J Inf Process Syst, Vol.15, No.6, pp.1462~1471, December 2019
https://doi.org/10.3745/JIPS.01.0047

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Runtime Software Monitoring Based on Binary Code Translation for Real-Time Software

Kiho Choi*, Seongseop Kim*, Daejin Park*, and Jeonghun Cho*

Abstract
Real-time embedded systems have become pervasive in general industry. They also began to be applied in such domains as avionics, automotive, aerospace, healthcare, and industrial Internet. However, the system failure of such domains could result in catastrophic consequences. Runtime software testing is required in such domains that demands very high accuracy. Traditional runtime software testing based on handwork is very inefficient and time consuming. Hence, test automation methodologies in runtime is demanding. In this paper, we introduce a software testing system that translates a real-time software into a monitorable real-time software. The monitorable real-time software means the software provides the monitoring information in runtime. The monitoring target are time constraints of the input real-time software. We anticipate that our system lessens the burden of runtime software testing.

Keywords
Binary Translation, Dynamic Testing, Software Monitoring

1. Introduction

Real-time embedded systems have become pervasive in general industry. They also began to be applied in such domains as avionics, automotive, aerospace, healthcare, and industrial Internet. For examples, in automotive, drive by wire is such a component where the traditional steering system was replaced by electronic control systems using electromechanical actuators and human-machine interfaces. However, the system failure of such domains could result in catastrophic consequences. Therefore, the correctness of real-time software is very important, and software testing could be one of ways to accomplish it. Software testing is an array of activities to ensure that the implemented software meets performance and constraints. Particularly, in real-time software, time constraints are very important elements of software testing.

Testing methodology of real-time software could be divided into two categories. One is a static-analysis-based methodology. It analyzes the real-time software closely, and evaluates the performance and the constraints of the real-time software without its execution. Therefore, it helps to minimize the software testing time. However, the static-analysis-based software testing could not deal with the following two potential risks: (1) in compilation time, the unwanted and undesired execution code could

Manuscript received May 28, 2018; first revision July 24, 2018; accepted September 2, 2018.
Corresponding Author: Jeonghun Cho (jcho@knu.ac.kr)
* School of Electronics Engineering, Kyungpook National University, Daegu, Korea ({posjkh22, kss92318}@gmail.com, {boltanut, jcho} @knu.ac.kr)

be generated, and (2) a real-time system may not meet the requirements of the real-time software. These potential risks should be considered and checked in real-time software systems that requires very high accuracy such as avionics, automotive, aerospace, and healthcare.

The other is a runtime-testing-based methodology. In this methodology, software is evaluated in runtime. It means that the runtime software testing includes not only the evaluation of the integrity of the complied real-time software, but also that of the real-time systems. Other than run-time software testing, nothing can guarantee a high level of stability [1]. However, the problem of the runtime software testing is that it requires much more testing time than the static-analysis-based software testing. Traditional runtime software testing based on handwork, which includes such as inserting a monitoring code manually or recompiling the software, is very inefficient and time consuming. To reduce the software testing time, test automation methodology in runtime is very demanding. There are two types of software testing automation. One is to generate the software test cases automatically [2] and the other is to automate the software testing itself.

In this paper, we introduce a software testing system that automatcally translates a real-time software into a monitorable real-time software in binary code level. The monitorable real-time software means the software that provides the monitoring information in runtime. By translating into the monitorable real-time software with our system, we do not need to perform runtime software testing based on handworks. In our first trial, we implement a task-based/function-based lapse-time monitoring module. In implementation, the module of monitoring target is designed modularly so that other targets of monitoring could be added. Our system targets ARMv7-M [3] architecture that is most commonly used in the real-time embedded system.

The rest of this paper consists as follows. Section 2 discusses the related works. Section 3 introduces the overall structure of our system. Section 4 describes how the proposed system performs the code translation. Section 5 mentions the experiments that we have implemented. Section 6 discusses the limitations of our system and the future work. Finally, we summarize the paper and conclude in Section 7.

2. Related Works

Much research on code translation for software testing have been performed. The code translation for the monitorable software could be divided into three categories according to levels of translation language; high-level language translation (e.g., C/C++, Java, Python, MATLAB), Intermediate-level language translation (e.g., LLVM IR), and low-level language translation (e.g., x86/ARM assembly code).

