JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 8, NO. 12, DECEMBER 2005(pp. 1658-1668)

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 particularly 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

※ Corresponding Author : YangSun Lee, Address : (136-704) 16-1 JungneungDong SungbukKu Seoul, Korea, TEL : +82-2-940-7292, FAX : +82-2-919-0345, E-mail : yslee@skuniv.ac.kr

[†] Dept. of Computer Engineering, Seokyeong Univ., Korea.

[††] Dept. of Computer Engineering, Seokyeong Univ., Korea. (E-mail : jkpark@ pl.skuniv.ac.kr)

Receipt date : Aug. 29, 2005, Approval date : Sep. 13, 2005

※ This work was supported by grant No.(R01-2002-000-00041-0) from the Basic Research Program of the Korea Science & Engineering Foundation

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 complier 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(**J**ava **V**irtual **M**achine). 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, Oolong code, another type of java intermediate language, is much easier to read and write compared to class file format. Oolong 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 Oolong 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 Oolong decompiler, Gnoloo. Gnoloo 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(**S**tandard **I**ntermediate **L**anguage), 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

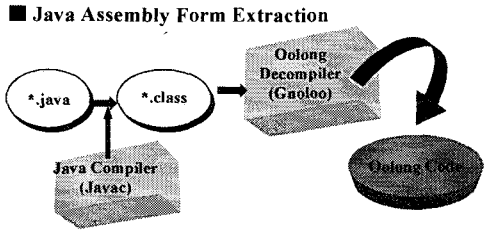

Fig. 1 Oolong Code Extracting Process.

to *.evm format through SIL assembler[4,7,18]. 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.

Table 1. Pseudo Code of SIL.

Pseudo Code	Meaning
.class	Defines the name of class created and access modifiers.
.field	Declares the field of class.
.method	Declares the method of class.
.maxstack	Restricts the number of maximum stacks of corresponding methods.
.throws	Declares exception generated from corresponding methods.
.catch	Sets an area for checking exception at corresponding methods.
.end	Indicator used to indicate the end of blocks
.locals	Declares local variables.

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(**E**mbedded **V**irtual **M**achine), 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[4,7,18,20,23].

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.

Arithmetic Operations(15)	
add	Pops two values, adds them, and pushes the result
sub	Pops two values, subtracts them, and pushes the result
neg	Pops an value, negates it, and pushes the result
...	...

Stack Operations(33)	
pop	Pop the top word from the operand stack
pup	Duplicate top operand stack word
...	...

Object Operations(7)	
ldfld	Pushes field, indicated by index, of object
strfld	Sets field, indicated by index, of object to value
new	Creates a new object on the heap, pushes reference
...	...

Flow Control Operations(23)	
ujp	Branch to offset
eq	Pop value1 and value2; if value1==value2, branch to offset
...	...

Convert Type Operations(4)	
convi	Pop value1, converts value1 to integer, and pushes the result
...	...

Other Operations(1)	
throw	Throw exception or error

SIL programs proposed in this paper. The secondpart 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.

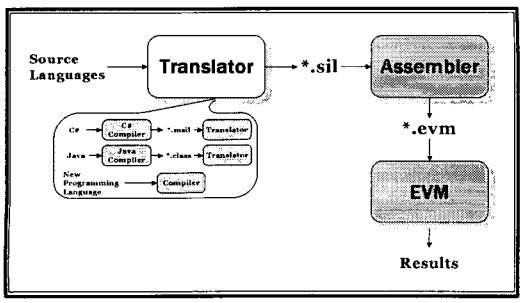

Fig. 2. EVM System Configuration.

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 Halcyon 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 Oolong code is drawn from java bytecode by using Oolong disassembler from the class file. Then extracted Oolong 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 Oolong code is drawn from the class file, and this extracted Oolong code is converted into SIL code by the translator.

3.2 Structure of Translator System

Bytecode-to-SIL translator system is a translator that Oolong 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 Oolong instruction with created lexical information, it delivers Oolong 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 Oolong 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, noderep 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 Oolong 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.

