# 재설계된 자바 클래스 파일을 위한 가상기계의 설계 및 구현

고 광 만†


## 요 약

가상 기계(Virtual Machine; VM)는 언어에 대한 장치 독립성 및 플랫폼 독립성을 지원하는 프로그래밍 실행 환경이다. 현재까지 자바 언어를 위해 JVM, KVM 등이 다양한 환경에서 사용되고 있으며 유사한 가상 기계가 개발되어 활용되고 있다. 본 논문에서는 자바 클래스 파일(*.class)에서 PDA와 같은 소규모 장치에서 반드시 필요한 요소를 추출하고 실행 효율성을 위해 클래스 파일의 포맷을 재구성한 클래스 파일(*.rclass)을 설계하고 변환기를 개발하였다. 또한 재설계된 클래스 파일을 입력으로 받아 실행 결과를 생성하는 가상기계를 구현하였다.


**키워드 : 자바 클래스 파일, 번역기, 가상기계, 링커&로더, 인터프리터**

# Design and Implementation of the Virtual Machine for the Redesigned Java Class File

Kwang-Man Ko†


## ABSTRACT

The virtual machine is a programming environment that supports device and platform independence. So far, virtual machines such as JVM and KVM have been used in a variety of environments for the Java language. Some virtual machines similar to them are also being developed and used. This paper presents the experiences of extracting elements essential for small sized devices such as PDA from Java Class files(*.class) and designing a converted class file(*.rclass) for runtime efficiency by modifying its class file format and developing its translator. In addition, a virtual machine is developed to receive the translated class file entered and output the runtime results.


**Key Words : Java Class File, Translator, Virtual Machine, Linker&Loader, Interpreter**

## 1. Introduction

The virtual machine is a programming run-time environment that supports device and platform independence. So far, virtual machines such as JVM and KVM have been used in a variety of environments for the Java programming language. Some virtual machines similar to them are also being developed and used. Recently, however, developing virtual machines for embedded systems to run applications written in an advanced language such as Java in small-sized devices other than computer systems are beginning to be developed[8].

Since applications are susceptible to the hardware and operating systems when they are executed, source files should be compiled to create native codes suitable to the machine used. Therefore, any change in hardware and operating systems will result in the development of new compilers, API modification and the rebuilding of the development environment. To address this issue, the virtual machine is used to interface between the system and the application. The Java compiler doesn't create any object code for a specific machine, but creates an intermediate code for the virtual machine and runs it in the JVM.

There are several constraints in putting the developed virtual machine in a platform. Especially when putting it on a small device, you will face several constraints that you wouldn't in Windows or UNIX; the virtual machine should be small in size. Mobile devices manage data in database, not in the file system, as it uses the flash memory. This is the most distinguished difference between small mobile devices and PCs. It is the virtual machine that allows an application to run in different devices without modifying or translating the application. Currently virtual machines are being developed for differ-

※ 이 논문은 한국과학재단의 특정기초연구(과제번호 : R01-2002-000-00041-0) 지원에 의한 것임.
† 종신회원 : 상지대학교 컴퓨터정보공학부 조교수
논문접수 : 2004년 10월 6일, 심사완료 : 2005년 5월 9일

ent systems and increasingly adopted among small devices. However, since small devices have limited system resources, some solutions should be come up with to manage the resources efficiently[3].

The size of Java classfiles has long been recognized as an issue, especially for embedded devices. A number of efforts have been made to reduce the size, both as a transmission format and as an execution format. Paugh[10] developed techniques for compressing classfile components for transmission, and achieved sizes ranging from 17% to 49% of the comparable JAR files. Rayside et al[9], in contrast, focused on execution format, specially reducing constant pool size and code size. Reductions in JAR file size of roughly 50% were obtained through optimizations of the constant pool, and smaller improvements were realized through code optimizations. Clausen et al[11]. developed a compressed bytecode scheme using macros, achieving spaces savings of around 30%, but at a runtime cost of up to 30%. Tip et al[12]. developed techniques for extracting only the components necessary for a particular application form a set of classfiles, with the resulting achieves reduced to 37.5% of their original size.

This paper introduces the experiences of designing and implementing a virtual machine based on JVM, KVM and Waba virtual machine. To do this, first a translator was developed to convert Java class files automatically. Second, a virtual machine for translated class files was designed and implemented. Finally, the developed virtual machine was verified by showing that the virtual machine received the translated class file and output the runtime results.