The code translation in high-level language or intermediate-level language is to insert a testing code in high-level language or intermediate-level language. Many related works on code translation in high/intermediate-level have been performed. (e.g., code translation in Java [4], MATLAB [5], Python [6], and LLVM IR [7]). However, the code translation for software testing in high/intermediate level language requires the compilation for generating binary code. It means that such code translation entails the compile time.

In contrast, the code translation in low-level language does not require compilation. The code translation in low-level language decodes the binary code and only translates some binary code that need to change. In this paper, we propose a system that executes code translation in low-level language, and we apply the system for real-time software testing. The system generates the monitorable binary code automatically. It could lessen more software testing time.

3. Overall Structure

Our monitorable code translation system for runtime software testing, called MCTS, is described in Fig. 1. The inputs of the system are the executable linkable format (ELF) [8] binary code and the symbol of target function to be monitored. The output of the system is the monitorable binary code. The monitorable binary code provides the monitoring information in runtime as mentioned above. The information is stored in a designated memory address. It could be stored in a specific monitoring information storage. The system consists of five modules; ELF code parser, ELF static analyzer, update section analyzer, word/relative branch updater, and shell code/monitoring code injector. The ELF code parser parses the input real-time software. The ELF static analyzer generates some information that is used for generating a monitoring code and software profiles. The update section analyzer divides memory space into update sections. The 'update' represents the proper code translation. The 'update section' refers to an area that is updated identically. Word/relative branch updater updates the words and the relative branch instructions. The 'word' refers to a memory space that contains a memory address value. The code injector inserts a shell code into a target function, and insert the generated monitoring code.

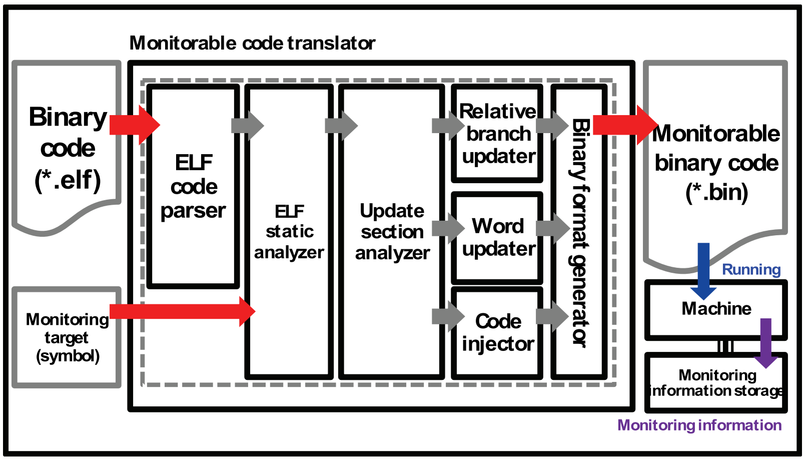


Fig. 1. Overall structure of the proposed system.

4. Code Translation

Code translation for the monitorable real-time software entails the insertion of some binary code into the input real-time software. Hence, proper translations on the input real-time software are required.

4.1 Code Translation Strategies

The strategies of the code translation that our system adopts are as follows. First, function trampoline method is used for calling the monitoring function, and shell code [9,10] is injected into the target function that would be monitored. Our system adopts the function trampoline methods [11,12] for code translation. The monitoring function refers to the monitoring code in each monitoring module. The shell code means a set of binary code that branch to the monitoring function. The target function means the entry function of the task that would be monitored.

Second, each monitoring code is inserted at the end of text segment. Our system inserts an array of software monitoring code at the end of text segment. No change of function addresses would happen with this strategy. This makes the code translation simple.

Third, no modification is made to the input real-time software other than the target task to be monitored. To accomplish it, proper word update and branch instruction update are required. ARMv7-M architecture generally uses a relative branch instruction such as 'bl' for calling a function. It also uses word to contain a memory address in general.

4.2 ELF Code Parser

The executable ELF file includes vector table, vector interrupt service routine (ISR), text section, data section, and symbol table. The ELF code parser divides ELF file into 4 segments: vector table segment (vector table), text segment (vector ISR and text section), data segment (data section), and symbol segment (symbol section). The information of each segment is used for the software static analysis or the proper code translation.

The vector table segment is used for the interrupt-related monitoring code such as system timer. The text segment is used for generating the monitoring function and function call graphs. The data segment is used for monitoring data. The symbol table is used by the ELF static analyzer. The ELF code parser is based on the binary file descriptor libraries [13].