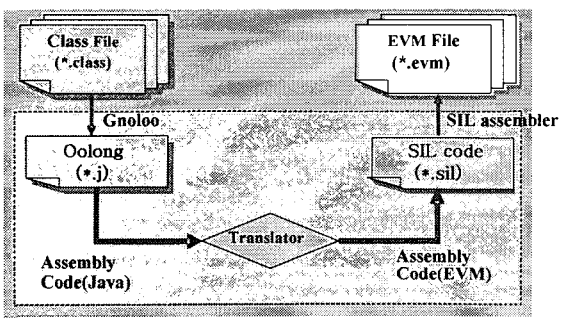

Fig. 3. Oolong-to-SIL Translator Model.

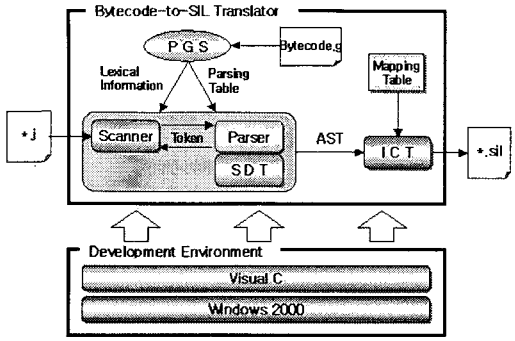

Fig. 4. Translator System.

```
BYTECODE_G   -> HEAD_PART METHOD_PART=> PROGRAM;
HEAD_PART -> HEAD_LIST
                                    => HEAD_PART;
HEAD_LIST  -> HEAD_PSEUDO HEAD_LIST;
           ->;
HEAD_PSEUDO->'.source' 'IDENT'          => HPSEUDO_SOURCE;
           ->'.class' MODIFIER_ST 'IDENT' => HPSEUDO_CLASS;
           ->'.super' SUPER_NAME        => HPSEUDO_SUPER;
           ->'.interface' 'PATH'        => HPSEUDO_INTERFACE;
           ->'.implements' 'PATH'       => HPSEUDO_IMPLEMENTS;
           ->'.field' MODIFIER_ST 'IDENT' DESCRIPTOR_TYPE
                                        => HPSEUDO_FIELD;
           ->'.throws' 'PATH'           => HPSEUDO_THROWS;
SUPER_NAME->'IDENT';
           ->'PATH';
            :
METHOD_PART -> METHOD_PART METHOD  => METHOD_PART;
             -> METHOD;
             ->;
METHOD->'.method' METHOD_START METHOD_BODY METHOD_END
                                    => METHOD;
METHOD_START->MODIFIER_ST METHOD_NAME DESCRIPTOR
                                    => METHOD_START;
METHOD_NAME -> '<init>'             => METHOD_INIT;
            ->'IDENT'               => METHOD_NAME;
DESCRIPTOR->'(' ARGUMENT_LIST ')' DESCRIPTOR_TYPE
                                    => DESCRIPTOR;
ARGUMENT_LIST-> ARGUMENT_ST         => ARGUMENT;
ARGUMENT_ST  -> ARGUMENT ARGUMENT_ST;
             -> ;
ARGUMENT -> DESCRIPTOR_TYPE;
         ->;
METHOD_BODY->METHOD_PSEUDO STATEMENTS => METHOD_BODY;
METHOD_PSEUDO -> PSEUDO_ST;
               -> METHOD_PSEUDO PSEUDO_ST;
               -> ;
               :
STATEMENT->'.line' 'NUM'                => SOURCELINE;
          -> LABEL SOURCE;
LABEL -> LABEL_NAME                     => LABEL;
      ->;
LABEL_NAME->'LABEL'                     => LABEL_NAME;
SOURCE->'OPCODE' SOURCE_ARGS            => SOURCE;
       ->TABLE_OP                       => TABLE_OP;
SOURCE_ARGS->ONE_ARGS                   => ONE_PARAM;
            ->TWO_ARGS                  => TWO_PARAM;
            ->THREE_ARGS                => THREE_PARAM;
            ->                          => NON_PARAM;
             :
```

Fig. 5. The Bytecode Instruction Grammar with AST node.

