EFFICIENT JAVA INTERFACE INVOCATION USING IZONE

by

Xiaojing Wu
B.Sc., Fudan University, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Xiaojing Wu 2004
SIMON FRASER UNIVERSITY
June 2004

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Xiaojing Wu

Degree: Master of Science

Title of thesis: Efficient Java Interface Invocation Using IZone

Examining Committee: Dr. Fred Popowich
Chair

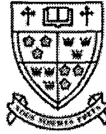
Dr. Robert Cameron, Senior Supervisor

Dr. Uwe Glässer, Supervisor

Dr. Lou Hafer, SFU Examiner

Date Approved: June 29, 2004

SIMON FRASER UNIVERSITY



Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

This thesis addresses the problem of improving the efficiency of interface invocation in the Java Virtual Machine (JVM). In current JVM implementations, interface method invocation is not so efficient as virtual method invocation, because of the need to support multiple interface inheritance in Java. This leads to the mistaken impression that Java interface invocation is inherently inefficient. This thesis will show that, with proper implementation, the performance of interface invocation can be substantially improved.

A new approach – IZone based interface invocation – is proposed in this thesis. IZone is a new data structure associated with an interface type in the method area. It is composed of several implementation lookup areas, one for each subclass of the interface. IZone is populated as subclasses are loaded and resolved, ensuring that the lookup areas within it are arranged in the resolution order of the corresponding subclasses. A fully constructed IZone contains pointers to different subclasses' implementation of the interface methods. Within a lookup area, the pointers are arranged according to the corresponding methods' declaration order by the interface. By taking advantage of class resolution orders and method declaration orders, IZone provides a quick access to the implementation of interface methods. As the experimental results demonstrate, with moderate space overhead, IZone-based interface invocation is the fastest approach after lightweight optimizations, and the second fastest after heavyweight optimizations.

To my parents!

Acknowledgments

I would like to express my sincere gratitude and appreciation to all those who gave me the possibility to complete this thesis.

I am deeply indebted to my senior supervisor Dr. Robert Cameron. He has inspired my efforts through his own sincere interest in the areas of software engineering. His stimulating suggestions, encouragement and support helped me in all the time of research for and writing of this thesis. I learned many things from him, and it was a great pleasure for me to conduct this thesis under his supervision.

I want to thank my supervisor Dr. Uwe Glässer who encouraged me to go ahead with my thesis and gave me constructive comments during my defense time as well as on the preliminary versions of this thesis.

I am obliged to the examiner Dr. Lou Hafer who read my thesis so carefully and gave me so much valuable advice on approach evaluation and test case refinement. The chair Dr. Fred Popowich was of great help in my thesis defense. The defense would not have gone so smoothly without his coordination.

Especially, I would like to give my special thanks to my husband Leo whose patient love and unrelenting support enabled me to complete this work. He has made this time in my life a truly wonderful experience.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgments v

Contents vi

List of Tables ix

List of Figures x

List of Programs xi

1 Introduction 1

2 Java Virtual Machine 3

2.1 JVM Instruction Set 4

2.2 JVM Runtime Data Areas 6

2.3 Method Invocation and Method Table 7

3 Problem Definition 11

3.1 Method Invocation in Single Class Inheritance 12

3.2 Method Invocation in Multiple Interface Inheritance 15

4 Previous Approaches 19
4.1 Naïve Implementation . . . 19
4.2 Interface Table . . . 19
4.2.1 Searched ITable . . . 20
4.2.2 Directly Indexed ITable . . . 21
4.3 Selector Indexed Table . . . 23
4.3.1 Sparse Interface Virtual Table in SableVM . . . 24
4.3.2 Interface Method Table in Jikes RVM . . . 25
4.4 C++ Solution for Multiple Inheritance . . . 28

5 Proposed Approach 29
5.1 Interface Zone . . . 29
5.2 Class Resolution Order . . . 31
5.3 Method Declaration Order . . . 32
5.4 Dispatch Mechanism Based on IZone . . . 33
5.5 Pseudocode . . . 35
5.5.1 IZone Construction . . . 35
5.5.2 IZone Updating . . . 37
5.5.3 Interface Invocation . . . 37

6 Approach Evaluation 39
6.1 Jikes RVM . . . 39
6.1.1 Object Model . . . 39
6.1.2 Compiler . . . 41
6.2 Approaches for Comparison . . . 42
6.3 Real World Test Case . . . 42
6.4 Artificial Test Cases . . . 44
6.5 Time Cost with Baseline Compiler . . . 46
6.6 Time Cost with Optimizing Compiler . . . 50
6.6.1 Lightweight and Heavyweight Optimizations . . . 50
6.6.2 Experimental Results with Lightweight Optimizations . . . 53
6.6.3 Experimental Results with Heavyweight Optimizations . . . 55
6.7 Space Cost . . . 57

7 Future Work 60

8 Conclusion 61

A Modifications to Jikes RVM 62
A.1 com.ibm.JikesRVM.VM_Configuration 62
A.2 com.ibm.JikesRVM.VM_TIBLayoutConstants 63
A.3 com.ibm.JikesRVM.VM_Entrypoints 65
A.4 com.ibm.JikesRVM.classloader.VM_Class 66
A.5 com.ibm.JikesRVM.classloader.VM_Method 68
A.6 com.ibm.JikesRVM.classloader.VM_InterfaceInvocation 69
A.7 com.ibm.JikesRVM.VM_Compiler (IA32 version) 76
A.8 com.ibm.JikesRVM.opt.OPT_ConvertToLowlevelIR 78

B XML Files 80

C Artificial Test Cases 86
C.1 Interface Invocation: Test Case 1 86
C.2 Interface Invocation: Test Case 2 93
C.3 Interface Invocation: Test Cases 3 $\sim$ 6 100
C.4 Virtual Invocation: Test Case v-1 101
C.5 Virtual Invocation: Test Case v-2 108

Bibliography 117

List of Tables

6.1 Parse results of *DOMCount* 43
6.2 Parse results of *SAXCount* 44
6.3 Artificial test cases 45
6.4 Average execution time with baseline compiler 48
6.5 Average execution time after lightweight optimizations 54
6.6 Stable execution time after heavyweight optimizations 56
6.7 Space overhead comparison 59

List of Figures

2.1 How a Java program is executed 4
2.2 JVM runtime data areas 8
2.3 *invokeinterface* instruction 9
2.4 *invokevirtual* / *invokespecial* / *invokestatic* instructions 10

3.1 Method table of a non-abstract class that implements no interface 14
3.2 Method table of a non-abstract class that implements one or more interfaces 15
3.3 Method tables of classes ***A*** and ***B*** in Program 3.2 17

4.1 Searched itable 20
4.2 Directly indexed itable 22
4.3 Selector indexed table 24
4.4 IMT dictionary with 5 entries 26
4.5 IMT table with 5 entries 26

5.1 Interface zone 30
5.2 How to find the correct IZone given an interface call 31
5.3 Java class file format 33

6.1 Object model of Jikes RVM 40
6.2 Execution time of 5,000-iteration inner loop with baseline compiler 47
6.3 Execution time of inner loop in test 1 with optimizing compiler 51
6.4 Execution time of 5,000-iteration inner loop with optimizing compiler 52
6.5 Execution time of 5,000,000-iteration inner loop with optimizing compiler . . 53

List of Programs

3.1 Example program for virtual method invocation 13

3.2 Example program for interface method invocation 16

4.1 A conflict resolution stub for four interface methods 27

5.1 Program for IZone construction 36

5.2 Program for IZone updating 37

5.3 Program for interface invocation 38

6.1 The nested loop in the test cases 45

Chapter 1

Introduction

Over the last several years, Java [21] has become one of the most popular object-oriented programming languages. Java programs are compiled into class files before execution. On a specific platform, class files are loaded and linked by a runtime system called the *Java Virtual Machine (JVM)* [23], and then translated into *bytecode* instructions for execution. As the main source for Java platform independence, the JVM is the cornerstone of Java technology. Platform independence does not come for free; it is achieved at the cost of slow execution speed. As generally understood, Java programs are not so fast as programs compiled to native machine code from languages like C++.

Multiple inheritance is a controversial issue in object-oriented programming languages. In C++, a class may have multiple direct superclasses. This inheritance mechanism adds both complexity and ambiguity to the languages. Java defines an alternative: a class may have multiple direct superinterfaces, but only one direct superclass. Early experience with the Java language has encouraged the assumption that Java interface invocation is not so efficient as virtual method invocation. This is not necessarily correct. Although the implementation of interface dispatch is not efficient in the current Java Virtual Machines, it is not inherently inefficient. The thesis demonstrates a new technique for efficient implementation of interface invocation.

The remaining part of this thesis includes 7 chapters.

Chapter 2 reviews background information on the Java Virtual Machine. It describes how a typical JVM works, the data types it supports, its instruction set and runtime data areas.

Chapter 3 compares multiple interface inheritance and single class inheritance, and analyzes the efficiency problems of interface invocation. By examining the dispatch mechanism of virtual and interface invocation, the main source for interface invocation inefficiency is revealed: variable interface method offsets in method tables.

Chapter 4 describes the previous approaches for Java interface invocation. They are classified into three categories: naïve implementation, itable, and selector-indexed table. The itable category is further broken down into searched itable and directly indexed itable. For selector-indexed table category, sparse interface virtual table of SableVM and interface method table (IMT) of Jikes RVM are reviewed. This chapter also describes the C++ solution for multiple inheritance.

Chapter 5 proposes a new approach – IZone based interface invocation. An IZone is a data structure associated with an interface. It is built up during the resolution of the interface's subclasses. The new approach suggests that the JVM records class resolution orders and method declaration orders, and updates IZone entries to contain the appropriate virtual method pointers based on the order information. The pseudocode for IZone construction, IZone updating, and IZone-based interface invocation is given at the end of the chapter.

Chapter 6 evaluates the proposed approach by comparing it with 5 previous approaches. The experimental results show that the IZone approach is efficient in terms of both interface invocation speed and space overhead.

Chapter 7 proposes other possible ways in which interface invocation could be improved. Chapter 8 concludes that the misimpression about interface invocation is not necessarily correct.

Chapter 2

Java Virtual Machine

The Java programming language is a general-purpose, concurrent, class-based, object-oriented language with three unique features: *platform independence, security, and network mobility* [26]. The same Java code can be transferred across a network and run on all the computers and devices. The built-in security framework of Java helps to build trust in a distributed system. Because of its platform independence, Java is intended to be a language for networked computing environments.

At the heart of Java's network-orientation is the *Java Virtual Machine (JVM)*, which makes the three features of Java possible. The JVM is an *abstract* computing machine. It does not assume any particular implementation technology, host hardware, or host operating system. The JVM specification [23] defines the features that should be possessed by every Java Virtual Machine, but gives designers the freedom to choose the method of implementation, or, to add new features not addressed by the specification. Like a real computing machine, the JVM has an instruction set – *bytecode instructions* – and manipulates various memory areas at run time, which are called *runtime data areas* [23, 26].

The main responsibility of a Java Virtual Machine is to load class files and execute the *bytecode instructions* [23] they contain. The JVM executes a bytecode instruction by interpreting it into native codes that are executable on the system upon which JVM is running. Bytecode instructions are platform-independent, but the native codes are not. By supplying different versions of JVM for different platforms, the platform dependence of native code is hidden behind the JVM, and transparent to Java applications. The Java Virtual Machine promises that Java programs are "compile once, run everywhere".

Figure 2.1 shows the procedure in which Java programs are executed on a specific platform. Java source files (*.java) are compiled, by a Java compiler, into binary files (*.class), which include type information and bytecode instructions. On a specific platform, the Java Virtual Machine loads and links class files, and then executes bytecode instructions.

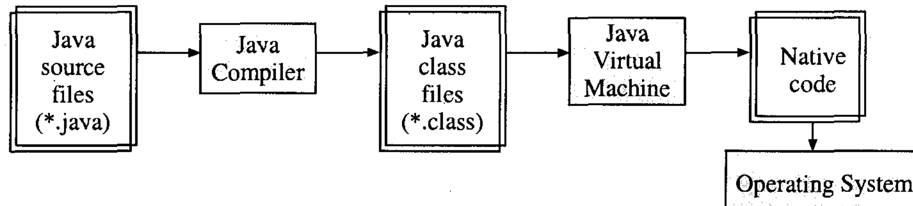


Figure 2.1: How a Java program is executed

Like the Java programming language, the Java Virtual Machine operates on two kinds of data types [23]:

- Primitive types: including byte, short, int, long, char, float, double, boolean, and returnAddress[1].
- Reference types: including class, interface and array.

The Java Virtual Machine contains explicit support for *objects* [21]. An object is either a dynamically allocated class instance or an array. An object of class or interface type is called *scalar object*, and an object of array type is called *array object*. The value of a reference type is a reference (pointer) to an object, i.e., a dynamically created class instance, an array, or a class instance or an array that implements the interface, respectively. A reference value may also be the special *null* reference.

2.1 JVM Instruction Set

The instructions of the Java Virtual Machine are called *bytecode instructions*. A bytecode instruction consists of a one-byte *opcode* and zero or more *operands*. The opcode specifies

[1]The value of a returnAddress type is a pointer to the opcode of a Java Virtual Machine instruction. returnAddress is the only JVM primitive type that is not directly associated with a Java programming language type.

the operation to be performed, and the operands supply arguments or data that are used by the operation. The operands may be generated at compile time and embedded within the instructions, as well as calculated at run time and supplied on the operand stack[2]. Many instructions have no operands, consisting only of an opcode. Limiting the opcode to a byte makes the code compact, while at the same time, limits the size of the instruction set.

Most of the opcodes encode type information about the operands on which they operate. For instance, the *iload* instruction loads the contents of an *int* variable onto the operand stack. The *fload* instruction does the same with a *float* value. These two instructions may be implemented in an identical way, but have distinct opcodes.

Bytecode instructions can be divided into the following categories [23]:

- Load and store instructions: to transfer values between the local variables and the operand stack.
- Arithmetic instructions: to compute a result that is typically a function of two values on the operand stack, and, push the result back on the operand stack.
- Type conversion instructions: to convert a value between Java Virtual Machine numeric types (byte, short, int, long, char, float, and, double).
- Object creation and manipulation instructions: to create and manipulate class instances and arrays, such as, to access instance or static fields of classes, to load array components onto the operand stack, etc.
- Operand stack management instructions: to manipulate the operand stack directly, such as popping or duplicating top word on the stack.
- Control transfer instructions: to cause conditional or unconditional execution of Java Virtual Machine instructions.
- Method invocation and return instructions: to invoke class / instance / interface methods and to return.
- Exception and finally instructions: to throw an exception, and, to implement the *finally* keyword.

[2]See section 2.2 on page 6.

- Synchronization instructions: to support synchronization of methods, or, sequences of instructions within a method.

2.2 JVM Runtime Data Areas

When the Java Virtual Machine executes a program, it needs to store a number of things in the memory: objects, method parameters, return values, local variables, intermediate results, and so on. A typical Java Virtual Machine organizes the memory it needs into several runtime data areas. Some of them are used by individual threads and the others are shared among all threads. Per-thread data areas are created when a thread is created and destroyed when the thread exits. Shared data areas are created when the JVM starts and destroyed when it exits. Five types of runtime data areas are typically defined inside the Java Virtual Machine:

- Method area: this area is shared among all threads; it contains the class data for all the types loaded by the JVM. The class data of a type includes type information[3], constant pool[4], field information[5], method data[6], all non-final class variables declared in the type[7], etc.

- Heap: this area is also shared among all threads; it contains all the objects instantiated by the program. For each object, it includes the instance variables declared in the object's class and all its superclasses, and some kind of pointer into the method area.

- PC (program counter) register: this area is used by a single thread; it contains the address of the current JVM instruction being executed by the thread.

- Java stack: this area is used by a single thread; it stores a thread's state in discrete frames. A Java stack frame is created for each method being invoked. Each stack

[3]Type information includes the fully qualified name of the type, its direct superclass and direct superinterfaces, whether or not the type is a class or an interface, the type's modifier, etc.

[4]Constant pool is an ordered set of constants used by the type.

[5]Field information includes the field names, types and modifiers.

[6]Method data include the method names, return types, parameters, modifiers, bytecode instructions, exception tables, local variable sections of the stack frame, etc.

[7]The final class variables are in the constant pool.

frame includes local variables[8], operand stack, and frame data[9].

- Native method stack: this area is used by a single thread; it supports the execution of native methods that are written in a programming language other than Java.

The JVM specification [23] does not impose restrictions on how these areas are implemented inside the Java Virtual Machine. The structural details of runtime data areas are decided by the designers of individual implementations.

Figure 2.2 gives an example on the structures of the runtime data areas and how they are correlated with each other. In this figure, class C's object is stored in the heap and its class data are stored in the method area. The Java program only deals with C's reference, which is a pointer to the corresponding object in the heap. The object contains instance variables, together with a pointer to the method area. Following the pointer, it is easy to find the class data of C. The class data contains type information, constant pool, field information, method data, and so on. When C's method, m, is called, a new stack frame is created and pushed into the current Java stack. The new frame contains necessary information for method execution, including pointers to the method data of m. During the execution of m, the PC register of the thread in which m is invoked is always pointing to the bytecode instructions of m, so that, when the current instruction finishes, the virtual machine knows which instruction should be executed next.

2.3 Method Invocation and Method Table

In the JVM instruction set, four opcodes are used for method invocation:

- *invokevirtual*: to invoke an instance method of an object.
- *invokeinterface*: to invoke a method that is declared by an interface and implemented by a non-abstract class.
- *invokespecial*: to invoke an instance method that requires special handling, such as an instance initialization method, a private method, and a superclass method.

[8] Local variables include the reference to the class instance upon which the method is invoked, the method parameters, and other local variables inside the method.

[9] Frame data include a pointer to constant pool, a pointer to method exception table, and the data for debugging and method completion.

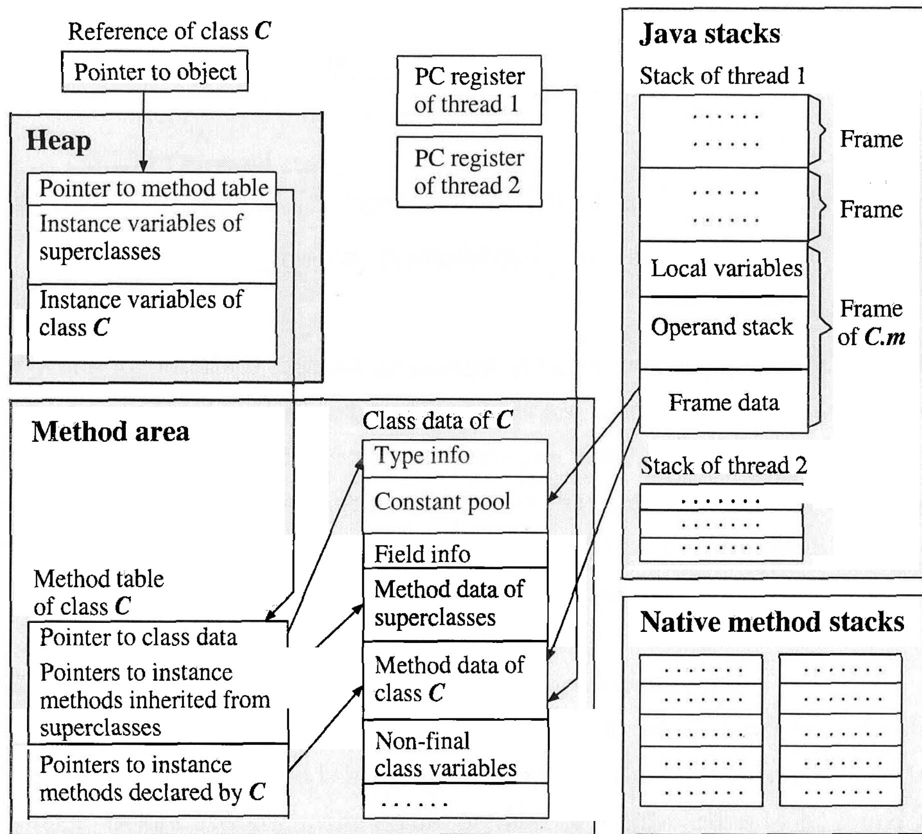


Figure 2.2: JVM runtime data areas

- *invokestatic*: to invoke a class (static) method.

Taking *invokeinterface* as an example, its instruction format and execution effect on the operand stack are shown in Figure 2.3.

Format

invokeinterface (0xb9)
indexbyte1
indexbyte2
count
0

Operand stack

$$\ldots, \; objref, \; [\, arg1, \; [\, arg2, \; \ldots \,]\,] \Longrightarrow \ldots$$

Figure 2.3: *invokeinterface* instruction

The opcode for interface method invocation is $0xb9$; *invokeinterface* is the mnemonic. The unsigned *indexbyte1* and *indexbyte2* operands are used to construct an index into the runtime constant pool of the current class, where the value of the index is $(indexbyte1 << 8) \mid indexbyte2$. The runtime constant pool entry at that index must be a symbolic reference to an interface method, which gives the name and descriptor of the interface method, as well as a symbolic reference to the interface, in which the method is declared. The *count* operand gives the number of arguments. The fourth operand byte exists to reserve space for an additional operand used in certain of Sun's implementations.

