A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms

YangSun Lee[1] and YunSik Son[2*]

[1]*Dept. of Computer Engineering, Seokyeong University*
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Korea

[2]*Dept. of Computer Engineering, Dongguk University*
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, Korea

yslee@skuniv.ac.kr, sonbug@dongguk.edu
**Corresponding Author: sonbug@dongguk.edu*

Abstract

In the existing smart devices, the contents for each platform there are a unique development environment, and thus are developed in a suitable method and the development of language is also different. These issues cause additional costs when developing the contents on various target platforms it is impossible to make the contents compatible on the various devices or platforms.

Proposed The Smart Virtual Machine (SVM) based on the language independent intermediate language is a newly developed virtual machine solution which is aimed towards solving these problems. It uses an intermediate language, the Smart Intermediate Language (SIL), which can cover the object oriented languages such as C++, Java, Objective C and more.

In this paper, we deal with a virtual machine, SVM, based on stack and capable of being run on various smart devices. SVM receives a SIL code which is semantically equivalent to a program created with different languages and interprets it based on stack on a software level. Then it runs the programs so that specific smart device operation systems and devices can load them and therefore have the advantage of being platform independent.

Keywords: *Smart Virtual Machine (SVM), Smart Intermediate Language (SIL), Smart Platform, Virtual Machine, Stack Interpretation, Runtime Model*

1. Introduction

Existing smart phone contents developing environments requires target codes to be made depending on the target device or platform. Each platform also has different development languages [1, 2]. Therefore even if the contents are identical, according to the target device, they are required to be redeveloped and require different compilers for different target devices. This leads to inefficient contents development. The SVM(Smart Virtual Machine) is a solution invented to solve this drawback. It is a stack based virtual machine which allows application programs to be downloaded and run platform independently once loaded on smart devices. It is run by inserting the SIL(Smart Intermediate Language) designed by our research team.

The SVM solution largely consists of three parts; a compiler, assembler and virtual machine. It is designed in a hierarchical way which minimizes the burden of the retargeting

process. In this research, a virtual machine, SVM, has been specifically de-signed and created to be run on various smart devices after receiving a SIL code input [3-7].

2. Related Studies

2.1. Smart Intermediate Language (SIL)

SVM's virtual machine code, SIL [3-7], has been designed as a standard model of virtual machine codes for ordinary smart phones and embedded systems. SIL is a set of stack based commands which has the characteristics of language independence, hardware independence and platform independence. In order to accommodate various programming languages, SIL has been defined based on the analysis of existing virtual codes such as .NET IL [8, 9], bytecode [10-11] and etc. It possesses an operation code set which can accommodate both object-oriented language and procedural language. 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 of operation codes and Figure 1 shows the category of SIL operation codes.

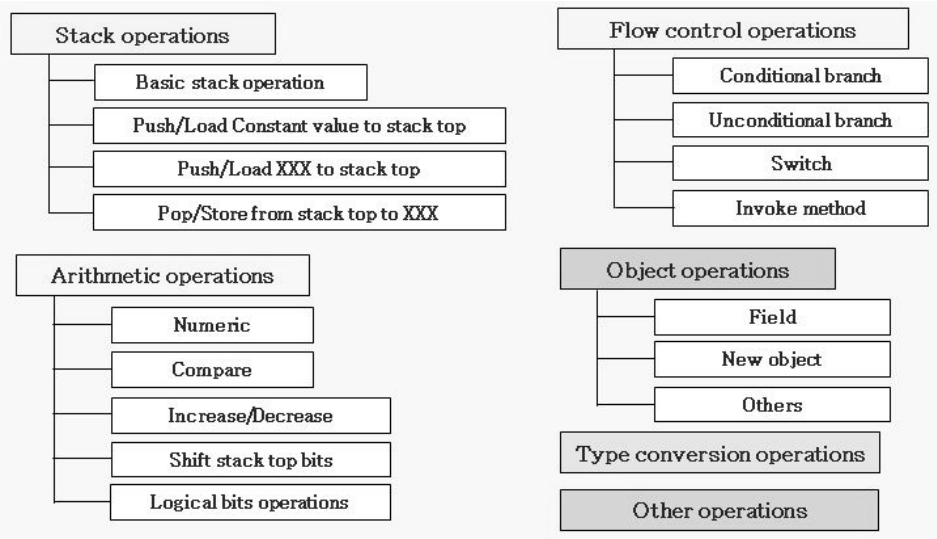

Figure 1. Category of SIL Operation Codes

2.2. Smart Assembly Format (SAF)

The code created using high level programming language is converted into SVM's assembly format, through the code converter. The SAF format consists of pseudo code and operation code. This is then converted into a Smart Executable Format (SEF) through the assembler and is run using the SVM regardless of the system's operating system or structure. Table 1 shows the descriptions of SAF's major mnemonics.

