International Journal of Smart Home
Vol.8, No.1 (2014), pp.229-240
http://dx.doi.org/10.14257/ijsh.2014.8.1.24

A Study on the Interpretation Optimization to Improve the Performance of the Stack based Virtual Machine on Smart Platforms

YunSik Son[1], SeMan Oh[1], JaeHyun Kim[2] and YangSun Lee[2]*

[1]Dept. of Computer Engineering, Dongguk University
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, Korea
[2]Dept. of of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Korea
*{sonbug, smoh}@dongguk.edu, statsr@skuniv.ac.kr, *Corresponding Author: yslee@skuniv.ac.kr*

Abstract

The previous development environments for smart phone contents are needed to generate specific target code depending on target devices or platforms, and each platform has its own developing language. Therefore, even if the same contents are to be used, it must be redeveloped depending on the target machine and a compiler for that specific machine is needed, making the contents development process very inefficient.

The Smart Cross Platform was developed for executing contents written in various programming languages – C, C++, Java, and Objective-C – on iOS or Android based smart devices. The contents developed in each programming language are translated into intermediate language called SIL (Smart Intermediate Language) by the compiler. And, the translation results - intermediate programs are executed on the SVM (Smart Virtual Machine) – a core module of the Smart Cross Platform – without device dependency. Intermediate language based SVM has an advantage of execution on multiple target devices without considerations about device specific features, but it has also a problem which low performance by the software-based execution, consequently. Therefore, to improve the performance of SVM is very important issue.

In this paper, we deal with two kinds of optimization technique to optimize stack based SVM which can execute on various smart devices. And, to improve performance of the SVM on execution engine aspect, we apply the one of these optimization techniques. For verification of this optimization technique, we profile and analyze performance of the original/optimized SVM. As a result of the experiments, the optimized SVM has 23~27% reduced execution times than the original SVM.

Keywords: *Smart Virtual Machine, Optimization, Smart Platform, Stack Interpretation, Execution Engine*

1. Introduction

The existing developmental environments for producing smart phone contents are tightly coupled with target platforms and devices. It means that the programming language to develop the contents is pre-determined and the generated target code by compiler depended on specific target devices or platforms [1, 2]. Therefore, even if the same contents are to be used, it must be redeveloped depending on the target machine and a compiler for that specific machine is needed, making the contents development process very inefficient.

ISSN: 1975-4094 IJSH

The Smart Cross Platform [3-6] is a virtual machine based solution which aims to solve such problems. It supports various programming languages – C / C + +, Java, Objective-C. And the contents developed in each programming language are translated into SIL to executing on the SVM, device independently. Intermediate language based SVM has an advantage of execution on multiple target devices without considerations about device specific features, but it has also a problem which low performance by the software-based execution, consequently.

In this paper, we deal with 2 kinds of optimization technique to optimize stack based SVM which can execute on various smart devices. And, to improve performance of the SVM on execution engine aspect, we apply the one of these optimization techniques. For verification of this optimization technique, we profile and analyze performance of the original/optimized SVM.

This paper is organized as follows. In Chapter 2, concept of the Smart Cross Platform, intermediate language and execution file format used in SVM, execution methods are examined. In Chapter 3, the optimization techniques to enhance the execution performance are examined. In Chapter 4, results of the execution engine aspect optimization shown. Lastly in Chapter 5, the conclusion of the paper and future direction of research are discussed.

2. Related Studies

2.1. Smart Cross Platform

The Smart Cross Platform is a stack based virtual machine solution which is loaded on smart devices. It is a stack based virtual machine based solution which can independently download and run application programs. The SVM consists of three main parts; compiler, assembler and virtual machine. It is designed in a hierarchal structure to minimize the burden of the retargeting process. Figure 1 shows a system configuration of the Smart Cross Platform [3-6].

