Translation Java Bytecode to EVM SIL Code for Embedded Virtual Machines

YangSun Lee*, JinKi Park**

ABSTRACT

This paper presents the bytecode-to-SIL translator which enables the execution of the java program in EVM(Embedded Virtual Machine) environment without JVM(Java Virtual Machine), translating bytecodes produced by compiling java programs into SIL(Standard Intermediate Language) codes. EVM, what we are now developing, is a virtual machine solution that can download and execute dynamic application programs written in sequential languages like C language as well as object oriented languages such as C#, Java, etc. EVM is a virtual machine mounted on embedded systems such as mobile device, set-top box, or digital TV, and converts the application program into SIL, an assembly language symbolic form, and execute it. SIL is a virtual machine code for embedded systems, based on the analysis of existing virtual machine codes such as bytecode. MSIL, etc. SIL has such features as to accommodate various programming languages, and in particular has an operation code set to accept both object-oriented languages and sequential languages. After compiling, a program written in java language is converted to bytecode, and also executed by JVM platform but not in other platform such as .NET, EVM platform. For this reason, we designed and implemented the bytecode-to-SIL translator system for programs written in java language to be executed in the EVM platform without JVM. This work improves the execution speed of programs, enhances the productivity, and provides an environment for programmers to execute application programs at various platforms.

Keywords: EVM, SIL, JVM, Bytecode, Intermediate Language Translator

1. INTRODUCTION

A virtual machine is a conceptual computer with a logical system configuration, made of software unlike physical systems made of hardware. Use of virtual machine technology does not require modification of application programs though processors or operating systems are changed. Typical virtual machines include JVM that executes Java bytecode[2,8,9].

At present, a research of virtual machine that can accommodate both MS’s C# language and SUN’s Java language is in progress. We are now developing the virtual machine solution named EVM which can download and execute the dynamic application programs written in sequential languages like C language as well as object-oriented languages such as C#, Java, etc. to be run at virtual machine mounted on embedded systems by converting the programs into .evm file format through .sil, an assembly language symbolic form.

SIL, the virtual machine code of EVM, is the target code of a translator that inputs java bytecode or .NET MSIL. If source programs are translated into SIL through the translator of the EVM system regardless of programming languages, they can be converted into .evm format through SIL.
assembler. SIL is designed as the standard intermediate language of a virtual machine for embedded systems. Thus, it was defined based on an analysis of virtual machine codes such as bytecode, MSIL, etc. widely used to accommodate various programming languages, and in particularly has an operation code set to accept both object oriented languages and sequential languages[4,10,14].

Java, one of the most widely used programming languages recently, is the language invented by James Gosling at Sun Microsystems, which is the next generation language independent of operating systems and hardware platforms. Java source code is compiled into bytecode as intermediate code independent of each platform by compiler, and also executed by JVM, the Java interpreter. Therefore, Java programming language is both compiler and interpreter language, but more universal than normal compilers and runs faster and more efficiently than usual interpreters[2,9,10,19].

This paper presents the bytecode-to-SIL intermediate language translator which enables the execution of the java program in EVM environments without JVM, translating bytecodes produced by compiling java programs into SIL codes. After compiling, a program written in Java language is converted to bytecode, and also executed by JVM platform but not in other platform. For this reason, we designed and implemented the bytecode-to-SIL translator system for java programs to be executed in the EVM platform without JVM. This work provides an environment for programmers to develop application programs without limitations of programming languages.

2. RELATED WORKS

2.1. Bytecode

Bytecode can be considered as a machine language for JVM (Java Virtual Machine). It is acquired in the stream format for each method within a class when JVM loads a class file. At this time, the stream is in the binary stream format of a 8-bit byte. Furthermore, a bytecode basically has a stack-oriented structure, originally intended for being used through an interpreter. In other words, it is either interpreted by JVM, or compiled when the class is loaded [2,9,10,19].

