A Study on the Java Compiler for the Smart Virtual Machine Platform*

YunSik Son[1] and YangSun Lee[2]

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

Abstract. SVM(Smart Virtual Machine) is the virtual machine solution that supports various programming languages and platforms, and its aims are to support programming languages like ISO/IEC C++, Java and Objective-C and smart phone platforms such as Android and iOS. Various contents that developed by supported language on SVM can be execute on Android and iOS platforms at no additional cost, because the SVM has the platform independent characteristic by using SIL(Smart Intermediate Language) as an intermediate language. In this paper, we will introduce the Java compiler to support the contents written in Java language on SVM which generates platform independently stack-based SIL code as target code.

Keywords: SVM(Smart Virtual Machine), SIL(Smart Intermediate Language), Java Compiler, Compiler Construction.

1 Introduction

The previous development environments for smart phone contents are needed to generate specific target code depending on target devices or platforms, and each platform has its own developing language. Therefore, even if the same contents are to be used, it must be redeveloped depending on the target machine and a compiler for that specific machine is needed, making the contents development process very inefficient. SVM(Smart Virtual Machine) is a virtual machine solution which aims to resolve such problems, and it uses the SIL(Smart Intermediate Language) code which designed by our research team as an input at the execution time[1-4].

In this study, a compiler for use in a program designed in the Java programming language[5] to be used on a SVM is designed and implemented. In order to effectively

* 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).

T.-h. Kim et al. (Eds.): MulGraB/BSBT/IUrC 2012, CCIS 353, pp. 135–140, 2012.

implement the compiler, it was designed to five modules; syntax analysis, class file loader, symbol information collector, semantic analyzer and code generator.

2 Relative Studies

2.1 SVM(Smart Virtual Machine)

The SVM is a platform which is loaded on smart phones. It is a stack based virtual machine solution which can independently download and run application programs. The SVM consists of three main parts; compiler, assembler and virtual machine. It is designed in a hierarchal structure to minimize the burden of the retargeting process.

The SVM is designed to accommodate successive languages, object-oriented languages and etc. through input of SIL as its intermediate language. It has the advantage of accommodating C/C++ and Java, which are the most widely used languages used by developers. SIL was a result of the compilation/translation process and it is changed into the running format SEF(SIL Executable Format) through an assembler. The SVM then runs the program after receiving the SEF[1-4].

2.2 SIL(Smart Intermediate Language)

SIL[6], the virtual machine code for SVM, is designed as a standardized virtual machine code model for ordinary smart phones and embedded systems. SIL is a stack based command set which holds independence as a language, hardware and a platform. In order to accommodate a variety of programming languages, SIL is defined based on the analysis of existing virtual machine codes such as bytecode[7], .NET IL[8] and etc. In addition, it also has the set of arithmetic operations codes to accommodate object-oriented languages and successive languages.

SIL is composed of meta-code(shows class declarations and specific operations) and arithmetic codes (responds to actual commands). Arithmetic codes are not subordinate to any specific hardware or source languages and thus have an abstract form. In order to make debugging of the languages such as the assembly language simple, they apply a name rule with consistency and define the language in mnemonics, for higher readability. In addition, they have short form arithmetic operations for optimization. SIL's arithmetic codes are classified into seven and each category has its own detailed categories.

3 Java to SIL Compiler

In this study, the Java to SIL compiler was designed as can be seen in Fig. 3 it has five parts and 10 detailed modules.

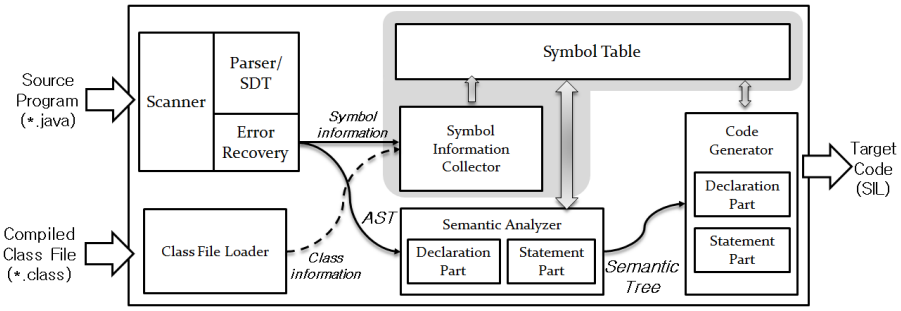

Fig. 1. Java to SIL Compiler Model

The Java to SIL compiler embodies the characteristics of the Java language and therefore was designed with five different parts; syntax analysis, class file loader, symbol information collection module, semantic analysis and code generation. The detailed information for each part is as follows.