The interface method invocation begins by having *objref* (the reference to the object upon which the interface method is invoked) and *args* (method arguments) on the top of the operand stack. Before execution, the actual method to be invoked is to be found and then resolved. As a result of the execution of the *invokeinterface* instruction, *objref* and *args* are popped from the operand stack, with the remainder of the operand stack unaffected.

The instructions *invokevirtual*, *invokespecial* and *invokestatic* have simpler formats than *invokeinterface*. Their execution effects on the operand stack are similar to that of *invokeinterface*.

In order to speed up the access to an instance method invoked on an object – while at the same time keeping the miscellaneous types of data in the method area – *method tables* (see Figure 2.2) are used in many implementations of Java Virtual Machine, although it is not a

Format

invokevirtual (0xb6) / invokespecial (0xb7) / invokestatic (0xb8)
indexbyte1
indexbyte2

Operand stack

$\ldots, objref, [arg1, [arg2, \ldots]] \Longrightarrow \ldots$

Figure 2.4: *invokevirtual / invokespecial / invokestatic* instructions

necessary component required by the specification. The method table of a class is an array of direct references to all the non-private instance methods, i.e., virtual methods, that may execute with respect to an object of the class. The methods are either declared by the class, or, inherited from its superclass. Since the methods in the table are all virtual methods, it is also called *virtual method table (VMT).* A virtual method table does not help in the case of abstract classes or interfaces, because they do not contain concrete implementations of methods.

A method table is generated when a non-abstract class is resolved by the JVM, and stored in the method area as part of the class information. When a non-private instance method is invoked for the first time, the symbolic reference to this method in the constant pool of the current class is resolved to a concrete value: the method's offset in the method table. If the same instance method is invoked for the second time, the JVM just follows the concrete value in the constant pool to find the corresponding entry in the method table, and then, follows the reference in that entry to find the instructions of the method.

Chapter 3

Problem Definition

The network mobility of Java does not come for free. It comes at the cost of slower execution speed compared with other programming languages such as C++. Since the first Java Virtual Machine by Sun in 1995, the designers have been struggling on improving the performance of the Java Virtual Machine. With the advances of virtual machine technology in the recent years, the speed gap is shrinking, but not vanishing altogether. Interface invocation is one of those problems that needs solutions for better efficiency.

A Java *interface* is a reference type whose members are publicly accessible constants and abstract methods. In current JVM implementations, interface method invocation is not so effective as virtual method invocation. This problem comes from the *multiple interface inheritance* in Java: a class extends exactly only one class, but, can implement zero or more interfaces.

Some object-oriented programming languages like C++ allow a class to have multiple direct superclasses [16]. This form of inheritance, which is called *multiple inheritance*, is very controversial [9, 31], because of the complexities added to the syntax, the expensive implementation, and the ambiguities that occur when deriving an identically named member from different classes.

Java supports *multiple interface inheritance*, which proves to be an attractive alternative to multiple inheritance. In the multiple interface inheritance of Java, a class may have multiple direct superinterfaces, but only one direct superclass. Since no method code appears in the interfaces, there is no problem with name clashes or inheriting the same interface in multiple ways. If a class implements two distinct interfaces that declare the same method signature, it satisfies both interfaces by providing a single implementation of this method.

If an interface inherits more than one field with the same name, then a single ambiguous member results. This situation does not in itself cause a compile-time error. Only the attempt to use this ambiguous member by its simple name will result in a compile-time error.

3.1 Method Invocation in Single Class Inheritance

Before the Java Virtual Machine can execute the virtual method of a class, it should first locate the data structures in the method area that contain sufficient data to enable the virtual machine to invoke the method. In the implementations that support method table dispatch, the Java Virtual Machine will first find the invoked method in the method table, and then, follow the pointer in the method table to find the actual executable code in the method data of the class.

Figure 3.1 shows the method table of a non-abstract class, *C*, which does not implement any interface. The method table contains pointers to the virtual methods inherited from its superclass as well as that declared by class *C* itself, with the superclass methods placed before the methods of *C*. For the methods of either *C* or its superclass, pointers are stored in the order they appear in the corresponding class. If *C* overrides a superclass method, the pointer for the overridden method appears in the entry where it first appears in the superclass; the entry points to the method data of *C*, instead of that of the superclass.

When the Java Virtual Machine invokes a virtual method upon an object for the first time, it looks up the method in the method table that is associated with the type of the object, and resolves the symbolic reference of the method to a method table offset. In the subsequent invocations of the same method, the Java Virtual Machine can always depend on the offset, because single class inheritance implies that a method's offset in the superclass method table is always equal to its offset in the subclass method table, no matter the method is overridden or not. Therefore, given a virtual method and an object that the method is invoked upon, the method always occupies the same entry in certain method tables, irrespective of the actual type of the object.

Program 3.1 gives an example on virtual method invocation. Class ***A*** declares three virtual methods *a0, a1, m0.* Class ***B*** is a subclass of ***A***. ***B*** overrides ***A***'s method *m0* by defining its own implementation.

When class *C* is compiled, the two *a.m0()* instructions in its main function will be both

```
public class A {
  public void a0() { System.out.println("A: a0"); }
  public void a1() { System.out.println("A: a1"); }
  public void m0() { System.out.println("A: m0"); }
}

public class B extends A {
  public void b0() { System.out.println("B: b0"); }
  public void b1() { System.out.println("B: b1"); }
  public void m0() { System.out.println("B: m0"); }
}

public class C {
  public static void main(String[ ] args) {
    A a;
    a = new A();
    a.m0();
    a = new B();
    a.m0();
  }
}
```

Program 3.1: Example program for virtual method invocation

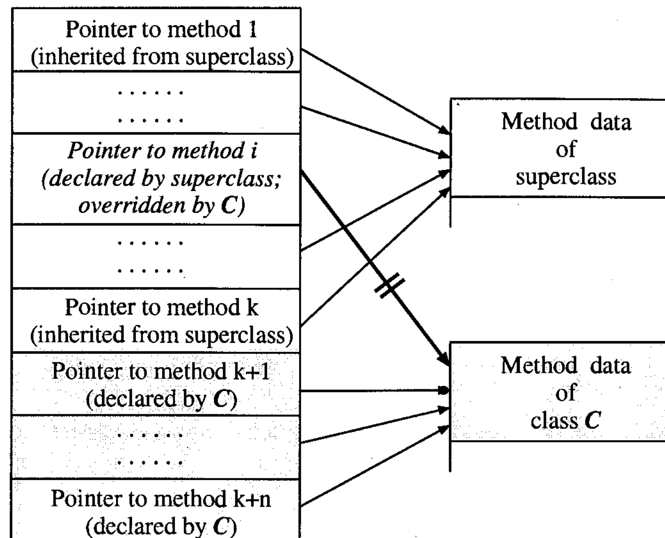


Figure 3.1: Method table of a non-abstract class that implements no interface

translated to the bytecode instruction: *invokevirtual indexbyte1 indexbyte2*. *indexbyte1* and *indexbyte2* together point to the symbolic reference of method $\boldsymbol{A}.m0$ in the constant pool of class $\boldsymbol{C}$. When the first *invokevirtual* instruction is executed, the symbolic reference of $\boldsymbol{A}.m0$ is resolved to its offset, supposing k, in $\boldsymbol{A}$'s method table. Since the object reference of class $\boldsymbol{A}$ is already pushed onto the operand stack, the Java Virtual Machine will deduce its type, $\boldsymbol{A}$, from the object reference, and then, load the method pointed to by the kth entry in $\boldsymbol{A}$'s method table. $\boldsymbol{A}$'s implementation of method *m0* is executed. When the second *invokevirtual* instruction is executed, symbolic reference resolution will not be performed again, since the symbolic reference is already replaced by an offset. Although the offset is with respect to the method table of $\boldsymbol{A}$ instead of $\boldsymbol{B}$, the Java Virtual Machine can depend on it, since *m0* must occupy the same offset in $\boldsymbol{B}$'s method table. Then, the Java Virtual Machine deduces the type $\boldsymbol{B}$ from the object reference on the operand stack, and loads the method pointed to by the kth entry in $\boldsymbol{B}$'s method table. This time, $\boldsymbol{B}$'s implementation of method *m0* is executed.

3.2 Method Invocation in Multiple Interface Inheritance

Interface method invocation is quite different from virtual method invocation, due to the inherent difference between single class inheritance and multiple interface inheritance.

The method table of a non-abstract class, C, which implements one or more interfaces is as in Figure 3.2. It contains pointers to the methods inherited from the superclass, declared by C itself, as well as that declared by the superinterface(s). The superclass methods are placed ahead of the methods of C, and the methods declared by superinterfaces are treated as part of C's methods. For the methods of either class C or its superclass, the pointers are stored in the order that they appear in the corresponding class. The pointers for the methods that are declared by superinterfaces and implemented by C point to the method data of C, because the corresponding superinterfaces do not contain implementation for abstract methods.

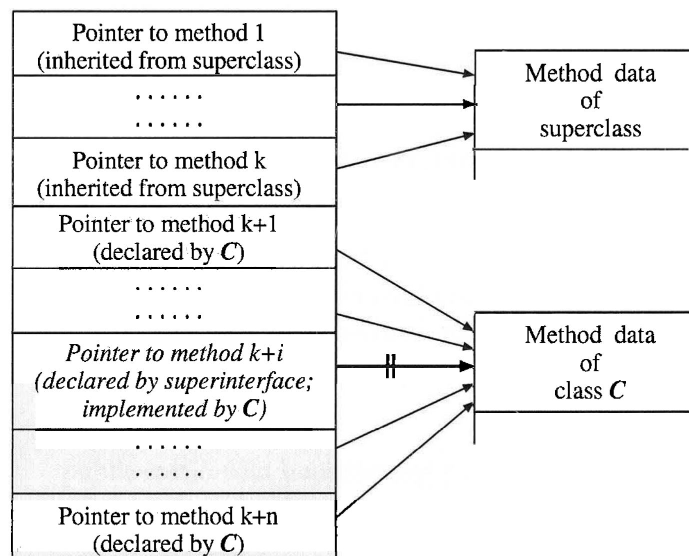


Figure 3.2: Method table of a non-abstract class that implements one or more interfaces

There is no guaranteed offset for an interface method in the method table. The offset of an interface method in the method table of a class depends on its appearance order in the class. As it may appear in different orders in different subclasses, the interface method is not necessarily at the same entry in all the subclass method tables. That is, the

method table offset is not always the same when an interface method is called several times. Therefore, symbolic reference resolution must be performed every time the Java Virtual Machine invokes an interface method.

Unlike the superclass methods, the superinterface methods are not placed ahead of the subclass methods in the method table, due to the potential multiple interface inheritance: even if they are placed ahead of the subclass methods, their offsets are not fixed in all method tables, if every subclass implements more than one interfaces.

Program 3.2 illustrates the problem associated with interface invocation. Interface ***I*** declares one abstract method *m0*. Two classes, ***A*** and ***B***, both implement ***I*** by giving their implementations for *m0*. The method tables for class ***A*** and ***B*** are shown in Figure 3.3.

```
public interface I {
  public void m0();
}

public class A implements I {
  public void m0() { System.out.println("A: m0"); }
  public void a1() { System.out.println("A: a1"); }
  public void a2() { System.out.println("A: a2"); }
}

public class B implements I {
  public void b1() { System.out.println("B: b1"); }
  public void b2() { System.out.println("B: b2"); }
  public void m0() { System.out.println("B: m0"); }
}
public class C {
  public static void main(String[ ] args) {
    I i;
    i = new A();
    i.m0();
    i = new B();
    i.m0();
  }
}
```

Program 3.2: Example program for interface method invocation

Since every class has ***java.lang.Object*** as a superclass (not necessarily a *direct* superclass), the first 11 entries (offset 0 ~ 10) in a method table contain the non-private instance methods declared by ***java.lang.Object***, while the remaining entries contain methods declared by the class. In the example, *m0* is at offset 11 in the method table of ***A***, and at offset 13 in the method table of ***B***. When class ***C*** is compiled, the two *i.m0()* instructions in the main function will both be translated to the bytecode instruction: *invokeinterface indexbyte1 indexbyte2*. *indexbyte1* and *indexbyte2* together point to the symbolic reference of method ***I****.m0* in the constant pool of class ***C***. When executing the first *invokeinterface* instruction, the Java Virtual Machine resolves the symbolic reference of ***I****.m0* to method table offset 11 by searching the method table of class ***A***. ***A***'s implementation of *m0* is loaded and executed. However, the offset is not constant. When the second *invokeinterface* instruction is executed, the naïve implementation will perform searching again, and the symbolic reference is resolved to an offset, 13, into ***B***'s method table. This offset is different from the previous one. This time, ***B***'s implementation of *m0* is loaded and executed.

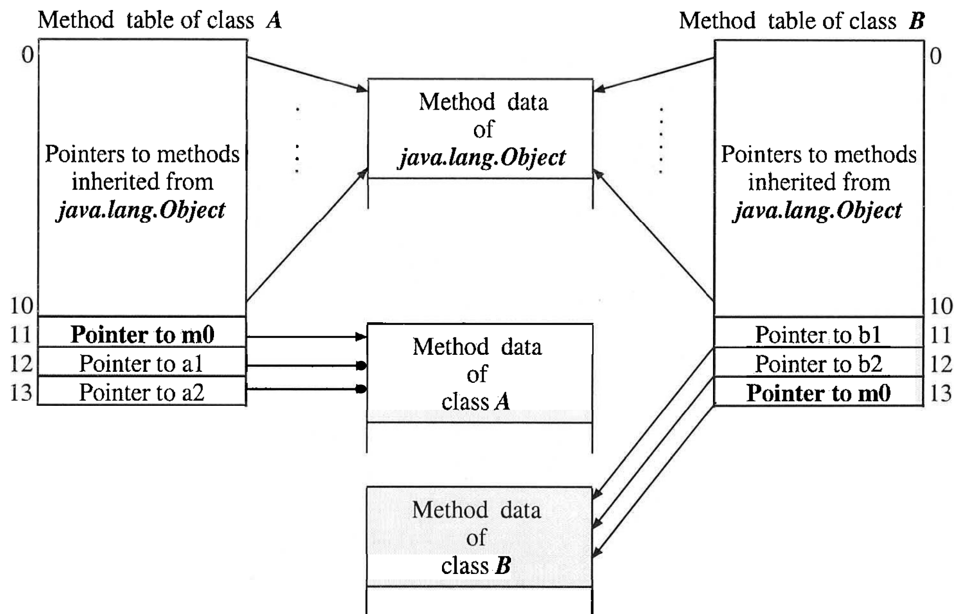


Figure 3.3: Method tables of classes ***A*** and ***B*** in Program 3.2

In summary, the overhead of naïve interface method invocation over virtual method invocation is that the Java Virtual Machine cannot simply resolve the symbolic reference

of an interface method to a fixed method table offset and depend on it in the following invocations. Instead, it has to search the appropriate method table for the implementation every time it encounters a method invoked upon an interface reference.

Chapter 4

Previous Approaches

To eliminate the overhead of interface invocation, research has been done in both academic and industrial settings. A number of solutions have been proposed, which can be classified into three categories: naïve implementation, interface table [17, 22] and selector-indexed table [14, 19, 4].

4.1 Naïve Implementation

The naïve implementation provides a straightforward but time-consuming solution for interface invocation, with the virtual method table used for interface dispatch. This approach has already been described in section 3.2. Whenever a method is invoked upon an interface reference, the JVM finds the appropriate virtual method table based on the object on the operand stack, searches the table to find the appropriate method that implements the interface method, and then executes it. The naïve implementation has no space overhead, since no additional data structure is needed. But its time overhead is high: as there is no guarantee that the interface will occupy the same entry in the virtual method tables of different subclasses, the JVM has to search one virtual method table each time an interface method is invoked.

4.2 Interface Table

The *interface table (itable)* approach is the most commonly used mechanism for interface invocation in high performance Java implementations. An itable is a virtual method table

for a (class, interface) pair, with each entry containing a pointer to the executable code of a method. This table is restricted to those methods that are declared by an interface and implemented by a class. Usually, the itable is an array in the method area; the pointers in it are arranged in the order that the corresponding methods are declared by the interface.

4.2.1 Searched ITable

The basic scheme of itable is called *Searched ITable* [17]. For each interface that is directly or recursively implemented by a class, the JVM constructs an itable in the method area. Each class that implements any interface is associated with an itable array, which contains pointers to the itables for every (class, interface) pair. The itable array is also in the method area, with size equal to the number of implemented interfaces. Usually, the itable array is directly referenced from the virtual method table of the class.

Figure 4.1 shows the structure of the searched itable. For a class C that implements k interfaces $I_1, I_2, \ldots, I_k$, the itable array contains pointers to the itables for the pairs (C, I_1), (C, I_2), ..., (C, I_k). The entries in each itable point to C's implementations of the corresponding methods. C's virtual method table contains a pointer to the itable array.

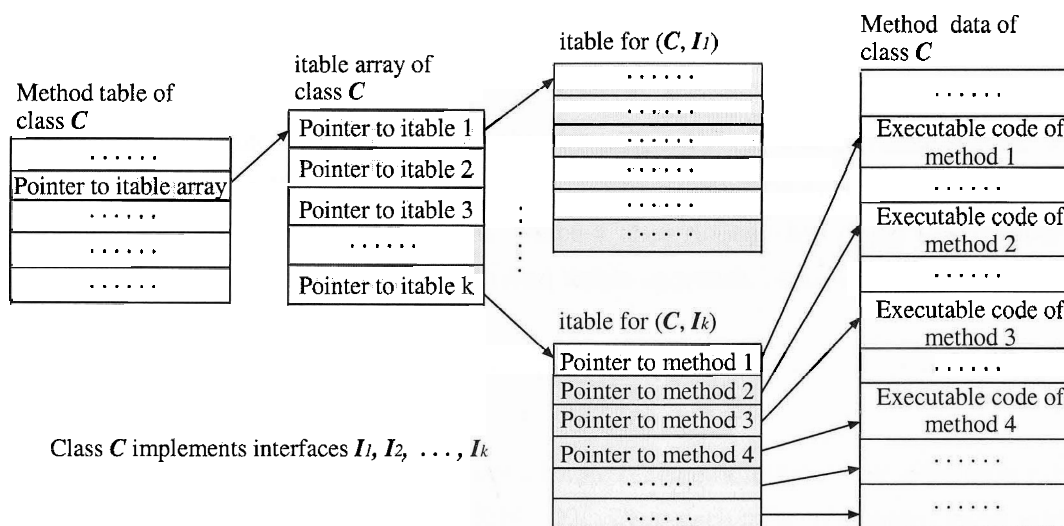


Figure 4.1: Searched itable

When an interface method is invoked, the Java Virtual Machine will locate the itable

array based on the object reference on the operand stack, search the array for the itable that corresponds to the appropriate (class, interface) pair, and then load the desired method from an appropriate offset. An interface method is at the same offset in all the itables for the interface, as the methods in an itable are typically arranged in their declaration orders[1] by the interface. The itable offset of a method can be determined from its declaration order, or, by searching the itable. In spite of this, the symbolic reference of an interface method cannot be replaced by an itable offset, as the offset itself does not provide information about the itable to which it applies; in the other words, itable search cannot be eliminated in searched itable approach, even though an interface method has a guaranteed itable offset. Thus, symbolic reference resolution must be performed every time when an interface method is invoked.

The overheads of searched itable-based interface invocation over virtual method invocation are: *a*) extra space for the itable array and itables, *b*) one extra dispatch for locating the itable array from the virtual method table, *c*) extra time spent on searching for an appropriate itable, and, *d*) extra time spent on deciding the method offset. It is obvious that the average of itable search time increases with the number of implemented interfaces. In order to reduce the search expense, some form of itable cache algorithm [15] has been exploited. One method of itable cache implementation is to let the first element of the itable array always point to the most recently constructed itable. Nevertheless, the search time is still significant in case of a cache miss.

The time overhead of the searched itable approach is mainly due to itable search. This approach outperforms the naïve implementation when dealing with an itable array that is smaller than the virtual method table. Since a class usually has much less number of superinterfaces than virtual methods, searched itable approach beats naïve implementation in almost all the cases.

4.2.2 Directly Indexed ITable

To reduce the time cost involved in searched itable scheme, another itable scheme is introduced, which is called *Directly Indexed ITable* [22]. This mechanism eliminates itable search by creating an entry in the itable array for each loaded interface, thus, the itable array contains pointers to the itables for each (class, loaded interface) pair. The itables are indexed

[1] See section 5.3 on page 32.

by *interface ids*, globally unique integral ids sequentially assigned to interfaces after they are loaded. If the class does not implement an interface, the corresponding itable is null, and, the corresponding itable array entry is empty. With a mostly empty itable array, the Java Virtual Machine can easily locate the appropriate itable from the itable array according to the interface id, with no need for searching.