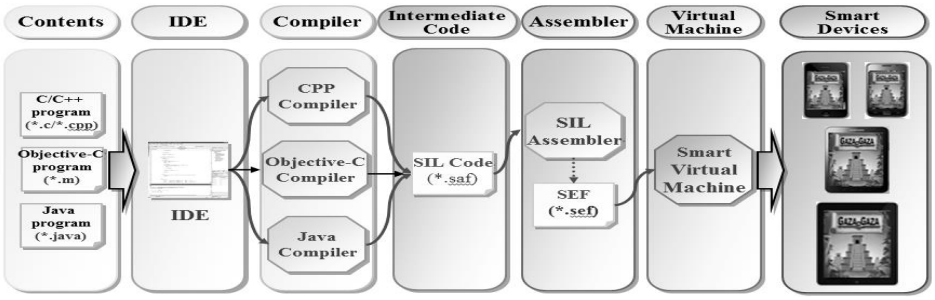

Figure 1. System Configuration of the Smart Cross Platform

It is designed to support procedural programming languages, object-oriented programming languages and *etc*. In the Smart Cross Platform, input contents are translate into SIL codes as intermediate language by compiler collection, and it is an advantage that the Smart Cross Platform cover the C/C++ and Java, which are the most widely used languages used by developers. SIL codes, the result of the compilation/translation process is changed into the running format SEF(SIL Executable Format) through an assembler. The SVM then runs the program after receiving the SEF.

The SIL code is a result of the compilation process and it is changed into the executing format SEF (Smart Executable Format) through an assembler. The SVM then runs the program after receiving the SEF. The SVM is composed by five major modules - SEF loader,

stack based interpreter, SVM built-in libraries, native interfaces, runtime environments, and runtime environments consist of exception handler, memory management, and thread scheduler. And the SVM is designed to easily add debugging interfaces, profiling interfaces, and *etc*.

2.2. SIL & SEF

SIL, the virtual machine code for SVM, is designed as a standardized virtual machine code model for ordinary smart devices and embedded systems [5, 7]. SIL is a stack based command set which holds independence as a language, hardware and a platform. In order to accommodate a variety of programming languages, SIL is defined based on the analysis of existing virtual machine codes such as bytecode [8-10], .NET IL [11, 12] and *etc*. In addition, it also has the set of arithmetic operations codes to cover procedural programming languages and object oriented languages.

SIL is composed of a meta-code which carries out particular jobs such as class creation and an operation code with responds to actual commands. An operation code has an abstract form which is not subordinated to specific hardware or source languages. It is defined in mnemonic to heighten readability and applies a consistent name rule to make debugging in assembly language levels easier. In addition, it has a short form operation code for optimization. SIL has 6 groups (except optimization group) of operation codes and Figure 2 shows the category of SIL operation codes.

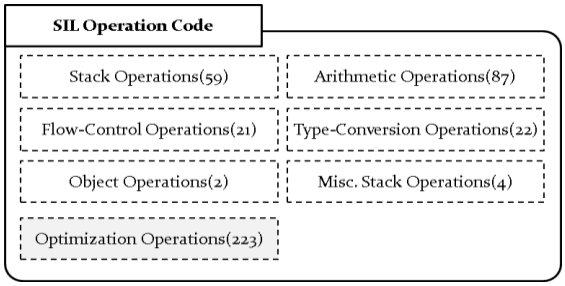

Figure 2. Category of SIL Operation Codes

SEF's structure largely consists of a header section which is in charge of expressing SEF files' composition, a program segment section and a debug section expresses debugging related information. The program segment section can be divided again into three sections which express codes and data [13]. The following Figure 3 is a simple diagrammed form of the SEF structure.

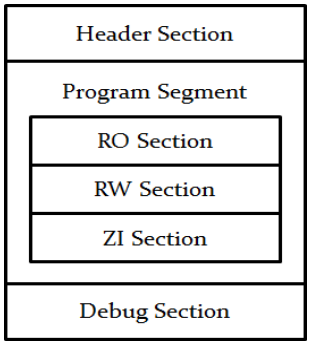

Figure 3. Structure of the SEF

In the header section, the detailed composition of program segments is expressed while information on programs' entry points is recorded. In addition, information related to the SEF file's header section is exclusively read to predict the entire memory expected to be used, and it is composed so that easy approaches can be made using detail sections as entry points.

Program segments are a form which is loaded and run in the SVM's memory and consists of pure codes and data. It separates data such as symbol tables and debugging information which are unimportant when running a SEF's program segment. This specific design is aimed towards minimizing the loading speed and memory space required by the SVM memory.

