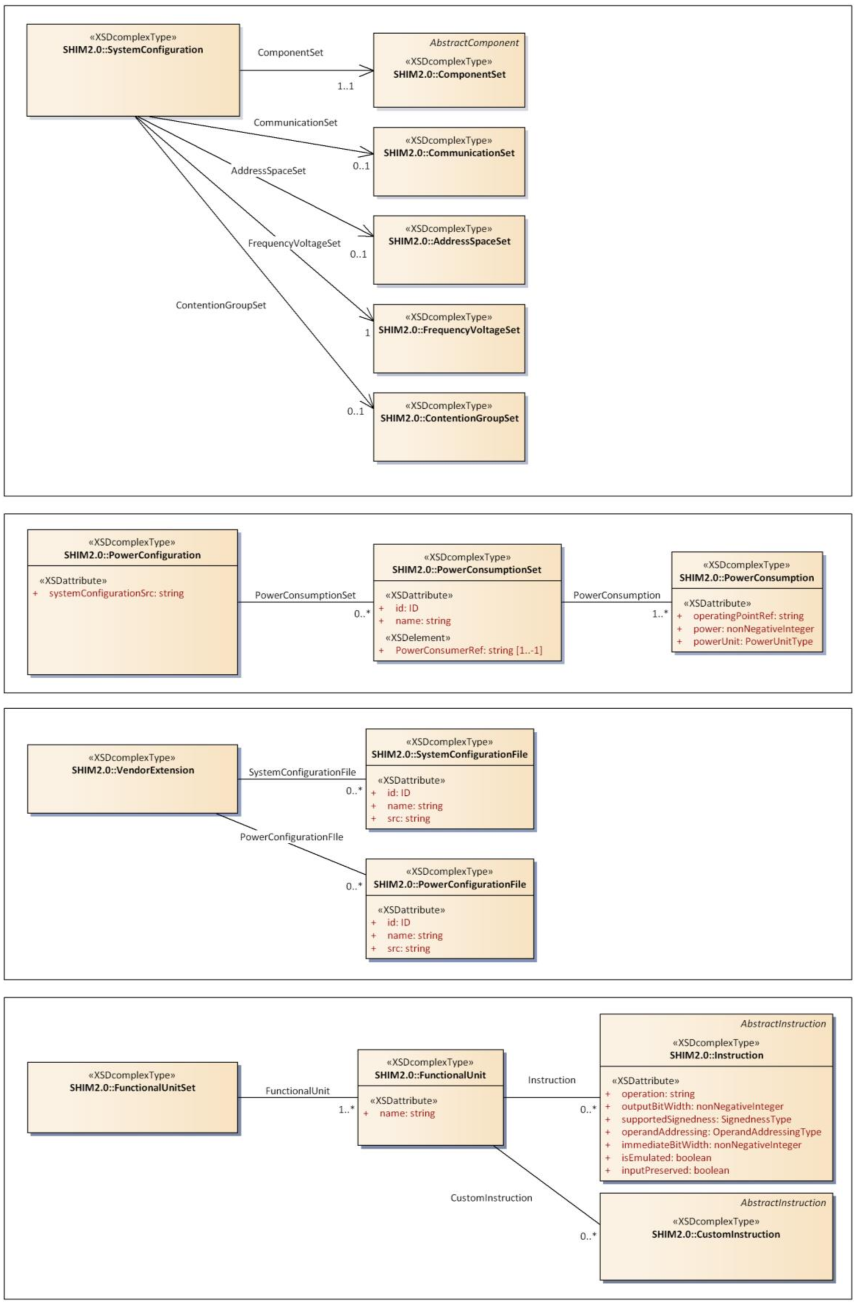

«XSDcomplexType» SHIM2.0::SystemConfiguration
ComponentSet
1..1
AbstractComponent
«XSDcomplexType» SHIM2.0::ComponentSet
CommunicationSet
0..1
«XSDcomplexType» SHIM2.0::CommunicationSet
AddressSpaceSet
0..1
«XSDcomplexType» SHIM2.0::AddressSpaceSet
FrequencyVoltageSet
1
«XSDcomplexType» SHIM2.0::FrequencyVoltageSet
ContentionGroupSet
0..1
«XSDcomplexType» SHIM2.0::ContentionGroupSet
«XSDcomplexType» SHIM2.0::PowerConfiguration
«XSDattribute» + systemConfigurationSrc: string
PowerConsumptionSet
0..*
«XSDcomplexType» SHIM2.0::PowerConsumptionSet
«XSDattribute» + id: ID + name: string
«XSDelement» + PowerConsumerRef: string [1..-1]
PowerConsumption
1..*
«XSDcomplexType» SHIM2.0::PowerConsumption
«XSDattribute» + operatingPointRef: string + power: nonNegativeInteger + powerUnit: PowerUnitType
«XSDcomplexType» SHIM2.0::VendorExtension
SystemConfigurationFile
0..*
«XSDcomplexType» SHIM2.0::SystemConfigurationFile
«XSDattribute» + id: ID + name: string + src: string
PowerConfigurationFile
0..*
«XSDcomplexType» SHIM2.0::PowerConfigurationFile
«XSDattribute» + id: ID + name: string + src: string
«XSDcomplexType» SHIM2.0::FunctionalUnitSet
FunctionalUnit
1..*
«XSDcomplexType» SHIM2.0::FunctionalUnit
«XSDattribute» + name: string
Instruction
0..*
AbstractInstruction
«XSDcomplexType» SHIM2.0::Instruction
«XSDattribute» + operation: string + outputBitWidth: nonNegativeInteger + supportedSignedness: SignednessType + operandAddressing: OperandAddressingType + immediateBitWidth: nonNegativeInteger + isEmulated: boolean + inputPreserved: boolean
CustomInstruction
0..*
AbstractInstruction
«XSDcomplexType» SHIM2.0::CustomInstruction

Figure 25: Class diagram representation of the SHIM XML schema (top-level elements) [6]

Appendix B. Instruction Relation Table

Table 6: Instruction Relation Table (Cortex-A53)

Instruction Variable	LLVM-IR	Assembler
arithmetic	add, sub, ashr, shl, xor, icmp,	add, sub, cmp, lsr, asr, lsl, neg, cset, and
mul	mul	mul, smull
div	sdiv, udiv, srem, urem	div, udiv, sdiv
float	fadd, fsub, fcmp	fadd, fcmp, fcmpe, fneg, fsub
fmul	fmul	fmul
fdiv	fdiv, frem	fdiv
load	load	ldr, ldp, ldrsw, ldrb
store	store	str, stp, strb
callret	call, ret	ret, bl
others1	sitofp, fptosi, sext, zext, bitcast, fptoui, uitofp, ptrtoint, inttoptr, fpext, trunc, fptrunc	scvtf, scvtzs, sxtw, fcvtzs, fcvt, ldrsw, cset
others2	getelementptr, alloca, phi, br, indirectbr, invoke, resume, unrechable, extractelement, insertelement, shufflevector, extractvalue, insertvalue, fence, cmpxchg, atomicrmw, select, va_arg, landingpad, switch	b, ble, blt, bgt, bne, beq, bge, bhi, bmi, bpl, csel, adrp, nop, csneg, mov, fmov, movk, mvn