4.3 ELF Static Analyzer

ELF static analyzer generates the information that is related to the code translation and the monitoring code. It analyzes the symbol table segment, and finds out the memory address of each function. It also analyzes the text segments and generates a function call graph. Such generated information as the memory addresses, and function call graphs are used to determine the update section in the update section analyzer.

4.4 Update Section Analyzer

Update section analyzer divides memory space into update sections and determine how the sections would be updated. Fig. 2 describes how the update section analyzer divides memory address into update sections. One target function divides memory space into two update sections as shown in Fig. 2(a). (e.g., one section that has a lower memory address than that of shell code in the target function, and the other section that has a higher address than that of shell code in the target function). In this case, the section that requires a proper update is the latter section. This is because shell code that would be injected into the target function causes a memory address change. If k-bytes shell code is injected into the target function, all the memory address in the latter section is required to increase by k.

In the same way, if there are the n target functions, the update section analyzer divides memory space into n+1 sections (e.g., n update sections and one no-update section) as shown in Fig. 2(b). The no-update section refers to the section where no increase in memory address is required. We denote the n-th update section as s_n in a numerical order except for s_0. s_0 is a no-update section. A set of the memory address of shell code is denoted as C. The memory address of the shell code in *n-th* target function is denoted as

c_n. The update section S_{up} is defined as follows.

$$C = \{c_1, c_2, c_3, \dots, c_n\} \tag{1}$$

$$S_{up} = \{s_0, s_1, s_2, \dots, s_n\} \tag{2}$$

$$c_n < region\ of\ s_n < c_{n+1} \tag{3}$$

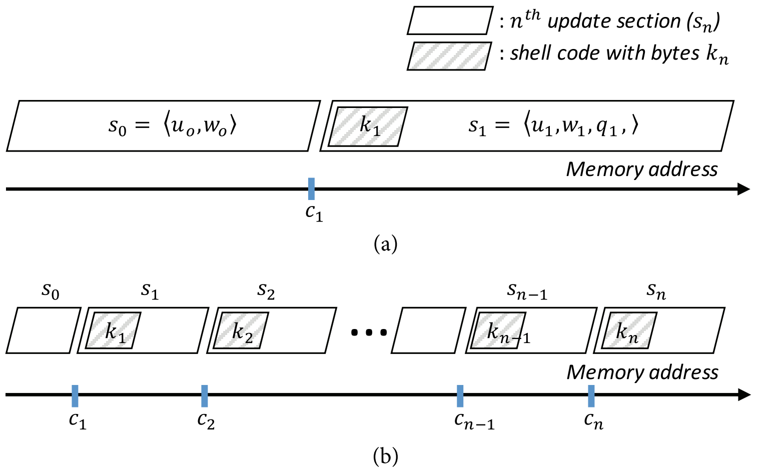
$$end\ address\ of\ s_n < c_{n+1} < entry\ address\ of\ s_{n+1} \tag{4}$$

A set of the bytes of shell code is denoted as K. If the shell code has k_n bytes in the *n-th* target function, update bytes in each update section has its own update bytes. k_0 refers to the bytes of shell code in s_0 and $k_0 = 0$. We denote a set of the update bytes as U. u_n refers to the update bytes in s_0 and it could be described as follows.

$$K = \{k_0, k_1, k_2, k_3, \dots, k_n\} \tag{5}$$

$$U = \{u_0, u_1, u_2, u_3, \dots, u_n\} \tag{6}$$

$$u_n = \textstyle\sum_{i=1}^{n} k_i \tag{7}$$



: n^{th} update section (s_n)
: shell code with bytes k_n
$s_0 = \langle u_o, w_o \rangle$
k_1
$s_1 = \langle u_1, w_1, q_1, \rangle$
Memory address
c_1
(a)
s_0
s_1
s_2
s_{n-1}
s_n
k_1
k_2
k_{n-1}
k_n
Memory address
c_1
c_2
c_{n-1}
c_n
(b)

Fig. 2. Update section analyzer and update section: (a) one target function and (b) n-target functions.