## 2. Related works

### 2.1 Java Class File

Class files have six main sections: header, constant pool, fields, methods and attributes. The constant pool is similar to traditional compiler symbol tables. The bytecode notion of attribute is interesting : every class, field and method mat have attributes of arbitrary structure. Sun predefines a number of attributes, such as 'Code' that contain the instructions of a method body. Physically, class files are a stream of 8-bit unsigned bytes. All values that require more than 8-bits are represented as a sequence of bytes. The JVM specification uses the notation u1, u2, and u4 to represent one-byte, two-byte and four-byte sizes respectively(the u stands for 'unsigned') [2][9][10].

The Constant pool is a table of variable length structures of type cp_info. cp_info records are used to represent class names, field names, method names, other string constants and numeric constants. The constant_pool_count is the number of cp_info entries in the constant_pool array. Attribute structures are used extensively throughout class files to describe classes, fields, methods and even other attributes. The JVM specification defines a set of standard attributes, but compiler writers are free to create new attribute types. The methods array contains all the methods(both instance and static) declared in this class or interface. Each entry is a variable length structure of the type method_info that contains a complete description of the JVM code for that particular method. The methods_count is the number of methods declared in this class. It should be noted that a variable number of attribute_info structures can be included in this structure. The code attribute is only used in the attributes table in the method_info structure. It contains all the JVM opcodes for a single Java method. Every method may contain at most one code attribute. A JVM instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands. Each opcode is represented by one byte, and therefore the JVM can support up to 256 different opcodes, although only 204 are defined. The opcodes can be classified into distinct groups according to the type of operands they accept. Opcodes for method invocation take an index to a Methodref record; opcodes for object creation take an index to a Classref record in the constant pool. Another opcode group that is related to transferring control of execution accepts a 2-byte operand that is used an an offset to the code array. There are only two opcodes with a variable number of operands: tableswitch and lookupswitch, which are used for compiling switch statements.

### 2.2 Waba VM

Waba VM[13] is a Java virtual machine runtime environment developed for small devices using Palm and Window CE as their OS. It follows Java grammar rules and operates on the information of the Java class files, but it uses new API, as it is not compatible with the API for the existing JVM.

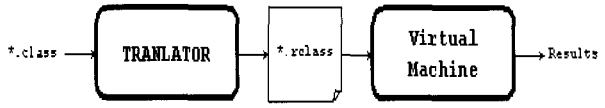
Waba uses 202 bytecodes defined according to the JVM specification, but no bytecodes are defined for long and double variables, thread and exception handling related commands. The class file format follows the definition by the JVM specification, but the some information is omitted. When the virtual machine begins to operate, it allocates the stack and heap memory area, goes through the initiation, loads class files and gains the runtime in-

formation to run bytecodes. The runtime information generated while the loading process is stored. The information referred while running has a structure to store constant pool, fields and methods. The first step of the class loader is to check whether the class to load is loaded in memory. If not, check whether the class is a system class. If it is a system file, modify the path to Waba API class path to prevent the existing Java class file from loading. Use nativeClassLoad(), which is used to obtain the class file from actual file, to load the class file to memory to get the start point. And then use each offset information of the class to parse information on Constans pool, this class, super class, field, method. In the last step of loading, go through ClassHook() binding process to call native method. If the file to load is a static class, run cinit() method to initialize the static field value of the class. At this time, information parsed while loading is stored in WClass. These loaded classes are in hash structure of WClass and referred when the interpreter is run.

## 3. Virtual Machine for the Redesigned Java Class File

### 3.1 Overview

The total system is composed of the translator to convert the Java class file(*.class) to redesigned class file(*.rclass) and the virtual machine that receives the redesigned class file as a input and output runtime results as shown in (Fig. 1).




(Fig. 1) Overview of the Virtual Machine for the Redesigned Java Class File

### 3.2 Redesigned Class file format and Translator

In the redesigned class file, the followings are redesigned for runtime efficiency by modifying its class file format and developing its translator as shown in (Fig. 2).