Figure 4.2 gives an example for the directly indexed itable. Class C implements k interfaces: $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$; $i_1, i_2, \ldots, i_k$ are their ids. The Java Virtual Machine has so far loaded n interfaces: $I_0, I_1, I_2, \ldots, I_{n-1}$ ($k \leq n$, $0 \leq i_1 < i_2 < \ldots < i_k \leq n-1$). The itable array of class C contains pointers to the itables for the pairs (C, I_0), ..., (C, I_{i_1}), ..., (C, I_{i_2}), ..., (C, I_{i_k}), ..., (C, I_{n-1}), with the itables for some pairs being null. The size of itable array is equal to the number of interfaces already loaded. Some of the entries in the itable array are empty if C does not implement the corresponding interfaces.

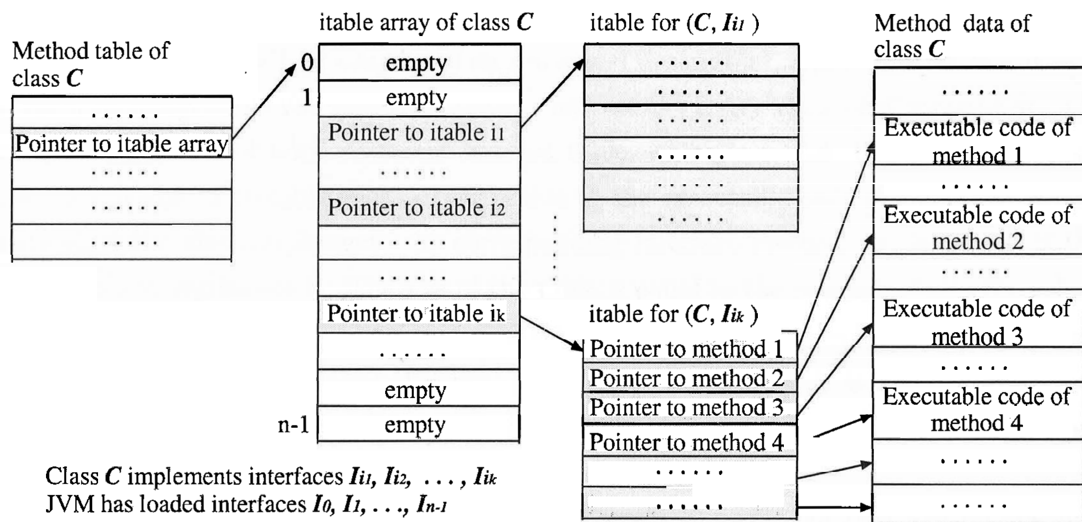


Figure 4.2: Directly indexed itable

Whenever an interface method is invoked, the Java Virtual Machine finds the itable array based on the object reference on the operand stack, locates the appropriate itable based on the interface id, calculate the method's offset in the itable, and then, loads the desired method. The interface id can be deduced from the interface that is indicated by the symbolic reference of the interface method. The itable offset of a method can be determined from its declaration order, or, by searching the itable.

The overheads of directly indexed itable based interface invocation over virtual method invocation are: *a*) extra space for the itable array and itables, *b*) one extra dispatch for locating the itable array from the virtual method table, and, *c*) extra time spent on deciding the method offset. The directly indexed itable mechanism avoids itable search by sacrificing more space on itable array. Resultantly, the performance of this approach is not much affected by the number of interfaces.

The CACAO JVM [22] is one of the implementations that employ the directly indexed itable approach. To reduce the space cost on the mostly empty itable array, CACAO truncates all the empty entries after the last non-empty one. This optimization saves considerable space when a class implements interfaces with small ids.

4.3 Selector Indexed Table

The *Selector Indexed Table* [14, 19, 4] is sometimes called *Signature Indexed Table.* The *signature* of a Java method consists of its name and descriptor[2], and a *selector* is a unique integral id assigned to the interface method signature. Every class that implements any interface is associated with a selector indexed table, which is a table indexed by interface method selector. The table entry either points to the executable code of the class' virtual method, if the class implements the corresponding interface method, or, is empty, if the class does not implement it. The size of the table is equal to the number of selector values assigned to the interface method signatures. As a result, the selector indexed table is sparse, with most of the entries empty. Usually, the selector indexed table is directly referenced from the virtual method table of a class. In some implementation, this table is embedded in the virtual method table [4].

Figure 4.3 illustrates the main idea of this approach. Suppose the Java Virtual Machine has assigned n selectors $(0, 1, 2, \dots, n-1)$ to interface methods; class $\boldsymbol{C}$ implements k interface methods $m_1, m_2, \dots, m_k$ $(k \leq n)$. The selector indexed table of $\boldsymbol{C}$ has n entries: k of them contain pointers to $\boldsymbol{C}$'s implementation of the corresponding interface methods, and the other $n-k$ entries are empty.

Whenever a method is called upon an interface reference, JVM first finds the appropriate selector index table based on the object reference on the operand stack, locates the entry in

[2]A method descriptor represents the parameters that the method takes and the value that it returns. See [23].

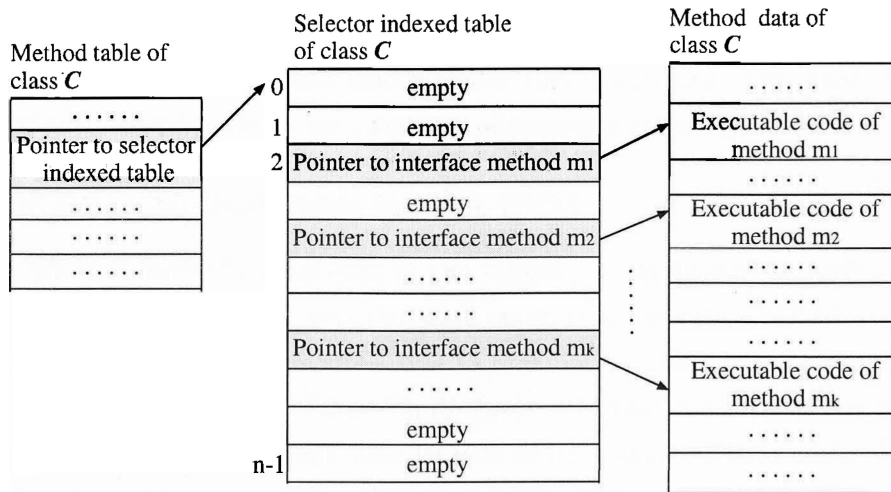


Figure 4.3: Selector indexed table

the table according to the selector of the method, and then, loads the desired method. The overheads of selector indexed table based interface invocation over virtual method invocation are: *a*) large space consumed by the sparse selector indexed table, and, *b*) one extra dispatch for locating the selector indexed table, if applicable.

Comparing with itable approaches, selector indexed table is much more space-intensive. In order to reduce the space overhead, several solutions have been explored, such as the sparse interface virtual table in SableVM [19], and the interface method table [4] in Jikes RVM[3] [2].

4.3.1 Sparse Interface Virtual Table in SableVM

The sparse interface virtual table in SableVM is a sparse array indexed by interface method id (selector). It is created at the same time when a class' normal virtual method table is created, and grows down from the normal virtual method table of the class. The size of the interface virtual table is equal to the largest id of all methods declared in the direct and indirect superinterfaces of the class. For any method declared by a superinterface, its entry in the table contains a pointer to the class' implementation of it. For all other methods, the

[3]Jikes RVM is an enhanced open source version of the code developed under the IBM Jalapeño research project since October 2001.

entries in the table are empty, since they are not implemented by the class.

To reduce the space overhead in the interface virtual table, the free space is recycled to the memory manager to store class loader related data structures. This kind of recycling mechanism adds complexities to both memory allocation and garbage collection.

4.3.2 Interface Method Table in Jikes RVM

The Jikes RVM reduces the space cost of the selector indexed table by hashing multiple interface method ids (selectors) into the same entry of a constant-sized selector indexed table and creating a *conflict resolution stub* to differentiate them [4]. The constant-sized table is called *Interface Method Table (IMT)*. Each class that implements any interface is associated with one IMT. The IMT may be part of the virtual method table of the class, or, directly referenced from the virtual method table. In these cases, the IMTs are called *embedded IMT* and *indirect IMT*, respectively.

In the current implementation of Jikes RVM, a very simple function is used for hashing: the interface method id, modulo the IMT size, is mapped directly into IMT entry. Therefore, all the methods whose ids are congruent modulo the IMT size are hashed into the same IMT entry. Obviously, the smaller the IMT, the more the conflicts.

An *IMT dictionary* is used to facilitate the construction of the IMT for a class. The IMT dictionary is an array with the equal size as the related IMT. Each entry in the dictionary points to a linked list [10]. Every linked list contains one or more references to the interface method signatures whose ids are congruent modulo the IMT size. The signatures within a linked list are arranged in the ascending order of their ids. The IMT dictionary only contains the signatures of those interface methods implemented by the class. Methods declared by different interfaces may be stored in the same linked list. Figure 4.4 gives an example for IMT dictionary with five entries.

Once all the implemented interface methods have been linked into the IMT dictionary, the Jikes RVM constructs the IMT for the class based on its IMT dictionary. For a linked list containing only one signature, the IMT entry, at the same index as the linked list in the dictionary, points to the class' virtual method which implements the corresponding interface method. If a linked list contains more than one signature, a conflict resolution stub will be generated, and the IMT entry will point to the stub. Figure 4.5 shows the IMT table that is constructed from the dictionary in Figure 4.4.

The main responsibility of a conflict resolution stub is to decide which virtual method is

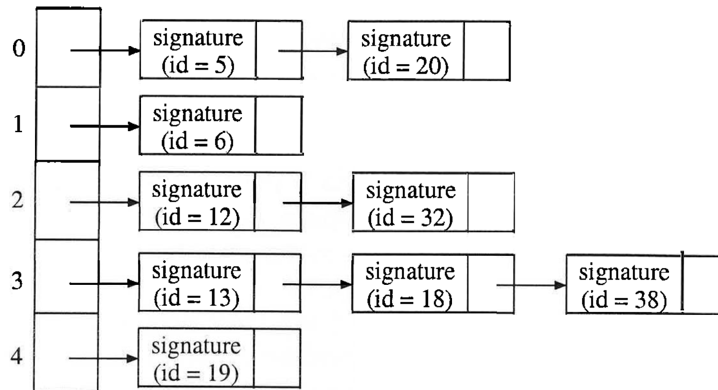


Figure 4.4: IMT dictionary with 5 entries

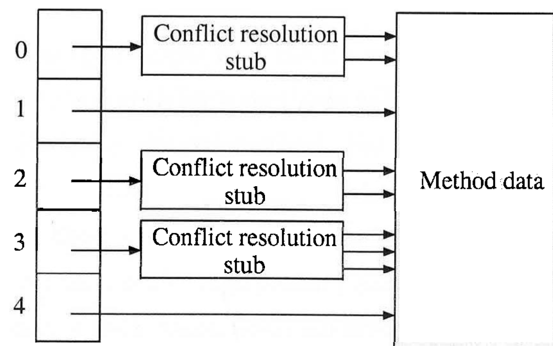


Figure 4.5: IMT table with 5 entries

to be executed based on the input method id. The pseudocode of a conflict resolution stub for four interface methods $m_{1\sim4}$ [4] is shown in Program 4.1.

```
     // Register s1 contains id of the interface
     // method being called;
     // id1 < id2 < id3 < id4;
     compare s1 with id2 (id of m2)
     if (s1 < id2) branch to L1
     if (s1 > id2) branch to L2
     load VMT entry for m2
     move m2 address to pc register
     branch to this address
L1:  load VMT entry for m1
     move m1 address to pc register
     branch to this address
L2:  compare s1 with id3 (id of m3)
     if (s1 > id3) branch to L3
     load VMT entry for m3
     move m3 address to pc register
     branch to this address
L3:  load VMT entry for m4
     move m4 address to pc register
     branch to this address
```

Program 4.1: A conflict resolution stub for four interface methods

The conflict resolution stub works in a binary search-like manner. Given an interface method id, a conflict resolution stub for n methods will perform at most $\lceil \log_2{}^n \rceil$ comparisons and branches before it finds the desired method. For instance, Program 4.1 will perform 2 comparisons and 2 branches before it finds the method m_4.

When an interface method is invoked, the Jikes RVM hashes its id into an IMT offset, and then loads the virtual method, or, the conflict resolution stub, that is pointed to by the IMT entry. The overheads of IMT based interface invocation over virtual method invocation are: *a*) extra space for the IMT dictionary and IMT table, *b*) extra time for executing the conflict resolution stub in case of conflicts, *c*) extra time spent on calculating the IMT offset of the invoked method, and, *d*) one extra dispatch for locating the IMT, in case of indirect IMT. The performance of the IMT approach is affected by the IMT size and the number of implemented interface methods. As the IMT size decreases or the interface method number

increases, more conflicts occurs, and more methods are hashed into the same IMT entry. As a result, more comparisons and branches in average may be needed for the conflict resolution stub to find the desired method.

Embedded IMT

The embedded IMT is part of the *TIB* (Type Information Block, the Jikes RVM analog of virtual method table) of a class, therefore, it can be directly accessed from an object.

Indirect IMT

The indirect IMT is different from the embedded IMT only in that the IMT is not part of the TIB but a separate table that can be accessed from TIB by following an IMT pointer in it. Thus, the overhead of indirect IMT over embedded IMT is one extra indirect dispatch for locating the IMT.

4.4 C++ Solution for Multiple Inheritance

Different from Java, the C++ programming language allows inheritance of multiple direct superclasses, and inheritance of different implementations of methods with the same signature [16]. To support multiple inheritance, a C++ class is associated with multiple method tables, with one table for each direct superclass [25]. For a class, C, which has k direct superclasses, its object is composed of $k+1$ contiguous sub-objects: the first k sub-objects for the superclasses, and the last one for class C itself [25]. A sub-object contains the fields of the corresponding class and a pointer to the method table of that class. The relative position of a sub-object inside the full object is known to the compiler.

Before the execution of a method, the object pointer is adjusted to point to the beginning of the appropriate sub-object, according to the actual type of the object reference. Object pointer adjustment is trivial as every sub-object's relative position is already known. After the adjustment, virtual method dispatch in multiple inheritance is identical to that in single inheritance.

Chapter 5

Proposed Approach

This thesis proposes an approach which substantially reduces the overhead of naïve interface invocation by introducing a new data structure – *interface zone (IZone)* – into the method area, and taking into account *class resolution order* and *method declaration order.* The proposed approach is called *IZone-Based Interface Invocation.*

5.1 Interface Zone

An interface zone, or, IZone, is associated with an interface that is implemented by at least one class. IZone is not constructed for a class, or, an interface that is not implemented by any class. By adding one more field, *IZone pointer*, into the virtual method table of an interface, the IZone can be directly accessed from the interface's virtual method table.The IZone of an interface is composed of several *class implementation lookup areas*, one for each subclass of the interface. A lookup area contains pointers to the corresponding subclass' implementation (executable code) of the interface methods. Its size is equal to the number of methods declared by the interface. Inside a lookup area, the pointers are arranged according to the interface method *declaration orders*, therefore, the same interface method will be at the same offset in different lookup areas.An IZone is not necessarily of fixed size. It may expand dynamically when more subclasses are loaded and resolved by the Java Virtual Machine. *A priori* knowledge about the subclasses is not necessary for IZone construction. The lookup areas are constructed in the *resolution* phase of the subclasses, and, arranged according to the subclass *resolution order.* As each subclass of an interface is loaded and resolved, one more lookup area is installed into its IZone. Figure 5.1 shows the IZone of

an interface, I, that declares j methods and has k subclasses. Each class implementation lookup area has j entries, with one for each interface method. The IZone of interface I has $(j \times k)$ entries in total.

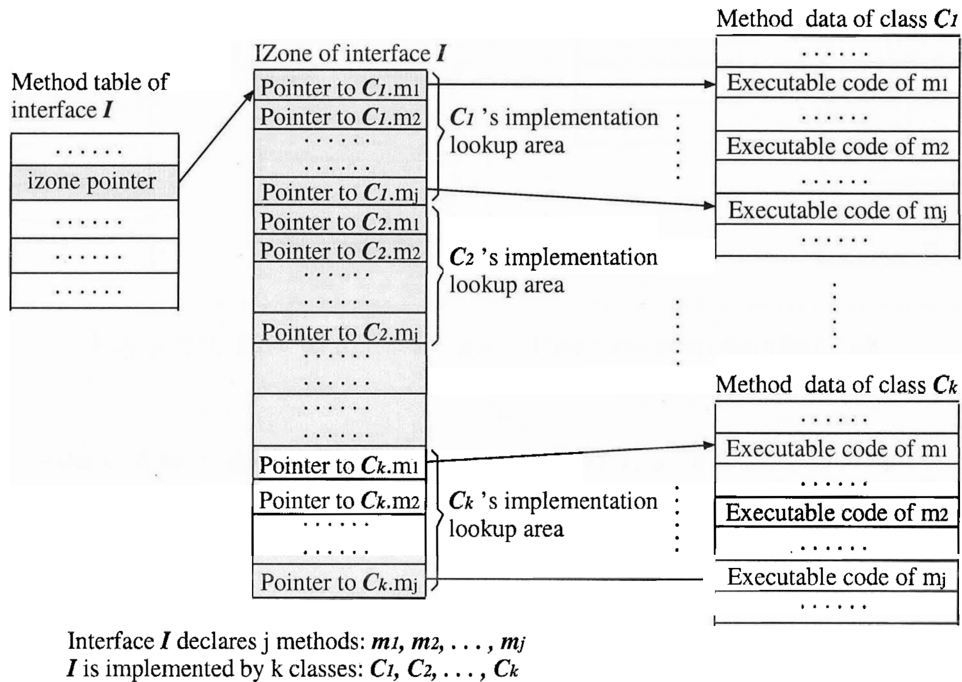


Figure 5.1: Interface zone

Given an interface method call, the appropriate IZone could be found following the procedure described in Figure 5.2. Suppose the interface method $I.m$ is called in method a of class C. Figure 5.2 shows part of the runtime data areas associated with class C and interface I. The bytecode of $C.a$ contains an *invokeinterface* instruction, within which, the operands *indexbyte1* and *indexbyte2* give the index of $I.m$'s symbolic reference in C's constant pool. The declaring class, I, of the method $I.m$ is indicated by its symbolic reference. It is easy to locate the method table, and then the IZone of the declaring class I.

The IZone approach is different from either itable or selector indexed table approach in the following aspects: *a*) The IZone is associated with an interface instead of a class, *b*) The IZone only contains the methods declared by one interface, *c*) The IZone entries point to multiple classes' implementations of interface methods, and, *d*) an interface method's offset

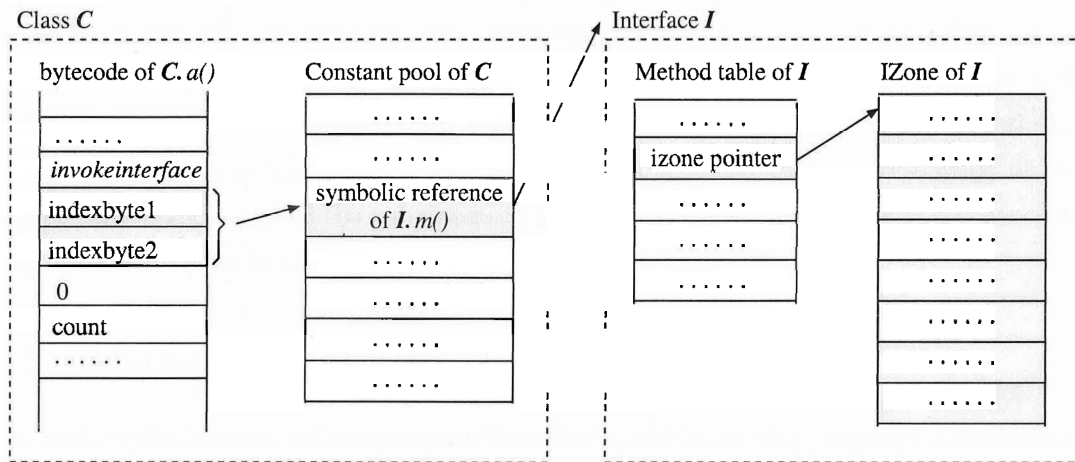


Figure 5.2: How to find the correct IZone given an interface call

in IZone is decided by its declaration order, as well as the resolution order of its implementing class.

5.2 Class Resolution Order

The class resolution order is with respect to a specific superinterface of the class. It represents an interface's view on the order in which a subclass is resolved. For a class that implements multiple interfaces, it may have different resolution orders with respect to different interfaces.

In order to record the resolution orders, each class, *C*, is associated with a *resolution order dictionary* indexed by *interface ids*. The dictionary is an integer array, *int orderDict[]*, with the element at index i being the interface $i(id = i)$'s view on C's resolution order. The resolution order begins from 1, while 0 represents an invalid order. If several classes implement the same interface *I*, from the viewpoint of *I*, the first resolved subclass has order 1, the second one has order 2, and so on. Since interface id begins from 0, the size of a resolution order dictionary is equal to the maximum id of the superinterfaces plus one. If the element at any index in the order dictionary has value 0, it implies that the class does not implement the corresponding interface. Apparently, the resolution order dictionary is "sparse". Most of the elements in it are of invalid values 0s. Interface types do not need an

order dictionary, as they do not implement any superinterface.

