Intermediate Language Translator for Execution of Java Programs in .NET Platform

YangSun Lee*, SeungWon Na**, DaeHoon Hwang***

ABSTRACT

This paper presents the java bytecode-to-.NET MSIL intermediate language translator which enables the execution of the java program in .NET environments without JVM(Java Virtual Machine), translating bytecodes produced by compiling java programs into MSIL codes. 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. .NET language such as C# and .NET platform in Microsoft Corp. has been developed to meet the needs of programmers, and cope with Java and JVM platform of Sun Microsystems. After compiling, a program written in .NET language is converted to MSIL code, and also executed by .NET platform but not in JVM platform. For this reason, we designed and implemented the java bytecode-to-.NET MSIL translator system for programs written in java language to be executed in the.NET platform without JVM. This work improves the execution speed of programs, enhances the productivity, and provides an environment for programmers to develop application programs without limitations of programming languages.

Keywords: Bytecode, MSIL, Intermediate Language, Translator, JVM, .NET Platform

1. INTRODUCTION

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 is a language that generates execution files through compiling. The generated files are in the bytecode form, which is interpreted and executed by JVM(Java Virtual Machine), 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[2,5,8].

On the other hand, C# language provided by Microsoft .NET platform is offered as a platform-independent language like Java of Sun Microsystems, the development of components by C# is becoming widespread. C# is a language based on the characteristics of high coding convenience and easy maintenance of Microsoft Visual Basic and flexibility and intensity of C++ which enables much faster and easier development of COM++ and web services[1,3,4,14].

On the other hand, one of studies about a translator is about an intermediate language translator[19–22] that generates a native code of the
target machine from bytecode for improving an execution speed of java applications. It, however, cannot be used in other computer environments because of being dependent on a target machine.

Microsoft has developed the JLCA translator[23] which converts a java program into a C# program in a source language level. With the iNET translator[24] 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[25] 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.

On this environment of .NET platform, we can create any code that, once converted to MSIL, can generate an execution file optimized for the target platform regardless of the language used. Therefore, by converting the java bytecode to .NET MSIL, we can run the java programs on the Windows .NET platform even without JVM.

In summary, this paper presents a method to translate java bytecode to .NET MSIL code using a mapping table and the macro translation method, to construct a platform independent information system. Indeed, we designed and implemented a translator using this method, and translate a java bytecode program to MSIL code to run it on the .NET platform without JVM.

2. BYTECODE AND MSIL CODE

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.[2,5,8,15,16].

JVM(Java Virtual Machine) 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, Oolang code, another type of java intermediate language, is much easier to read and write compared to class file format. Oolang code is based on Jasmin language of John Meyer, and designed to work in the level of bytecode[5,13].

Fig. 1 schematically shows the process of extraction of Oolang 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 Oolang decompiler, Gnoloo. Gnoloo carries out the function to extract only the source code-related contents from the various data in the class file.

Fig. 1. Oolang Code Extracting Process

2.2 MSIL Code

MSIL(MicroSoft Intermediate Language) is an intermediate language of .NET language such as C#, comprising a set of stack-based instructions designed to be easily generated from source codes
through a compiler or other tools. The instructions are classified into arithmetic/logical operations, flow control, DMA, exception handling, and method calling. In addition, the virtual method calling, field access, array access, object allocation and initialization, which affect the programming structure, are also types of instructions directly supported by MSIL. The MSIL instruction set searches data types in stacks, and is directly interpreted.

Furthermore, MSIL is independent of hardware and platform, and was originally designed for JIT. Unlike Java, it was designed to be language independent of the first and targeted at generic programming. Consequently, the language adapts very well to the change of program functions and structures [1,3,6,7,9,11,13].

Fig. 2 illustrates the extraction of an MSIL code in reverse order to find out the resulting format of the translator.

3. CODE GENERATION AND CODE CONVERSION METHODS

3.1 Code Generation Method

In general, there are two methods for generation of object codes from an intermediate language: using code expander or code generator.

The code expander method uses the routine to convert an intermediate code to the concrete target machine code. The benefit of this method is that it can generate a target machine code within a short time. However, the quality of the generated code is lower than the code generator method [3,7,17,18].

The code generator method is to generate codes through pattern matching technique referencing the code generation rules described in the target machine code table (MDT: Machine Description Table). The benefit of using this method is that we can get high-quality target machine codes. On the other hand, its disadvantage is that there is a time delay while generating a target machine code through pattern matching techniques [3,7,17,18].