Program segments can be classified into the RO section, the RW section or the ZI according to the characteristics of the program components. Each of these areas has the following definition. First, the RO section stores codes and literal data which have read-only approach characteristics. Next, the RW section stores all global variable data which have initialization values on the source codes which have read-write approach characteristics. Finally, the ZI section refers to the section of global variables which do not have initialization values on the source codes. The debug section is space for expressing the debugging information of application programs stored in SEF. It is not loaded on the SVM's running memory and is used through IDE (Integrated Development Environment) or the debugger tool. According to the SVM compiling options, the debug section on SEF exists selectively and does not influence the running of the program.

2.3. Comparison of the Virtual Machine (VM) and the Native Execution Method

Aspects of executing contents, there are differences between the VM method and the Native. Firstly, the VM method cans easily executing the same VM application even if H/W changed because it has the H/W independence. Also, it has the stability for the target system when the errors exist in the VM contents and can be easily ported to various H/W.

On the other hand, the execution performance of the algorithm code is slower than the native method. In the case of the native method, need the changes of the contents by the characteristics of hardware platforms and OS, and it can affect the system due to the errors in the contents. But, the executing performance of the algorithm code is faster than the VM method [4, 6, 14, 15]. Table 1 shows the difference of the VM and the native execution method.

Table 1. Comparison of the VM and the Native

	Virtual Machine	Native
Execution Mechanism	Software, Intermediate Code / Software Register based JIT, Hot spot	Hardware, Machine Code
Hardware Dependency	Hardware Independent	Hardware Dependent
Execution Performance	Low	High
Contents Porting	Easy	Difficult
Stability on Error	High	Low

3. The Optimization Techniques for the SVM

SVM's detailed module configuration is shown in Figure 4. Largely, it combined 5 components; SEF loader which is load input SEF files on the memory, interpreter for stack based evaluating instructions in memory, managing module group for runtime environment, built-in SVM library, and native interface which is used for interaction with native platform. And, it also designed for the additional components like debugging and profiling interface.

The interpreter is SVM's core module which is SIL codes execution routine from loaded SEF file. The interpreter has action procedures that mapped on each SIL code, and it executes instructions with reference which is stored metadata by loader. On execution, evaluated data is stored and managed in stack or heap, and if error occurred while executing then the exception handler catch the occurred error and output the related error message and halt the VM instance for the given program.

In the portability of the contents, the SVM has an advantage, but it has the low performance due to the S/W interpretation for the instructions. Optimization techniques for solving these problems can be viewed from two major aspects. Firstly, the optimization of the instruction codes itself in the contents. The optimized code, because it reduces the cost of the interpretation of the virtual machine, is a very important issue in the virtual machine optimization. Next, the optimization of the interpretation method is one of the important issues.

The VM requires efficient interpretation method because the stack based interpretation under the general fetch-decode-dispatch method has very poor performance. In this chapter, we will discuss these issues.

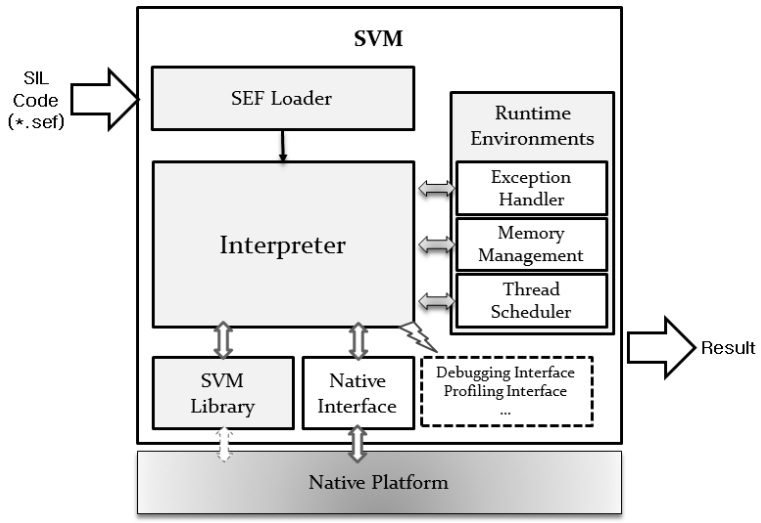

Figure 4. System Configuration of the Smart Virtual Machine

3.1. Code Level Optimization

Diverse optimization techniques have been deployed lately in the compilation process of a source program for improving the program's execution speed and reducing the size of the source code. Optimization techniques in the compiler development stage can be categorized into target machine-independent intermediate code optimization [16, 17] and target machine-dependent target code optimization. Furthermore, since recent attention regarding complier development has been focused on the retargetable optimization compiler that facilitates

applications in various target machines, there is an increasing need for target machine-independent intermediate code optimization.