- Insert source file index in the header part
- code area reallocation
  - move bytecode information in the code attribute section to code area
- 1 byte tag insert

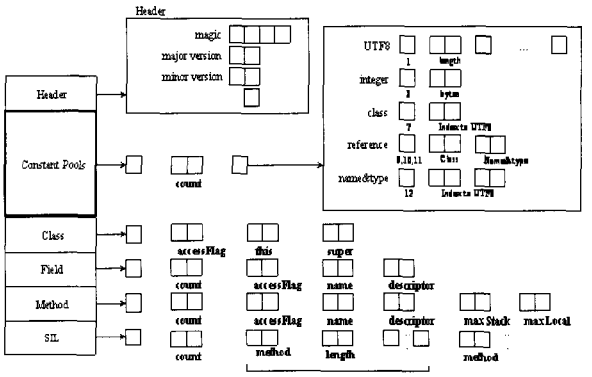
And the followings are removed from the original Java class file format(*.class).

- Remove interface part
  - not supports the interface
- attribute information remove in the field and method area
  - move attribute information in the field and method area to code area.

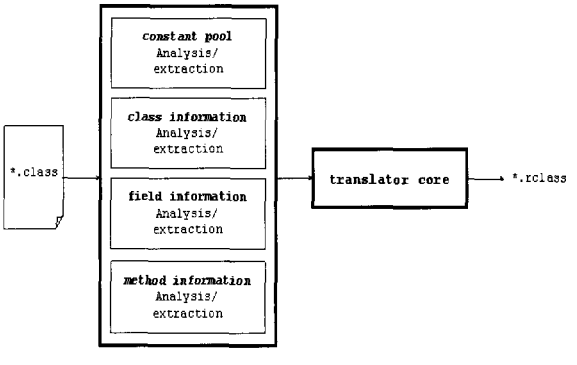
In the redesigned parts, the method area is separated to the code part of bytecode area. This improves the handling efficiency when linking/loading class files in a virtual machine by separating bytecodes from information and meta information in each area of class. Collecting and managing the distributed bytecodes in property information of method areas provides easy access to each bytecode.

The translator reads and analyzes binary information in class files through a process shown (Fig. 3) and produces class information according to the new *.rclass format.

The translator receives *.class files as input, extract relevant information in each area, store it in the array or structure format and then rearrange the information according to *.rclass format. The information added to the new format is one obtained from analyzing the extracted information and when the file is rearranged, it is recorded in the binary file. When the method area is processed, the bytecode is recorded in the end part of the format.




(Fig. 2) Redesigned Class File Structure




(Fig. 3) Java Class File Translator(from *.class to *.rclass)

### 3.3 Virtual Machine Implementation

The virtual machine is implemented through a process as following(core routine). The virtual machine is largely divided into definition, initialization and runtime parts. The virtual machine is developed divided into one core file and two porting-related sources. The program source starts from the main() function and receives class file name as function parameter when it runs.

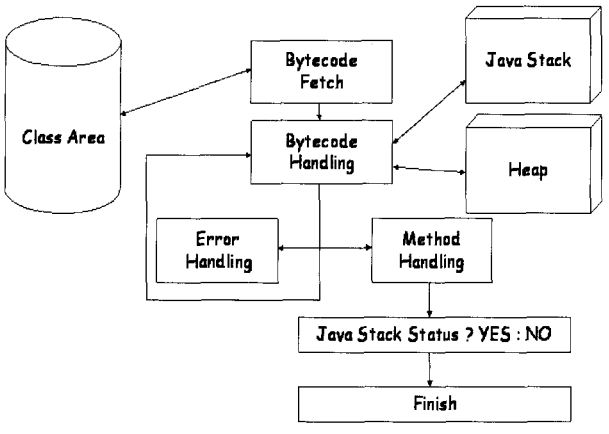
***Type definition :*** Defines the data structure to use for storing class file information and implement the virtual machine and mnemonic name for each bytecode to refer to when implement the interpreter.

***Class access :*** When the class file is loaded, the binary information stored in the file is also loaded onto memory. These are macro-defined functions to obtain information on the class file. These functions are helpful when using the pointer values of the file image to get specific values of each area. These are used most when loading/linking and by interpreter.

***VM initialization :*** The virtual machine should go through a process of initializing its components. Class (method) area, Java stack, heap memory area and global variable are initialized and the values of elements required for performance are set.

***getClass() Loader :*** It analyze the class information from the class file and stores it in the class area, while working as a loader that loads classes stored in forms of files. As the virtual machine adopted the dynamic loading, any class required for operation is loaded at any time.