In this paper, we used the code generator to convert codes through a pattern matching technique with the intermediate language instruction code table.

3.2 Code Conversion Method

The code conversion method is to generate codes by using the routine of converting the intermediate code to a concrete target machine code. The code conversion method utilizes the macro conversion. The macro definition format consists of a macro name, parameters, and the macro definition part to be expanded between `{` and `}`.

Table 1. Macro definition format

<table>
<thead>
<tr>
<th>macro's name</th>
<th>definition of the macro</th>
<th>call for another macro</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ </code>parameter`</td>
<td>parameter</td>
<td><code>argument``)</code> }</td>
</tr>
</tbody>
</table>

The following is an example of macro definition of an MSIL code for a bytecode instruction.

<table>
<thead>
<tr>
<th>istore_2</th>
<th>iadd</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iconst_2</code></td>
<td>iadd</td>
</tr>
<tr>
<td><code>ldc.i4 2</code></td>
<td><code>add</code></td>
</tr>
<tr>
<td><code>//push 2 onto the stack as i4</code></td>
<td><code>//add to values, returning a new value</code></td>
</tr>
</tbody>
</table>
4. JAVA BYTECODE-TO-.NET MSIL IL TRANSLATOR

For the translation system from Java bytecode to .NET MSIL code to construct a platform independent information system, we designed and implemented the code translator by using the mapping table and the macro conversion method. We must convert the matching between codes through the mapping table so that the instruction mapping part and the function format structure conversion part, which are the main components of the translator, would become functionally equivalent to the relationship between bytecode and MSIL code. Furthermore, in the actual implementation, we will refer to the table corresponding to each of the codes by using the macro conversion method.

4.1 System Configuration

Fig. 3 shows the system configuration of the translator from Java bytecode, namely Oolang code, to .NET MSIL code. For the process of extracting the Oolang code from class file, with the class file as an input, the Oolang disassembler, Gooloo, was used to output the Oolang code in the text assembly format. Next, with the code converter, we generated an MSIL code from the Oolang code used as the input file. Finally, we created an *.exe file from the MSIL code through ilasm.exe provided by the .NET platform.

4.2 Java Bytecode-to-.NET MSIL Translator

With the extracted Oolang code as an input to the translator, we used the instruction mapping table and the function format structure conversion table to generate MSIL code, which is the resulting code of the translator. As shown in Fig. 4, we obtained an MSIL code from the Oolang code through the extraction process, and the result is generated from the two processes of mapping and conversion.

The development environment for the translator is divided into the JDK and Oolang part to create and disassemble the java class file to be input to the translator, and the .NET platform SDK part to convert the translated MSIL file into a *.exe file.

Fig. 4. Internal System Configuration of Translator

4.2.1 Data Type Mapping Table

This is the table for the basic data type mapping used by the translator for conversion(Table 2). There are some basic data types for Oolang code, which are included in the 17 data types used by MSIL.

Table 2. Data Type Mapping Table

<table>
<thead>
<tr>
<th>Oolang</th>
<th>MSIL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd, fadd, ladd, dadd</td>
<td>add</td>
<td>addition</td>
</tr>
<tr>
<td>imul, fmul, lmul, dmul</td>
<td>mul</td>
<td>multiplication</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>newarray</td>
<td>newarr</td>
<td>creating array</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>unicode string</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Table 2. Continued

<table>
<thead>
<tr>
<th>Type</th>
<th>MSIL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>int8</td>
<td>signed 8-bit integer</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
<td>unsigned 16-bit integer</td>
</tr>
<tr>
<td>int</td>
<td>int32</td>
<td>unsigned 32-bit integer</td>
</tr>
<tr>
<td>long</td>
<td>int64</td>
<td>unsigned 64-bit integer</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

4.2.2 Instruction Mapping Table

Table 3 lists only the basic instructions. The instructions are classified based on the MSIL, to which the corresponding Oolang instructions are mapped.