4.5 Word Updater

'For' statement, 'while' statement, and indirect branch using a pointer variable generally use a word in ARMv7-M architecture. The word contains a memory address to branch. A set of the memory address of the word is denoted as W. If the memory address of the word w_n is within the update section s_n, it should be updated according to the update bytes u_n. A set of the updated memory address of the words is denoted as W', and w'_n is described as follows.

$$W = \{w_1, w_2, w_3, \dots, w_n\} \tag{8}$$

$$W' = \{w'_1, w'_2, w'_3, \dots, w'_n\} \tag{9}$$

$$w'_n = update(w_n) = w_n + u_n \tag{10}$$

4.6 Relative Branch Updater

ARMv7-M architecture generally uses a relative branch instruction such as 'bl' to call a function. The relative branch updater divides the relative branch into two types; one is intra-update-section branch and the other is inter-update-section branch. Fig. 3(a) describes the intra-update-section branch. The intra-update-section branch does not need to update. There is no relative change because the branch occurs within the one section.

Fig. 3(b) describes the inter-update-section branch. The inter-update-section branch jumps from one update section to other section. Therefore, it requires updating the relative branch instruction (e.g., bl). A set of the variations of the relative address is denoted as R. If the n-target functions determine the n+1-update sections S_{up} with its own update bytes K, and if an inter-update-section branch is within the update section s_x and it jumps to a function within the update section s_y, then a variation of the inter-update section branch is denoted as $r_{x \to y}$. It is described as follows.

$$R = \langle u_x, u_y \rangle,\ u_x \neq u_y,\ u_x \in U, u_y \in U \tag{11}$$

$$R = \{\, r_{0\to 1}, r_{0\to 2}, \dots, r_{0\to n}, r_{2\to 0}, r_{2\to 1}, \dots\ , r_{n\to n-1}\} \tag{12}$$

$$r_{x\to y} = u_y - u_x \tag{13}$$

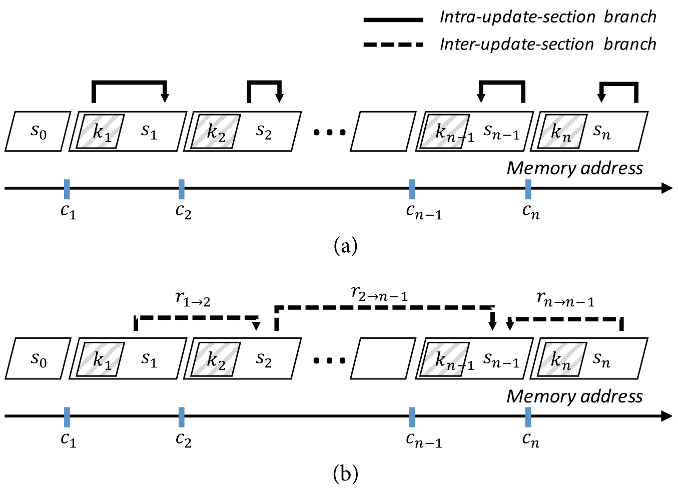


Fig. 3. Intra-update-section branch (a) and inter-update-section branch (b).

4.7 Code Injector of Shell Code and Monitoring Code

The code injector inserts the shell code and monitoring code. The shell code is injected into a target function. The memory address of shell code C is determined by the ELF static analyzer. The shell code calls a designated monitoring code. Fig. 4 illustrates an example of the injected shell code that calls the task-based/function-based lapse time monitoring code.

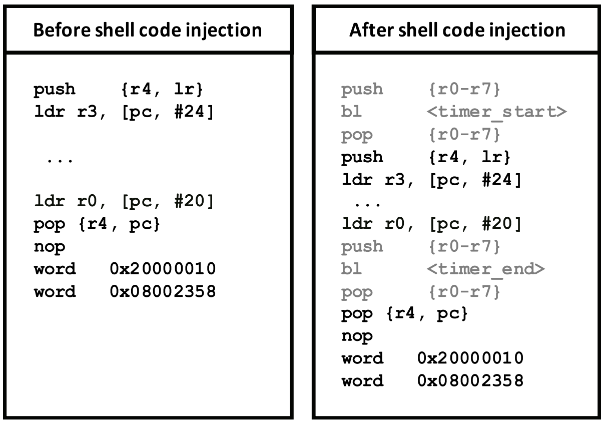
The shell code consists of three instructions; push registers, branch to monitoring code, and pop the registers. The 'push registers' saves a context of registers before the function call occurs. The 'branch to monitoring code' calls the monitoring code. The 'pop the registers' restore the saved context of registers. The monitoring code is generated through the information from the ELF static analyzer.