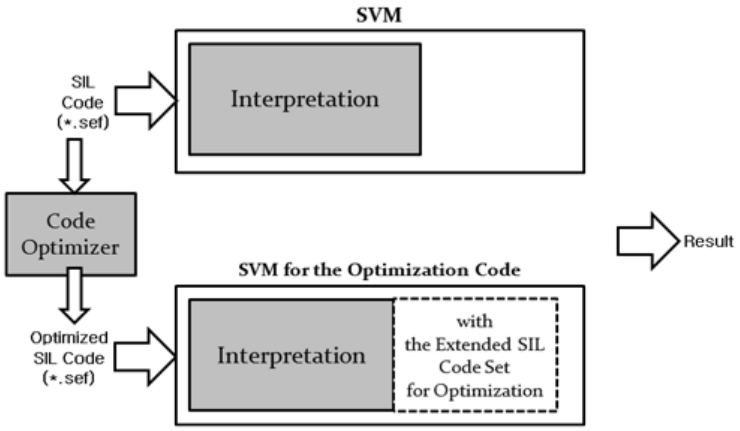

Figure 5. Code Optimization Model for SVM

As shown in Figure 5, if the SIL is optimized by the code optimizer, more efficient code execution is possible and the executing performance is increased. Also, reduce the contents size and improve the performance at the same time by the adding optimization specific SIL code can be abbreviate multiple instructions to smaller command [4, 6, 17].

3.2. System Level Optimization

Differently from the code level optimization perspective, the Virtual Machine (VM) system optimization may be considered. Especially, there are a variety of methodologies to improve the performance through the interpretation of a stack-based content, because it has a big impact on the degradation of the performance of the VM executing. Firstly, JIT, Back-End, Decompilation and Hotspot Compilation are the native executing method for the contents to avoid the disadvantages of the VM method. Such ways are the method to convert to native code and execute rather than directly executing the intermediate code in the VM [18].

4. Interpretation Optimization & Experimental Results

In this paper, we propose the interpretation routine optimization for the Smart Virtual Machine (SVM), the proposed technique is system level optimization. Figure 6 shows the typical VM interpretation model, and SVM has an implemented execution engine based on this general execution model.

This model executes matched instruction or calls API by lookup routine after load an instruction from SEF file lookup. In the this model, the lookup routine is hotspot cause it needed highly operation time. Following Table 1 shows profiling result of the needed time in the each module to executing game contents on the SVM.

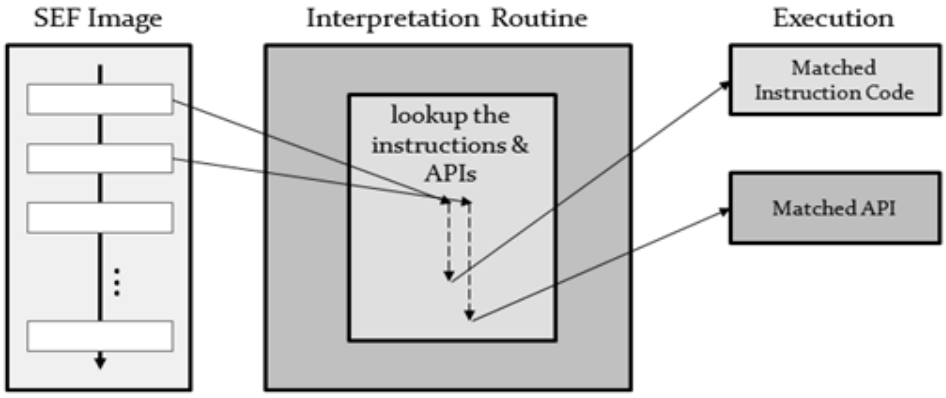

Figure 6. Typical Interpretation Model of SVM

Table 2 shows the top 10 SVM modules that needed highly operation time while executing the target content, and most time consumed module is the lookup routine which is to find instructions and APIs. This routine raises the execution time cost extremely, thus the performance of contents is decreased by frequently searching for to execute the each instruction in the interpretation routine.