JVM saves and executes the source code by java programming language using class file format. Since the class file is in binary format, it is very difficult to analyze and modify. On the other hand, Ooolong code, another type of java intermediate language, is much easier to read and write compared to class file format. Ooolong code is based on Jasmin language of John Meyer, and designed to work in the level of bytecode[9].

Figure 1 schematically shows the process of extraction of Ooolong code from class file. In order to obtain an assembly format file as an input for the translator from the class file acquired by java compiler, javac, we used the Ooolong decompiler, Gooloo. Gooloo carries out the function to extract only the source code-related contents from the various data in the class file.

2.2. SIL Code and EVM

1) SIL Code

SIL (Standard Intermediate Language), the virtual machine code of EVM, is the target code of a translator that inputs Java Bytecode or .NET IL. If source programs are translated into SIL through the translator of the EVM system regardless of programming languages, they can be converted in

[Image: Ooolong Code Extracting Process]
to *.evm format through SIL assembler[47,18]. SIL is designed as the standard intermediate language of a virtual machine for embedded systems. Thus, it was defined based on an analysis of virtual machine codes such as Oolong, .NET IL, etc. widely used to accommodate various programming languages.

SIL consist of pseudo codes which are the directives of specific works such as class declaration and operation codes corresponding to actual instructions executed by virtual machines. Operation codes are a group of stack-based instructions, independent of specific programming languages and of hardware and platform. Therefore, the mnemonic of operation codes has an abstract format independent of specific hardware or source language.

SIL codes consist of a group of operation codes to accommodate both object-oriented languages and sequential languages, securing the readability of codes by applying coherent naming rules to make easy debugging on the level of symbolic assembly language. Also, it has operation codes with short parameter form for optimization and operation codes in pseudo form.

The mnemonic of pseudo codes improved readability by using the keyword on the level of source codes as it is, and defined a total of indicators such as class, field, method, exception handling, and etc. Table 1 shows all pseudo codes and their meanings.

The operation code of SIL corresponds to actual instructions executed by virtual machines. Operation codes are stack-based instructions, and expressed by one byte. The mnemonic of operation codes basically defines English words meaning tasks executed by corresponding operation code. Establishing the alphabet meaning simplified forms and the combination of integers as a principal, if information on types in relation to the execution of operation codes is required, add a character ‘.’ expressing types using ‘.’. Operation codes must begin with the alphabet, and no special symbols except for ‘.’ are used.

Operation codes can have instruction parameters necessary for operation, and the number of instruction parameters differs according to operation codes. To provide the polymorphism of operation codes, the operation code of Arithmetic category and of Type conversion operations category enables the operand to have the type information of the operand. There are 83 SIL operation codes, which can be largely divided into 6 main categories. Each category can have subcategories. Table 2 shows the category of SIL operation code.

2) EVM

EVM(Embedded Virtual Machine), what we are now developing, is a virtual machine solution that can download and execute dynamic application programs written in sequential languages like C language as well as object oriented languages such as C#, Java, etc. to be run at virtual machine mounted on embedded systems such as mobile device, set-top box, or digital TV by converting the programs into SIL, an assembly language symbolic form[47,18,20,23].

Figure 2 shows the structure of EVM. The system is largely divided into three components. The first part translates programs written in high-level programming languages such as C# or Java into
Table 2. Operation Code of SIL.

<table>
<thead>
<tr>
<th>Arithmetic Operations(15)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Pops two values, adds them, and pushes the result</td>
</tr>
<tr>
<td>sub</td>
<td>Pops two values, subtracts them, and pushes the result</td>
</tr>
<tr>
<td>neg</td>
<td>Pops an value, negates it, and pushes the result</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Operations(33)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>Pop the top word from the operand stack</td>
</tr>
<tr>
<td>pup</td>
<td>Duplicate top operand stack word</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Operations(7)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ldfl</td>
<td>Pushes field, indicated by index, of object</td>
</tr>
<tr>
<td>strfl</td>
<td>Sets field, indicated by index, of object to value</td>
</tr>
<tr>
<td>new</td>
<td>Creates a new object on the heap, pushes reference</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow Control Operations(23)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>uip</td>
<td>Branch to offset</td>
</tr>
<tr>
<td>eq</td>
<td>Pop value1 and value2; if value1==value2, branch to offset</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Convert Type Operations(4)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>convi</td>
<td>Pop value1, converts value1 to integer, and pushes the result</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other Operations(1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>Throw exception or error</td>
</tr>
</tbody>
</table>