Every interface maintains a variable *nextOrder*: the resolution order that will be assigned to the next subclass. It has an initial value 1. Every time when one more subclass of the interface is resolved, its current *nextOrder* value is assigned to that class, and then, it increases by 1. Using this assignment scheme, it is guaranteed that the resolution order is sequentially assigned to all the subclasses. The class stores the newly obtained order at the appropriate entry in its order dictionary for the future usage.

5.3 Method Declaration Order

The method declaration order represents the order in which a method appears in its declaring class or interface. It is assigned to a method when its declaring class or interface is being loaded. The Java Virtual Machine loads a class by reading in its *.class* file and then retrieving the type, constants, fields, and, methods-related information.

A Java class file is a binary form of Java program that can be run by the Java Virtual Machine. It plays a critical role in supporting Java's platform independence and network mobility. Carrying type information and bytecode instructions, the Java class file gives the definition for a class or interface even though it knows little about underlying hardware platforms and operating systems. The pseudostructure of a Java class file [23] is shown in Figure 5.3.

In the Java class file format, the item *methods* is a table containing all methods declared by this type, including static methods, instance methods, instance initializer *<init>*, and, class/interface initializer *<clinit>* if any. Each element in the table is a *method_info* structure, which gives a complete description for a method: the access permission to and properties of the method, its simple name, descriptor, and attributes. The length of the *methods* table, i.e., the total number of declared methods, is defined by the item *methods_count*.

When a class is being loaded, the Java Virtual Machine will read in all the items in the class file, and save some of them in predefined formats in its internal data structures. The proposed approach adds declaration order assignment into the loading phase of a class: when the *methods* table is being processed, every method is assigned a declaration order that is equal to its index in the *methods* table. The method declaration order begins from 0. The first method in the table has order 0, the second one has order 1, and so on.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 5.3: Java class file format

5.4 Dispatch Mechanism Based on IZone

Like the itable or selector indexed table approach, the central issue in IZone-based interface invocation is how to locate the correct entry in the appropriate IZone given an *invokeinterface* instruction. Locating the IZone is as Figure 5.2 has shown. In order to find the correct entry for the invoked method in the IZone, this approach utilizes a very simple and straightforward formula for offset calculation, instead of a search process.

Suppose a method $\boldsymbol{m}$, declared by interface $\boldsymbol{I}$, is called upon an object of class $\boldsymbol{C}$, then the correct entry for $\boldsymbol{m}$ is at the following offset in the IZone of $\boldsymbol{I}$:

$$\mathit{offset} = (cOrder - 1) \times nMethods + mOrder \tag{5.1}$$

where, $cOrder$ is interface $\boldsymbol{I}$'s view on class $\boldsymbol{C}$'s resolution order (1, 2, 3,...), $nMethods$ is the number of methods declared by $\boldsymbol{I}$ (0, 1, 2, ...), and $mOrder$ is the invoked method's declaration order (0, 1, 2, ...). The entry at the calculated offset contains a pointer to class $\boldsymbol{C}$'s implementation of interface method $\boldsymbol{m}$, the actual code to be executed.

In the implementation, the expensive multiplication operation is replaced by a cheaper

left shift operation. To support this change, $nMethods$ is incremented to the nearest integer that is a power of 2 and no less than $nMethods$, and the offset calculation is changed to:

$$offset = (cOrder - 1) << lg + mOrder \tag{5.2}$$

where, lg is the smallest integer that is no less than the logarithm (base 2) of $nMethods$, i.e., $lg = \lceil \log_2^{nMethods} \rceil$. As a result, the size of each implementation lookup area is extended from $nMethods$ to 2^{lg}, or, $2^{\lceil \log_2^{nMethods} \rceil}$.

Now the benefits of dividing an IZone into multiple implementation lookup areas are revealed: since every lookup area contains the same number of entries, it is easy to calculate the relative position of a lookup area based on the resolution order of the corresponding class; and, since the pointers within a lookup area are arranged in the interface method declaration order, it is easy to calculate the offset of the invoked interface method based on its declaration order and the relative position of the corresponding lookup area. In short, the implementation lookup areas make it possible to take advantage of class resolution orders and method declaration orders.

Given an *invokeinterface* instruction, IZone-based interface invocation works in the following main steps:

1. Find the symbolic reference of the invoked method in the constant pool
2. Deduce the declaring interface of the invoked method
3. Find the IZone of the declaring interface
4. Obtain the object reference, upon which the method is invoked, from the operand stack
5. Deduce the class type of the object reference
6. Calculate the offset of the invoked method in the IZone
7. Follow the pointer at the calculated IZone offset to load the virtual method which implements the interface method
8. Execute the virtual method

Although the symbolic reference of an interface method is resolved to a concrete IZone offset before execution, the symbolic reference can not be replaced by the offset for the

future use, as the offset is not constant. Next time when the same interface method is invoked again, symbolic reference resolution has to be performed for the second time.

The overheads of IZone-based interface invocation over virtual method invocation are: *a*) extra space for the IZone and order dictionary, *b*) extra time spent on calculating the IZone offset for the invoked method, and, *c*) one extra dispatch for locating the IZone. The performance of the IZone approach does not change much with either the number of declared interface methods, or the number of subclasses, since the appropriate IZone offset is calculated using a simple formula rather than a search.

5.5 Pseudocode

Before an interface method can be invoked, the IZone of the related interface should have already been constructed and the appropriate class implementation lookup area should have already been installed. When the machine code of a virtual method is updated, the corresponding entries in IZones should also be updated. This section describes how an IZone is constructed and updated and how an interface method is invoked by supplying pseudocode. The pseudocode is written in Java-like notation.

5.5.1 IZone Construction

When a non-abstract class, C, is being resolved, the Java Virtual Machine will execute Program 5.1 to obtain a resolution order for it from each of its direct and indirect superinterfaces, initialize IZones for the superinterfaces as needed, and install C's implementation lookup areas into those IZones. This process is called IZone construction. Program 5.1 only lists the main steps for IZone construction. Type checking, such as method accessibility and property checking, is omitted to save the space.

For each interface, I, that class C directly implements or recursively inherits from its superclass or superinterfaces, the Java Virtual Machine will perform the following checks and take actions accordingly:

1. Check whether I declares any method. If it does not, no method will be called upon its reference during the Java program execution, therefore, no IZone will be constructed for it.

2. Check I's view on class C's resolution order. If it is non-zero, i.e., a valid order,

// $\boldsymbol{C}$: the non-abstract class that is being resolved
for (each interface $\boldsymbol{I}$ implemented by $\boldsymbol{C}$)
    if ($\boldsymbol{I}$ declares no method) return;
    if ($\boldsymbol{I}$'s view on $\boldsymbol{C}$'s resolution order != 0) return;
    obtain *nextOrder* from $\boldsymbol{I}$ and add it to $\boldsymbol{C}$'s resolution order dictionary;
    obtain the value of lg from $\boldsymbol{I}$;
    if (IZone of $\boldsymbol{I}$ == null) create IZone for $\boldsymbol{I}$ with size = 1 << lg;
    else expand IZone of $\boldsymbol{I}$ to size (current size + 1 << lg);
    for (each method m declared by $\boldsymbol{I}$)
        if (m is class initializer) continue;
        find $\boldsymbol{C}$'s virtual method vm which implements m;
        calculate the offset of m in the IZone of $\boldsymbol{I}$;
        let the IZone entry for m point to vm's code;

Program 5.1: Program for IZone construction

it indicates that $\boldsymbol{C}$ has already obtained a resolution order from $\boldsymbol{I}$ and stored it in its order dictionary. Since lookup area installation is immediately following the obtainment of resolution order (see step 3), it can be concluded that $\boldsymbol{C}$'s lookup area has already been installed into the IZone of $\boldsymbol{I}$. Otherwise, it indicates that $\boldsymbol{C}$ has not obtained a resolution order from $\boldsymbol{I}$ yet. $\boldsymbol{C}$ should obtain an order first, and then, further actions should be taken as follows.

3. Check whether the IZone of interface $\boldsymbol{I}$ has been initialized. If not, it implies that $\boldsymbol{C}$ is the first resolved class that implements $\boldsymbol{I}$; initialize $\boldsymbol{I}$'s IZone to have the exact number of entries for a lookup area. Otherwise, $\boldsymbol{C}$ is not the first resolved subclass; expand the IZone so that it can contain one more lookup area, with the new entries appended at the end and the old entries not affected.

4. Find $\boldsymbol{C}$'s implementation for each method declared by $\boldsymbol{I}$, and let the corresponding IZone entry point to the implementation. In order to find the implementation for an interface method, the virtual method table of $\boldsymbol{C}$ will be searched from the beginning, until a method is found with the same name and descriptor as the interface method.

Program 5.1 just illustrates how the IZones of all the superinterfaces are affected by the resolution of a single class. From the perspective of a single interface, its IZone is constructed gradually as its direct and indirect subclasses are resolved one after another: the IZone is

initialized when the first subclass is being resolved; one more implementation lookup area is installed during the resolution of a subclass; the complete IZone is built up after all the subclasses are resolved.

5.5.2 IZone Updating

During the execution of a Java program, the virtual method of a class may be updated, for instance, the bytecode of the method is recompiled into more efficient machine code by an optimizing compiler [2]. As a result, the corresponding entries in some IZones are pointing to an out-of-date address. If such a situation occurs, Program 5.2 will be executed by the Java Virtual Machine to update the IZone entries.

```
// C: the class whose virtual method has been updated
// vm: the updated virtual method of C
for (each interface I implemented by C)
    find its declared method, im, which is implemented by vm;
    if (im does not exist) continue;
    calculate the offset of im in the IZone of I;
    update the IZone entry for im to point to vm's code;
```

Program 5.2: Program for IZone updating

In case that a virtual method of a class is updated, the Java Virtual Machine will check each interface that the class directly or indirectly implements, to see whether any of its method is implemented by the updated virtual method, i.e., whether it declares a method with the same name and descriptor as the virtual method. If no such method exists, no change will be made to its IZone; otherwise, update the corresponding IZone entry to point to the new code of the virtual method. Since, in multiple interface inheritance, a virtual method may have satisfied multiple interface methods, this procedure will continue until all the implemented interfaces have been examined.

5.5.3 Interface Invocation

Program 5.3 gives the pseudocode for dispatching interface invocation. Before the interface method is executed, the object reference upon which the method is invoked, together with method arguments, would already be on the top of the operand stack, with the arguments

above the object reference. By looking into the interface method descriptor, the Java Virtual Machine can get the types and the number of arguments the method takes, and then, based on them, calculate the offset of the object reference in the stack. Then, it is easy to locate the object reference in the operand stack. The class type will then be deduced from the object reference, and, the method declaring class will be deduced from the invoked interface method. Knowing the class type of the object and the declaring class of the interface method, it is easy to obtain the class resolution order, invoked method declaration order, and the size of each lookup area, and then, calculate the IZone offset using the formula presented in section 5.4.

// m: the interface method being invoked
locate the object reference upon which m is invoked;
$\boldsymbol{C}$ = class type of the object reference;
$\boldsymbol{I}$ = declaring class of m;
$cOrder$ = $\boldsymbol{I}$'s view on $\boldsymbol{C}$'s resolution order;
obtain the lg value from $\boldsymbol{I}$;
$mOrder$ = m's declaration order;
m's *offset* in $\boldsymbol{I}$'s IZone = $(cOrder - 1) << lg + mOrder$;
locate the IZone of $\boldsymbol{I}$;
follow the pointer at the IZone *offset* to run the method;

Program 5.3: Program for interface invocation

Chapter 6

Approach Evaluation

This chapter assesses the IZone-based interface invocation approach by comparing it with five previous ones: searched itable, directly indexed itable, embedded IMT, indirect IMT, and naïve implementation. Experiments are conducted on a modified Jikes RVM [2] that deploys these six approaches, and the experimental results are analyzed to study the nature of different approaches and evaluate their performance in terms of both time and space cost.

6.1 Jikes RVM

The Jikes RVM is a virtual machine for Java servers. It is the first self-hosted virtual machine written in the Java language [3]. By assembling the fundamental core services – class loader, object allocator, and, compiler – into an executable *boot image* prior to running the virtual machine, the Jikes RVM can get started and load all remaining services required for normal operation. Unlike the JVM that runs on top of another JVM, the Java code of Jikes RVM runs on itself without requiring a second virtual machine.

6.1.1 Object Model

In the Jikes RVM, a scalar object is composed of an object *header* and several instance fields; an array object is composed of an object *header* and all the array elements. An object header contains GC information, a pointer to a *Type Information Block (TIB)*, status information, and some other miscellaneous fields. TIB is the Jikes RVM's analog of a virtual method table, but it contains more information than a typical method table does. A TIB

contains type description, superclass information, superinterface information, (pointer to) interface dispatch data structure[1], pointer to array element TIB[2], and, a virtual method table. Figure 6.1 gives an example for the object and TIB of a class. Since an IZone is only associated with an interface, the TIB in the example does not contain pointer to IZone. The "pointer to IZone" occupies the same slot in an interface TIB as the "pointer to itable / indirect IMT" in a class TIB.

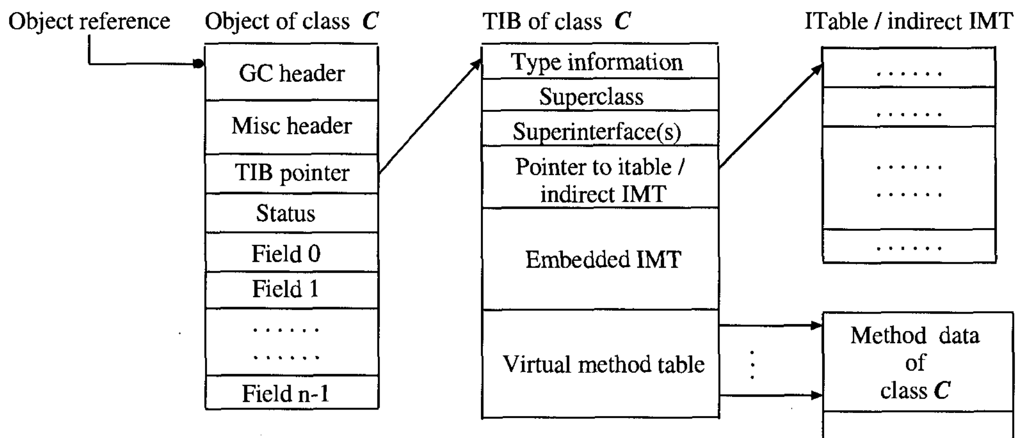


Figure 6.1: Object model of Jikes RVM

For a class type, its fields and method pointers span different data structures: instance fields are in the object of the class, pointers to virtual methods are in the TIB, static fields and pointers to static methods are in the *Jikes RVM Table of Contents (JTOC)*. Unlike a TIB, a per-type data structure, the JTOC is share by all types. Besides static fields and static method pointers from all the classes, the JTOC contains pointers to the TIBs of each type in the system as well. Since different types of data are stored in the JTOC, another equal-sized array is used to describe the elements in the JTOC. Based on different data structures, the dispatch mechanism differs for static and virtual methods.

[1]For the embedded IMT approach, TIB contains the whole IMT table; for the IZone, itable, or, indirect IMT approach, TIB contains a pointer to those data structures.

[2]If this is an array type.

6.1.2 Compiler

The Jikes RVM supports two architecture families: IA-32 [13] and PowerPC [20]. Before executing a program, the compiler inside the Jikes RVM first compiles bytecode into machine code that conforms to the underlying architectures. The Jikes RVM adopts a dynamic, or "lazy", class loading strategy: when the compiler encounters a bytecode, such as *invokevirtual*, that refers to a class not yet loaded, it does not load the class immediately; instead, the compiler emits code that, when executed, first ensures that the referenced class is loaded, resolved and instantiated, and then performs the operation.

The Jikes RVM employs two different, but cooperating, compilers – the *baseline compiler* and the *optimizing compiler* [8] – that address distinct design points. The baseline compiler is a quick compiler that does not generate high-performance machine code. The optimizing compiler generates high-performance code for the *hottest* methods. The cost of running the optimizing compiler is quite expensive, therefore, it is called on a method only after the method has executed many times.

During the optimization of a method, the bytecode is transformed into HIR (high-level intermediate representation), LIR (low-level intermediate representation), MIR (machine intermediate representation), and finally into machine code [2]. HIR instructions are closely patterned after bytecode, except that HIR instructions operate on symbolic register operands instead of an implicit stack, and that HIR contains separate operators to check for runtime exceptions. LIR expands HIR instructions into operations that are specific to Jikes RVM's object layout and parameter-passing conventions. MIR instructions reflects the IA-32 or PowerPC architecture.

Various optimizing transformations are performed at each level of IR (intermediate representation) [2]. In the bytecode-to-HIR transformation, some on-the-fly optimization is performed, such as copy propagation, constant propagation, register renaming for local variables, dead code elimination, etc. HIR optimization includes: common expression elimination, redundant exception check elimination, redundant load elimination, flow insensitive optimization across extended basic blocks [12], and in-line expansion of method calls [6]. LIR optimization includes local common subexpression elimination and dependence graph [11] construction. MIR optimization includes live variable analysis and linear scan global register allocation [24].

When a method is executed for the first time, it is compiled with the baseline compiler.

It is the *adaptive optimization system* [5] that makes decision on whether a method should be recompiled using the optimizing compiler and when. The adaptive optimization system keeps track of method execution using on-line measurement such as profiling and sampling. When a certain threshold is reached, an *optimization plan* is be built to describe which methods should be recompiled with what optimization levels. Then, the optimizing compiler is invoked automatically and the methods are compiled accordingly.

6.2 Approaches for Comparison

Six interface invocation approaches are experimented in this chapter: IZone, searched itable, directly indexed itable, embedded IMT, indirect IMT, and naïve implementation. They are implemented in Jikes RVM v2.3.0. Their performance is evaluated from two aspects: the time they spend on executing interface methods, and, the extra space they consume compared with virtual method invocation.

Different IMT sizes 5, 40, and 100 (IMT-5/40/100) are tried in IMT approaches, in order to explore how the IMT size affects the approach performance. In the two itable approaches, method declaration order is utilized to decide a method's offset in the itable.

All the experimental results reported below are obtained on a Pentium-III computer with 655 MHZ CPU and 512M RAM. The operating system installed is Red Hat Linux 9.0, which runs in single user mode [18] to minimize the interference from other processes.

6.3 Real World Test Case

We verify the implementation correctness by running a real world Java application, Xerces-J Parser 1.4.4 [1], on top of the modified Jikes RVM versions that adopt the six interface invocation approaches.

Xerces-J Parser 1.4.4 is an XML parser written in Java. It supports W3C's XML Schema recommendation version 1.0 [28, 29, 30], DOM Level 2 version 1.0 [27], and SAX Version 2 [7]. Tree-based DOM and event-based SAX are two most popular interfaces for programming XML. Xerces-J Parser provides two types of parsers: DOM parser and SAX parser, which are invoked by the classes *DOMCount* and *SAXCount* respectively. Both *DOMCount* and *SAXCount* parse input XML files and output the total parse time, along with counts of elements, attributes, text characters, and ignorable whitespace characters.

Xerces-J Parser involves intensive interface invocation. In the package ***org.w3c.dom***, there are 17 interfaces and only 1 class. In the package ***org.xml.sax***, there are 11 interfaces and 6 classes. In total, Xerces-J Parser has about 334 classes and 162 interfaces. Many classes implement more than one interfaces, many interfaces are implemented by multiple classes, and many methods are invoked upon interface references. All these make this software is a very good testbed for verifying implementation correctness.

Table 6.1 and 6.2 shows the parse results of *DOMCount* and *SAXCount* when working with the XML file for periodic table *allelements.xml*[3] from *ibiblio.org*. Both tables show that all the interface invocation approaches implemented in the Jikes RVM (with optimizing compiler) can generate the same output (except the parse time) as the standard Sun JVM. Additional tests with other xml files also prove the implementation correctness.

JVM versions	Output of *DOMCount*
Sun J2SDK 1.4.1	1207 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (IZone)	510 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (directly indexed itable)	499 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (searched itable)	524 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-5)	507 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-40)	519 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-100)	522 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-5)	505 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-40)	497 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-100)	502 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (naïve implementation)	2327 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)

Note: all the Jikes RVM versions in this table are with optimizing compiler

Table 6.1: Parse results of *DOMCount*

The modified Jikes RVM versions which support the baseline compiler only, also run *DOMCount* and *SAXCount* on the XML test file, and the output proves implementation correctness as well.

[3]See Appendix B on page 80.