*Execute-Interpreter:* The core of the virtual machine which is a function handling methods. It gets method information from the class area and handles bytecodes contained in methods. All data generated during the interpreter operation is stored and managed in the Java stack and heap memo.




(Fig. 4) Interpreter design: execution procedure

The interpreter executes the bytecodes of the class files. The interpreter stores and maintains the class information that is saved by the loader and refer to the information when it is executed. Data generated during the process is stored and managed in the Java stack and heap. When an error occurs, the interpreter is supposed to display an error message and exit. Since the methods terminate in the opposite order to the calls, if there is no frame information in Java stack, the interpreter regards it as a termination status and stops the operation. (Fig. 4) is a detailed design for interpreter implementation.

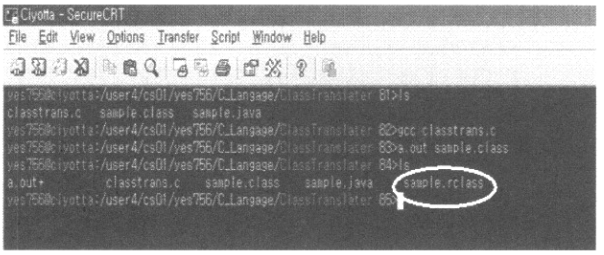
## 4. Experiments

To test the implemented small virtual machine, 'Sample.java' source was written and its results were verified. In addition, a print related method was defined in APIClass.java to call a native function supporting the output and a relevant native function was implemented in the virtual machine to check the output during execution. A sentence to output a multiplication table in the main() and a method to test a repetitive sentence are inserted in Sample.java source. (Fig. 5) represents Sample.java and APIClass.java sources, respectively.

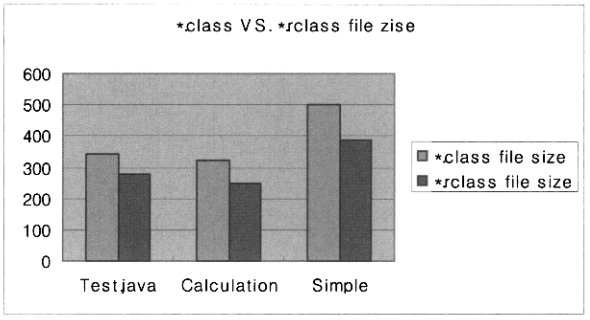
```
// Sample.java
public class Sample {
  public static void main(String[] args) {
    for (int i=2; i<=9; i++) {
      for (int j=1; j<=9; j++) {
        APIClass.print(i); APIClass.prints(" * "); APIClass.print(j);
        APIClass.prints(" = "); APIClass.print(i*j); APIClass.printsln("");
      }
    }
    hello(7);
    }
    public static void hello(int value) {
      for (int i=2; i<value; i++) {
        APIClass.prints("print : "); APIClass.println(i);
      }
    }
}
// APIClass.java
public class APIClass {
  public static native void print(int value);  public static native void
  println(int value);
  public static native void printsln(String value); public static native void
  prints(String value);
}
```

(Fig. 5) Sample.java and APIClass.java sources

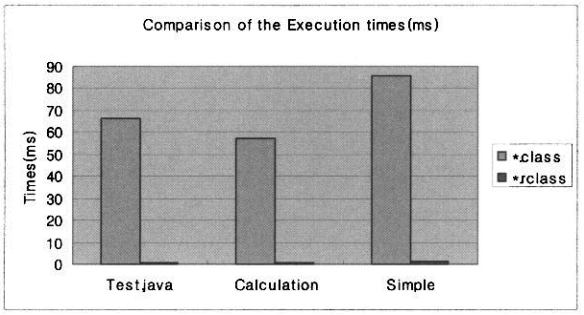
(Fig. 6) shows the results of executing the virtual machines including debugging information and results of each stage. When each class file is loaded, such information as file image output information, initialization process and method call information is printed.




(Fig. 6) *.rclass generation




(Fig. 7) Comparison of *.class and *.rclass file sizes(byte)




(Fig. 8) Comparison of the execution times(ms) of *.class and *.rclass file

(Fig. 7) shows a comparison of the size of applications(Test.java, Calculation.java, Simple.java).

The sizes are all smaller for the *.rclass format. Because of the the essential elements were extracted to reduce the size the class file during the conversion.