SAF includes a pseudo code which carries out class creation and other specific jobs and an operation code which responds to the actual commands run in the virtual machine. The operation code is a set of stack based commands which is not subordinate to specific

programming languages, therefore possessing language independence, hardware independence and platform independence. As a result, an operation code's mnemonic has an abstract form as it is not subordinate to any specific hardware or source languages [4-7, 12].

Table 1. Selected Major Mnemonics for SAF

Mnemonic	Description
%HeaderSection[Start/End]	Define the range of the header section.
%CodeSection[Start/End]	Define the range of the code section.
%DataSection[Start/End]	Define the range of the data section.
%DebugSection[Start/End]	Define the range of the debug section.
%DefinedLiteralCount	Number of literals.
%IntializedVariableCount	Number of initialized global variables.
%UninitializedVariableCount	Number of uninitialized global variables.
%ExternalVariableCount	Number of external variables.
%ExternalFunctionCount	Number of external functions.
%InitFunctionName	Name of the initialize function for object.
%EntryFunctionName	Name of the entry point function for program execution.
%SourceFileName	Describe the program source file name.
%Function[Start/End]	Define the range of the function.
%Label	Describe the program source file name.
%Line	Describe the program source file name.
%LiteralTable[Start/End]	Define the range of the literal table section.
%InternalSymbolTable[Start/End]	Define the range of the internal symbol table section.
.func_name	Describe the function name.
.func_type	Describe the function types.
.param_count	Describe the number of parameters for the function.
.opcode_[start/end]	Define the range of the operation code section for the function.

2.3. Smart Executable Format (SEF)

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 [4-7, 12]. The following Figure 2 is a simple diagrammed form of the SEF structure.

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.

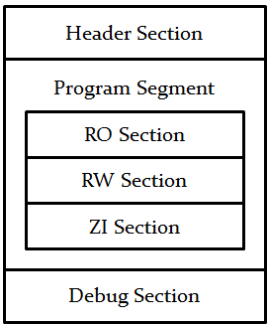

Figure 2. Structure of SEF

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.

3. The Smart Virtual Machine for Smart Platforms

SVM is a stack based virtual machine solution, loaded on smart devices, which al-lows dynamic application programs to be downloaded and run platform independently. SVM is designed to use an intermediary language, SIL, which is capable of accommodating both procedural and object-oriented languages. It has the advantage of accommodating languages such as C/C++, Java, and Objective-C used in the iOS, which are currently used by a majority of developers. Figure 3 is a diagram of SVM system.

The SVM system consists of three parts; a compiler which compiles application programs to create a SAF form file made from SIL code, an assembler which converts the SAF file into the execution formation SEF, and a virtual machine which receives the SEF form file and runs the program. SVM's system is designed in a hierarchical manner to minimize the burden of retargeting processes caused by different devices and execution environments. SIL which is generated during the compile/translate process is converted into SEF through the assembler and SVM receives SEF input to run a program [4-7, 12].