Table 2. Performance Profiling Result for SVM

% time	cumulative seconds	self seconds	function name
93.11	6.76	6.76	Interpret::findFuncIndex
3.44	7.01	0.25	PutPixelSet
0.41	7.15	0.03	PutPixel
0.41	7.18	0.03	GetColor
0.28	7.20	0.02	DrawLineSet
0.28	7.22	0.02	CopyImageSet
0.14	7.23	0.01	SetColor
0.14	7.24	0.01	_op_call
0.14	7.25	0.01	_op_lod_i
0.14	7.26	0.01	Assemble::assembleOpCode

To solve this problem, as shown in Figure 7, the repetitive searching can be removed by the using of the mapping table for instructions/APIs and executing routines instead of lookup routine. In addition, by the executing of the mapped routine on the native level, the executing performance of the VM can be enhanced.

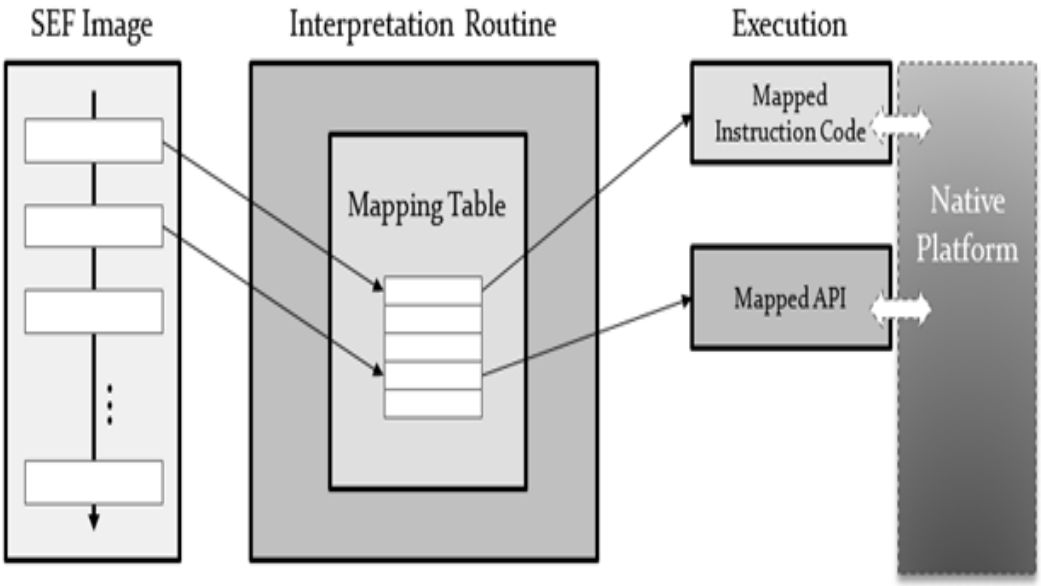

Figure 7. Enhanced Interpretation Routine Model for SVM

Using this proposed model, we had optimized the interpretation routine of the SVM and profiling the optimized SVM by the game content for Table 2. Following Table 3 shows the profiling result of the optimized interpretation routine. As the profiling result, dominant interpretation routine was eliminated in top 10 highly time consumed modules, and load of the stack based interpretation became lower cause overall modules related content execution are used.

Next, we show the experimental result that performance variation of before and after interpretation routine optimization. The test program is shown in Table 4 that selected to check the performance according to the complexity of algorithms [19], and input parameters are shown in Table 4. Experimental H/W is Samsung Galaxy Tab 10.1 and the version of the Android is 4.2.