The syntax analysis part carries out syntax analysis regarding the given input program(*.java) and converts it into an AST(Abstract Syntax Tree) which holds the equivalent semantics. There are largely three steps in the syntax analysis part; lexical analysis, syntax analysis and error recovery [9-11].

The Class file loader is the module to extract symbol information needed to syntax analysis, semantic analysis and code generation from the pre-compiled class files. The Class file loader extracts class information from the inputted class files(*.class), and stores it in the symbol table through symbol information collector.

The module for symbol information collection consists of symbol information collection routines and a symbol table. First, the symbol information collection routine carries out the job of saving information into the symbol table which is obtained by inserting ASTs and rounding the tree. The routines consist of the interface, protocol, class member, ordinary declarations and others(given the characteristics of the Java language). Next, the symbol table is used to manage the symbols(names) and information on the symbols within a program.

The semantic analysis part is composed of the declarations semantic analysis module and the statements semantic analysis module. The declarations semantic analysis module checks the process of collecting symbol information on the AST level, to verify cases which are grammatically correct but semantically incorrect. Semantic analysis of the declarations part is handled by two parts; semantic error and semantic warning. The statements semantic analysis module uses the AST and symbol table to carry out semantic analysis of statements and creates a semantic tree as a result. A semantic tree is a data structure which has semantic information added to it from an AST[4,10]. It is responsible for all that has not been taken care of during the syntax analysis process and then it is used to generate codes as it has been designed to generate codes easily.

The code generation part receives the semantic tree as an input after all analysis is complete and it generates a SIL code which is semantically equal to the input program (*.java). For this, the SIL code is expressed as symbols so it is convenient to generate and handle them. For type conversion code lists, the same data structure is kept so that the code generation process can take place efficiently. Type conversion code lists are

data structures that pre-calculate the process of converting a semantic code into a SIL code when generating a code. A code generator visits each nodes of the semantic tree to convert them into SIL codes[10].

4 Implementation and Experiments

To implement the Java to SIL compiler, first the language's grammar was chosen and then using this a LALR(1) parsing table was created. The grammar used was based on JDK 6.0 and the information on the grammar parsing table can be seen in Table 1.

Table 1. Java Grammar, Parsing Table, Tree Information

Name	Count
Grammar Rules	380
Terminal Symbols	105
Nonterminal Symbols	152
Parsing Table Kernels	650
AST Nodes	153
Semantic Tree Nodes	236

Next, we show the process of converting the source program's code(written in Java language) into the target code, the SIL code, using the implemented Java to SIL compiler. Table 2 has been created so that the characteristics of the declarations and syntax of the example program can be seen using the Java language.

Table 2. Example Program(TicTacToe.java)

```
public class TicTacToe extends Component
{
...

    public TicTacToe() {
     this.player = 0;
     this.computer = 0;
     ...
    }

    public void paint( Graphics g) {
     g.setColor( getBackground());
     g.fillRect( 0, 0, getSize().width,
                 getSize().height);
     g.setColor( gridColor);
     int fieldSize = this.getFieldSize();
     g.drawLine( 0, fieldSize, 3*fieldSize,
                       fieldSize);
     g.drawLine( 0, fieldSize+1,
                    3*fieldSize, fieldSize+1);
     ...
    }
    ...
}

…
```

Table 3 shows the AST structures generated from the input program. You can see that the syntax have been expressed using the AST nodes defined earlier on. Table 4 shows a part of the SIL code that has been generated using a semantic tree.