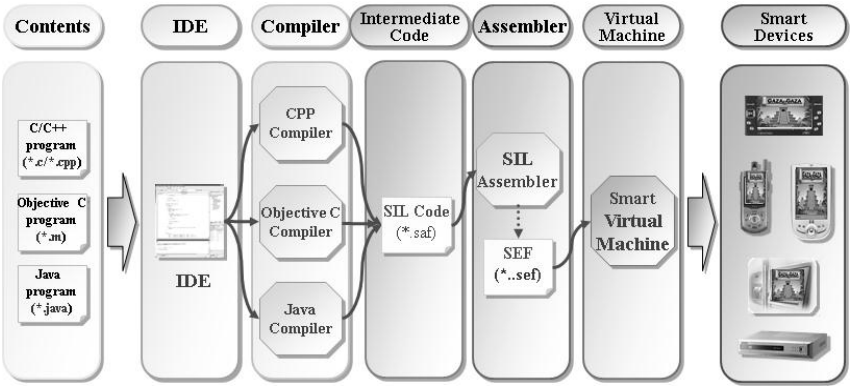

Figure 3. System Configuration of SVM

3.1. Composition of the Smart Virtual Machine

The composition of SVM detailed modules is as Figure 4. It is composed of four factors; a SEF loader which loads inserted files on to the memory, an interpreter which calculates commands to stack bases, a module for managing the execution environment and a library. In addition, it has a native interface in order to use the native platform's function, an interface for debugging and profiling and additional components.

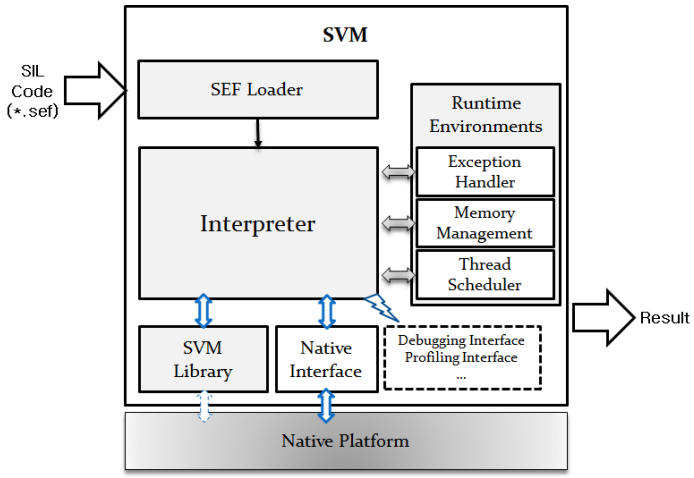

Figure 4. System Configuration of the Smart Virtual Machine

The core model of SVM is the interpreter which is the actual routine which runs the SIL codes of SEF files. Actual processing actions for each SIL codes has been written and formulated. It refers to the Meta information stored through the loader and carries out commands. The data that occurs during program execution are stored and controlled in the stack and hip.

While running a program, if an error occurs, an exception occurs. This exception put out as a message and will be shut down through the exception processor.

3.2. Interpreter Runtime Workspace

Being a virtual machine, SVM claims to develop and run contents that can be inter-converted regardless of what device is being used. In order to provide such flexible portability,

SVM must be transferred to various devices and have a runtime workspace structure which is not subordinate to any hardware or platforms.

Due to the characteristics of a virtual machine, SVM uses a certain section's memory that it is delegated from hardware or platform. Of this memory, the memory space required for the SIL interpreter to carry out calculations is called Runtime Workspace. Runtime workspace consists of three elements; runtime stack, activation record and display vector. The diagram of the system composition and the relationship between each element is as can be seen in Figure 5.