JVM versions	Output of *SAXCount*
Sun J2SDK 1.4.1	746 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (IZone)	264 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (directly indexed itable)	258 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (searched itable)	258 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-5)	265 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-40)	260 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (embedded IMT-100)	258 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-5)	265 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-40)	261 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (indirect IMT-100)	261 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)
Jikes RVM (naïve implementation)	1538 ms (1897 elems, 936 attrs, 0 spaces, 25523 chars)

Note: all the Jikes RVM versions in this table are with optimizing compiler

Table 6.2: Parse results of *SAXCount*

6.4 Artificial Test Cases

Six artificial test cases, tests 1∼6[4], are adopted to evaluate the time cost of the six interface invocation approaches. In each test, 20 interfaces (Interface 1∼20) are defined with the same number of declared methods; all the interfaces are implemented by two classes A and B. Using the same notations as in [4], the class' implementation of interface methods are either *trivial calls* (simply return a integer value between 0 and 1200) or *normal calls* (either return an integer between 0 and 1200 or call another method, based on the true / false value of an instance field of the class). All the interface methods are implemented either as trivial calls or as normal calls in every test case. Table 6.3 below gives the descriptions for the 6 test cases.

The number of methods declared by each interface in the six test cases are 1, 1, 100, 100, 200, 200, respectively. In tests 1, 3, and 5, all the interface methods are implemented as trivial calls, while in tests 2, 4, and 6, all the interface method implementations are normal calls. For the experiments in the following sections, an instance field of the class is always set to true in run time so that the normal calls in tests 2, 4, and 6 always return an integral value without calling another method. However, setting the instance field to true in run time does not mean the branch in a normal call is eliminated after compilation, as the value of the instance field is not known at compile time and may change in run time.

[4]See the source code in Appendix C on page 86.

Test cases	Number of methods declared by each interface	Interface methods implemented as
1	1	trivial calls
2	1	normal calls
3	100	trivial calls
4	100	normal calls
5	200	trivial calls
6	200	normal calls

Table 6.3: Artificial test cases

Thus, normal call has more bytecode instructions than trivial calls. The tests will study the performance of interface invocation approaches with trivial calls and normal calls, and how they are affected by the number of interface methods.

The core of the test cases is a nested loop: the outer loop runs 10 iterations, while the inner loop runs 5,000 or 5,000,000 iterations[5]. The 10 execution times of the inner loop are recorded for further analysis. Program 6.1 shows the nested loop in the test cases.

```
for (int i = 0; i < 10; i ++) {              // outer loop
   long result = 0;
   for (int j = 0; j < 5000; j ++) {       // inner loop
      ......               ......
      result += in1.a1();    result += in2.b2();    result += in3.c3();
      result += in4.d4();    result += in5.e5();    result += in6.f6();
      result += in7.g7();    result += in8.h8();    result += in9.i9();
      result += in10.j10();  result += in11.m11();  result += in12.n12();
      result += in13.p13();  result += in14.q14();  result += in15.r15();
      result += in16.s16();  result += in17.t17();  result += in18.u18();
      result += in19.v19();  result += in20.w20();
   }
   System.out.println(result);
}
```

Program 6.1: The nested loop in the test cases

[5]When experimenting with baseline compiler, the inner loop runs 5,000 iterations; for the experiments with optimizing compiler, both 5,000 and 5,000,000 iterations will be tried. See the following sections for details.

Within the inner loop, the invoked methods a1, b2, ..., w20 are declared by Interface 1, Interface 2, ..., Interface 20, respectively, and, implemented by both class A and B. The variables *in1∼20* are all interface references, which are casted from the same class reference. The object that the class reference points to is of either class A or class B type, which can be decided only at run time. Therefore, none of the interface invocations will be virtualized or inlined by the Jikes RVM optimizing compiler. The execution time of the inner loop is affected not only by the interface invocations in the test cases, but also by the Jikes RVM versions, as interface invocations occur inside the Jikes RVM as well.

A few simpler test cases have also been tried, although not included in the thesis: the interface methods are always called upon a fixed object. After intensive optimization, the interface invocation takes similar time, regardless of which approach is used, as it has been converted to an appropriate virtual invocation (virtualization), and, takes almost the same time as the virtual method.

6.5 Time Cost with Baseline Compiler

For a Jikes RVM version that includes only a baseline compiler, each method will be compiled into machine code by the baseline compiler the first time it is invoked, and afterward, the same segment of machine code is executed every time the method is called. No optimization will be ever applied to the method. Thus, the execution time of the method body (not including the compilation time) will remain the same no matter how frequently the method is invoked.

In order to compare the performance of the six approaches with the baseline compiler, tests 1∼6 are executed, with the inner loop running 5,000 iterations and the outer loop running 10 iterations. The execution time of the inner loop are recorded. Theoretically, among the 10 times of the inner loop, the first one should be larger, and the other 9 ones are almost the same with each other, no matter which approach is used. The first time is larger as it includes the time for compiling the method into machine code: the method is compiled before the first execution. As the same machine code is executed in the subsequent invocations, the other 9 times are almost the same. Figure 6.2 testifies this statement. It shows the execution time (measured in millisecond) of the inner loop in tests 1∼6 when using the IZone and directly index itable approaches. The execution time with other approaches demonstrates the same tendency.

— IZone —
Test 1: 339, 325, 327, 326, 326, 327, 328, 327, 327, 327
Test 2: 347, 330, 329, 329, 331, 330, 330, 329, 329, 330
Test 3: 448, 335, 336, 337, 336, 336, 336, 336, 337, 336
Test 4: 452, 337, 337, 336, 337, 336, 336, 336, 336, 336
Test 5: 582, 330, 330, 330, 330, 335, 330, 329, 330, 329
Test 6: 590, 333, 334, 336, 332, 333, 332, 333, 333, 331

— Directly indexed itable —
Test 1: 556, 545, 543, 545, 545, 544, 544, 545, 546, 544
Test 2: 567, 550, 546, 549, 551, 550, 550, 551, 552, 552
Test 3: 664, 551, 553, 550, 551, 550, 550, 551, 550, 550
Test 4: 672, 553, 552, 554, 553, 554, 553, 553, 554, 553
Test 5: 777, 521, 522, 521, 521, 522, 521, 523, 521, 521
Test 6: 788, 525, 525, 524, 524, 524, 524, 525, 524, 524

Figure 6.2: Execution time of 5,000-iteration inner loop with baseline compiler

By calculating the average of the last 9 times, we get the average execution time of the inner loop, which does not include method compilation time. The average execution time (in millisecond) of the inner loop with each approach is in Table 6.4. The average execution time of virtual methods is also recorded when invoking the trivial or normal calls directly in the similar nested loop.

From Table 6.4, it is easy to see the following facts about the execution time of each test case:

$$Time_{test1} < Time_{test2}, \quad Time_{test3} < Time_{test4}, \quad Time_{test5} < Time_{test6} \tag{6.1}$$

$$Time_{test1} < Time_{test3} < Time_{test5}, \quad Time_{test2} < Time_{test4} < Time_{test6} \tag{6.2}$$

Tests 1, 3, and 5 only contain trivial calls, while tests 2, 4, and 6 only contain normal calls. Trivial calls contain less bytecode than normal calls, so, generally speaking, they take less time than normal calls, no matter which approach is used.

Condition (6.2) applies to IMT and naïve implementation only. In tests 1 and 2, the number of methods per interface is only 1; in tests 3 and 4, the number is 100; in tests 5 and 6, the number is 200. For embedded / indirect IMT approach (with a specific IMT size), more interface methods mean more conflicts, i.e., more methods are handled by a

Dispatch mechanism	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Virtual invocation	trivial calls: 146; normal calls: 150					
Embedded IMT-5	243	247	255	257	272	273
Embedded IMT-40	245	241	264	264	265	269
Embedded IMT-100	245	247	259	264	264	265
Indirect IMT-5	247	247	257	260	271	273
Indirect IMT-40	242	245	260	267	263	268
Indirect IMT-100	255	256	265	269	270	272
IZone	326	329	336	336	330	333
Directly indexed itable	544	551	550	553	521	524
Searched itable	714	714	720	722	721	723
Naïve implementation	839	848	33,393	18,624	60,461	60,783

Table 6.4: Average execution time with baseline compiler

conflict resolution stub. On average, therefore, it takes longer time to jump to a desired method in the resolution stub. For naïve implementation, more interface methods mean more time spent on searching for a desired method in a larger virtual table. Therefore, both the IMT and naïve approaches need more time to execute an interface method as the number of methods per interface increases. The execution time of the itable and IZone approaches does not change much with interface method number, since the offset of an interface method in itable or IZone is calculated from its declaration order, or, from both its declaration order and the proper subclass resolution order, which are already known before the method is executed. By calculating the offset, the entry for the method can be easily located without searching any table.

Based on the experimental data in Table 6.4, we rank the approach performance as follows, with ">" meaning "faster than":

$$IMT > IZone > directly\ indexed\ itable > searched\ itable > naïve \tag{6.3}$$

$$IMT\text{-}100 > IMT\text{-}40 > IMT\text{-}5 \tag{6.4}$$

$$virtual\ invocation > interface\ invocation \tag{6.5}$$

Among all the interface invocation approaches, IMT has the best performance with the baseline compiler, because of the nature of the conflict resolution stub. The conflict resolution stub is a short piece of machine code, while all the other approaches are implemented

in the Java language, involving intensive method calls and method returns, and being translated to a long segment of machine code, if no optimization applied. Furthermore, all the 6 test cases do not impose a big burden on the "branches" inside the conflict resolution stub. For IMT-5 that handles 200 interface methods, each conflict resolution stub deals with 40 methods. Since the conflict resolution stub uses a form of binary search, at most 6 ($40 < 64 = 2^6$) "branch" operations are performed before the appropriate method is found. However, in the case that a conflict resolutions stub has to deal with a large enough number of methods, IMT performance might be worst than the itable and IZone approaches, as the execution time of these two approaches are not affected much by the method number. The embedded IMT and indirect IMT approaches have quite similar performance as the only difference between them is one extra indirect dispatch. This difference will become more obvious in case of optimizing compilation (see section 6.6 on page 50).

The naïve implementation has the worst performance because it is too "naïve": every time an interface method is called, this approach will search the whole virtual method table of the implementing class linearly from the very beginning.

Condition (6.4) applies to both the embedded and indirect IMT approaches. With a fixed number of interface methods, a smaller IMT size leads to more methods that are handled by a conflict resolution stub, and therefore, a larger average execution time for interface invocation. This is not so obvious for tests 1 and 2 with IMT-40 and IMT-100, because each interface has only 1 method in tests 1 and 2, and thus, no conflict ever occurs between interface methods and conflict resolution stub is not needed. But for tests $3 \sim 6$, with interface method number being 100 or 200, conflicts occur with IMT-5, IMT-40 and IMT-100, and resolution stubs are generated to resolve conflicts; it is not difficult to find that a smaller IMT size leads to a larger execution time.

When working with the baseline compiler, all the interface invocation approaches are slower than virtual invocation: directly invoking trivial calls upon a class object is much faster than invoking the corresponding interface methods in test 1, 3, or 5; directly invoking normal calls upon a class object is much faster than invoking the corresponding interface methods in tests 2, 4, and 6. This is because that the virtual method offset is reliable, but interface method offset is not. All the approaches here have to search for the interface method or calculate its offset in the appropriate table every time it is invoked.

One point that is obvious, though not demonstrated by the experimental data in Table 6.4, is that the performance of IZone and directly indexed itable is not affected by the

number of interfaces implemented by a class. This can be easily inferred from their dispatch mechanism. More implemented interfaces mean more implementation lookup areas for a class in the IZone approach, or, more non-empty itables for a class in the directly indexed itable approach. Nevertheless, the IZone approach calculates method offset based on the known class resolution order, method declaration order, etc.; the directly indexed itable approach locates the invoked method based on the interface id and then the method declaration order. Both of them don't have to do searching. Therefore, no matter how many interfaces are implemented by a class, or, how many lookup areas or itables have been constructed for a class, it does not affect the time used for interface invocation. This does not apply to the IMT or searched itable approach, since more implemented interfaces mean more methods to be handled by each conflict resolution stub in the IMT approach, or, a larger itable array to be searched in the searched itable approach.

6.6 Time Cost with Optimizing Compiler

For a Jikes RVM version that supports both a baseline and an optimizing compiler, the situation is totally different. When a method is called for the first time, it is compiled into machine code by the baseline compiler. Afterward, if there are enough invocations of the method, it, or the method it will further call, will be passed to the optimizing compiler for optimization. In the first optimization pass, the method may be optimized at a modest level. As it is called again and again, it will be passed to the optimizing compiler again for optimization at a higher level, and so on. Higher level optimization needs more compilation time, but the resultant method takes less execution time.

6.6.1 Lightweight and Heavyweight Optimizations

Since more than one level of optimization may be applied to the same method, the 10 execution times of the inner loops in the 6 test cases may not be stable after the first iteration, which is different from the baseline compiler case. Figure 6.3 testifies this point.

Figure 6.3 gives the 10 execution times (in millisecond) of the inner loops in test 1 with the IZone and directly indexed itable approaches. The inner loops run 5,000 / 10,000 / 50,000 / 100,000 / 500,000 / 1,000,000 / 5,000,000 iterations for both approaches, and the execution time is recorded. The number in brackets are the iterations of the inner loops. The following tendencies are observed based on the experimental data in the figure:

— IZone —

(5,000)	47, 45, 45, 46, 45, 46, 45, 45, 46, 45
(10,000)	93, 91, 91, 90, 91, 91, 90, 91, 91, 90
(50,000)	453, 452, 453, 453, 480, 823, 454, 455, 454, 503
(100,000)	908, 906, 1288, 904, 904, 905, 904, 667, 308, 306
(500,000)	4923, 4568, 2078, 1502, 1501, 1493, 1491, 1485, 1487, 1515
(1,000,000)	5935, 3006, 2998, 2994, 3012, 3014, 2994, 2972, 2972, 2995
(5,000,000)	24605, 15905, 16040, 15707, 15664, 15800, 15458, 15521, 15650, 15659

— Directly indexed itable —

(5,000)	68, 65, 65, 66, 66, 65, 66, 66, 66, 65
(10,000)	135, 134, 133, 133, 134, 133, 134, 134, 133, 134
(50,000)	670, 667, 683, 747, 660, 660, 242, 134, 134, 134
(100,000)	1338, 1430, 1326, 1315, 1301, 1136, 223, 220, 215, 213
(500,000)	4120, 1120, 1018, 993, 984, 966, 965, 992, 987, 987
(1,000,000)	6728, 2010, 1994, 1998, 2002, 2028, 2080, 2069, 2065, 2064
(5,000,000)	14985, 10006, 9969, 9535, 9619, 9475, 9415, 9409, 9396, 9408

Figure 6.3: Execution time of inner loop in test 1 with optimizing compiler

- The execution time with both approaches does not change proportionally with the iterations of the inner loop. This is because more iterations lead to more optimization passes on the methods, and therefore, less execution time for the methods.
- If the inner loop runs 10,000 iterations or less, the 10 execution times do no demonstrate much difference. This is because the optimizations at low levels did not bring noticeable change to the execution time.
- If the inner loop runs 50,000 iterations or more, the 10 execution times demonstrate a decreasing tendency: the first time is relatively larger while all the others are smaller; the preceding time is usually larger than all the succeeding ones. The reason is: a method which has been optimized at a lower level may be sent to the optimizing compiler again for optimization at a higher level, if it has been invoked for enough times, thus, its execution time could be further reduced.

Although only the IZone and directly indexed itable approaches are cited as examples here, all the other approaches demonstrate the similar tendency as well.

If the inner loop runs 10,000 iterations or less, the resultant optimizations do not obviously change the method execution time. This kind of optimization behavior is called *lightweight optimization* is this thesis. If the inner loop runs for 50,000 iterations or more, the optimizations triggered lead to 10 decreasing execution times of the inner loop. This kind of optimization behavior is called *heavyweight optimization* here.

The 5,000- and 10,000-iteration inner loops result in lightweight optimizations not in test 1 only , but in all the six test cases, no matter which approach is used. Similarly, the $50,000^{+}$-iteration inner loops result in heavyweight optimizations in all the test cases for whatever approach. Figure 6.4 and Figure 6.5 testify this statement.

— IZone —
Test 1: 47, 45, 45, 46, 45, 46, 45, 45, 46, 45
Test 2: 73, 70, 69, 70, 69, 70, 69, 70, 70, 69
Test 3: 66, 50, 50, 50, 50, 51, 50, 50, 50, 51
Test 4: 89, 68, 68, 68, 68, 69, 68, 68, 68, 68
Test 5: 100, 51, 50, 50, 51, 50, 51, 50, 51, 50
Test 6: 122, 69, 82, 69, 68, 68, 68, 68, 77, 66

— Directly indexed itable —
Test 1: 68, 66, 66, 66, 66, 66, 65, 66, 66, 65
Test 2: 69, 65, 65, 65, 65, 66, 65, 65, 65, 65
Test 3: 80, 65, 65, 65, 65, 65, 65, 65, 65, 65
Test 4: 91, 71, 72, 71, 72, 71, 71, 72, 70, 71
Test 5: 110, 64, 64, 64, 64, 64, 65, 64, 64, 64
Test 6: 119, 70, 71, 77, 81, 68, 68, 68, 68, 71

Figure 6.4: Execution time of 5,000-iteration inner loop with optimizing compiler

Figure 6.4 gives the 10 execution times (in millisecond) of the 5,000-iteration inner loop. All the times in the figure are smaller than the corresponding times in Figure 6.2 for the baseline compiler. For both IZone and directly indexed itable approach, the first execution time is larger, while the other 9 times are quite similar with each other, as the lightweight optimizations have not substantially reduced execution time.

Figure 6.5 gives the 10 execution times (in millisecond) of the 5,000,000-iteration inner loop. Every test demonstrates the same tendency: the first time is the largest one, while all the other times decrease gradually. For the IZone approach, the last execution times are 16173, 17682, 18060, 20249, 19073, 20569 in tests 1, 2, ..., 6; for the directly indexed itable

— IZone —
Test 1: 24605, 15905, 16040, 15707, 15664, 15800, 15458, 15521, 15650, 15659 ~ 15659
Test 2: 21415, 18176, 18308, 18188, 18139, 17970, 17870, 17972, 18131, 17587 ~ 17587
Test 3: 22650, 16192, 16511, 16535, 16115, 16091, 16091, 15971, 15972, 15990 ~ 15990
Test 4: 25397, 18817, 18911, 18889, 18723, 18704, 18842, 18771, 18496, 18111 ~ 18111
Test 5: 29822, 17440, 17795, 17710, 17838, 17746, 17117, 16999, 17102, 17251 ~ 17251
Test 6: 24564, 19386, 19295, 19328, 19107, 19136, 19371, 19273, 19401, 19276 ~ 19276

— Directly indexed itable —
Test 1: 14985, 10006, 9969, 9535, 9619, 9475, 9415, 9409, 9396, 9408 ~ 9408
Test 2: 13601, 11080, 11018, 10899, 11140, 10811, 10703, 10603, 10599, 10639 ~ 10639
Test 3: 17078, 9227, 9295, 9745, 9253, 9109, 9101, 9101, 9114, 9102 ~ 9102
Test 4: 16145, 11422, 11247, 10866, 10772, 10896, 10943, 11092, 11051, 11001 ~ 11001
Test 5: 19726, 9580, 9413, 9480, 9601, 9189, 9115, 9129, 9137, 9185 ~ 9185
Test 6: 13945, 11650, 11582, 11397, 11384, 11159, 11264, 11311, 11203, 11040 ~ 11040

Figure 6.5: Execution time of 5,000,000-iteration inner loop with optimizing compiler

approach, the last execution time is 8298, 9642, 8311, 9673, 8355, 9773, respectively. The numbers after "~" give the last execution time.

In the following two sections, the performance of the six interface invocation approaches are evaluated with both lightweight and heavyweight optimizations.

6.6.2 Experimental Results with Lightweight Optimizations

In order to evaluate the six approaches' performance with lightweight optimizations, 6 test cases are executed with the inner loop running 5,000 iterations. All the 6 approaches demonstrate the similar tendency: the first execution time is the largest while the other 9 are similar with each other. By calculating the average of the last 9 times, we get the average execution time of the 5,000-iteration inner loop, which excludes the baseline compilation time, but includes the appropriate optimization compilation time.

The average execution time (in millisecond) of each approach is in Table 6.5. The shortest execution time for each test is enclosed in square brackets. The execution time in Table 6.5 is much smaller than that in Table 6.4, because the lightweight optimization reduces the method execution time, though not so much as the heavyweight optimization.

The following facts can be deduced from the table about the execution time of the 6 test

Dispatch mechanism	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**
Virtual invocation	trivial calls: 30; normal calls: 33					
IZone	[45]	69	[50]	[68]	[50]	[70]
Directly indexed itable	65	65	65	71	64	71
Embedded IMT-5	53	57	66	73	72	84
Embedded IMT-40	51	50	68	80	72	89
Embedded IMT-100	52	[50]	60	74	66	85
Indirect IMT-5	54	59	68	75	78	88
Indirect IMT-40	53	51	72	89	75	93
Indirect IMT-100	53	51	57	73	68	81
Searched itable	77	81	80	83	82	92
Naïve implementation	444	483	9,766	3,989	19,476	12,320

Table 6.5: Average execution time after lightweight optimizations

cases:

$$Time_{test1} < Time_{test2}, \quad Time_{test3} < Time_{test4}, \quad Time_{test5} < Time_{test6} \tag{6.6}$$

$$Time_{test1} < Time_{test3} < Time_{test5}, \quad Time_{test2} < Time_{test4} < Time_{test6} \tag{6.7}$$

Condition (6.6) holds since trivial calls take less time than normals calls.

