DOI https://doi.org/10.24297/ijct.v22i.9231

LLVM Instruction Latency Measurement for Software-Hardware Interface for Multi-many-core

Hiro Mikami, Kei Torigoe, Makoto Inokawa, and Masato Edahiro

Graduate School of Informatics, Nagoya University, Japan

pdsl@ertl.jp

Abstract

The increasing scale and complexity of embedded systems and the use of multi-many-core processors have resulted in a corresponding increase in the demand for software development with a high degree of parallelism. The degree of parallelism in software and the accuracy of performance estimation in the early design stages of model-based development can be improved by estimating performance of blocks in models and utilizing the estimate for parallelization. Research is therefore being performed on a software performance estimation technique that uses the IEEE2804-2019 hardware feature description called software-hardware interface for multi-many-core (SHIM). In SHIM, each LLVM-IR instruction is associated with the execution cycle of the target processor. Because several types of assembly instruction sequences for the target processor are generated from a given LLVM-IR instruction, it is not easy to estimate the number of execution cycles. In this study, we propose a regression analysis method to estimate the execution cycles for each LLVM-IR instruction. It is observed that our method estimated the execution cycles within the target error of ±20% in experiments using a Raspberry Pi3 Model B+.

Keywords: SHIM, embedded system, multi-many-core, estimation

1. Introduction

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

In addition, the man-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 errors. In model-based development, the control system is represented by an abstract model and the design is performed on the model.

Owing to these factors, there is a need for parallelization to support multi-many-core model-based development. One approach for parallelization is the code-based approach, in which 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 met after parallelization, the system must be redesigned from scratch, resulting in rework costs.

Another approach for parallelization is the model-based approach [3]. In this approach, parallelization is 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 parallelism, it is necessary to consider the processing time of each block in the model for load balancing. To achieve this, it is crucial to devise a method for estimating the performance of the model design.

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

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

In SHIM, in order to make tools using SHIM target-independent, the LLVM-IR instruction set [8] from the LLVM project [7] is used for the processor instructions instead of the instruction set of the target processor. The use of LLVM-IR has two issues for performance estimation. 1) Since the LLVM-IR assumes infinite number of virtual registers, there are no register spilling in LLVM-IR code. 2) Several code sequences of the target code are generated from an LLVM-IR instruction.

In this study, to overcome these issues, we propose A) a method for software performance estimation using SHIM and B) a method for performance estimation of each LLVM-IR instruction to be stored in SHIM based on a regression analysis. In design flow, first, SHIM is created for the target processor using the proposal B) with the target-dependent environment. Then, performance of a system is estimated with the created SHIM using the proposal A) with the target-independent tools. The key contribution of this study is a reduction in the cost of producing SHIM so as to facilitate the application of SHIM to performance estimation. A case study on a Raspberry Pi3 Model B+ shows that the error of the estimated performance with SHIM is within the target value of $\pm 20\%$.

The rest of this paper is organized as follows: Section 2 provides a brief background of related work and previous studies on performance estimation with SHIM. Section 3 proposes the two methods related to performance estimation with SHIM. Section 4 describes a case study for the performance estimation of LLVM-IR instructions to be stored in SHIM for a Raspberry Pi3 Model B+. Section 5 presents an evaluation of the software performance estimation with SHIM described in Section 4. Section 6 concludes the study and discusses future work.

2. Performance Estimation with SHIM

2.1. SHIM [6]

The widespread use of multi-many-core processors has led to the emergence of architectures with a variety of characteristics from which the user has to select the optimal processor for the specific application. This has led to an increased burden on vendors of tools for the analysis, optimization, automatic parallelization, and real-time operating systems, who are forced to deal with 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 the reading of voluminous manuals and an understanding of the characteristics of each processor.

