

在翻譯爪哇中介碼成 X86 組語環境下的動態呼叫實作

學生：陳俊元

指導教授：楊武 博士

國立交通大學資訊科學研究所

摘要

為了達到跨平台的特性，Java 程式會被編譯成一種以堆疊操作為導向，與硬體無關的中間型式，這種中間型式稱為 Bytecode，一般執行 Bytecode 的工具是直譯器。Java 程式語言雖然能符合在網路環境下發展程式的特性，但用直譯器執行 Bytecode 的過程十分緩慢，其速度無法和傳統語言如 C 和 C++ 相比。為了改善效能問題，因而衍生出幾種改進方式。如：JIT 編譯器、Native Compiler、Java chip。

本篇論文是屬於 Native Compiler 範疇，提出了一種將 Bytecode 轉成 X86 組合語言的方法，研發出來的翻譯器可以將類別檔轉成 X86 的組合語言，我們將詳細介紹 Java 的執行環境與 Native Compiler 的理論，並說明翻譯器如何把 JVM 的 stack machine 對應至 register machine，及如何翻譯各種類型的 JVM 指令（其中包括複雜的動態呼叫），及如何做一些基本的最佳化。最後會將此方法與 SUN 提出的 Interpreter 及 JIT 編譯器做效能比較，然後討論影響執行效能的原因。根據實驗結果，翻譯出來的組合語言有不錯的執行效能。

Dynamic Dispatch Implementation

Based On Translating Java Bytecode To X86 Assembly Environment

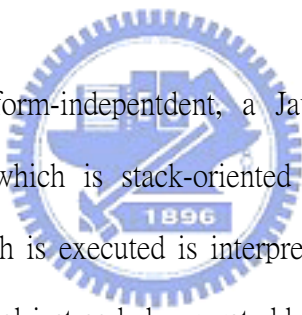
Student: Chun-Yuan Chen

Advisor: Dr. Wu Yang

Institute of Computer and Information Science

National Chiao Tung University

Abstract

The logo of National Chiao Tung University is a circular emblem with a gear-like border. Inside the circle, there is a stylized building and a banner with the year '1896'.

For the purpose of platform-independent, a Java program is compiled to an intermediate form, bytecode, which is stack-oriented and hardware-independent. The common tool of bytecode which is executed is interpreter. The performance of such an interpreter is less comparable to object coded generated by programming languages such as C and C++. To improve the performance, lead to some improvement types. For example, JIT compiler, Native Compiler, Java chip.

This paper belong to a category of Native Compiler. A method of translating bytecode to X86 assembly code is proposed. Translator which is researched can translate class file to X86 assembly code. We will introduce the execution environment and Native Compiler theory, and explain those translations of every type of bytecode by translator (including dynamic dispatch), and do some basic optimization. Finally, we will compare the performance of our translator with SUN interpreter and JIT, then discuss the factors that affect performance. Experimental results show that generated assembly code has good performance.

Acknowledgments

First of all, I deeply appreciate to my advisor, Dr. Wu Yang, for all that he has done. Without his careful guidance and suggestion, it is impossible to accomplish this thesis.

Then, I would like to thank all the members in the PLAS laboratory for their encouragement. Their friendliness made me feel not lonely in the past two years.

Furthermore, I am grateful to my girl friend, Ching-Wen Chang. When I am discouraged by setbacks, she always spirits me up with her love and tolerance.

Finally, I would dedicate this thesis to my dear family, especially my parent. Without their support in spirit, I would not be able to get my M.S. degree. I want to offer my heartfelt thanks to them.



Contents

Abstract (in Chinese)	i
Abstract (in English)	ii
Acknowledgements	iii
Contents	iv
List of Figures	vi
List of Tables	viii
1 Introduction	1
1.1 The Java Programming Language	1
1.2 Research Motivation	2
1.3 Goals	3
1.4 Thesis Organization	4
2 Variations of Java Bytecode Execution	5
2.1 Just-In-Time Code Generators	5
2.2 Native Compilers	7
2.3 Bytecode Annotator	8
2.4 Bytecode Optimization	8
3 System Design and Implementation	9
3.1 System Architecture	9
3.1.1 Execution Environment	10
3.1.2 Translation Environment and Flow	12
3.1.3 Map Java Stack Machine to X86 Register Machine	18
3.2 System Implementation	21
3.2.1 Start JVM From Assembly Code	22
3.2.2 Call Java Bytecode From Assembly Code	23
3.2.3 General Bytecode Translation	27
3.2.3.1 Stack and Local Variables	27
3.2.3.2 Array	31
3.2.3.3 Arithmetic 、 Logic 、 Type Conversion	39
3.2.3.4 Flow Control	44
3.2.4 Special Bytecode Translation	51
3.2.4.1 Object Association	51
3.2.4.2 Method Invocation	63
3.2.5 Template Match	79
4 Performance Evaluation and Analysis	80
4.1 Performance Evaluation and Comparison	80

4.2	Benchmarks Analysis	82
5	Conclusion and Future Work	85
5.1	Conclusion	85
5.2	Future Work	85
	References	86
	Appendix	89



List of Figures

1.1 Compiling and Running Java programs	1
2.1 Data structure of Intel VM	5
2.2 Fast, effective code generation in JIT compiler	6
2.3 Structure of the optimizing compiler	6
2.4 Impact NET compiler translation path	7
3.1 The three compiling techniques of JBCC	10
3.2 Execution environment	11
3.3 Bcc32 assemble and link assembly codes with JVM	11
3.4 The simple translation environment	12
3.5 Translation flow	13
3.6 Translator's data structure	14
3.7 Summary predefined assembly codes	15
3.8 The diagram of the internal data structure	16
3.9 Internal tables of the translator	17
3.10 Initial asm codes of three type methods	19
3.11 Flow of translator translating	21
3.12 The assembly code calls the startvm.c function	23
3.13 The sketch of the assembly code call the bytecode	23
3.14 The JNIEnv interface pointer	24
3.15 Example of call Java bytecode from assembly code	26
3.16 Example of stack and local variable associated bytecodes translating	30
3.17 Example of the primitive array	33
3.18 The constructing order of multianewarray implementation	34
3.19 Example of the three dimension array	38
3.20 Example of flow control	46
3.21 Example of lookupswitch	47
3.22 Example of lcmp	49
3.23 Example of dcmpl	51
3.24 Example of the assembly code of the new instruction	53
3.25 The total flow of the translator creates an object	54
3.26 Example of assembly code of getfield and putfield instructions	56
3.27 Example of assembly code of getstatic and putstatic instructions	60
3.28 Example of the instanceof instruction	61

3.29 The flow of dynamic dispatching for user methods67
3.30 The translating flow of invoking system API67



List of Tables

3.1 The JNI function relationship of bytecodes associated array	31
3.2 The total JNI functions associated with object	62
3.3 The JNI functions of method invocation	68
4.1 Loop test	80
4.2 Arithmetic tests	80
4.3 Array test	81
4.4 Field test	81
4.5 User method tests	82
4.6 Excellent examples tests	82