The monitoring code is selected according to the selected monitoring target. The selected monitoring code is inserted at the end of the text segment. The monitoring code differs depending on the monitoring target. For example, the task-based/function-based lapse time monitoring code consists of three functions; timer_start, timer_end, and timer_systick. The timer_start function and timer_end function is called by the two inserted shell code. The timer_systick function is used for timer interrupt service routine. The memory address of the timer_systick function is overwritten on the vector table. The timer_start operates timer_systick. The timer_systick increases time data. The timer_end stops timer_systick and saves the time data. Though many tasks are operated concurrently and scheduled by RTOS [14], the target task could be monitored in a proper way.

One thing that we have to consider is that the memory address of the end of text segment prior to code translation differs from the address posterior to code translation. It is because the shell code is injected. In other words, the relative address between the 'branch to monitoring code' and the monitoring code is required to update. A set of the variance of the relative address is denoted as Q. q_x refers to the variance of the relative address in s_n. q_0 is not defined because there is no shell code in s_0. q_x is described as follows.

$$Q = \{q_1, q_2, q_3, \dots, q_x\} \tag{14}$$

$$q_x = r_{x \to n} \tag{15}$$


Before shell code injection

```
push    {r4, lr}
ldr r3, [pc, #24]

 ...

ldr r0, [pc, #20]
pop {r4, pc}
nop
word    0x20000010
word    0x08002358
```

After shell code injection

```
push     {r0-r7}
bl       <timer_start>
pop      {r0-r7}
push     {r4, lr}
ldr r3,  [pc, #24]
 ...
ldr r0,  [pc, #20]
push     {r0-r7}
bl       <timer_end>
pop      {r0-r7}
pop {r4, pc}
nop
word    0x20000010
word    0x08002358
```

Fig. 4. Shell code injection.

5. Experimental Results

The case study tries to apply our system to a real-time software on MCUs configured with ARMv7-M architecture. For MCUs, STM32F407VG and STM32F401RE are considered. Each STM32F407VG and STM32F401RE is configured with 32-bit Cortex-M4. Table 1 shows detail information of considered micro-controllers.

We implemented an experiment that compares the manually measured execution time of each benchmark and lapse-time information given by our system. The selected case studies are the following benchmarks: fast Fourier transform (FFT), StringSearch, DhryStone 2.1 [15], WhetStone 1.1 [16], Calculation-A, and Calculation-B as shown in Table 2. The FFT and StringSearch are selected from MiBench 1.1 [17]. The DhryStone and WhetStone are well-known synthetic computing benchmark programs that are representative of general processor performance. The Calculation-A and Calculation-B are our own implementations. As shown in Table 2, each of time difference ratios is under 1.6, and the mean of difference ratio is 0.59. In consideration that each of time difference ratio is positive and minor value, it is thought that the reason of the time difference might be caused by console output function for debugging such as 'printf' and 'trace_printf'.

Table 1. Microcontroller information

MCU	DMIPS/MHz	RAM (kb)	Remark
STM32F407VG	1.25	192	32-bit
STM32F401RE	1.25	96	32-bit

Table 2. Benchmarks and experimental results

	Time difference ratio (%)	
	STM32F407VG	STM32F401RE
FFT	0.01	0.01
StringSearch	0.45	0.44
DhryStone	1.21	1.21
WhetStone	1.56	1.56
Calculation-A	0.15	0.15
Calculation-B	0.14	0.14

6. Limitations and Future Works

Our system adopts function trampoline method, and shell code is injected into a target function. The challenge in inserting shell code is how to insert shell code at the start of the target function or at the end of the target function. It is the reason that there should be no effect on the target task/function.

One simple way to solve it is to inject shell code right before calling the target function or right after calling the target function. However, this approach is limited if the target function is called with indirect ways. In addition, in this approach, shell code should be inserted into a function that calls the target function, which is too inefficient (e.g., the increased size of binary code) and complicated to implement.

Hence, our system inserts shell code into the target function. However, in our approach, inserting shell code at the end of the target function is challenging. In Fig. 4, the second shell code that calls timer_end is inserted before 'pop {r4, pc}'. It is because these kind of instructions that return to prior PC and restore registers should be at the end of the function. In other words, the shell code injector has to recognize the instruction set that return to prior PC and restore registers.

In our system, the shell code injector recognizes the proper end point of the target function through many secured cases. However, other case could be added depending on the complier. Our future work will include transforming the way of restoring registers and returning to prior PC into one identical way in the target function after analyzing how the target function restores register and returns to prior PC.