To address this issue, the Multicore Association [2] has 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 IEEE in 2019 [6]. Figs. 1 to 5 show a schematic diagram of SHIM, what SHIM is, what SHIM can, relationship between SHIM and tools, and its use cases. In addition, the class diagram representation of the SHIM

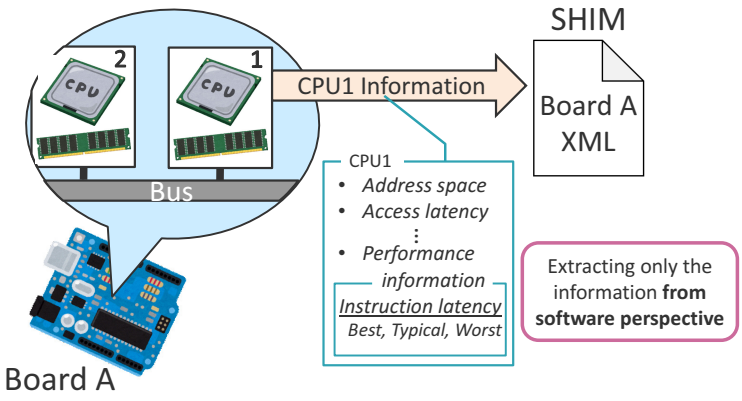

Figure 1: SHIM schematic

XML schema at top-level view [6] is shown in appendix.

- 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

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

2.2. Relationship between SHIM and LLVM-IR

The LLVM-IR instruction set [8] is used in SHIM for the processor instructions instead of the instruction set of 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 hardwares. Each LLVM-IR instruction is associated with its corresponding performance information in SHIM. This performance information is used in conjunction with the LLVM tool chain to estimate software performance for an arbitrary processor without the use of processor-specific tools.

2.3. Related Work

There are many studies [5, 10, 14, 11, 15] on performance estimation at the target instruction level that do not rely on

- 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

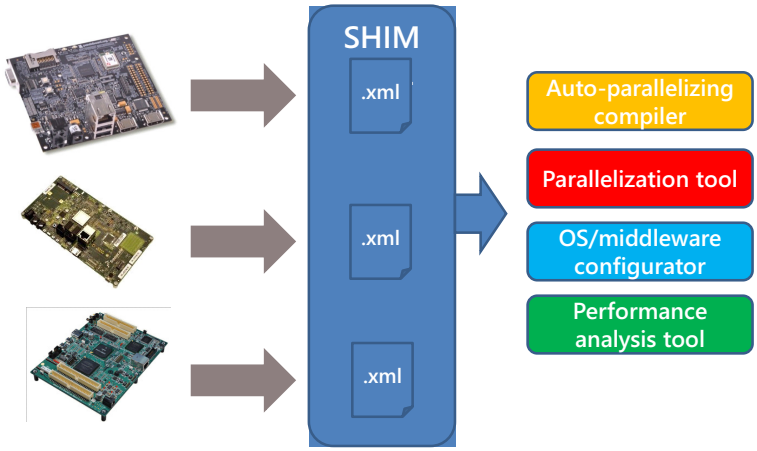

Figure 4: SHIM and tools

SHIM. In general, the number of loop iterations and branch directions cannot be easily estimated in static analysis, whereas dynamic analysis requires an actual machine or ISS, which is not desirable for model-level design. In this study, we do not discuss these issues but consider only the SHIM-specific issues.

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

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

$$Execution\ Cycles = \sum_i (IR\ Execution\ Times_i \times Latency_i), \tag{1}$$

where $IR\ Execution\ Times_i$ is the execution count of $Instruction_i$ in LLVM-IR obtained from the LLVM profiler and $Latency_i$ is the number of execution cycles for $Instruction_i$ stored in SHIM.

The execution cycles in SHIM were obtained with two methods: 1) extracting from the user manual of the target processor, or 2) calculating difference of clock cycles between a program WITH the target instruction (Target Program) and that WITHOUT it (Base Program) on an evaluation environment of the target processor (Fig.6). For both methods, it takes very long time to obtain cycle counts of all instructions. In addition, these methods force us to read

- 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

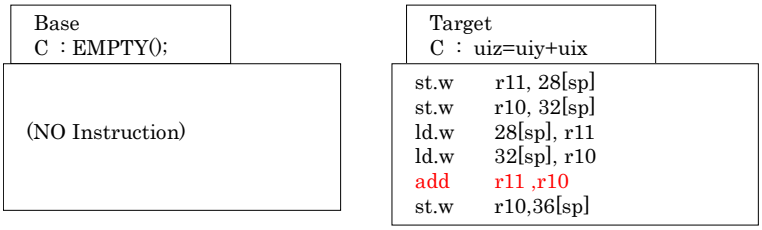

Figure 6: Measurement of worst-case "add" cycle counts for SHIM

voluminous amount of documents or write a number of assembly programs.