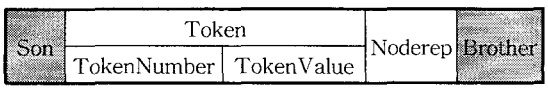

Fig. 6. Node Structure of AST.

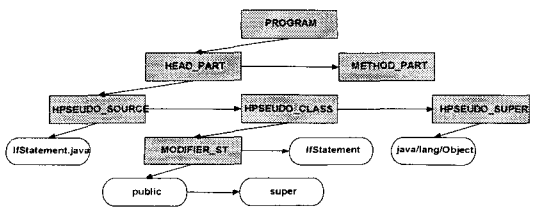

Fig. 7. Structure of AST.

Table 3. Instruction Mapping Table.

Category	Oolong Code	Mapping	SIL Code
Arithmetic Instruction	*add (* - Type)	N : 1	add
	mul (- Type)		mul
	and (- Type)		and
	or (- Type)		or
	...	...	...
Flow Instruction	Ifne	1 : N	ldc 0, ne
	ifeq		ldc 0, eq
	if_icmple	1 : 1	Le
	if_icmpge		ge
	...	...	...
Object Instruction	getfield	1 : 1	ldfld
	putstatic		strsfld
	invokespecial		call
	invokevirtual		calli
	...	...	...
Constant Instruction	*const_k (* - Type, k - integer constant)	N : 1	ldc <parameter>
	...	...	...
Convert Instruction	d2l	1 : 1	convl
	l2i		convi
	...	...	...
Stack Instruction	dup2	1 : N	dup, dup
	...	...	...
Variable Instruction	*load (* - Type)	N : 1	ldl <parameter>
	store (- Type)		str <parameter>
	...	...	...

2) Translation Algorithm

Oolong 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 Oolong 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 Oolong 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.

```
void translator () {
   scaner(); // Token division
   root = parser(); // Input program Parsing
   writeTree(root); // AST creation
   writeSIL(root); // SIL code creation
}
void writeSIL (Node *pt) {
   locals(root); // Local variable information maintenance
   codegen(root); // Translate the node of AST
}
void locals (Node *pt) {
   switch (pt->token.number) {
      case NON_PARAM: ONE_PARAM: {
         if (opnum >=0 && opnum <= 47) {
            // The Operation which it load/store in local variable
            // Store to the type table
         }
       }break;
      }
}
void codegen (Node *pt) {
   switch (pt->token.number) {
      case PSEUDO_NODE: {
         // Translate the pseudocode
         // If exist subnode, recursive call
      }break;
      case LOCALS: {
         for (i=methoddepth; i<=methoddepth; ++i) {
            for (j=1; j<=stackNum; ++j) { // Used local variable counter
               if (localNum[i][j] == 1) { // Search from the type table
      }break;
   case NON_PARAM: ONE_PARAM: TWO_PARAM:
THREE_PARAM:{
         if (tmp->token.opnum != NULL) {
            if (opcode >= 132 && opcode <= 139) {
               // Translate the 1:N mapping
            }
            else { // Translate the 1:1 mapping
               if (param != NULL) { // Exist parameter
                  // Fetch the parameter
(NUM, IDENT, PATH, PARAMETER)
               }
            }
         }
      }break;
   }
}
```

Fig. 8. Translation Algorithm.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Translation Method

Figure 9 shows a process that translator converts Oolong code into SIL code traversing created AST by parser. Translator translates Oolong 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 Oolong 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 Oolong code extracting from the class file that compiles java thread program of program 1 compiles and creates the class file.

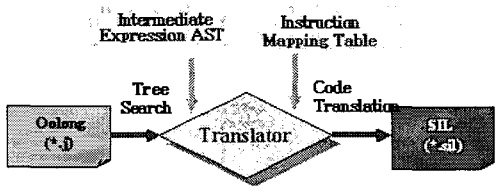

Fig. 9. Translation using AST.

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

Program 1. Java Program.