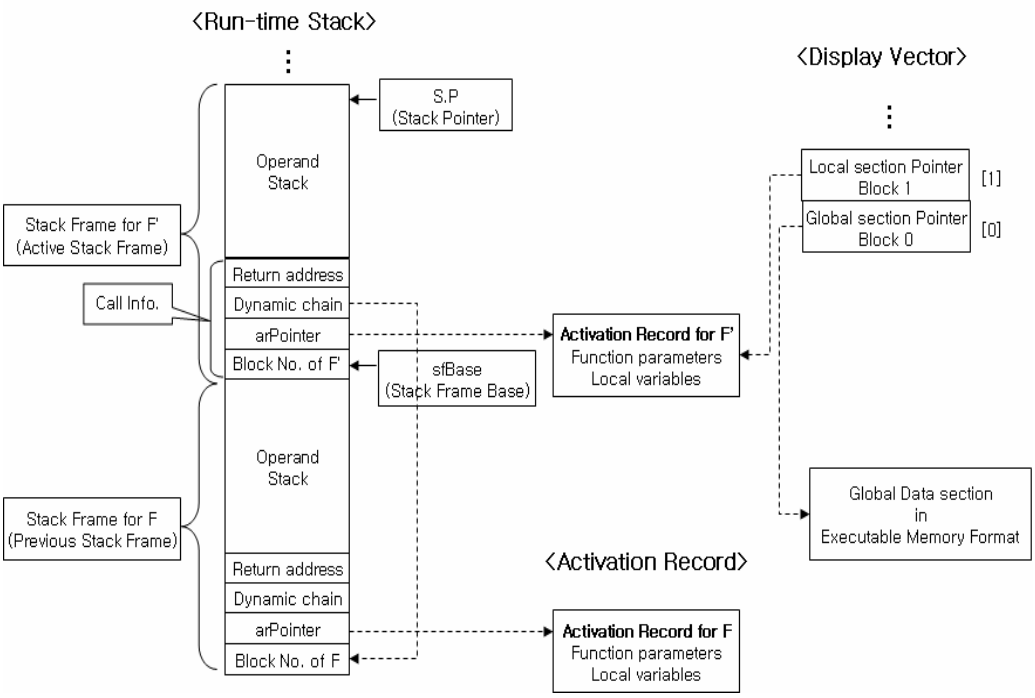

Figure 5. Configuration of Runtime Workspace

Runtime stack is the storage space for the operand used by the SIL interpreter during calculations and includes call information related to function calls. The local variable storage space, Activation Record, is being referred to through the Call Information's arPointer category. Also, in order to refer to the higher blocks (which involve one's function) within a function or all sections' variables, the Display Vector information must be maintained.

Firstly, runtime stack is a 4 byte signed integer type sequence. The size of runtime stack is set to be static and the runtime space used for each function call is set as stack frame. Thus, runtime stack is a linear stack structure made up of more than one stack frame. The size of runtime stack is set by the interpreter depending on the size of the operation stack used by an average function and the acceptable call depth. Figure 6 is a diagram of runtime stack.

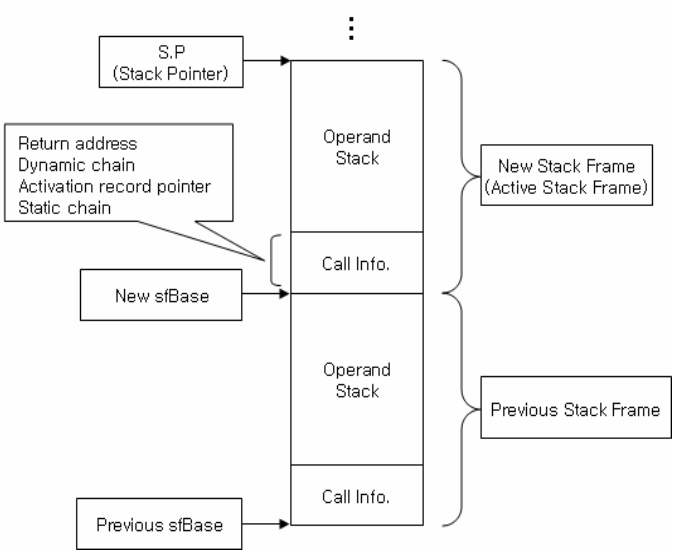

Figure 6. Configuration of Runtime Stack

Next, activation record is space for storing local variables. Thus, for one stack frame, one activation record can be delegated. When the function is terminated and the stack frame is collected, the activation record too is cleared. Figure 7 is a diagram of an Activation Record.

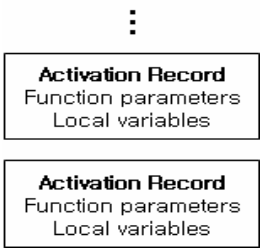

Figure 7. Diagram of Activation Record

Activation records are created in separate space from runtime stack and have pointers for Activation Records in Stack Frame's Call Information. This value is set in the display vector when activating stack frame and carries out simple local variable reference. The size of activation records is the sum of the size of the function's factor and the local variable. The size is received from the proc operation codes' instruction parameter after which memory is delegated. In the case of variable factor functions, the size of the factor can be known at the function call point. As a result, in the compiler, before the variable factor function call, the code for loading the factor's size on the stack is created and the local variable's size is created as the first factor of the procva operation code. The data within activation records can be approached in a byte unit offset and this value is generated in the compiler.