Table 3. AST for an Example Program Segment

```
Nonterminal: PROGRAM                                  Nonterminal: DCL_SPEC
…                                                       Nonterminal: INT_TYPE
  Nonterminal: CLASS_DCL                              Nonterminal: VAR_ITEM
    Nonterminal: PUBLIC                                 Nonterminal: SIMPLE_VAR
      Terminal: TicTacToe                                 Terminal: player
    Nonterminal: EXTENDS                              Nonterminal: FIELD_DCL
      Nonterminal: CLASS_INTERFACE_TYPE                   Nonterminal: PRIVATE
        Nonterminal: SIMPLE_NAME                          Nonterminal: DCL_SPEC
          Terminal: Component                               Nonterminal: INT_TYPE
        Nonterminal: CLASS_BODY                           Nonterminal: VAR_ITEM
          Nonterminal: FIELD_DCL                            Nonterminal: SIMPLE_VAR
            Nonterminal: PRIVATE                              Terminal: computer
                                                  …
```

Table 4. Generated SIL Code for Example Program

```
%%HeaderSectionStart
…
%%HeaderSectionEnd
%%CodeSectionStart
   %FunctionStart
      .func_name
         &TicTacToe::TicTacToe$
            0
      .func_type    2
      .param_count    0
      .opcode_start
         proc    16    1    1
         lod.p    1    0
         ldc.p    0
         add.p
         ldc.i    0
         sti.i
         lod.p    1    0
         ldc.p    4
         add.p
         ldc.i    0
         sti.i
%Label    ##0
      lod.i    1    4
      lod.i    1    0
      le.i
      fjp    ##1
      ldc.i    0
      str.i    1          12
      ldc.i    1
      str.i    1          8
%Label    ##3
      lod.i    1    8
      lod.i    1    4
      ldc.i    2
      div.i
      le.i
      fjp    ##4
      lod.i    1    4
      lod.i    1    8
      mod.i
      ldc.i    0
      eq.i
      fjp    ##6
         lod.i    1    12
         lod.i    1    8
         add.i
         str.i    1    12
            add.p
            ldi.p
            …
            .opcode_end
         %FunctionEnd

      …
%%CodeSectionEnd
%%DataSectionStart
…
%%DataSectionEnd
```

5 Conclusions and Further Researches

Virtual machines refer to the technique of using the same application program even if the process or operating system is changed. It is the core technique that can be loaded onto recently booming smart phones, necessary as an independent download solution software technique. In this study, the Java to SIL compiler was designed and virtualized to run a program that was originally created for another platform to enable its use on a SVM. In this paper, we defined five modules to create a compiler and generate a SIL code for use on a SVM which is independent of platforms. As a result, programs developed for use as Java contents could be run on a SVM using the compiler developed throughout the study and therefore expenses required when producing such contents can be minimized.

In the future, there is need for research on an Android Java-SIL compiler so that Android contents can be run on a SVM. Further research on optimizers and assemblers for SIL code programs are also needed so that SIL codes that have been generated can run effectively on SVMs.

References

1. Lee, Y.S.: The Virtual Machine Technology for Embedded Systems. Korea Multimedia Society 6, 36–44 (2002)
2. Oh, S.M., Lee, Y.S., Ko, K.M.: Design and Implementation of the Virtual Machine for Embedded Systems. Journal of Korea Multimedia Society 8(9), 1282–1291 (2005)
3. Lee, Y.S., Oh, S.M., Son, Y.S.: Development of C++ Compiler for Embedded Systems. Industry-Academia Cooperation Foundation of Seokyeong University (2006)
4. Son, Y., Lee, Y.: Design and Implementation of an Objective-C Compiler for the Virtual Machine on Smart Phone. In: Kim, T.-H., Gelogo, Y. (eds.) MulGraB 2011, Part I. CCIS, vol. 262, pp. 52–59. Springer, Heidelberg (2011)
5. The Java Language & Virtual Machine Specifications, Oracle, `http://docs.oracle.com/javase/specs/index.html`
6. Yun, S.L., Nam, D.G., Oh, S.M., Kim, J.S.: Virtual Machine Code for Embedded Systems. In: International Conference on CIMCA, pp. 206–214 (2004)
7. Meyer, J., Downing, T.: JAVA Virtual Machine. O'Reylly (1997)
8. Lindin, S.: Inside Microsoft.NET IL Assembler. Microsoft Press (2002)
9. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. Addison-Wesley (2007)
10. Son, Y.S.: 2-Level Code Generation using Semantic Tree, Master Thesis, Dongguk University (2006)
11. Graham, S.L., Haley, C.B., Joy, W.N.: Practical LR Error Recovery. In: Proceedings of the SIGPLAN Sym. on Compiler Construction, SIGPLAN Notices, vol. 13(8), pp. 168–175 (1979)