Table 3. Performance Comparison of the SVM and the Optimized SVM

% time	cumulative seconds	self seconds	function name
36.11	0.13	0.13	PutPixelSet
11.11	0.25	0.14	DrawLineSet
5.56	0.31	0.02	Assemble::assembleOpCode
2.78	0.32	0.01	PutPixel
2.78	0.33	0.01	GetColor
2.78	0.34	0.01	CopyImageSet
2.78	0.35	0.01	loadTextureFromPNG
2.78	0.36	0.01	png_read_filter_row
0.00	0.36	0.00	DrawLine
0.00	0.36	0.00	SetImageAlpha

Table 4. Performance Comparison of the SVM and the Optimized SVM

Test Programs	Performance(millisecond)	
	SVM	SVM (using optimized interpretation routine)
PerfectNumber.sef (parameter: 2000)	3256	2519
PrimeNumber.sef (parameter: 1000)	1780	1130
MagicNumber.sef (parameter: 49)	28	18

Figure 8 shows the performance comparison SVM and SVM with optimized interpretation routine. Optimized SVM shows 23~37% reduced execution time than original SVM. This means that the optimization on interpretation routine affects performance of VM significantly. The performance of the SVM applied enhanced interpretation routine are shown in experiments. The test suit was selected as the program with high computational complexity algorithm to verify the performance of the interpretation routine. The selected test programs in Table 4 is suitable the experiment because we need to measure the interpretation time of the VM's arithmetic codes to examine the performance of the interpretation routine. Results of the experiments was confirmed, an average of 32% improvement in performance when applied to the optimized interpretation routine.

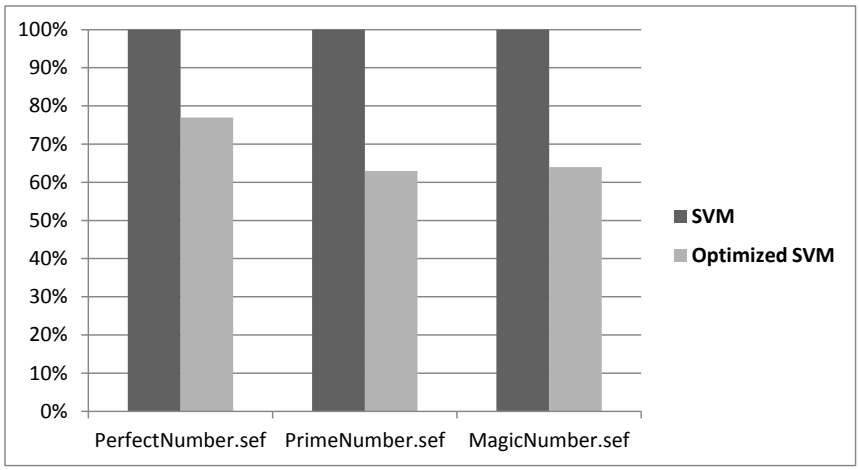

Figure 8. Execution Time Rate of SVM and Optimized SVM

5. Conclusions and Further Researches

A virtual machine has the characteristic of enabling application programs to be used without alteration even if processors or operating systems are changed. It is a core technology for executing a variety of contents in the recent mobile, embedded and smart systems. The virtual machine solution has the advantage on the portability, but it has the low performance due to the S/W interpretation for the instructions.

In this paper, we examined 2 kinds of optimization technique in order to increase the performance of the SVM. Using these optimization techniques, enables the execution of complex and various contents because cover the disadvantage - lower performance - of the

VM. And, to improve performance of the SVM and verify proposed optimization technique on execution engine aspect, we applied system level optimization based on SVM profiling results. The optimized SVM has 23~27% reduced execution time than the original SVM.

In the future, we will adapt the code level optimization on the optimized SVM, and verify the validity of the optimization techniques have considered in this paper by the performance evaluation.