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

Nonterminal: PROGRAM
        Nonterminal: HEAD_PART
                Nonterminal: HPSEUDO_SOURCE
                        Terminal: example.java
                Nonterminal: HPSEUDO_CLASS
                        Nonterminal: MODIFIER_ST
                                Terminal: public
                                Terminal: super
                        Terminal: example
                Nonterminal: HPSEUDO_SUPER
                        Terminal: java/lang/Object
            :
            :
157: aload_3
158: invokevirtual
ThreadTest/start ()V

Nonterminal: LABEL
    Nonterminal: LABEL_NAME
        Terminal: 157:
Nonterminal: SOURCE
    Terminal: aload_3
    Nonterminal: NON_PARAM
Nonterminal: LABEL
    Nonterminal: LABEL_NAME
        Terminal: 158:
Nonterminal: SOURCE
    Terminal: invokevirtual
    Nonterminal: TWO_PARAM
 Nonterminal: TWO_PARAM_METHOD
            Terminal: ThreadTest/start
            Nonterminal: DESCRIPTOR
                Nonterminal: ARGUMENT
                Nonterminal: VOID_DESC

193: iload_1
194: ifne 164

Nonterminal: LABEL
    Nonterminal: LABEL_NAME
        Terminal: 193:
Nonterminal: SOURCE
    Terminal: iload_1
    Nonterminal: NON_PARAM
Nonterminal: LABEL
    Nonterminal: LABEL_NAME
        Terminal: 194:
Nonterminal: SOURCE
    Terminal: ifne
    Nonterminal: ONE_PARAM
        Nonterminal: ONE_PARAM_IDENT
            Terminal: 164
                :
```

Program 2. AST and Oolong Code.

Oolong code		SIL code
.source example.java .class public super example .super java/lang/Object ...	Class (1:1)	.class public super example .super java/lang/Object ...
.method public static main ([Ljava/lang/String;)V .limit stack 3	Method (1:1)	.method public static V main ([Ljava/lang/String;]
.limit locals 5		.locals o i i o o
.catch java/lang/ ArithmeticException from l61 to l97 using l100	Exception (1:1)	.catch java/lang/ ArithmeticException from l61 to l97 using l100
l0: iconst_0 l1: istore_1	Assignmetn (1:1)	l0: ldc 0 l1: str 1
...		...
l4: new ThreadTest l7: dup l8: ldc "SimpleThread" l10: invokespecial ThreadTest/<init> (Ljava/lang/String;)V l13: astore_3	Class Objet (1:1)	l4: new ThreadTest l7: dup l8: ldc "SimpleThread" l10: call V ThreadTest/<init> (Ljava/lang/String;) l13: str 3
...		...
l57: aload_3 l58: invokevirtual ThreadTest/start ()V	Thread Start (1:1)	l57: ldl 3 l58: calli V ThreadTest/start ()
...		...
l93: iload_1 l94: ifne l64	If Statement (1:1)	l93: ldl 1 l94: ldc 0 l94: ne l64
l97: goto l113 <sub_expression> ... l113: return .end method		l97: ujp l113 <sub_expression> ... l113: ret .end

Program 3. Oolong Code and SIL Code.

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.j 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.j 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

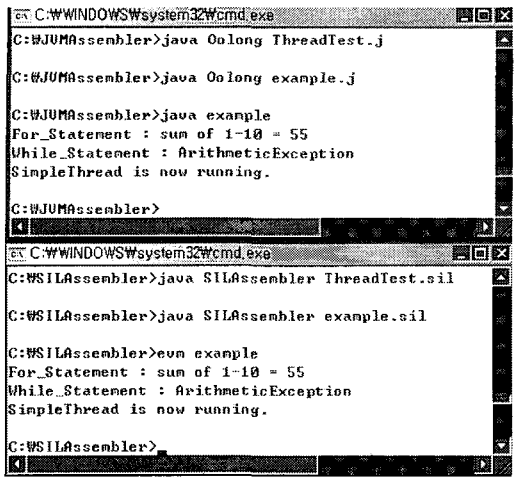

Fig. 10. Execution Result.

Table 4. The Result of Performance Evaluation.

Program	Java Bytecode	EVM SIL Code
Perfect number	634ms	445ms
Palindrome number	263ms	183ms
Quick sort	277ms	188ms
Class	338ms	265ms
Extended Class	363ms	273ms
Exception Handling	336ms	264ms
Thread	342ms	270ms

the execution speed of java applications, and provides an environment for programmers to develop application programs without limitations of programming languages.

6. CONCLUSION

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.

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.