SIL programs proposed in this paper. The second part is an assembler that converts SIL codes into *.evm format that can be executed by virtual machines. The third part is a virtual machine which is mounted on hardware and runs *.evm file. The virtual machine part of EVM is hierarchically designed to minimize a burden in a retargeting process.

2.3. TRANSLATOR

One of studies about a translator is about an intermediate language translator[1,3,5,16] that generates a native code of the target machine from bytecode for improving an execution speed of java applications. It, however, can't be used in other computer environments because of being dependent on a target machine.

Microsoft has developed the JLCA translator[12] which converts a java program into a C# program in a source language level. With the iNET translator[6] the Haicyon Soft generates java source program from .NET MSIL code, an intermediate code of .NET programming languages. The Remotesoft, on the contrary, has developed the Java.NET translator[15] which translates a java source program into a .NET MSIL code.

It can't guarantee the security of source programs that a source program is translated into an intermediate code like JCLA or Java.NET. It also takes more time, and is hard to have good performance and is difficult to overcome the semantic gap, because translating an intermediate code into a source program such as iNET should have several translation steps. To solve the above-mentioned problems, we developed an intermediate language translator such as Bytecode-to-MSIL translator, MSIL-to-Bytecode translator[20,21,22,23].

3. JAVA BYTECODE TO EVM SIL TRANS- LATOR

3.1 An Outline of Translator System

Translator System is implemented on purpose to execute java program in the embedded virtual
machine, EVM, and converts java bytecode into SIL code. Translating process of code is the followings. First, easily utilizing Oolang code is drawn from java bytecode by using Oolang disassembler from the class file. Then extracted Oolang code converts into SIL code by the translator. Translated SIL code is created to an executable file(*.evm) by SIL assembler, and executed in the embedded virtual machine, EVM; therefore it is able to execute the java program in the embedded system without JVM. Figure 3 shows the process that Oolang code is drawn from the class file, and this extracted Oolang code is converted into SIL code by the translator.

3.2 Structure of Translator System

Bytecode-to-SIL translator system is a translator that Oolang code extracted from the class file is inputted, and converts into SIL code, an intermediate language of EVM. The translator is designed for more effective code conversion by executing an optimization for the information possessed by AST using compiler technique with utilizing AST; it is different from the existing translators that utilize a one-pass form for establishing easily at other platforms.

Figure 4 displays the structure of the translator system. Translator system is composed of scanner, parser, SDT(Syntax-Directed Translation), AST(Abstract Syntax Tree), and ICT(Intermediate Code Translator). Context-free grammar is designed for implementing scanner and parser with analyzed bytecode instruction. Lexical information and parsing table are created by PGS(Parser Generating System) and by inputting the grammar. After scanner analyzes token of Oolang instruction with created lexical information, it delivers Oolang code to parser as token units. Parser executes syntax analysis in reference to the parsing table delivered token from the scanner. Parser creates AST showing the structure of Oolang program, an input program through SDT in execution of a shift-reduce parsing.

Figure 5 displays the grammar with AST node for the bytecode instruction.

Figure 6 displays the structure of node of AST. The inner expressive method defines that left link son points son node by using left son/right brother and right link brother points brother node. And, nodered is a field in order to display whether the node is terminal node or not; and token is a structure having tokenNumber for storing sorts of token and tokenValue for storing values of token. Figure 7 shows the basic structure of AST created by parser.