Condition (6.7) applies to the IMT and naïve approaches only, as their performance is affected by the number of methods declared by the interface. The execution time of the IZone and itable approaches does not vary much with the number of interface methods.

The execution time with the IZone, directly indexed itable, and searched itable approaches does not increase with the number of implemented interfaces, just like the case for the baseline compiler.

Based on the experimental data in Table 6.5, we rank the approach performance as follows, with ">" meaning "faster than":

$$IZone > directly\ indexed\ itable,\ IMT > searched\ itable > na\ddot{\imath}ve \tag{6.8}$$

$$embedded\ IMT > indirect\ IMT \tag{6.9}$$

$$IMT\text{-}100 > IMT\text{-}40 > IMT\text{-}5 \tag{6.10}$$

$$virtual\ invocation > interface\ invocation \tag{6.11}$$

The IZone, directly indexed itable and IMT approaches are faster than the searched itable approach, which in turn is faster than naïve implementation. Based on the overall performance, IZone is the fastest approach, as it beats all the other approaches in 5 tests (1, $3 \sim 6$). Directly indexed itable beats embedded and indirect IMT approaches in the tests $3 \sim 6$, while IMT approaches beat directly indexed itable only in the tests 1 and 2, when when very few conflicts occur: after lightweight optimization, the conflict resolution stub which is written in machine code is no longer an advantage for IMT approaches.

After optimization, the difference between embedded IMT and indirect IMT approaches becomes more apparent. Indirect IMT is a little slower than embedded IMT, since it involves one extra indirect dispatch.

Both the embedded IMT and indirect IMT approaches work better with a larger IMT table, as each conflict resolution stub deals with less interface methods.

After lightweight optimizations, interface invocation is still much slower than virtual invocation, as the low level optimizations do not substantially reduce method execution time.

6.6.3 Experimental Results with Heavyweight Optimizations

In order to evaluate the six approaches' performance with heavyweight optimizations, the test cases are executed with inner loop running 5,000,000 iterations. By collecting the last execution time, we get the execution time (in millisecond) of the 5,000,000-iteration inner loop after heavyweight optimizations, as in Table 6.6[6]. This time excludes the baseline compilation time, but includes the appropriate optimization compilation time. The shortest execution time for each test is enclosed in square brackets.

From Table 6.6, we could deduce the following fact:

$$Time_{test1} < Time_{test2}, \quad Time_{test3} < Time_{test4}, \quad Time_{test5} < Time_{test6} \tag{6.12}$$

$$Time_{test1} < Time_{test3} < Time_{test5}, \quad Time_{test2} < Time_{test4} < Time_{test6} \tag{6.13}$$

Condition (6.12) holds as trivial calls are faster than normal calls. Condition (6.13) applies to the IMT and naïve implementation, as their performance are affected by the number of interfaces methods.

[6]The naïve approach has not been tested with the tests 3~6, as the execution times are too long (more than 10 hours). For the 100,000-iteration inner loops in the tests 3~6, their execution times with the naïve approach are already larger than the other approaches

Dispatch mechanism	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Virtual invocation	trivial calls: 6,873, normal calls: 8,965					
Directly indexed itable	9,408	10,639	[9,102]	[11,001]	[9,185]	[11,040]
IZone	15,659	17,587	15,990	18,111	17,251	19,276
Embedded IMT-5	10,405	10,670	23,228	16,814	30,299	32,430
Embedded IMT-40	[7,450]	10,081	22,167	24,386	22,787	26,585
Embedded IMT-100	7,615	[9,683]	12,588	15,763	16,660	19,641
Indirect IMT-5	9,640	11,426	23,823	17,530	30,974	33,625
Indirect IMT-40	8,496	10,326	23,327	26,127	23,728	25,678
Indirect IMT-100	8,179	10,492	11,736	13,810	17,140	20,561
Searched itable	39,917	42,196	40,283	41,839	40,254	41,642
Naïve implementation	61,245	66,649	–	–	–	–

Table 6.6: Stable execution time after heavyweight optimizations

Based on the experimental data in Table 6.5, we rank the approach performance as follows, with ">" meaning "faster than":

$$directly\ indexed\ itable > IMT,\ IZone > searched\ itable > na\ddot{i}ve \tag{6.14}$$

$$embedded\ IMT > indirect\ IMT \tag{6.15}$$

$$IMT\text{-}100 > IMT\text{-}40 > IMT\text{-}5 \tag{6.16}$$

Directly indexed itable has the best overall performance. It is faster than all other approaches with the tests 3 $\sim$ 6, although it is a little slower than certain IMT approaches with the tests 1 and 2.

IMT approaches are faster than directly indexed itable and IZone when there are very few conflicts between interface methods, for instance, in the tests 1 and 2. With large enough IMT tables, the time cost of the embedded IMT approaches (embedded IMT-40/100) with the tests 1 and 2 is close to that of virtual invocation: with IMT table sized 40 or 100, no conflict occurs in these two tests, therefore, the conflict resolution stub is eliminated, and 2 indirect dispatches are needed to find the method implementation, starting from the TIB of a class. In contrast, directly indexed itable needs 3 indirect dispatches from the TIB to the method implementation, while IZone needs 2 indirect dispatches besides the offset calculation.

Nevertheless, the performance of IMT is largely affected by the number of interface methods while IZone is not. IZone beats IMT when the IMT table size is too small compared

with the total number of interface methods. For example, IZone is faster than IMT in the tests 5 and 6. It is faster than embedded and indirect IMT-5/40 in the tests 3 and 4. IZone is slower than directly indexed itable in all test cases.

The searched itable approach is faster than the naïve approach only.

After heavyweight optimization, the difference between the embedded and indirect IMT approaches becomes even more evident.

6.7 Space Cost

This section studies the space overhead of interface invocation over virtual invocation using different approaches. To make things simple, we just look at the extra space associated with a single class for interface invocation. If there are multiple classes residing in the Java Virtual Machine, it is easy to calculate the total extra space for interface invocation.

First of all, we make the following assumptions:

- The JVM has loaded m interfaces: $I_1, I_2, \ldots, I_m$, with id: $id_1, id_2, \ldots, id_m$ ($0 \leq id_{1\sim m} < m$)
- A class $\boldsymbol{C}$ directly and recursively implements k interfaces: $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$ ($k \leq m, 1 \leq i_1 < i_2 < \ldots < i_k \leq m$), which declare $n_1, n_2, \ldots, n_k$ methods respectively.
- Among the methods declared by $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$, no two methods have the same signature.

The space overhead of the searched itable approach comes from: $\boldsymbol{C}$'s itable array which contains pointers to the itables for the implemented interfaces, and, the itables for each ($\boldsymbol{C}$, I_{i_j}) pair ($j = 1, 2, ..., k$). Since the itable for the pair ($\boldsymbol{C}$, I_{i_j}) has n_j entries, the total size of the itables is $\sum_{j=1}^{k} n_j$. The size of the itable array is equal to the number of implemented interfaces, k. Therefore, the size of the extra space associated with class $\boldsymbol{C}$ is:

$$SPACE_{SearchedITable} = \sum_{j=1}^{k} n_j + k$$

The space overhead of the directly indexed itable approach comes from: $\boldsymbol{C}$'s itable array which contains pointers to all the interfaces loaded so far, and, the itables for each ($\boldsymbol{C}$, I_{i_j}) pair ($j = 1, 2, ..., m$). The total size of the itables is the same as with the searched itable

approach, but, the size of the itable array is equal to the number of loaded interfaces by the JVM, not the number of implemented interfaces by C. Therefore, the size of the extra space associated with class C is:

$$SPACE_{DirectlyIndexedITable} = \sum_{j=1}^{k} n_j + m$$

The space overhead of the embedded or indirect IMT-p approach (p is IMT size) comes from the IMT table and IMT dictionary. The IMT table is of a fixed size, p, and the size of the IMT dictionary is equal to the size of the linked list pointer array which is the same as that of IMT table, plus the size of the links for all the interface methods implemented by C. The total number of implemented interface methods is $\sum_{j=1}^{k} n_j$, so the size of the links is $\sum_{j=1}^{k} n_j \times linkSize$. In the current implementation of Jikes RVM, a link in the IMT dictionary has the following fields: a pointer to the signature of an interface method, a pointer to the virtual method that implements the interface method, and a pointer to the next link, thus, $linkSize$ is 3. The size of extra space associated with class C is:

$$SPACE_{IMT-p} = p + (\sum_{j=1}^{k} n_j) \times linkSize + p = 3 \times (\sum_{j=1}^{k} n_j) + 2p$$

For the IZone approach, the space overhead comes from: C's implementation lookup areas in the IZones of interface $I_{i_1}, I_{i_2}, \ldots, I_{i_k}$, and, C's resolution order dictionary. C's implementation lookup area in the IZone of I_{i_j} has a size that is equal to the number of methods declared by I_{i_j}, n_j, if the formula (5.1) is used to calculate the IZone offset, or, $2^{\lceil \log_2^{n_j} \rceil}$, if the formula (5.2) is used. The size of C's resolution order dictionary is equal to the maximum id of the implemented interfaces plus one. Since an integer type occupies the same number of bytes as a pointer inside the JVM, the size of the extra space associated with class C is:

$$SPACE_{IZone} = \sum_{j=1}^{k} n_j + \max_{j=1}^{k} id_{i_j} + 1 = \sum_{j=1}^{k} n_j + t$$

$$\text{or,}\ SPACE_{IZone} = \sum_{j=1}^{k} 2^{\lceil \log_2^{n_j} \rceil} + \max_{j=1}^{k} id_{i_j} + 1 = \sum_{j=1}^{k} 2^{\lceil \log_2^{n_j} \rceil} + t$$

$$\text{where,}\ t \in [k,\ m]$$

The space overheads of the six approaches are listed in Table 6.7.

Dispatch mechanism	**Space overhead**
Naïve implementation	0
Searched itable	$\sum_{j=1}^{k} n_j + k$
IZone	$\sum_{j=1}^{k} n_j + t$, or, $\sum_{j=1}^{k} 2^{\lceil \log_2 {n_j} \rceil} + t$ $t \in [k,\ m]$
Directly indexed itable	$\sum_{j=1}^{k} n_j + m$
Embedded / indirect IMT-p	$3 \times (\sum_{j=1}^{k} n_j) + 2p$

Table 6.7: Space overhead comparison

The naïve implementation is the only approach with zero space overhead. The space overhead of searched itable is no greater than that of directly indexed itable. In real practice, k is far less than m ($k \ll m$), so, searched itable has quite lower space overhead than directly indexed itable.

IZone has different space overhead by using different formulas for offset calculation. If the formula (5.1) is used, the space overhead depends on the maximum id of the implemented interfaces: with the maximum id equal to the number of implemented interfaces minus one, i.e., the implemented interfaces having continuous ids starting from 0, IZone's space overhead is equal to that of searched itable; otherwise, it is higher than searched itable, but never exceeding that of directly indexed itable; only when the maximum id is $m - 1$, the space overhead of IZone is the same as directly indexed itable. On average, the space overhead of IZone is on halfway between that of searched itable and of directly indexed itable. If the formula (5.2) is used, it may use more space than needed in order to accelerate the offset calculation.

IMT has the highest space overhead, mainly due to the space consumed by the links in the IMT dictionary. IMT space overhead increases with the size p.

The comparison of the space overhead for different approaches illustrates the trade-off between space and execution time: if memory space is most concerned, execution time has to be sacrificed; on the other hand, in order to achieve faster execution, more memory space will be used. As computer memory becomes cheaper day after day, space is no longer a big issue for the decision of performance improvement.

Chapter 7

Future Work

Most of the approaches investigated in this thesis try to reduce the overhead of *invokeinterface* by introducing new data structure(s) into the method area. The new data structures facilitate interface dispatch, but lead to time and space overheads at the same time. Since Java programs are "compile once, run everywhere", the resultant overheads are unavoidable in every execution of interface methods.

A possible alternative is to optimize the Java compiler, so that, whenever an interface method is invoked upon an object reference whose type is known at compile time, it is translated into *invokevirtual* instruction instead of *invokeinterface* instruction, and the constant pool contains the symbolic reference to a virtual method instead of an interface method. To achieve this, the compiler should perform flow analysis and type checking before assembling the *invokevirtual* instruction. This compile-time virtualization conforms to the specifications of both the Java compiler and the Java Virtual Machine. Although it introduces some overhead to Java program compilation, it eliminates the time and space overhead demonstrated by the approaches examined in this thesis: interface invocation will be equivalently efficient as virtual method invocation. Considering the overheads avoided in every execution of method, the overhead introduced to the one-time compilation is insignificant.

For the IZone, itable and IMT approaches studied in this thesis, some of the entries in the dispatching data structures may not be ever used when the application exits, if the corresponding method has never been called by the program. Although it leads to unused memory space, it saves the lookup time as trade-off. How to keep the dispatching structure compact so that it only contains the methods to be called, while at the same time keeping the lookup equally efficient, is another possible future work of this research.

Chapter 8

Conclusion

This thesis studies the main source of inefficient interface invocation, overviews previous work on this problem, and proposes a new approach – IZone based interface invocation. IZone is a per-interface structure. It is built up gradually as the subclasses are being resolved. Based on the subclass resolution orders and interface method declaration orders, methods in IZone are arranged in special order for easy locating.

The IZone approach is evaluated together with five previous approaches: directly indexed itable, searched itable, embedded IMT, indirect IMT and naïve approaches. They are compared in terms of both time and space costs that are involved in interface method executions. IZone-based interface invocation proves efficient: with modest space cost, it is faster than three other approaches and slower than two approaches after baseline compilation; it is the fastest after lightweight optimizations, and the second fastest after heavyweight optimizations. IZone approach presents a new way for optimizing Java interface invocation, and disproves the long-time mistaken impression that Java interface invocation is inherently inefficient.

Appendix A

Modifications to Jikes RVM

Eight classes in Jikes RVM have been modified in order to support IZone-based interface invocation:

1. com.ibm.JikesRVM.VM_Configuration
2. com.ibm.JikesRVM.VM_TIBLayoutConstants
3. com.ibm.JikesRVM.VM_Entrypoints
4. com.ibm.JikesRVM.classloader.VM_Class
5. com.ibm.JikesRVM.classloader.VM_Method
6. com.ibm.JikesRVM.classloader.VM_InterfaceInvocation
7. com.ibm.JikesRVM.VM_Compiler (IA32 version)
8. com.ibm.JikesRVM.opt.OPT_ConvertToLowlevelIR

Part of the source code of the above eight classes are given below, with modification highlighted in italic and bold.

A.1 com.ibm.JikesRVM.VM_Configuration

This class defines configuration settings for Jikes RVM, such as the hardware architecture and operating system that the virtual machine is built upon, the compiler to be supported, the interface invocation approach to be used, etc. The public static fields *BuildForIMTInterfaceInvocation* and *BuildForITableInterfaceInvocation* are to indicate whether the IMT or itable approach is to be used for interface invocation.

A public field of boolean type, *BuildForIZoneInterfaceInvocation*, is added to the settings. When set to true, it instructs the virtual machine that IZone approach should be

used for interface invocation.

```
. . . . . .
public static final boolean BuildForIMTInterfaceInvocation = false;
public static final boolean BuildForIndirectIMT = true &&
                                        BuildForIMTInterfaceInvocation;
public static final boolean BuildForEmbeddedIMT = !BuildForIndirectIMT &&
                                        BuildForIMTInterfaceInvocation;

public static final boolean BuildForITableInterfaceInvocation = false;
public static final boolean DirectlyIndexedITables = true &&
                                        BuildForITableInterfaceInvocation;

public static final boolean BuildForIZoneInterfaceInvocation = true;

. . . . . .
```

A.2 com.ibm.JikesRVM.VM_TIBLayoutConstants

This class defines the format of a Type Information Block. An IZone pointer is inserted into the TIB, which occupies the same entry as the indirect IMT pointer or itable pointer. A new field of integer type, *TIB_IZONE_TIB_INDEX*, is added to to record the offset of the IZone pointer in the TIB. The value of the field *TIB_ITABLES_TIB_INDEX*, which gives the offset of the itable pointer, is changed accordingly after the IZone pointer is inserted.

```
// Number of slots reserved for interface method pointers.
static final int IMT_METHOD_SLOTS =
     VM.BuildForIMTInterfaceInvocation ? 100 : 0;

static final int TIB_INTERFACE_METHOD_SLOTS =
     VM.BuildForEmbeddedIMT ? IMT_METHOD_SLOTS : 0;

// First slot of tib points to VM_Type (slot 0 in above diagram).
static final int TIB_TYPE_INDEX = 0;
```

```
// A vector of ids for classes that this one extends.
// (see vm/classLoader/VM_DynamicTypeCheck.java)
static final int TIB_SUPERCLASS_IDS_INDEX = TIB_TYPE_INDEX + 1;

// "Does this class implement the ith interface?"
// (see vm/classLoader/VM_DynamicTypeCheck.java)
static final int TIB_DOES_IMPLEMENT_INDEX = TIB_SUPERCLASS_IDS_INDEX + 1;

// The TIB of the elements type of an array (may be null in fringe cases
// when element type couldn't be resolved during array resolution).
// Will be null when not an array.
static final int TIB_ARRAY_ELEMENT_TIB_INDEX = TIB_DOES_IMPLEMENT_INDEX + 1;

// If VM.BuildForIZoneInterfaceInvocation, then allocate 1 TIB entry to
// hold an IZone (this entry is only valid for an interface)
static final int TIB_IZONE_TIB_INDEX = TIB_DOES_IMPLEMENT_INDEX +
                         (VM.BuildForIZoneInterfaceInvocation ? 1 : 0);

// If VM.ITableInterfaceInvocation then allocate 1 TIB entry to hold
// an array of ITABLES
static final int TIB_ITABLES_TIB_INDEX = TIB_IZONE_TIB_INDEX +
                        (VM.BuildForITableInterfaceInvocation ? 1 : 0);

// If VM.BuildForIndirectIMT then allocate 1 TIB entry to hold a
// pointer to the IMT
static final int TIB_IMT_TIB_INDEX =
     TIB_ITABLES_TIB_INDEX + (VM.BuildForIndirectIMT ? 1 : 0);

// Next group of slots point to interface method code blocks
// (slots 1..K in above diagram).
static final int TIB_FIRST_INTERFACE_METHOD_INDEX = TIB_IMT_TIB_INDEX + 1;

// Next group of slots point to virtual method code blocks
// (slots K+1..K+N in above diagram).
static final int TIB_FIRST_VIRTUAL_METHOD_INDEX =
```

```
    TIB_FIRST_INTERFACE_METHOD_INDEX + TIB_INTERFACE_METHOD_SLOTS;

// Special value returned by VM_ClassLoader.getFieldOffset() or
// VM_ClassLoader.getMethodOffset() to indicate fields or methods that
// must be accessed via dynamic linking code because their offset is not
// yet known or the class's static initializer has not yet been run.
//
// We choose a value that will never match a valid jtoc-, instance-,
// or virtual method table- offset. Zero is a good value because:
//      slot 0 of jtoc is never used
//      instance field offsets are always negative w.r.t. object pointer
//      virtual method offsets are always positive w.r.t. TIB pointer
//      0 is a "free" (default) data initialization value
//
public static final int NEEDS_DYNAMIC_LINK = 0;
```

A.3 com.ibm.JikesRVM.VM_Entrypoints

This class defines the fields and methods of the virtual machine that are needed by compiler-generated machine code.

A new field, *invokeInterfaceIZoneOnlyMethod*, is introduced to this class to facilitate the invocation of the method *invokeInterface_IZoneOnly*. This method is declared in the class ***com.ibm.JikesRVM.classloader.VM_InterfaceInvocation***, and called from the class ***com.ibm.JikesRVM.classloader.VM_Compiler*** by its method *emit_invokeinterface*.

```
. . . . . .

public static final VM_Method invokeInterfaceIZoneOnlyMethod =
    getMethod("Lcom/ibm/JikesRVM/classloader/VM_InterfaceInvocation;",
              "invokeInterface_IZoneOnly",
              "(Ljava/lang/Object;II)Lcom/ibm/JikesRVM/VM_CodeArray;");

. . . . . .
```

A.4 com.ibm.JikesRVM.classloader.VM_Class

This class represents a Java "class" or "interface" type. It provides functions for class loading, resolution, instantiation, initialization, etc.

The class loading function is revised so that the methods declared by the current class or interface are assigned declaration orders when it is being loaded.