In order to apply the method [4] for performance estimation to general software, two issues related to LLVM-IR must 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. Because of the infinite virtual registers in LLVM-IR, there is no register spill and data in a 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 for the data placed in a register to be lost because of register spilling. The memory accesses that are generated in such cases are not reflected in the LLVM-IR programs. Such memory accesses may incur cache miss penalties. The handling of such events is an important consideration in performance estimation with SHIM.

Issue2 is related to the fact that because LLVM-IR is an intermediate representation in compilers, multiple types of assembly code for the target processor may be generated from a single LLVM-IR instruction. Fig. 7 shows an example of the conversion from an add instruction in LLVM-IR to the assembly code for the RH850/E1M-S2 [12] from Renesas Electronics Corporation. The converted assembler code can be a simple add instruction or an add instruction with several types of register replacement processes. Because 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.

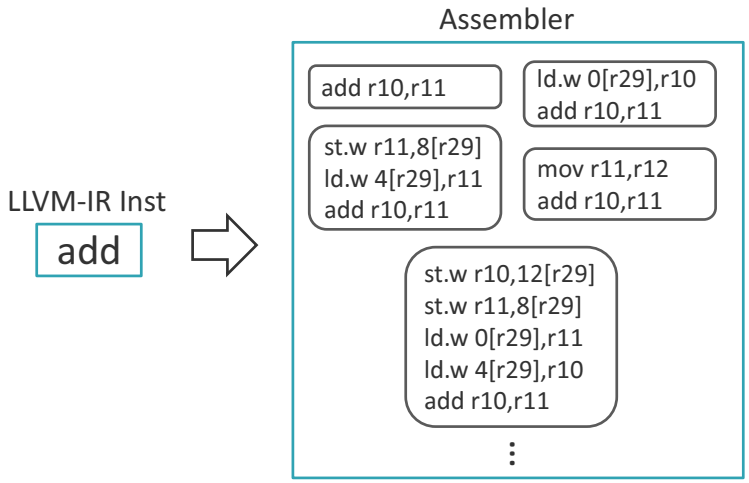

Figure 7: Example of conversion from LLVM-IR instructions to assembly code

The consideration of such occurrence probabilities becomes more difficult with the increase in hardware architecture complexity and the dependence of code generation on the optimizer.

3. Proposed Performance Estimation

In this section, we propose A) a method for software performance estimation using SHIM for early stage of model-based development, and B) a method for estimating the execution cycle of each LLVM-IR instruction to be stored in SHIM for a processor using regression analysis when SHIM is created for the processor.

In the proposal A), it is assumed that a model-based development tool can generate code for the model to be estimated, from which coverage information can be obtained by llvm-cov [9]. In addition, we assume that the cache miss and register spill ratios are known for the software generated from the model to be estimated. This assumption is usually true for embedded control systems such as automotive powertrain, where these ratios can be predicted from current design because of the use of iterative design. In the proposal B), we assume that a measuring method is available for the target processor, with which several types of values such as execution cycles can be measured for a number of programs.

These assumptions are summarized as follows:

- The coverage information from the llvm-cov tool [9] is available for the estimated software generated from the model.
- The cache miss ratio and register spill ratio are known for the estimated software generated from the model.
- A measuring method for the cycle counts of the program is available for the target processor at the SHIM creation.

It is important to note that provided that the assumptions hold true, designers can estimate the performance of software using SHIM files with a target-independent tool once the SHIM files are created for the candidate processors and their evaluation environments by vendors.

3.1. Software Performance Estimation

In this subsection, equation (1) is improved to give the following equation:

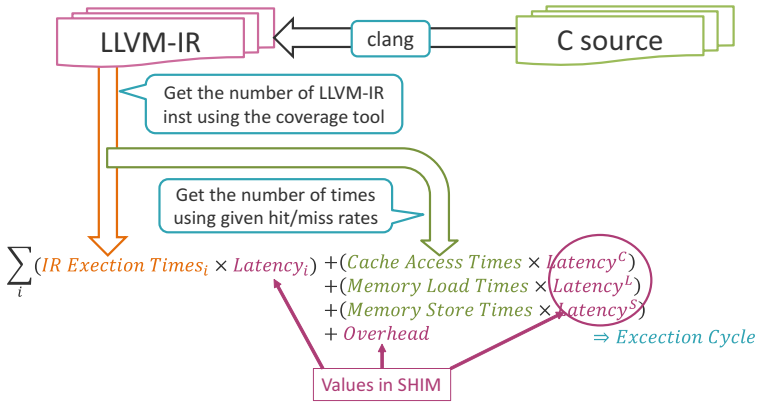

Figure 8: Performance estimation flow

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

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

To estimate software performance, as is shown in Fig. 8, $IR\ Execution\ Times_i$ is measured using llvm-cov, and *Cache Access Times*, *Memory Load Times*, and *Memory Store Times* are calculated using the results from llvm-cov and the given cache miss and register spilling ratios. *Execution Cycles* can be estimated using *Latencies* in SHIM and the system *Overhead*, if any.

3.2. Execution Cycle Estimation by Regression Analysis

In this subsection, we propose a method to estimate the performance values to be stored in SHIM by using regression analysis as a solution to overcome Issue2. The flow and regression formula of the proposed method are shown in Fig. 9. It is important to note that the regression formula in Fig. 9 is the same as that in equation (2) in Fig. 8. In the formula, the variables are *Latencies* and *Overhead*, and other values are obtained from llvm-cov or the evaluation environment of the target processor.

The measurement flow is as follows. First, multiple measurement programs are prepared. Next, *Cache Access Times*, *Memory Load Times*, *Memory Store Times*, and *Exection Cycles* are measured for the measurement programs in the evaluation environment of the target processor (*measurement phase*). The $IR\ Execution\ Times_i$s are obtained from these measurement programs using llvm-cov (*simulation phase*). The regression formula is solved (*regression analysis phase*) to calculate *Latencies* and *Overhead*.

Issue2 is therefore overcome in this method by estimating the latency using a number of measurement programs. It should be noted that the same set of measurement programs can be used to measure different hardware.

As a result, with our proposed method, the estimation cost of execution cycles for each LLVM-IR instruction to be stored

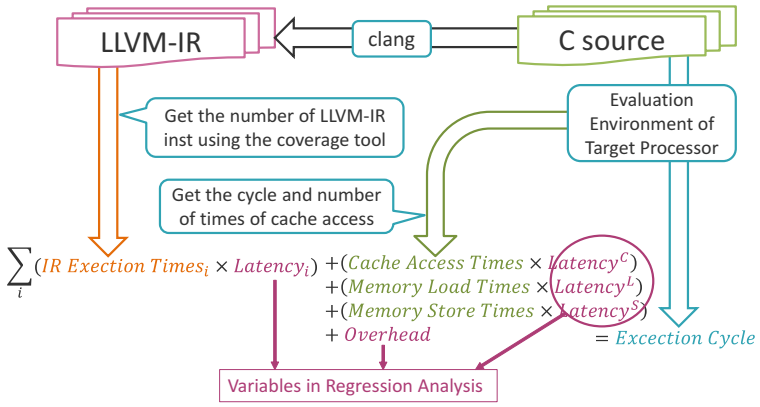

Figure 9: SHIM value measurement flow

in SHIM is reduced compared with previous methods where these values are obtained by extracting from voluminous amount of documents or writing a number of assembly programs. In addition, acceracy of software performance estimation should be improved, which is shown in the following sections.

4. Case Study

In this section, we describe each step to estimate the SHIM performance values introduced in the previous section through a case study on a Raspberry Pi3 Model B+. The tools described in this section are openly available on the OpenSHIM website [13].

4.1. Target Hardware

The target hardware in this case study is 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 first prepared 36 programs in C language for evaluation. The programs were designed to increase variants of LLVM-IR instructions.

We also set a lower limit for the number of execution cycles because the overhead has a greater effect on measurement programs with fewer execution cycles in our experimental environment. The lower limit was set to 10,000 cycles. For measurement programs in which the number of execution cycles was originally less than 10,000, the lower limit was exceeded by iterating the process in the main function.

4.3. Measurement Phase

In the measurement phase, the compiled measurement programs were executed to obtain the numbers of execution cycles, cache accesses, and memory loads and stores. These values were obtained using the PMU in the ARM processor on the Raspberry Pi3 Model B+. Each value was obtained through the following means:

Cache Access Times
Actual measurement.

Memory Load Times
Calculated using equation (3).

Memory Store Times
Calculated using equation (4).