Acknowledgements

This paper was extended from the previous research paper "A Study on Optimization Techniques for the Smart Virtual Machine Platform" in MulGraB 2012.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(No.20110006884), and in part by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and future Planning(No.2013R1A2A2A01067205)

References

[1] Apple, iOS Reference Library, iOS Technology Overview, http://developer.apple.com/devcenter/ ios.
[2] Goole, Android-An Open Handset Alliance Project, http://code.google.com/intl/ko/android/.
[3] Y. S. Lee, "The Virtual Machine Technology for Embedded Systems", Journal of Korea Multimedia Society, Korea Multimedia Society, vol. 6, no. 36, **(2002)**.
[4] Y. S. Son and Y. S. Lee, "A Study on the Smart Virtual Machine for Smart Devices", Information-an International Interdisciplinary Journal, International Information Institute, vol. 6, no. 1465, **(2013)**.
[5] Y. S. Lee and Y. S. Son, "A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms", International Journal of Smart Home, SERSC, vol. 6, no. 93, **(2012)**.
[6] S. M. Han, Y. S. Son and Y. S. Lee, "Design and Implementation of the Smart Virtual Machine for Smart Cross Platforms", Journal of Korea Multimedia Society, Korea Multimedia Society, vol. 16, no. 190, **(2013)**.
[7] S. L. Yun, D. G Nam, S. M. Oh and J. S Kim, "Virtual Machine Code for Embedded Systems", International Conference on CIMCA, vol. 206, **(2004)**.
[8] J. Meyer and T. Downing, JAVA Virtual Machine, O'REYLLY, **(1997)**.
[9] T. Lindholm and F. Yellin, "The Java™ Virtual Machine Specification", 2nd ed., Addison Wesley, **(1999)**.
[10] J. Engel, "Programming for the Java Virtual Machine", Addison-Wesley, **(1999)**.
[11] Microsoft, MSIL Instruction Set Specification, Microsoft Corporation, **(2000)**.
[12] S. Lindin, Inside Microsoft .NET IL Assembler, Microsoft Press, **(2002)**.
[13] H. S. Choi and Y. S. Lee, "Development of an Assembler for Generating the Executable File of the Ubiquitous Virtual Machine", Proc. of the 2007 Spring Conference, Korea Multimedia Society, vol. 10, no. 73, **(2007)**.
[14] J. E. Smith and R. Nair, "Virtual Machines", Morgan Kaufmann, **(2005)**.
[15] P. G. Vijayrajan, "Analysis of Performance in the Virtual Machines Environment", International Journal of Software Engineering and Its Applications, vol. 32, no. 53, **(2011)**.
[16] Y. S. Son and Y. S. Lee, "A Study on Optimization Techniques for the Smart Virtual Machine Platform", Lecture Notes in Computer Science(LNCS), Springer, vol. 7709, no. 167, **(2012)**.
[17] Y. S. Son and Y. S. Lee, "An Objective-C Compiler to Generate Platform-Independent Codes in Smart Device Environments", Information-an International Interdisciplinary Journal, International Information Institute, vol. 16, no. 1457, **(2013)**.
[18] Y. S. Lee, Y. K. Kim and H. J. Kwon, "Design and Implementation of the Decompiler for Virtual Machine Code of the C++ Compiler in the Ubiquitous Game Platform", Lecture Notes in Computer Science(LNCS), Springer, vol. 4413, no. 1, **(2007)**, pp. 511.
[19] H. K. Kim and R. Y. Lee, "Quality Validation for Mobile Embedded Software", International Journal of Advanced Science and Technology, vol. 1, no. 43, **(2008)**.

Authors

Yunsik Son, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. Currently, he is a Researcher of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, secure software, programming languages, compiler construction, and mobile/embedded systems.

Seman Oh, he received the B.S. degree from the Seoul National University, Seoul, Korea, in 1977, and M.S. and Ph.D. degrees from the Dept. of Computer Science, Korea Advanced Institute of Science and Technology, Seoul, Korea in 1979 and 1985, respectively. He was a Dean of the Dept. of Computer Science and Engineering, Graduate School, Dongguk University from 1993-1999, a Director of SIGPL in Korea Institute of Information Scientists and Engineers from 2001-2003, a Director of SIGGAME in Korea Information Processing Society from 2004-2005. Currently, he is a Professor of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.

JaeHyun Kim, he received the B.S. degree from the Dept. of Mathematics, Hanyang University, Seoul, Korea, in 1986, and M.S. and Ph.D. degrees from Dept. of Statistics, Dongguk University, Seoul, Korea in 1989 and 1996, respectively. He was a chairman of Dept. of Internet Information 2002-2007. Currently, he is a member of the Korean Data & Information Science Society and a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include mobile programming, cloud system and data analysis.

YangSun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2014, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006-2013 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2014. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.