A resolution order dictionary (int orderDict[]) is used to record the resolution orders of the class represented by the current ***VM_Class*** instance. An integral field *nextOrder* is used for resolution order assignment. The resolution order dictionary is valid with a class type, and the *nextOrder* field is valid only with an interface type. Some supporting functions are added to manage and retrieve order information in/from the order dictinary.

```
. . . . . .

VM_Class(VM_TypeReference typeRef, DataInputStream input)
    throws ClassFormatError, IOException {
    . . . . . .
    int numMethods = input.readUnsignedShort();

    setAdjustedNDeclaredMethodsLg(numMethods);

    if (numMethods == 0) {
      declaredMethods = emptyVMMethod;
    } else {
      declaredMethods = new VM_Method[numMethods];
      for (int i = 0; i<numMethods; i++) {
        int mmodifiers = input.readUnsignedShort();
        VM_Atom methodName = getUtf(input.readUnsignedShort());
        VM_Atom methodDescriptor = getUtf(input.readUnsignedShort());
        VM_MemberReference memRef = VM_MemberReference.findOrCreate(
                                   typeRef, methodName, methodDescriptor);
        VM_Method method =
                    VM_Method.readMethod(this, memRef, mmodifiers, input);

        method.setDeclaringOrder(i);
```

```
        declaredMethods[i] = method;
        if (method.isClassInitializer())
          classInitializerMethod = method;
      }
    }
    . . . . . .
}

private int nextOrder = 1; // next order to be assigned to a subclass
                            // valid only when this is an interface

// the initial size of an IZone = (IZONE_INITIAL_TIMES) *
// (# of method declared by the interface), i.e., the initial size can
// contain the implementation lookup areas of (IZONE_INITIAL_TIMES)
// classes.
public static final int IZONE_INITIAL_TIMES = 1;

public final synchronized int getNextOrder()  {
  return nextOrder++;
}


/* the following field and methods are valid only when this is a class,
   not an interface
*/

// superinterfaces' view of this class' order; the order is indexed by
// interface id, i.e., orderDict[i] is the interface (with id = i)'s
// view on this class' order. The initial value of the array elements
// is 0; valid order begins from 1.
private int orderDict[] = new int[100];

public int getOrderFromDict(VM_Class ainterface)  {
  int id = ainterface.getInterfaceId();
  return getOrderFromDict(id);
}
public int getOrderFromDict(int interfaceId)  {
```

```
  if(interfaceId < orderDict.length) return orderDict[interfaceId];
  else return 0;
}
public void addOrderToDict(VM_Class ainterface, int order)  {
  int id = ainterface.getInterfaceId();
  if(id >= orderDict.length) {
    int[] tmp = new int[interfaces.length];
    System.arraycopy(orderDict, 0, tmp, 0, orderDict.length);
    orderDict = tmp;
  }
  orderDict[id] = order;
}

// the smallest integer that is not less than the logarithm (base 2)
// of # declared methods; -1 reprsents a invalid value.
private int adjustedNDeclaredMethodsLg = -1;

// Set the value of adjustedNDeclaredMethodsLg
// parameter num: # of methods declared by this class
// if num = 0, keep adjustedNDeclaredMethodsLg as -1
public void setAdjustedNDeclaredMethodsLg(int num) {
  if (num > 0) adjustedNDeclaredMethodsLg =
                        (int) Math.ceil( Math.log(num) / Math.log(2) );
}

public int getAdjustedNDeclaredMethodsLg() {
  return adjustedNDeclaredMethodsLg;
}

. . . . . .
```

A.5 com.ibm.JikesRVM.classloader.VM_Method

This class represents the Java methods. It provides functions for method resolution and compilation.

One more field, *declaringOrder*, is added to record the current method's declaration order by its declaring class or interface. Two functions are added to set or get the value of

the declaration order.

```
. . . . . .

protected VM_Method(VM_Class declaringClass, VM_MemberReference memRef,
                    int modifiers, VM_TypeReference[] exceptionTypes) {
  super(declaringClass, memRef, modifiers & APPLICABLE_TO_METHODS);
  memRef.asMethodReference().setResolvedMember(this);
  this.exceptionTypes = exceptionTypes;

  this.declaringOrder = -1; // -1 is invalid value; set to valid value
                            // during class loading

}

// the order that the declaring class declares this method;
// the first declared method has order 0; the secondly has order 1
private int declaringOrder;

public void setDeclaringOrder(int order) {
  this.declaringOrder = order;
}
public int getDeclaringOrder() {
  return this.declaringOrder;
}

. . . . . .
```

A.6 com.ibm.JikesRVM.classloader.VM_InterfaceInvocation

This class defines runtime system mechanisms and data structures to implement interface invocation.

A few functions are added to this class for IZone construction, implementation lookup area installation, IZone based interface invocation, and IZone entry updataing.

The old implementation of the function *getITableIndex* is replaced by a more efficient one. Rather than searching the itable to determine an interface method's offset, the new implementation calculates the offset using the method declaration order.

```
. . . . . . .
public static VM_CodeArray invokeInterface(Object target, int mid)
    throws IncompatibleClassChangeError {
    VM_MethodReference mref = VM_MemberReference.getMemberRef(mid).
                                                  asMethodReference();
    VM_Method sought = mref.resolveInterfaceMethod();
    VM_Class I = sought.getDeclaringClass();
    VM_Class C = VM_Magic.getObjectType(target).asClass();

    if (VM.BuildForIZoneInterfaceInvocation) {
       Object[] tib_i = I.getTypeInformationBlock();
       Object[] iZone = (Object[])tib_i[TIB_IZONE_TIB_INDEX];
       return (VM_CodeArray)iZone[getIndexInIZone(C.getOrderFromDict(I),
       I.getAdjustedNDeclaredMethodsLg(), sought.getDeclaringOrder())];
    }
    else if (VM.BuildForITableInterfaceInvocation) {
      Object[] tib_c = C.getTypeInformationBlock();
      Object[] iTable = findITable(tib_c, I.getInterfaceId());
      return (VM_CodeArray)iTable[getITableIndex(sought)];
    }

    else {          // naive implementation
      if (!VM_Runtime.isAssignableWith(I, C))
                         throw new IncompatibleClassChangeError();
      VM_Method found  = C.findVirtualMethod(sought.getName(),
                                                  sought.getDescriptor());
      if (found == null) throw new IncompatibleClassChangeError();
      return found.getCurrentInstructions();
    }
  }

public static VM_CodeArray invokeInterface_IZoneOnly(Object target,
                             int declaringInterfaceId, int mOrder) {
  VM_Class C = VM_Magic.getObjectType(target).asClass();
  VM_Class I = VM_Class.getInterface(declaringInterfaceId);
  Object[] tib_i = I.getTypeInformationBlock();
```

```
  Object[] iZone = (Object[])tib_i[TIB_IZONE_TIB_INDEX];
  int idx = getIndexInIZone(C.getOrderFromDict(declaringInterfaceId),
                            I.getAdjustedNDeclaredMethodsLg(), mOrder);
  return (VM_CodeArray)iZone[idx];
}

public static void initializeDispatchStructures(VM_Class klass) {
  // if klass is abstract, we'll never use the dispatching structures.
  if (klass.isAbstract()) return;
  VM_Class[] interfaces = klass.getAllImplementedInterfaces();
  if (interfaces.length == 0) return;

  if (VM.BuildForIMTInterfaceInvocation) {
      IMTDict d = buildIMTDict(klass, interfaces);
      if (VM.BuildForEmbeddedIMT) {
          populateEmbeddedIMT(klass, d);
      } else {
          populateIndirectIMT(klass, d);
      }
  }
  else if (VM.DirectlyIndexedITables) {
      populateITables(klass, interfaces);
  }

  else if (VM.BuildForIZoneInterfaceInvocation) {
      populateIZones(klass, interfaces);
  }

}

/**
 * Populate IZones for all interfaces that "this" class implements
 * @param klass the klass that implements some interfaces
 * @param super_interfaces the interfaces that klass implements
 */
private static void populateIZones(VM_Class klass,
```

```
                                             VM_Class[] super_interfaces) {
   for (int i = 0; i < super_interfaces.length; i ++) {
      synchronized (super_interfaces[i]) {
        installIZone(klass, super_interfaces[i]);
      }
    }
}

/**
 * Install IZone for an interface that "this" class implements
 * @param klass the klass that implements some interfaces
 * @param super_interface an interface that klass implements
 */
private static void installIZone(VM_Class klass,
                                 VM_Class super_interface) {
  int lg = super_interface.getAdjustedNDeclaredMethodsLg();
  // no IZone will be created if the interface declares no method
  if (lg == -1) return;

  int cOrder = klass.getOrderFromDict(super_interface);
  if(cOrder != 0)  return; // the order is already in the dictionary

  // add order into the dictionary
  cOrder = super_interface.getNextOrder();
  klass.addOrderToDict(super_interface, cOrder);

  Object tib[] = super_interface.getTypeInformationBlock();
  // iZone is associated with super_interface, not klass

  Object iZone[] = (Object[])tib[TIB_IZONE_TIB_INDEX];
  int iZoneInitSize = (1 << lg) * VM_Class.IZONE_INITIAL_TIMES;
  if (iZone == null) {  // initialize IZone if it is null
      iZone = new Object[iZoneInitSize];
      tib[TIB_IZONE_TIB_INDEX] = iZone;
  } else if ( (cOrder << lg) > iZone.length ) {
      // expand IZone if needed
      Object tmp[] = new Object[iZone.length + iZoneInitSize];
```

```
      System.arraycopy(iZone, 0, tmp, 0, iZone.length);
      iZone = tmp;
      tib[TIB_IZONE_TIB_INDEX] = iZone;
  }

  // create klass' lookup area in super_interface's IZone
  VM_Method iMethods[] = super_interface.getDeclaredMethods();
  for (int i = 0; i < iMethods.length; i ++) {
      VM_Method im = iMethods[i];
      if (im.isClassInitializer()) continue;
      if (VM.VerifyAssertions)
                     VM._assert(im.isPublic() && im.isAbstract());

      VM_Method vm = klass.findVirtualMethod(im.getName(),
                                             im.getDescriptor());
      if (vm == null || vm.isAbstract()) {
          vm = VM_Entrypoints.raiseAbstractMethodError;
      } else if (!vm.isPublic()) {
          vm = VM_Entrypoints.raiseIllegalAccessError;
      }

      int indexInIZone = getIndexInIZone(cOrder, lg, i);
      if (vm.isStatic()) {
          vm.compile();
          iZone[indexInIZone] = vm.getCurrentInstructions();
      } else {
          Object tib_c[] = klass.getTypeInformationBlock();
          iZone[indexInIZone] =
            (VM_CodeArray)tib_c[vm.getOffset()>>LOG_BYTES_IN_ADDRESS];
      }
  }
}

/**
 * @param cOrder the resolution order of the class which implements
 *               this interface method
 * @param adjustedNMethodsLg the smallest integer that is no less than
```

```
 * the log (base 2) of # of methods that the interface has declared
 * @param mOrder the interface method's declaring order
 * @return the interface method's index in the IZone
 */
private static int getIndexInIZone(int cOrder, int adjustedNMethodsLg,
                                   int mOrder) {
  return ( (cOrder-1) << adjustedNMethodsLg ) + mOrder;
}

/**
 * Return the index of the interface method im in the itable
 * Using declaring order makes this method more efficient.
 */
public static int getITableIndex(VM_Method im) {
  if (VM.VerifyAssertions)
              VM._assert(VM.BuildForITableInterfaceInvocation);
  if (VM.VerifyAssertions)
              VM._assert(im.getDeclaringClass().isInterface());
  return im.getDeclaringOrder() + 1;
}

/**
 * If there is an an IZone or IMT or ITable entry that contains compiled
 * code for the argument method, then update it to contain the current
 * compiled code for the method.
 *
 * @param klass the VM_Class who's IMT/ITable is being reset, or who
 *              implements an interface method by declaring m (IZone)
 * @param m the method that needs to be updated, klass' virtual method.
 *
 * This method is only called by VM_Class.updateTIBEntry(VM_Method m)
 */
public static void updateTIBEntry(VM_Class klass, VM_Method m) {
  if (VM.BuildForIZoneInterfaceInvocation) {
      if (klass.isAbstract()) return;
      VM_Class[] interfaces = klass.getAllImplementedInterfaces();
```

```
    for (int i = 0; i < interfaces.length; i ++) {
        VM_Class itf = interfaces[i];
        VM_Method iMethod = itf.findDeclaredMethod(m.getName(),
                                                  m.getDescriptor());
        if(iMethod == null) continue;
        int index = getIndexInIZone(klass.getOrderFromDict(itf),
                              itf.getAdjustedNDeclaredMethodsLg(),
                              iMethod.getDeclaringOrder());
        Object[] tib = itf.getTypeInformationBlock();
        Object[] iZone = (Object[])tib[TIB_IZONE_TIB_INDEX];
        iZone[index] = m.getCurrentInstructions();
    }
}

else if (VM.BuildForIMTInterfaceInvocation) {
    Object[] tib = klass.getTypeInformationBlock();
    VM_Method[] map = klass.noIMTConflictMap;
    if (map != null) {
      for (int i = 0; i<IMT_METHOD_SLOTS; i++) {
        if (map[i] == m) {
          if (VM.BuildForIndirectIMT) {
              VM_CodeArray[] IMT =
                          (VM_CodeArray[])tib[TIB_IMT_TIB_INDEX];
              IMT[i] = m.getCurrentInstructions();
          } else {
              tib[i+TIB_FIRST_INTERFACE_METHOD_INDEX] =
                                     m.getCurrentInstructions();
          }
          return; // all done -- a method is in at most 1 IMT slot
        }
      }
    }
}
else if (VM.BuildForITableInterfaceInvocation) {
    Object[] tib = klass.getTypeInformationBlock();
```

```
      if (tib[TIB_ITABLES_TIB_INDEX] != null) {
        Object[] iTables = (Object[])tib[TIB_ITABLES_TIB_INDEX];
        VM_Atom name = m.getName();
        VM_Atom desc = m.getDescriptor();
        for (int i=0; i<iTables.length; i++) {
          Object[] iTable = (Object[])iTables[i];
          if (iTable != null) {
            VM_Class I = (VM_Class)iTable[0];
            VM_Method [] interfaceMethods = I.getDeclaredMethods();
            for (int j=0; j<interfaceMethods.length; j++) {
              VM_Method im = interfaceMethods[j];
              if (im.getName() == name && im.getDescriptor() == desc) {
                iTable[getITableIndex(im)] = m.getCurrentInstructions();
              }
            }
          }
        }
      }
  }
}
. . . . . .
```

A.7 com.ibm.JikesRVM.VM_Compiler (IA32 version)

This is the baseline compiler class for the IA32 architecture. It translates bytecode instructions into IA32 machine code.

The method *emit_invokeinterface* is modified so that it is able to translate the bytecode instruction *invokeinterface* into the machine code that are specific to the IZone approach.

```
. . . . . .
protected final void emit_invokeinterface(VM_MethodReference methodRef) {
  . . . . . .
  // (2) Emit interface invocation sequence.
```

```
  if (VM.BuildForIZoneInterfaceInvocation) {
    if (resolvedMethod == null) {
      int methodRefId = methodRef.getId();
      asm.emitPUSH_RegDisp(SP, (count-1)<<LG_WORDSIZE);
                                          // "this" parameter is obj
      asm.emitPUSH_Imm(methodRefId);      // id of method to call
      genParameterRegisterLoad(2);        // pass 2 parameter words
      asm.emitCALL_RegDisp(JTOC,
             VM_Entrypoints.invokeInterfaceMethod.getOffset());
                  // invokeinterface(obj, id) returns address to call
      asm.emitMOV_Reg_Reg (S0, T0);       // S0 has address of method
      genParameterRegisterLoad(methodRef, true);
      asm.emitCALL_Reg(S0);               // the interface method
    } else {
      int declaringInterfaceId =
                  resolvedMethod.getDeclaringClass().getInterfaceId();
      int mOrder = resolvedMethod.getDeclaringOrder();
      asm.emitPUSH_RegDisp(SP, (count-1)<<LG_WORDSIZE);
      asm.emitPUSH_Imm(declaringInterfaceId);
                                    // id of the declaring interface
      asm.emitPUSH_Imm(mOrder);     // declaration order
      genParameterRegisterLoad(3);  // pass 2 parameter words
      asm.emitCALL_RegDisp(JTOC,
           VM_Entrypoints.invokeInterfaceIZoneOnlyMethod.getOffset());
                   // call invokeinterface_IZoneOnly(obj, id, morder)
      asm.emitMOV_Reg_Reg (S0, T0);
      genParameterRegisterLoad(methodRef, true);
      asm.emitCALL_Reg(S0);
    }
  }

  else if (VM.BuildForIMTInterfaceInvocation) {
      . . . . . .
  }
  . . . . . .
}
. . . . . .
```

A.8 com.ibm.JikesRVM.opt.OPT_ConvertToLowlevelIR

This class is to convert the instructions with HIR-only operators into an equivalent sequence of LIR operators. The method *callHelper* is modified so that it could generate LIR instructions that are specific to the IZone approach.

```
. . . . . .
static OPT_Instruction callHelper(OPT_Instruction v, OPT_IR ir) {
  . . . . . .

  if (VM.BuildForIZoneInterfaceInvocation) {
    OPT_RegisterOperand realAddrReg =
                        ir.regpool.makeTemp(VM_TypeReference.CodeArray);
    VM_Method target;
    OPT_Instruction vp;
    if( ! methOp.hasTarget() ) {
      // Create an instruction of the Call instruction format
      // with 2 variable arguments
      target = VM_Entrypoints.invokeInterfaceMethod;
      vp = Call.create2(CALL, realAddrReg, I(target.getOffset()),
                         OPT_MethodOperand.STATIC(target),
                         Call.getParam(v, 0).asRegister().copyU2U(),
                         I(methOp.getMemberRef().getId())  );

    } else {
      target = VM_Entrypoints.invokeInterfaceIZoneOnlyMethod;
      vp = Call.create3(CALL, realAddrReg, I(target.getOffset()),
            OPT_MethodOperand.STATIC(target),
            Call.getParam(v, 0).asRegister().copyU2U(),
            I(methOp.getTarget().getDeclaringClass().getInterfaceId()),
            I(methOp.getTarget().getDeclaringOrder()) );
    }
    vp.position = v.position;
    vp.bcIndex = RUNTIME_SERVICES_BCI;
    // Insert vp immediately before v in the instruction stream
    v.insertBack(vp);
    callHelper(vp, ir);
    Call.setAddress(v, realAddrReg.copyD2U());
```

```
  } else if (VM.BuildForIMTInterfaceInvocation) {
    . . . . . .
  }
  . . . . . .
}
. . . . . .
```

Appendix B

XML Files

Implementation correctness is verified by running Xerces-J Parser 1.4.4 on the Jikes RVM versions that adopt one of the six interface invocation approaches. XML files are feed to the DOM and SAX parsers and the outputs are compared with that generated by the standard Sun JVM.