Overhead
Assumed to occur once each time and put into the regression equation as a constant term.

$$Memory\ Load\ Times = Load\ Times \times Cache\ Miss\ Rate \tag{3}$$

$$Memory\ Write\ Times = Store\ Times \times Cache\ Miss\ Rate \tag{4}$$

4.4. Simulation Phase

In the simulation phase, the number of LLVM-IR instructions for the measurement programs were obtained using llvm-cov. In the following, we present several techniques to improve the estimation.

4.4.1. Variable Settings

In our method, the latency of the LLVM-IR instruction is assigned to the variables in equation (2). However, because a simple one-to-one assignment will result in too many variables and increase the number of sample programs required, we use a grouping technique in which instructions with similar behaviors are grouped into a single variable. Because an overly rough assignment will decrease the estimation cycle accuracy, the assignment of LLVM-IR instructions to variables is an important issue. This assignment must be performed such that the number of variables is not increased significantly while minimizing the latency differences for each variable.

In this study, we classified the instructions according to their processing similarities and number of cycles. Because there are many distinct processing instructions in LLVM-IR, we performed a rough classification of these instructions as shown in Table 1 to avoid an excessive number of variables. The characteristics of each instruction variable are as follows:

arithmetic	Integer arithmetic instruction set
div	Integer division instruction
float	Floating-point instruction set
fdiv	Floating-point division instruction

Table 1: Instruction classification table (part of the instructions)

Instruction Variable	Instruction			
arithmetic	add	sub	mul	srem
	urem	icmp		
div	sdiv			
float	fadd	fsub	fmul	fcmp
fdiv	fdiv			
load	load			
store	store			
callret	call	ret		
others1	sext	zext	sitofp	fptosi
others2	getelementptr	alloca	phi	br

load Load Instruction

store Store Instruction

callret Call instruction and Ret instruction

others1 Cast instruction set

others2 Other instruction groups

4.4.2. Classification of Instructions with Operand Dependencies

Although a mechanism to forward register values is present in modern hardware, delays are still incurred for some instructions such as load. In this study, we distinguish instructions that cause delays from those with *operand dependencies*. We say that an instruction has *an operand dependency* when its result is used by the next instruction in LLVM-IR. We classify the instructions div, float, fdiv, and load into two types of variables: *inst* and *inst_d*.

4.5. Regression Analysis Phase

In the regression analysis phase, the latency is calculated using the values obtained from the measurement and simulation phases.

We used the error least-squares method for the regression analysis in which the error was defined as the difference between the estimated number of cycles and that executed on the actual machine. We calculated the square of the error for each measurement program and added the squares of all the programs to obtain the sum of the squares of the error. The combination of the variables *Latencies* and *Overhead* that minimizes the sum of squared errors is obtained through regression analysis with the following two constraints:

- The value of each latency is non-negative.
- The error of the estimated number of cycles is less than $\pm 20\%$.

We used the solver function in Microsoft Excel because the constraints could be easily set and the generated outputs in the CSV files easily handled. In addition, the solver function in Excel output the best solution even if there was no solution that satisfied the constraint conditions; therefore, the latency would be output for all situations.

Table 2: Overview of the evaluation programs

No.	Program Name	Description
1	CoreMark.c [1]	Benchmark program for embedded systems developed by EEMBC
2	Selectionsort.c	Program to perform selection sorting
3	Functions3.c	Program that calls a recursive function and performs integer and floating-point operations within the function
4	Functions4.c	Program to perform integer arithmetic with recursive functions and compare the size of two numbers
5	Functions11.c	Program to perform floating point arithmetic, including division with recursive functions and compare the size of two numbers

5. Evaluation for Software Performance Estimation

In this section, we evaluate the estimation of the SHIM software performance for the Raspberry Pi3 Model B+ described in the previous section.

5.1. Evaluation Method

The accuracy of the estimation is evaluated through the *error*, which is defined as

$$Error = \frac{(Total\ Cycles - Estimation\ Result)}{Total\ Cycles} \tag{5}$$

The programs used in our evaluation are outlined in Table 2. Other than CoreMark.c, the remaining four programs were self-made. The four programs were designed such that the overlaps in the trends of the number of executions were minimized for effective evaluation.