3.3 Implementation of Translator System

1) Instruction Mapping Table

SIL code mapping for Oolang code is composed of 1:1, 1:N, N:1 in accordance to sorts of instruction. Table 3 is a sort of mapping SIL code by categories.
Table 3. Instruction Mapping Table.

<table>
<thead>
<tr>
<th>Category</th>
<th>Oolang Code</th>
<th>Mapping SIL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*add ( * - Type)</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>*mul ( * - Type)</td>
<td>mul</td>
<td></td>
</tr>
<tr>
<td>*and ( * - Type)</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>*or ( * - Type)</td>
<td>or</td>
<td></td>
</tr>
<tr>
<td>Flow Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifne</td>
<td>ldc 0, ne</td>
<td></td>
</tr>
<tr>
<td>ifeq</td>
<td>ldc 0, eq</td>
<td></td>
</tr>
<tr>
<td>if_icmple</td>
<td>Le</td>
<td></td>
</tr>
<tr>
<td>if_icmpge</td>
<td>ge</td>
<td></td>
</tr>
<tr>
<td>Object Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getfield</td>
<td>ldfld</td>
<td></td>
</tr>
<tr>
<td>putstatic</td>
<td>strfld</td>
<td></td>
</tr>
<tr>
<td>invokespecial</td>
<td>call</td>
<td></td>
</tr>
<tr>
<td>invokevirtual</td>
<td>call</td>
<td></td>
</tr>
<tr>
<td>Constant Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*const_k</td>
<td>ldc</td>
<td></td>
</tr>
<tr>
<td>(* - Type, k - integer constant)</td>
<td>&lt;parameter&gt;</td>
<td></td>
</tr>
<tr>
<td>Convert Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2l</td>
<td>convl</td>
<td></td>
</tr>
<tr>
<td>i2l</td>
<td>convi</td>
<td></td>
</tr>
<tr>
<td>Stack Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dup2</td>
<td>dup, dup</td>
<td></td>
</tr>
<tr>
<td>Variable Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*store ( * - Type)</td>
<td>ldc</td>
<td></td>
</tr>
<tr>
<td>(* - Type)</td>
<td>&lt;parameter&gt;</td>
<td></td>
</tr>
</tbody>
</table>

2) Translation Algorithm

Oolang code consists of one class field and more than one method, composed of one method field and more than one operation. Operation is composed of all 201 parts such as arithmetic operation and object operation, and each operation has a type. However, SIL code is characterized by that operation does not have a type, and locals pseudo code declares numbers and type of local identifier. This type of translation from Oolang code to SIL code is related with operation coming out mostly in the
method pseudo code.

Translation of each operation is not effective because it has a pretty complex structure. Thus, it creates AST, the form of intermediate expression, categorized the operation into NON, ONE, TWO, THREE on the basis of the numbers of parameter by using scanner that Ooolong code is divided into token unit by utilizing standardized technique of compiler, parser that is parsing, and SDT module showing semantic rules. Traversing the created AST node, it maintains a type in the type table from the operation having a type of local identifier, and translates the operation by sorts using mapping table.

Figure 8 displays an algorithm of translating bytecode to SIL code.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Translation Method

Figure 9 shows a process that translator converts Ooolong code into SIL code traversing created AST by parser. Translator translates Ooolong code to the appropriate SIL code in reference to instruction mapping table by traversing each node of the AST.

4.2 Example of Translating Bytecode into SIL Code

1) Java Program

The following is an example of using a Bytecode-to-SIL translator for implementing thread program by Java programming. Program 1 is Java thread program.

Program 2 shows an expression of AST utilizing the bytecode-to-SIL translator that implements Ooolong code extracting from the class file that compiles java thread program of program 1 compiles and creates the class file.