Fig. 8 shows a comparison of the execution times of applications(Test.java, Calculation.java, Simple.java).

The execution times of the virtual machine for the three benchmark programs(*.rclass) is measured over more than range from 52% to 83% of the *.class files. Because of the extracted essential elements are executed, and Constants pools informations in the redesigned class file are compacted.

## 5. Conclusion

Since applications are susceptible to the hardware and operating systems when they are executed, source files should be compiled to create native codes suitable to the machine used. Therefore, any change in hardware and operating systems will result in the development of new compilers, API modification and the rebuilding of the development environment. To address this issue, the virtual machine is used to interface between the system and the application. The Java compiler doesn't create any object code for a specific machine, but creates an intermediate code for the virtual machine and runs it in the JVM.

This paper introduces the experiences of designing and implementing a virtual machine, which runs in small devices such as mobile devices to address Java applications, based on JVM, KVM and Waba virtual machine. To do this, first a translator was developed to convert Java class files automatically. Only the essential elements were extracted to reduce the size the class file during the conversion and the file's internal structure was transformed for better runtime efficiency. Second, a virtual machine for translated class files was designed and implemented. All components including core components such as loader/linker, internal information structure and interpreter were newly designed and implemented. Finally, the developed virtual machine was verified by showing that the virtual machine received the translated class file and output the runtime results.

## References

[1] Jon Meyer & Troy Downing , "Java Virtual Machine", March, 1997.

[2] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification 2nd edition", Addision-Wesley, 1999.

[3] Bill Blunden, "Virtual Machine Design and Implementation in C/C++", Wordware Publishing, Inc., 2002.

[4] Sun Microsystems, "The K Virtual Machine(KVM) White Paper Technical Report", Sun Microsystems, 1999.

[5] Rainer Leupers and Peter Marwedel, "Retargetable Compiler Technology for Embedded System: Tools and Applications", Kluwer Academic Publishers, 2001.

[6] John R. Levine, "Linkers and Loaders", Morgan Kaufmann Publishers, 2000.

[7] Nik Shaylor, Douglas N. Simon, William R. Bush, "A Java Virtual Machine Architecture for Very Small Devices", In the Proceddings of ACM SIGPLAN Conferences on Languages, Compilers, and Tools Embedded Systems 2003(LCTES '03), ACM Press, PP. 34-41, 2003.

[8] John Whaley, "Joeq: A Virtual Machine and Compiler Infrastructure", In the Proceddings of ACM SIGPLAN

Conferences on Interpreters, Virtual Machines and Emulators 2003(IVME '03), ACM Press, pp.58-66, 2003.

[9] Derek Rayside, Evan Mamas, Erik Hons, "Compact Java Binaries for Embedded System", Proceddings of the 9th NRC/IBM centre for Advanced Studies Confercence (CASCON '99), pp.1-14, 1999.

[10] W. Paugh, "Compressing Java class files", In the Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation(PLDI) '99, pp.247-258, May, 1999.

[11] Clausen, L.R., Schultz, U.P., Consel, C., Muller, G., "Java Bytecode Compression for Low-End Embedded Systems", ACM TOPLAS, Vol.22, No.3, pp.471-489, May, 2000.

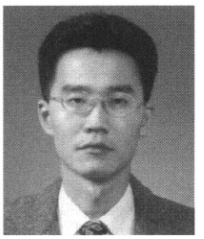
[12] Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D., "Practical Extraction Techniques for Java", ACM TOPLAS, Vol.24, No.5, pp.625-666, Nov., 2002.

[13] "Jike Research Virtual Machine", http://www.ibm.com/developerworks/oss/jikesrvm

[14] "Waba Programming Platform", http://www.wabasoft.com

[15] "Joeq Virtual Machine", http://sourceforge.net/projects/jeoq http://www.stanford.edu/~jwhaley

## 고 광 만



e-mail : kkman@sanji.ac.kr

1991년 원광대학교 컴퓨터공학과(공학사)

1993년 동국대학교 컴퓨터공학과(공학석사)

1998년 동국대학교 컴퓨터공학과(공학박사)

1998년~2001년 광주여자대학교 컴퓨터과학과 전임강사

2002년~2003년 Queensland University of Technology 연구교수

2001년~현재 상지대학교 컴퓨터정보공학부 조교수

관심분야 : 프로그래밍 언어 설계 및 구현