5.2. Results of estimation accuracy evaluation experiment

The results of applying the estimated latency to the five evaluation programs are presented in Fig.10, where our proposed method is compared with the previous method [4]. The numbers in each graph in the figure correspond to those in Table 2. The errors for the proposed method were within the SHIM performance target of ±20% for all the evaluation programs, and the error of CoreMark.c was within ±10%. CoreMark.c is a well-known benchmark for embedded systems and is the most practical program among the evaluation programs. In addition, error rates for the proposed method are reduced in all evaluation programs compared with the previous method.

Conclusion

In this study, we proposed A) a method for software performance estimation with SHIM and B) a method for estimating the performance of each LLVM-IR instruction stored in SHIM using regression analysis. The proposed methods were implemented and tested on a Raspberry Pi3 Model B+. The target error of ±20% was achieved in the SHIM performance value, and error rates were reduced in all experiments compared with the previous method. Furthermore, our method for estimation of execution cycles for each LLVM-IR instruction is easily applicable to other processors

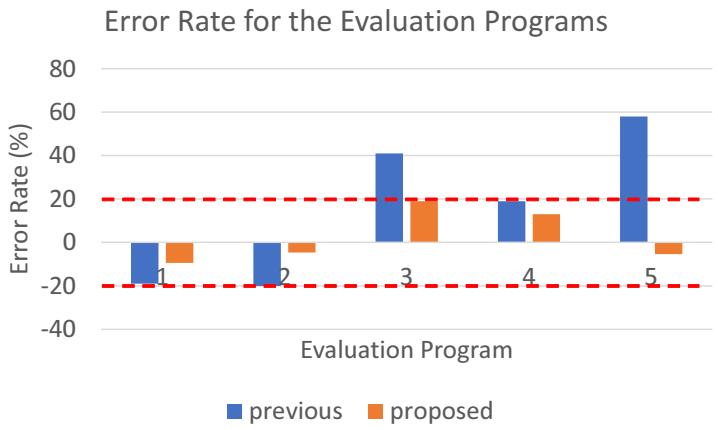

Figure 10: Error rate for the evaluation programs

with different instruction set. Thus, our proposed method will greatly reduce the cost of estimation.

In the future, we will provide support for system calls and library functions in the measurement program. Furthermore, one of the main goals of this study is to apply the proposed methods to different hardware. It is necessary to test the applicability of our method to other hardware and to make further improvements.

References

[1] EEMBC, "CoreMark", (Accessed 2022-05-01). https://www.eembc.org/coremark/.

[2] EMC, "The Multicore Association Specifications", (Accessed 2022-05-01). https://www.embeddedmulticore.org/the-multicore-association-specifications/.

[3] eSOL Co. Ltd., "eMBP (Model Based Parallelizer)", (Accessed 2022-05-01). https://www.esol.co.jp/embedded/mbp.html.

[4] Gondo, M., Arakawa, F., & Edahiro, M. (2014, April). Establishing a standard interface between multi-manycore and software tools-SHIM. In 2014 IEEE COOL Chips XVII (pp. 1-3). IEEE.

[5] Hwang, Y., Abdi, S., & Gajski, D. (2008, March). Cycle-approximate retargetable performance estimation at the transaction level. In Proceedings of the conference on Design, automation and test in Europe (pp. 3-8).

[6] IEEE, "IEEE Standard for Software-Hardware Interface for Multi-Many-Core", IEEE 2804-2019, (Accessed 2022-05-01). https://standards.ieee.org/ieee/2804/7477/.

[7] LLVM project, "The LLVM Compiler Infrastructure",(Accessed 2022-05-01). https://llvm.org/.

[8] LLVM project, "LLVM Language Reference Manual", (Accessed 2022-05-01). https://llvm.org/docs/LangRef.html.

[9] LLVM project, "llvm-cov", (Accessed 2022-05-01). https://llvm.org/docs/CommandGuide/llvm-cov.html.

[10] Patel, R., & Rajawat, A. (2013). Recent trends in embedded system software performance estimation. Design Automation for Embedded Systems, 17(1), 193-213.

[11] Ray, A., Srikanthan, T., & Wu, J. (2010). Rapid techniques for performance estimation of processors. Journal of Research and Practice in Information Technology, 42(2), 147-165.

[12] Renesas electronics, "RH850/E1M-S2", (Accessed 2022-05-01). https://www.renesas.com/jp/en/products/microcontrollers-microprocessors/rh850-automotive-mcus.