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 de-signed 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 enhances the productivity of programs, and provides an environment for programmers to develop application programs without limitations of programming languages.

7. REFERENCES

[1] A.Krall and R.Grafl, "CACAO: A 64 bit Java VM Just-in-Time Compiler," *Concurrency: Practice and Experience,* 1997.

[2] Bill Venners, *Inside the JAVA Virtual Machine*, Second Edition, McGraw-Hill, Dec. 1998.

[3] C.A. Hsieh, M.T. Conte, and T.L. Johnson, "Java Bytecode to Native Code Translation: the Caffeine Prototype and Preliminary Results," *Proceedings of the IEEE 29th Annual International Symposium on Microarchitecture*, Dec. 1996.

[4] DongKeun Nam, SungLim Yun, and SeMan Oh, "Assembly Language of Virtual Machines," *Proceedings of the KIPS'2003 Spring Conference*, Vol.10, No.1, pp.783-786, 2003.

[5] Harlan McGhan and Mike O'Conner, "PicoJava: A Direct Execution Engine for Java Bytecode,"

IEEE Computer, pp.22-30, 1998.

[6] Halcyon Soft, iNET, 2003. http://www.halcyonsoft. com/

[7] JiHoon Jung, JinKi Park, and YangSun Lee, "Java Bytecode-to-.NET MSIL IL Translator," *Proceedings of the KIPS'2003 Fall Conference*, Vol.10, No.2, pp.663-666, 2003.

[8] John Gough, *Compiling for the .NET Common Language Runtime(CLR)*, Prentice Hall, 2002.

[9] Jon Meyer and Troy Downing, *Java Virtual Machine*, O'REILLY, Mar. 1997.

[10] Joshua Engel, *Programming for the Java Virtual Machine*, Addison-Wesley, Jan. 1999.

[11] Ken Arnold and James Gosling, *The Java™ Programming Language*, 3rd ed., Addison-Wesley, 2000.

[12] Microsoft, JLCA; Java-Language-to-C# Conversion Assistant, 2002. http://www.microsoft.com/korea/press/pressroom/2002/02/02.htm

[13] Microsoft Corporation, *C# Language Specification*, Nov. 2000.

[14] Microsoft Corporation, *MSIL Instruction Set Specification*, Nov. 2000.

[15] Remotesoft, Java.NET, 2003. http://www.remotesoft. com/

[16] Ronald Veldema, "Jcc, A Native Java Compiler," *Vrije Universiteit Amsterdam*, July 1998.

[17] Serge Lidin, *Inside Microsoft .NET IL Assembler*, Microsoft Press, 2002.

[18] SungKyou Choi, JiHoon Jung, and YangSun Lee, "A Study on Translator of C# MSIL Code into Oolong Code for Embedded Systems," *Proceedings of the KIPS'2003 Spring Conference*, Vol.10, No.1, pp.983-986, 2003.

[19] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1999.

[20] YangSun Lee, et al., "Java Bytecode-to-.NET MSIL Translator for Construction of Platform Independent Information Systems," *LNAI 3215*, Springer-Verlag, pp.726-732, Sept. 2004.

[21] YangSun Lee, et al., "Intermediate Language Translator for Execution of Java Programs in .NET Platform," *Journal of Korea Multimedia Society*, Vol.7, No.6, pp.824-831, Jun. 2004.

[22] YangSun Lee, et al., "Design and Implementation of a MSIL-to-Bytecode Translator to execute .NET Programs in JVM Platform," *Journal of Korea Multimedia Society*, Vol.7, No.6, pp.976-984, Jun. 2004.

[23] YangSun Lee, et al., "IL Translator for JIT Execution of Java Bytecode in .NET Environments," *International Conference on Cybernetics and Information Technologies, Systems and Applications(CITSA'2004)*, pp. 80-85, Jul. 2004.

[24] YangSun Lee, et al., "Design and Implementation of a Virtual Machine for Embedded Systems," *Journal of Korea Multimedia Society*, Vol.8, No.9, pp.1282-1291, Sept. 2005.

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 Technology

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 Engineerings