Finally, the display vector is a Byte* type sequence which holds purpose in maintaining the Static Chain between the data sections for each stack frame. Figure 8 shows the diagram of the Display Vector.

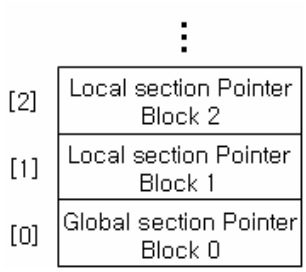

Figure 8. Diagram of Display Vector

The first index of the Display Vector sets a pointer for the global data section and when all sections' variables of the program are referred to, this value is used as the base to take the offset calculation approach. The value of the first index is set at the point of contents loading and does not change even when exiting the program. From the second index, when calling functions an Activation Record pointer is set on the function's the index for the relevant block No. Block No. is called up from the second instruction parameter of the proc operation code and shows the functions comprehension depth. Thus, for a function with the block No. 2 which refers to the local variable that comprehends itself, the display vector's index value can be used as a base for approach. The values of the display vector, excluding the first index, change every time a stack frame is delegated/collected. The size of the display vector is defined by the Workspace Manager after consideration of the acceptable functions' comprehensive depth.

Next, in case of function call, the runtime workspace changes explain. Figure 9 shows a function call example code for runtime workspace.

```
void main()
{
  int x, y;
  ...
  x = f(1, 2);
}
```

```
int f(int a, int b)
{
  char c;
  int i;
  …
  return i;
}
```

Figure 9. Example Code

Firstly, at the time of the main function execution, the activation record and the stack frame for main function have been assigned in the runtime workspace. And the pointer for the activation record is stored in arPointer of the call information. A pointer value for the global data section is stored in the display vector's [0] index when loading the program, and arPointer of the main function is assigned to [1] index of the display vector at the start of the main function.

Next, the function f is called in the main function, the stack frame and the activation record for the f are newly allocated in the runtime workspace, and the sfBase and the stack top pointer are adjusted for the new execution conditions. The arPointer is set up to the call information of f, and the sfBase value of the caller (main) function is stored for use as dynamic chain. In the activation record of the function f, storage for the function arguments (a, b) and the local variables (c, i) is allocated, and the arPointer value of the f is stored in the

display vector's [1] index. Preparing to run the function f is a completed process. Figure 10 shows the configuration of the runtime workspace for the example code in Figure 9.