[13] SHIM Working Group, "SHIM Latency Measurement and Insertion", (Accessed 2022-05-01). https://github.com/openshim/shim2/tree/master/shim-measure.

[14] Wang, S., Zhong, G., & Mitra, T. (2017). CGPredict: Embedded GPU performance estimation from single-threaded applications. ACM Transactions on Embedded Computing Systems (TECS), 16(5s), 1-22.

[15] Wijesundera, D., Prakash, A., Srikanthan, T., & Ihalage, A. (2018). Framework for rapid performance estimation of embedded soft core processors. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11(2), 1-21.

Appendix A. Top-level class diagram of SHIM XML

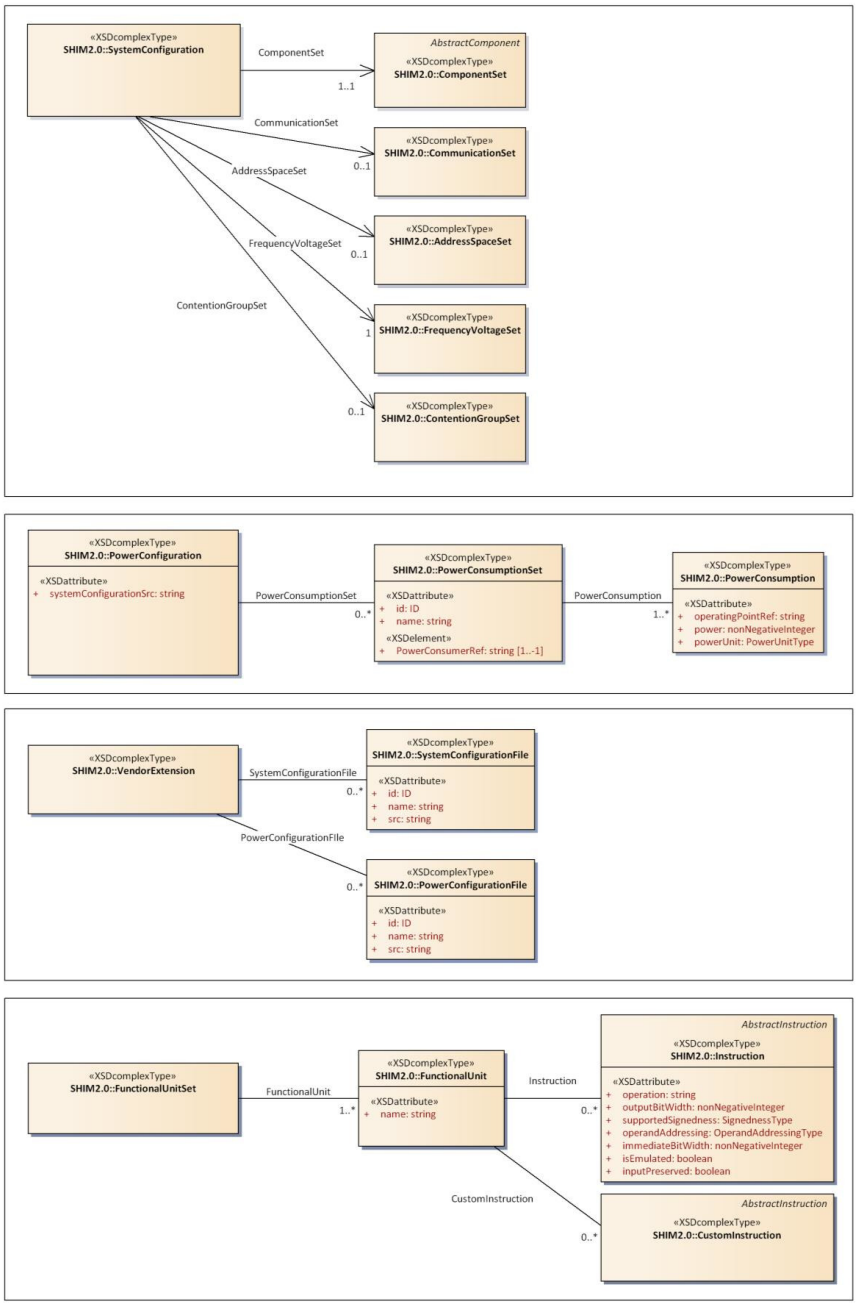

Figure 11: Class diagram representation of the SHIM XML schema (top-level elements) [6]