Program 3 shows a translation of SIL code utilizing the bytecode-to-SIL translator that implements Ooolong code extracting from the class file that compiles java thread program of program 1 compiles and creates the class file.
Program 1. Java Program.

```
Class ThreadTest extends Thread {
    public ThreadTest(String name) {
        super(name);
    }
    public void run() {
        //Thread Method
        System.out.println("is now running.");
    }
    public static void main(String[] args) {
        int i = 0;
        //Assignment
        int sum = 0;
        //Create Object
        ThreadTest t = new ThreadTest("Simple Thread");
        for(i = 0; i < 10; i++) {
            //For Statement
            sum += i;
        }
        System.out.println("ForStatement : sum = 1 to 10 = "+ sum);
        t.start();
        //Thread Start
        try {
            //Exception Check Block
            while(i != 0) {
                //While Statement
                i = i + 2;
            }
        } catch(ArithmeticException e) {
            //exception handler
            System.out.println("WhileStatement : ArithmeticException");
        }
    }
}
```

Program 2. AST and Oolong Code.

```
source example.java
.class public super example
super java/lang/Object

.method public static main
(Ljava/lang/String;)V
.limit stack 3
.limitlocals 5
    .catch java/lang/ArithmeticException
        from 31 to 39 using 1000
        .catch java/lang/NullPointerException
        from 40 to 48 using 1000
    .locals 10
        0: iconst_0
        1: astore_1

    .new java/lang/Thread
        .dup
        .ldc ";Simple Thread"
        .invokespecial ThreadTest.<init>:(Ljava/lang/String;)V
        .astore_3

    .aload_3
    .invokespecial ThreadTest/start:(IV)

    .aload_1
    .ifne 84

    .goto 113
    .check-cast <sub_expression>
    .return
.end method
```


Fig. 10 shows the result of the execution after translating the java Oolong program to SIL program and converting it into executable file. It shows the same results from the execution of ThreadTest.java extracted by the Oolong disassembler from the class file generated by the Java compiler and the execution of SIL file ThreadTest.sil generated by the translator with ThreadTest.sil file as input.

Table 4 shows the result of performance evaluation between java bytecode program in JVM and SIL code program in EVM platform. As the table displayed, it takes little time for SIL code programs to be executed rather than java programs, 20% speed on average. Therefore, SIL code programs can be executed in EVM platform without JVM with this bytecode-to-SIL translator, it improve
developing the virtual machine solution named EVM which can download and execute the dynamic application programs written in sequential languages like C language as well as object-oriented languages such as C#, Java, etc., to be run at virtual machine mounted on embedded systems by converting the programs into .evm file format through .sil, an assembly language symbolic form.

After compiling, a program written in Java language is converted to bytecode, and also executed by JVM platform but not in .NET platform. For this reason, we designed and implemented the bytecode-to-SIL translator to convert the bytecode, intermediate language for JVM, into the SIL code, intermediate language for the EVM platform. It enables the execution of Java programs on the EVM platform environment without JVM. This work improves the productivity of programs, and provides an environment for programmers to develop application programs without limitations of programming languages.

7. REFERENCES


YangSun Lee
1985 Computer Science, Dongguk University (B.S.)
1987 Computer Engineering, Dongguk University (M.S.)
1993 Computer Engineering, Dongguk University (Ph.D.)
1994~Present Associate Professor, Dept. of Computer Engineering, SeoKyeong University
1996~2000 Chief of Computer Center, SeoKyeong University
2000~2004 Director of Korea Multimedia Society
2005~Present General Director of Korea Multimedia Society
2001~2005 Director of SIGPLAN in KISS
2004~Present Director of KSII
2004~2005 Director of SIGGAME in KIPS
2005~Present President of SIGGAME in KIPS
Research Areas: Programming Languages, Embedded Systems, Mobile Computings, Game Engineering

JinKi Park
1985 Computer Science, Dongguk University (B.S.)
1989 Computer Engineering Dongguk University (M.S.)
2006 Computer Engineering SeoKyeong University, (Ph.D.)
2000~Present Director of Moneytek Inc.
2004~Present Part-time Instructor, Dept. of Computer Engineering, SeoKyeong University
Research Areas: Programming Languages, Embedded Systems, Mobile Computings, Game Engineering