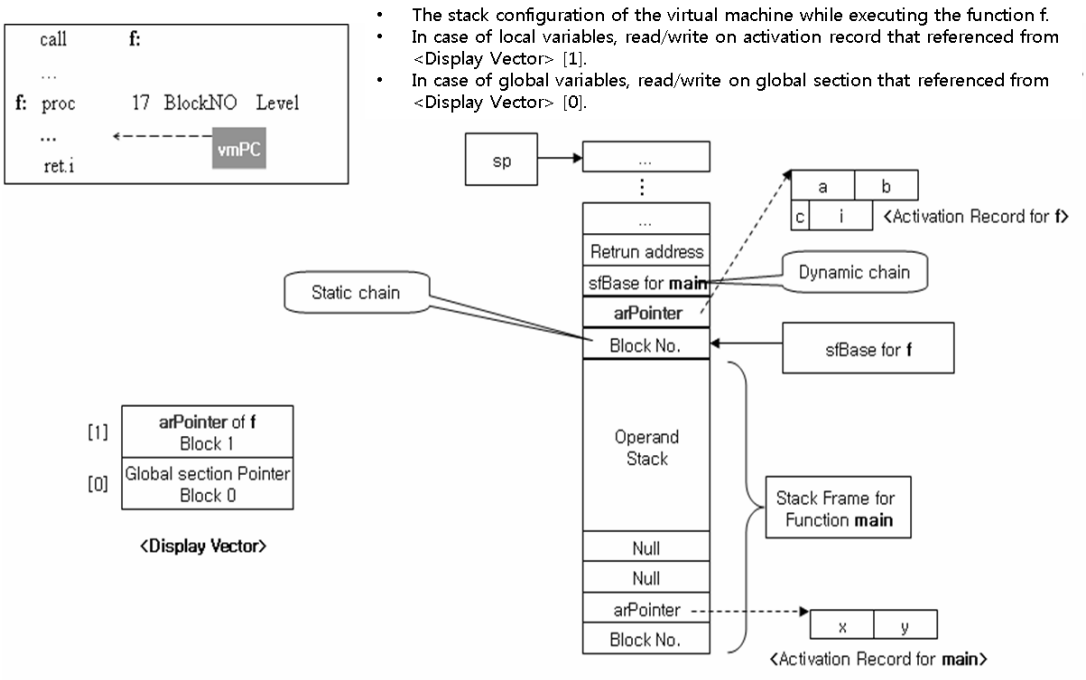

Figure 10. Runtime Workspace for the Example Code

After finish of the function f's execution, completion of the return function f, the activation record for the function f is released and the stack frame of f is collected. The stack top pointer is adjusted by the value of the f's sfBase and the sfBase value is reset by the dynamic chain information. The value of the display vector's [1] index is set to arPointer of the main function, and the return value of f stored in a temporary location is pushed on the stack. The code of the main function is ready to continue.

4. Experimental Results and Analysis

In this research, the virtual machine for smart devices was largely composed of four modules and this virtual machine was designed and formulated. In order to test the formulated SVM, the Objective-C compiler of the SVM system was used to compile a source program for calculating factorials and the SAF files created were the subject of experimentation. The contents of the SAF files created can be seen in Figure 11.

```
%%HeaderSectionStart
  %DefinedLiteralCount  6
  %InitializedVariableCount 0
  %UninitializedVariableCount 0
  %ExternalVariableCount  20
  %ExternalFunctionCount  0
  %InitFunctionName 0
  %EntryFunctionName  0
  %SourceFileName factorial2.m
%%HeaderSectionEnd
%%CodeSectionStart
  %FunctionStart
    .func_name  &Factorials::fact$1
    .func_type  2
    .param_count  1
    .opcode_start
      proc  16  1 1
      str.p 1 0
      str.i 1 4
      ldc.i 1
      str.i 1 8
      lod.i 1 4
      str.i 1 12
%Label  ##0
      lod.i 1 12
      ldc.i 0
      gt.i
      fjp ##1
      lod.i 1 8
      lod.i 1 12
      mul.i
      str.i 1 8
%Label  ##2
```

Figure 11. SAF Example for Calculating Factorial

To run the created SAF files on the SVM, they must be converted into SEF files through the assembler. Figure 12 shows the conversion results of converting the SAF files from Figure 11 into an executable format through the assembler.

```
00000000h: F8 04 00 00 5C 00 00 00 0C 07 00 00 FF FF FF FF
00000010h: D0 04 00 00 9D 00 10 00 00 00 01 00 01 00 2A 00
00000020h: 01 00 00 00 00 00 28 00 01 00 04 00 00 00 07 00
00000030h: 01 00 00 00 28 00 01 00 08 00 00 00 10 00 01 00
00000040h: 04 00 00 00 28 00 01 00 0C 00 00 00 10 00 01 00
00000050h: 0C 00 00 00 07 00 00 00 00 00 60 00 8E 00 88 00
00000060h: 00 00 10 00 01 00 08 00 00 00 10 00 01 00 0C 00
00000070h: 00 00 42 00 28 00 01 00 08 00 00 00 10 00 01 00
00000080h: 0C 00 00 00 03 00 07 00 01 00 00 00 3C 00 28 00
00000090h: 01 00 0C 00 00 00 8F 00 38 00 00 00 10 00 01 00
```

Figure 12. SEF Example for Figure 11

The results of running the SEF file is as can be seen in Figure 13.

```
>svm factorial.sef
>input: 10
>result: 3628800
>svm factorial.sef
>input: 5
>result: 120
```

Figure 13. Execution Result of Factorial Example

Next, the results of using the same method for the games Elemental Force and Aiolos on SVM can be seen. The contents of Elemental Force and Aiolos were in GNEX/WIPI contents form and were converted using the GNEX/WIPI-Objective C contents converter [13-19]. The converted contents were made to be executable on SVM using the Objective C-SIL compiler. Figure 14 shows the execution result of compiled contents.