Table 3. Instruction Mapping Table

<table>
<thead>
<tr>
<th>Oolang</th>
<th>MSIL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload.0</td>
<td>ldarg.0</td>
<td>load the argument number 0 on the stack.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>istore.0</td>
<td>stloc.0</td>
<td>store a value in local variable number 0 on the stack.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>iadd</td>
<td>add</td>
<td>addition</td>
</tr>
<tr>
<td>imul</td>
<td>mul</td>
<td>multiplication</td>
</tr>
<tr>
<td>ifnonnull</td>
<td>btrue</td>
<td>branch if &lt;value&gt; is nonzero.</td>
</tr>
<tr>
<td>putfield</td>
<td>stfld</td>
<td>store into a field of an object.</td>
</tr>
<tr>
<td>getfield</td>
<td>ldfld</td>
<td>load a field of an object.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

5. EXPERIMENTAL RESULTS AND ANALYSIS

The followings are the example of java array program. Program 1 below depicts the extraction of Oolang code from java program source. The translator used the extracted Oolang code as its input, and translated the .NET MSIL code corresponding to each Java Oolang code through the mapping table and macro conversion within the translator, which is exactly the case for program 2.

```java
Public class ArrayTest {
    public static void main()
    {
        int m[][1,2,3];
        for (int i=0; i<3; ++i)
            system.out.println(m[i]);
    }
}
```

Program 1. Java Program, Extracted Oolang Code and Corresponding MSIL Code

```java
.assembly extern mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
    .ver 1-0:3300:0
}
.class public auto ansi beforefieldinit ArrayTest
extends [mscorlib]System.Object
{
    .method public hidebysig specialname
tspecialname instance void .ctor() cil managed
    {
        .maxstack 1
        10: ldarg.0
        11: call instance void [mscorlib]
            System.Object::.ctor()
        14: ret
    }
```
Program 2. MSIL Code Generated by Translator

Fig. 5 shows the result of the execution after translating the Java Oolong program to .NET MSIL program and converting it into executable file. It shows the same results from the execution of ArrayTest.j extracted by the Oolong disassembler from the class file generated by the Java compiler and the execution of MSIL file ArrayTest.exe generated by the translator with ArrayTest.j file as input.

![Fig. 5. Execution Result](image)

Table 4 shows the result of performance evaluation between Java bytecode program in JVM and MSIL code program in .NET platform.

<table>
<thead>
<tr>
<th>Program</th>
<th>Java Bytecode</th>
<th>.NET MSIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>If Statement</td>
<td>335ms</td>
<td>171ms</td>
</tr>
<tr>
<td>While Statement</td>
<td>333ms</td>
<td>170ms</td>
</tr>
<tr>
<td>For Statement</td>
<td>339ms</td>
<td>171ms</td>
</tr>
<tr>
<td>While Statement</td>
<td>335ms</td>
<td>172ms</td>
</tr>
<tr>
<td>Class</td>
<td>338ms</td>
<td>165ms</td>
</tr>
<tr>
<td>Extended Class</td>
<td>363ms</td>
<td>173ms</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>336ms</td>
<td>274ms</td>
</tr>
<tr>
<td>Thread</td>
<td>342ms</td>
<td>165ms</td>
</tr>
</tbody>
</table>

As the table displayed, it takes little time for .NET programs to be executed rather than Java programs, double speed on average. Therefore, Java programs can be executed in .NET platform without JVM with this bytecode-to-MSIL translator, it improves the execution speed of Java applications, and provides an environment for programmers to develop application programs without limitations of programming languages.
6. CONCLUSIONS

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 Java bytecode-to-.NET MSIL translator system for java programs to be executed in the.NET platform without JVM. This work improves the execution speed of programs, enhances the productivity, and provides an environment for programmers to develop application programs without limitations of programming languages.

We are currently researching on the optimization method to generate the better codes to speed up and to be able to convert the entire code format for large programs as well.

7. REFERENCES

Yang Sun 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
2000~Present Director of Korea Multimedia Society
2001~Present Director of SIGPLAN in Korea
Research Areas: Programming Languages, Embedded Systems, Mobile Computing

Seung Won Na
1993 Agricultural Economics, Dankook University (B.S.)
1996 Electronic Information Management, Dankook University (M.S.)
2004 Computer Engineering, Dongguk University (Ph.D.)
1997~Present SK Telecom Platform Researcher
Research Areas: Mobile Computing, Mobile agent, Ubiquitous Computing, Programming Languages

Dae Hoon Hwang
1977 Mathematics, Dongguk University (B.S.)
1983 Computer Engineering, Chungang University (M.S.)
1991 Computer Engineering, Chungang University (Ph.D.)
1987~Present Professor, College of Software, Kyungwon University
2003~Present chief editor of journal of Korea Multimedia Society
Research Areas: XML, VRML and Internet application