7. Conclusion

In this paper, we introduce a software testing system that translates a real-time software into a monitorable real-time software. The monitorable real-time software means the software provides the monitoring information in runtime. In our first trial, we implement a task/function-based lapse-time monitoring module. In our implementation, the module of monitoring target are designed modularly so that other targets of monitoring could be added. The inputs of the system are the executable binary code and the symbol of target function to be monitored. The output of the system is the monitorable binary code. We anticipate that our system lessens the burden of runtime software testing.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2016R1D1A1B03934343 and NRF2018R1A6A1A03025109), and Ministry of Science and ICT (No. NRF2019R1A2C2005099).

References

[1] M. Belaoued and S. Mazouzi, "A chi-square-based decision for real-time malware detection using PE-file features," *Journal of Information Processing Systems*, vol. 12, no. 4, pp. 644-660, 2016.

[2] S. Sabharwal and M. Aggarwal, "Test set generation for pairwise testing using genetic algorithms," *Journal of Information Processing Systems*, vol. 13, no. 5, pp. 1089-1102, 2017.

[3] ARMv7-M Architecture Reference Manual [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403-e.b/index.html.

[4] D. Xu, "A tool for automated test code generation from high-level Petri nets," in *Application and Theory of Petri Nets*. Heidelberg: Springer, 2011, pp. 308-317.

[5] M. Conrad, "Testing-based translation validation of generated code in the context of IEC 61508," *Formal Methods in System Design*, vol. 35, no. 3, pp. 389-401, 2009.

[6] R. Marvie, "An introduction to test-driven code generation," [Online]. Available: http://ojs.pythonpapers.org/index.php/tpp/article/download/50/47.

[7] N. Hasabnis, R. Qiao, and R. Sekar, "Checking correctness of code generator architecture specifications," in *Proceedings of 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, 2015, pp. 167-178.

[8] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification version 1.2," 1995 [Online]. Available: http://refspecs.linuxbase.org/elf/elf.pdf.

[9] Shellcode [Online]. Available: https://en.wikipedia.org/wiki/Shellcode#cite_note-11.

[10] J. C. Foster, *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Rockland, MA: Syngress Publishing, 2005.

[11] Trampoline [Online]. Available: https://en.wikipedia.org/wiki/Trampoline_(computing).

[12] H. G. Baker, "CONS should not CONS its arguments, Part II: Cheney on the MTA," *ACM SIGPLAN Notices*, vol. 30, no. 9, pp. 17-20, 1995.

[13] C. DiBona and S. Ockman, *Open sources: Voices from the Open Source Revolution*. Sebastopol, CA: O'Reilly Media Inc., 1999.

[14] M. M. Tajwar, M. Pathan, L. Hussaini, and A. Abubakar, "CPU scheduling with a round robin algorithm based on an effective time slice," *Journal of Information Processing Systems*, vol. 13, no. 4, pp. 941-950, 2017.
[15] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013-1030, 1984.
[16] H. J. Curnow and B. A. Wichman, "A synthetic benchmark," *The Computer Journal,* vol. 19, no. 1, pp. 43-49, 1976.
[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (Cat. No. 01EX538)*, Austin, TX, 2001, pp. 3-14.

Kiho Choi https://orcid.org/0000-0003-2072-3508

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2017. He is currently a M.S. student in department of electronics engineering at Kyungpook National university, Daegu, Korea. His research interests include embedded software analysis algorithm for software safety and software testing code generations.

Seongseop Kim https://orcid.org/0000-0003-1694-093X

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2017. He is currently a M.S. student in department of electronics engineering at Kyungpook National University, Daegu, Korea. His research interests include hardware-accelerated signal processing algorithm and high performance microprocessor-based system implementation.

Daejin Park https://orcid.org/0000-0002-5560-873X

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2001, the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2003, and 2014, respectively. He is now with School of Electronics Engineering as full-time assistant professor in Kyungpook National University, Daegu, Korea and presidential research fellow.

Jeonghun Cho https://orcid.org/0000-0002-9330-6118

He received the B.S. degree in EE, the M.S. and the Ph.D. degree in EECS from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea in 1996, 1998, and 2003, respectively. He was a senior engineer at Hynix Semiconductor from 2003 to 2005. His main role was development of a C compiler for 8-bit microcontrollers. He is currently a professor with the School of EE of Kyungpook National University, Daegu, South Korea since 2005.