Figure 14. Execution Result of Elemental Force and Aiolos in SVM

Next, the compiler shown in Figure 15 was used to compile the game contents written in Objective C. Figure 15(a) is a screenshot from an iOS simulator during the execution of original game content written in Objective C; Figure 15(b) shows the execution of compiled code using the proposed compiler for the SVM on the android platform. The implemented compiler correctly generates SVM codes from the source contents for iOS.

a) Experimental Content in iOS

b) Compiled Code for Experimental Content in SVM-Android

Figure 15. Execution Screenshots

5. Conclusions

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.

In this study, a virtual machine which allows great numbers of applications to be downloaded and run on smart devices has been designed and formulated. The virtual machine formulated in this study enables contents to be run without alterations even if there is a change in platforms. It is also a software technology which can accommodate any development language such as C/C++, Objective-C and Java. Therefore

programmers need not be restricted by languages when developing application programs.

In the future, more research will be carried out to enhance the virtual machine's running speed, optimize executable codes and to actively analyze executable codes by researching the modules such as the debugging module within the virtual machine.

Acknowledgements

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).

This paper was extended from the previous research paper "A Study on the Virtual Machine for Smart Device" in AITS-MSA 2012.

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] S. M. Oh, Y. S. Lee and K. M. Ko, "Design and Implementation of the Virtual Machine for Embedded Systems", Journal of Korea Multimedia Society, vol. 8, no. 9, **(2005)**, pp. 1282-1291.
[4] Y. S. Lee, "The Virtual Machine Technology for Embedded Systems", Korea Multimedia Society, vol. 6, no. 2, **(2002)**, pp. 36-44.
[5] Y. S. Son and Y. S. Lee, "Design and Implementation of the Virtual Machine for Smart Devices", Proc. of the 2011 Fall Conference, Korea Multimedia Society, vol. 14, no. 2, **(2011)**, pp. 93-96.
[6] 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. 1, **(2007)**, pp. 73-76.
[7] 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, to be published, International Information Institute, **(2013)**.
[8] A. Troelsen, "C# and the .NET Platform", APRESS, **(2001)**.
[9] Microsoft, "MSIL Instruction Set Specification", Microsoft Corporation, **(2000)**.
[10] J. Engel, "Programming for the Java Virtual Machine", Addison-Wesley, **(1999)**.
[11] J. Meyer and T. Downing, "JAVA Virtual Machine", O'REYLLY, **(1997)**.
[12] Y. S. Son and Y. S. Lee, "Design and Implementation of an Objective-C Compiler for the Virtual Machine on Smart Phone", Multimedia, Computer Graphics and Broadcasting, CCIS, vol. 262, **(2011)**, pp. 52-59.
[13] Y. S. Lee, H. J. Choi and J. S. Kim, "Design and Implementation of the GNEX-to-iPhone Converter for Smart Phone Game Contents", Journal of Korea Multimedia Society, vol. 14, no. 4, **(2011)**, pp. 577-584.
[14] Y. S. Son, S. M. Oh and Y. S. Lee, "Design and Implementation of the GNEX C-to-Android Java Converter using a Source-Level Contents Translator", Journal of Korea Multimedia Society, vol. 13, no. 7, **(2010)**, pp. 1051-1061.
[15] Y. S. Lee and Y. S. Son, "A Study on the WIPI-to-Windows Mobile Game Contents Converter using a Resource Converter and a Platform Mapping Engine", Information: An International Interdisciplinary Journal, to be published, International Information Institute, **(2013)**.
[16] WIPI (Wireless Internet Platform for Interoperability), KWISF (Korea Wireless Internet Standardization Forum), **(2004)**.
[17] Y. S. Lee and Y. S. Son, "A Platform Mapping Engine for the WIPI-to-Windows Mobile Contents Converter", Multimedia, Computer Graphics and Broadcasting, Springer, CCIS, vol. 262, **(2011)**, pp. 69-78.
[18] Y. S. Lee, "Design and Implementation of the GNEX C-to-WIPI Java Converter for Automatic Mobile Contents Translation", Journal of Korea Multimedia Society, vol. 13, no. 4, **(2010)**, pp. 609-617.
[19] Y. S. Lee, "Automatic Mobile Contents Converter for Smart Phone Platforms", Korea Multimedia Society, vol. 15, no. 1, **(2011)**, pp. 54-73.

Authors

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, 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, and a President of a Society for the Study of Game at Korea Information Processing Society from 2006. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and mobile/embedded systems.

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.