We only show one file *allelements.xml* here. This file stores the periodic table in XML format. It is archived on the website *ibiblio.org*. It has 110 kilobytes in total.


```
<?xml version="1.0"?>
<PERIODIC_TABLE>

  <ATOM>
    <NAME>Actinium</NAME>
    <ATOMIC_WEIGHT>227</ATOMIC_WEIGHT>
    <ATOMIC_NUMBER>89</ATOMIC_NUMBER>
    <OXIDATION_STATES>3</OXIDATION_STATES>
    <BOILING_POINT UNITS="Kelvin">3470</BOILING_POINT>
    <SYMBOL>Ac</SYMBOL>
    <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
      10.07
    </DENSITY>
    <ELECTRON_CONFIGURATION>[Rn] 6d1 7s2 </ELECTRON_CONFIGURATION>
    <ELECTRONEGATIVITY>1.1</ELECTRONEGATIVITY>
```

```
  <ATOMIC_RADIUS UNITS="Angstroms">1.88</ATOMIC_RADIUS>
  <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
    22.5
  </ATOMIC_VOLUME>
  <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
    0.12
  </SPECIFIC_HEAT_CAPACITY>
  <IONIZATION_POTENTIAL>5.17</IONIZATION_POTENTIAL>
  <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
  <!-- At 300K -->
    12
  </THERMAL_CONDUCTIVITY>
</ATOM>

<ATOM>
  <NAME>Aluminum</NAME>
  <ATOMIC_WEIGHT>26.98154</ATOMIC_WEIGHT>
  <ATOMIC_NUMBER>13</ATOMIC_NUMBER>
  <OXIDATION_STATES>3</OXIDATION_STATES>
  <BOILING_POINT UNITS="Kelvin">2740</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">933.5</MELTING_POINT>
  <SYMBOL>Al</SYMBOL>
  <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
    2.7
  </DENSITY>
  <ELECTRON_CONFIGURATION>[Ne] 3s2 p1 </ELECTRON_CONFIGURATION>
  <COVALENT_RADIUS UNITS="Angstroms">1.18</COVALENT_RADIUS>
  <ELECTRONEGATIVITY>1.61</ELECTRONEGATIVITY>
  <ATOMIC_RADIUS UNITS="Angstroms">1.43</ATOMIC_RADIUS>
  <HEAT_OF_VAPORIZATION UNITS="kilojoules/mole">
    290.8
  </HEAT_OF_VAPORIZATION>
  <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
```

```
    10
  </ATOMIC_VOLUME>
  <HEAT_OF_FUSION UNITS="kilojoules/mole">
    10.7
  </HEAT_OF_FUSION>
  <IONIZATION_POTENTIAL>5.986</IONIZATION_POTENTIAL>
  <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
    0.9
  </SPECIFIC_HEAT_CAPACITY>
  <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
  <!-- At 300K -->
    237
  </THERMAL_CONDUCTIVITY>
</ATOM>

......
......

<ATOM STATE="GAS">
  <NAME>Argon</NAME>
  <ATOMIC_WEIGHT>39.948</ATOMIC_WEIGHT>
  <ATOMIC_NUMBER>18</ATOMIC_NUMBER>
  <BOILING_POINT UNITS="Kelvin">87.45</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">83.95</MELTING_POINT>
  <SYMBOL>Ar</SYMBOL>
  <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
    1.784
  </DENSITY>
  <ELECTRON_CONFIGURATION>[Ne] 3s2 p6 </ELECTRON_CONFIGURATION>
  <COVALENT_RADIUS UNITS="Angstroms">0.98</COVALENT_RADIUS>
  <ELECTRONEGATIVITY>0</ELECTRONEGATIVITY>
  <ATOMIC_RADIUS UNITS="Angstroms">0.88</ATOMIC_RADIUS>
  <HEAT_OF_VAPORIZATION UNITS="kilojoules/mole">
```

```
    6.506
  </HEAT_OF_VAPORIZATION>
  <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
    24.2
  </ATOMIC_VOLUME>
  <HEAT_OF_FUSION UNITS="kilojoules/mole">
    1.188
  </HEAT_OF_FUSION>
  <IONIZATION_POTENTIAL>15.759</IONIZATION_POTENTIAL>
  <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
    0.52
  </SPECIFIC_HEAT_CAPACITY>
  <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
  <!-- At 300K -->
    0.0177
  </THERMAL_CONDUCTIVITY>
</ATOM>

......
......

<ATOM>
  <NAME>Zinc</NAME>
  <ATOMIC_WEIGHT>65.39</ATOMIC_WEIGHT>
  <ATOMIC_NUMBER>30</ATOMIC_NUMBER>
  <OXIDATION_STATES>2</OXIDATION_STATES>
  <BOILING_POINT UNITS="Kelvin">1180</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">692.73</MELTING_POINT>
  <SYMBOL>Zn</SYMBOL>
  <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
    7.13
  </DENSITY>
  <ELECTRON_CONFIGURATION>[Ar] 3d10 4s2 </ELECTRON_CONFIGURATION>
```

```
  <COVALENT_RADIUS UNITS="Angstroms">1.25</COVALENT_RADIUS>
  <ELECTRONEGATIVITY>1.65</ELECTRONEGATIVITY>
  <ATOMIC_RADIUS UNITS="Angstroms">1.38</ATOMIC_RADIUS>
  <HEAT_OF_VAPORIZATION UNITS="kilojoules/mole">
    115.3
  </HEAT_OF_VAPORIZATION>
  <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
    9.2
  </ATOMIC_VOLUME>
  <HEAT_OF_FUSION UNITS="kilojoules/mole">
    7.38
  </HEAT_OF_FUSION>
  <IONIZATION_POTENTIAL>9.394</IONIZATION_POTENTIAL>
  <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
    0.388
  </SPECIFIC_HEAT_CAPACITY>
  <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
  <!-- At 300K -->
    116
  </THERMAL_CONDUCTIVITY>
</ATOM>

<ATOM>
  <NAME>Zirconium</NAME>
  <ATOMIC_WEIGHT>91.224</ATOMIC_WEIGHT>
  <ATOMIC_NUMBER>40</ATOMIC_NUMBER>
  <OXIDATION_STATES>4</OXIDATION_STATES>
  <BOILING_POINT UNITS="Kelvin">4682</BOILING_POINT>
  <MELTING_POINT UNITS="Kelvin">2128</MELTING_POINT>
  <SYMBOL>Zr</SYMBOL>
  <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
    6.51
  </DENSITY>
```

```
    <ELECTRON_CONFIGURATION>[Kr] 4d2 5s2 </ELECTRON_CONFIGURATION>
    <COVALENT_RADIUS UNITS="Angstroms">1.45</COVALENT_RADIUS>
    <ELECTRONEGATIVITY>1.33</ELECTRONEGATIVITY>
    <ATOMIC_RADIUS UNITS="Angstroms">1.6</ATOMIC_RADIUS>
    <HEAT_OF_VAPORIZATION UNITS="kilojoules/mole">
      590.5
    </HEAT_OF_VAPORIZATION>
    <ATOMIC_VOLUME UNITS="cubic centimeters/mole">
      14.1
    </ATOMIC_VOLUME>
    <HEAT_OF_FUSION UNITS="kilojoules/mole">
      21
    </HEAT_OF_FUSION>
    <IONIZATION_POTENTIAL>6.84</IONIZATION_POTENTIAL>
    <SPECIFIC_HEAT_CAPACITY UNITS="Joules/gram/degree Kelvin">
      0.278
    </SPECIFIC_HEAT_CAPACITY>
    <THERMAL_CONDUCTIVITY UNITS="Watts/meter/degree Kelvin">
    <!-- At 300K -->
      22.7
    </THERMAL_CONDUCTIVITY>
  </ATOM>

</PERIODIC_TABLE>
```

Appendix C

Artificial Test Cases

C.1 Interface Invocation: Test Case 1

Test case 1 defines 20 interfaces ***MyInterface1~20***, and two classes ***MyClassA*** and ***MyClassB***. ***MyInterface1~20*** are the direct superinterfaces for both classes. Each interface declares only one method, which is implemented as a trivial call by both classes.

```
interface MyInterface1 {
  public int a0();
}

interface MyInterface2 {
  public int b0();
}

interface MyInterface3 {
  public int c0();
}

interface MyInterface4 {
  public int d0();
}
```

```
interface MyInterface5 {
  public int e0();
}

interface MyInterface6 {
  public int f0();
}

interface MyInterface7 {
  public int g0();
}

interface MyInterface8 {
  public int h0();
}

interface MyInterface9 {
  public int i0();
}

interface MyInterface10 {
  public int j0();
}

interface MyInterface11 {
  public int m0();
}

interface MyInterface12 {
  public int n0();
}

interface MyInterface13 {
```

```
  public int p0();
}

interface MyInterface14 {
  public int q0();
}

interface MyInterface15 {
  public int r0();
}

interface MyInterface16 {
  public int s0();
}

interface MyInterface17 {
  public int t0();
}

interface MyInterface18 {
  public int u0();
}

interface MyInterface19 {
  public int v0();
}

interface MyInterface20 {
  public int w0();
}


class MyClassA implements MyInterface1, MyInterface2, MyInterface3,
```

```
          MyInterface4, MyInterface5, MyInterface6, MyInterface7,
          MyInterface8, MyInterface9, MyInterface10, MyInterface11,
          MyInterface12, MyInterface13, MyInterface14, MyInterface15,
          MyInterface16, MyInterface17, MyInterface18, MyInterface19,
          MyInterface20  {
public boolean b = false;

public int a0()  { return 0;}
public int b0()  { return 1;}
public int c0()  { return 2;}
public int d0()  { return 3;}
public int e0()  { return 4;}
public int f0()  { return 5;}
public int g0()  { return 6;}
public int h0()  { return 7;}
public int i0()  { return 8;}
public int j0()  { return 9;}

public int m0()  { return 10;}
public int n0()  { return 11;}
public int p0()  { return 12;}
public int q0()  { return 13;}
public int r0()  { return 14;}
public int s0()  { return 15;}
public int t0()  { return 16;}
public int u0()  { return 17;}
public int v0()  { return 18;}
public int w0()  { return 19;}

public void mya0() {System.out.println("0");}
public void mya1() {System.out.println("1");}
public void mya2() {System.out.println("2");}
public void mya3() {System.out.println("3");}
```

```
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
         mya0(); mya1(); mya2(); mya3(); mya4();
         mya5(); mya6(); mya7(); mya8(); mya9();
         return 100;
  }
}

class MyClassB implements MyInterface20, MyInterface19, MyInterface18,
            MyInterface17, MyInterface16, MyInterface15, MyInterface14,
            MyInterface13, MyInterface12, MyInterface11, MyInterface10,
            MyInterface9, MyInterface8, MyInterface7, MyInterface6,
            MyInterface5, MyInterface4, MyInterface3, MyInterface2,
            MyInterface1  {
  public boolean b = false;

  public int w0()  { return 119;}
  public int v0()  { return 118;}
  public int u0()  { return 117;}
  public int t0()  { return 116;}
  public int s0()  { return 115;}
  public int r0()  { return 114;}
  public int q0()  { return 113;}
  public int p0()  { return 112;}
  public int n0()  { return 111;}
  public int m0()  { return 110;}

  public int j0()  { return 109;}
```

```
  public int i0()  { return 108;}
  public int h0()  { return 107;}
  public int g0()  { return 106;}
  public int f0()  { return 105;}
  public int e0()  { return 104;}
  public int d0()  { return 103;}
  public int c0()  { return 102;}
  public int b0()  { return 101;}
  public int a0()  { return 100;}

  public void mya0() {System.out.println("0");}
  public void mya1() {System.out.println("1");}
  public void mya2() {System.out.println("2");}
  public void mya3() {System.out.println("3");}
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
         mya0(); mya1(); mya2(); mya3(); mya4();
         mya5(); mya6(); mya7(); mya8(); mya9();
         return 100;
  }
}

public class Test1 {
  public static void main(String[] args)  {
    Test1 mytest = new Test1();
    mytest.aaa(Integer.parseInt(args[0]));
  }
```

```
public void aaa(int rounds) {
  Object classArray[] = new Object[2];
  MyClassA myA = new MyClassA(); myA.b = true;
  MyClassB myB = new MyClassB(); myB.b = true;
  classArray[0] = myA;
  classArray[1] = myB;

  MyInterface1  in1;  MyInterface2  in2;  MyInterface3  in3;
  MyInterface4  in4;  MyInterface5  in5;  MyInterface6  in6;
  MyInterface7  in7;  MyInterface8  in8;  MyInterface9  in9;
  MyInterface10 in10; MyInterface11 in11; MyInterface12 in12;
  MyInterface13 in13; MyInterface14 in14; MyInterface15 in15;
  MyInterface16 in16; MyInterface17 in17; MyInterface18 in18;
  MyInterface19 in19; MyInterface20 in20;

  long times[] = new long[10];

  for (int i = 0; i < times.length; i ++) {
    long result = 0;
    long time_1 = System.currentTimeMillis();
    for (int j = 0; j < rounds; j ++) {
      Object ob = classArray[j%2];
      in1  = (MyInterface1) ob;   in2  = (MyInterface2) ob;
      in3  = (MyInterface3) ob;   in4  = (MyInterface4) ob;
      in5  = (MyInterface5) ob;   in6  = (MyInterface6) ob;
      in7  = (MyInterface7) ob;   in8  = (MyInterface8) ob;
      in9  = (MyInterface9) ob;   in10 = (MyInterface10) ob;
      in11 = (MyInterface11) ob;  in12 = (MyInterface12) ob;
      in13 = (MyInterface13) ob;  in14 = (MyInterface14) ob;
      in15 = (MyInterface15) ob;  in16 = (MyInterface16) ob;
      in17 = (MyInterface17) ob;  in18 = (MyInterface18) ob;
      in19 = (MyInterface19) ob;  in20 = (MyInterface20) ob;
```

```
        result += in1.a0(); result += in2.b0(); result += in3.c0();
        result += in4.d0(); result += in5.e0(); result += in6.f0();
        result += in7.g0(); result += in8.h0(); result += in9.i0();
        result += in10.j0(); result += in11.m0(); result += in12.n0();
        result += in13.p0(); result += in14.q0(); result += in15.r0();
        result += in16.s0(); result += in17.t0(); result += in18.u0();
        result += in19.v0(); result += in20.w0();
      }

      long time_2 = System.currentTimeMillis();
      times[i] = time_2 - time_1;
      System.out.print(result);
    }
  }
}
```

C.2 Interface Invocation: Test Case 2

Test case 2 defines 20 interfaces ***MyInterface1∼20***, and two classes ***MyClassA*** and ***MyClassB***. ***MyInterface1∼20*** are the direct superinterfaces for both classes. Each interface declares only one method, which is implemented as a normal call by both classes.

```
interface MyInterface1 {
  public int a0();
}

interface MyInterface2 {
  public int b0();
}

interface MyInterface3 {
  public int c0();
}
```

```
interface MyInterface4 {
  public int d0();
}

interface MyInterface5 {
  public int e0();
}

interface MyInterface6 {
  public int f0();
}

interface MyInterface7 {
  public int g0();
}

interface MyInterface8 {
  public int h0();
}

interface MyInterface9 {
  public int i0();
}

interface MyInterface10 {
  public int j0();
}

interface MyInterface11 {
  public int m0();
}

interface MyInterface12 {
```

```
  public int n0();
}

interface MyInterface13 {
  public int p0();
}

interface MyInterface14 {
  public int q0();
}

interface MyInterface15 {
  public int r0();
}

interface MyInterface16 {
  public int s0();
}

interface MyInterface17 {
  public int t0();
}

interface MyInterface18 {
  public int u0();
}

interface MyInterface19 {
  public int v0();
}

interface MyInterface20 {
  public int w0();
```

```
}


class MyClassA implements MyInterface1, MyInterface2, MyInterface3,
            MyInterface4, MyInterface5, MyInterface6, MyInterface7,
            MyInterface8, MyInterface9, MyInterface10, MyInterface11,
            MyInterface12, MyInterface13, MyInterface14, MyInterface15,
            MyInterface16, MyInterface17, MyInterface18, MyInterface19,
            MyInterface20  {
  public boolean b = false;

  public int a0()  { if(b) return 0;  else return myb0();}
  public int b0()  { if(b) return 1;  else return myb0();}
  public int c0()  { if(b) return 2;  else return myb0();}
  public int d0()  { if(b) return 3;  else return myb0();}
  public int e0()  { if(b) return 4;  else return myb0();}
  public int f0()  { if(b) return 5;  else return myb0();}
  public int g0()  { if(b) return 6;  else return myb0();}
  public int h0()  { if(b) return 7;  else return myb0();}
  public int i0()  { if(b) return 8;  else return myb0();}
  public int j0()  { if(b) return 9;  else return myb0();}

  public int m0()  { if(b) return 10;  else return myb0();}
  public int n0()  { if(b) return 11;  else return myb0();}
  public int p0()  { if(b) return 12;  else return myb0();}
  public int q0()  { if(b) return 13;  else return myb0();}
  public int r0()  { if(b) return 14;  else return myb0();}
  public int s0()  { if(b) return 15;  else return myb0();}
  public int t0()  { if(b) return 16;  else return myb0();}
  public int u0()  { if(b) return 17;  else return myb0();}
  public int v0()  { if(b) return 18;  else return myb0();}
  public int w0()  { if(b) return 19;  else return myb0();}
```

```
  public void mya0() {System.out.println("0");}
  public void mya1() {System.out.println("1");}
  public void mya2() {System.out.println("2");}
  public void mya3() {System.out.println("3");}
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
        mya0(); mya1(); mya2(); mya3(); mya4();
        mya5(); mya6(); mya7(); mya8(); mya9();
        return 100;
  }
}

class MyClassB implements MyInterface20, MyInterface19, MyInterface18,
            MyInterface17, MyInterface16, MyInterface15, MyInterface14,
            MyInterface13, MyInterface12, MyInterface11, MyInterface10,
            MyInterface9, MyInterface8, MyInterface7, MyInterface6,
            MyInterface5, MyInterface4, MyInterface3, MyInterface2,
            MyInterface1  {
  public boolean b = false;

  public int w0()  { if(b) return 119;  else return myb0();}
  public int v0()  { if(b) return 118;  else return myb0();}
  public int u0()  { if(b) return 117;  else return myb0();}
  public int t0()  { if(b) return 116;  else return myb0();}
  public int s0()  { if(b) return 115;  else return myb0();}
  public int r0()  { if(b) return 114;  else return myb0();}
  public int q0()  { if(b) return 113;  else return myb0();}
  public int p0()  { if(b) return 112;  else return myb0();}
```

```
  public int n0()  { if(b) return 111;  else return myb0();}
  public int m0()  { if(b) return 110;  else return myb0();}

  public int j0()  { if(b) return 109;  else return myb0();}
  public int i0()  { if(b) return 108;  else return myb0();}
  public int h0()  { if(b) return 107;  else return myb0();}
  public int g0()  { if(b) return 106;  else return myb0();}
  public int f0()  { if(b) return 105;  else return myb0();}
  public int e0()  { if(b) return 104;  else return myb0();}
  public int d0()  { if(b) return 103;  else return myb0();}
  public int c0()  { if(b) return 102;  else return myb0();}
  public int b0()  { if(b) return 101;  else return myb0();}
  public int a0()  { if(b) return 100;  else return myb0();}

  public void mya0() {System.out.println("0");}
  public void mya1() {System.out.println("1");}
  public void mya2() {System.out.println("2");}
  public void mya3() {System.out.println("3");}
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
        mya0(); mya1(); mya2(); mya3(); mya4();
        mya5(); mya6(); mya7(); mya8(); mya9();
        return 100;
  }
}

public class Test2 {
  public static void main(String[] args)  {
```

```
    Test2 mytest = new Test2();
    mytest.aaa(Integer.parseInt(args[0]));
  }

  public void aaa(int rounds) {
    Object classArray[] = new Object[2];
    MyClassA myA = new MyClassA(); myA.b = true;
    MyClassB myB = new MyClassB(); myB.b = true;
    classArray[0] = myA;
    classArray[1] = myB;

    MyInterface1  in1;  MyInterface2  in2;  MyInterface3  in3;
    MyInterface4  in4;  MyInterface5  in5;  MyInterface6  in6;
    MyInterface7  in7;  MyInterface8  in8;  MyInterface9  in9;
    MyInterface10 in10; MyInterface11 in11; MyInterface12 in12;
    MyInterface13 in13; MyInterface14 in14; MyInterface15 in15;
    MyInterface16 in16; MyInterface17 in17; MyInterface18 in18;
    MyInterface19 in19; MyInterface20 in20;

    long times[] = new long[10];

    for (int i = 0; i < times.length; i ++) {
      long result = 0;
      long time_1 = System.currentTimeMillis();
      for (int j = 0; j < rounds; j ++) {
        Object ob = classArray[j%2];
        in1  = (MyInterface1) ob;   in2  = (MyInterface2) ob;
        in3  = (MyInterface3) ob;   in4  = (MyInterface4) ob;
        in5  = (MyInterface5) ob;   in6  = (MyInterface6) ob;
        in7  = (MyInterface7) ob;   in8  = (MyInterface8) ob;
        in9  = (MyInterface9) ob;   in10 = (MyInterface10) ob;
        in11 = (MyInterface11) ob;  in12 = (MyInterface12) ob;
        in13 = (MyInterface13) ob;  in14 = (MyInterface14) ob;
```

```
        in15 = (MyInterface15) ob;  in16 = (MyInterface16) ob;
        in17 = (MyInterface17) ob;  in18 = (MyInterface18) ob;
        in19 = (MyInterface19) ob;  in20 = (MyInterface20) ob;

        result += in1.a0(); result += in2.b0(); result += in3.c0();
        result += in4.d0(); result += in5.e0(); result += in6.f0();
        result += in7.g0(); result += in8.h0(); result += in9.i0();
        result += in10.j0(); result += in11.m0(); result += in12.n0();
        result += in13.p0(); result += in14.q0(); result += in15.r0();
        result += in16.s0(); result += in17.t0(); result += in18.u0();
        result += in19.v0(); result += in20.w0();
      }

      long time_2 = System.currentTimeMillis();
      times[i] = time_2 - time_1;
      System.out.print(result);
    }
  }
}
```

C.3 Interface Invocation: Test Cases 3 ~ 6

Test cases 3~6 define 20 interfaces ***MyInterface1~20***, and two classes ***MyClassA*** and ***MyClassB***. ***MyInterface1~20*** are the direct superinterfaces for both classes. Each interface declares 100, 100, 200, 200 methods respectively in the 4 test cases, i.e., both classes implement 2000, 2000, 4000, 4000 interface methods respectively. The interface methods in test cases 3 and 5 are implemented as trivial calls, and the interface methods in test cases 4 and 6 are implemented as normal calls.

The source code of these 4 test cases is omitted in order to save space.

C.4 Virtual Invocation: Test Case v-1

The objective of the test case v-1 is to get the execution time of virtual methods, which are all trivial calls. It is very similar with the test case 1 (for interface invocation), except that every method is invoked upon a class reference, instead of an interface reference.

```
interface MyInterface1 {
  public int a0();
}

interface MyInterface2 {
  public int b0();
}

interface MyInterface3 {
  public int c0();
}

interface MyInterface4 {
  public int d0();
}

interface MyInterface5 {
  public int e0();
}

interface MyInterface6 {
  public int f0();
}

interface MyInterface7 {
  public int g0();
}
```

```
interface MyInterface8 {
  public int h0();
}

interface MyInterface9 {
  public int i0();
}

interface MyInterface10 {
  public int j0();
}

interface MyInterface11 {
  public int m0();
}

interface MyInterface12 {
  public int n0();
}

interface MyInterface13 {
  public int p0();
}

interface MyInterface14 {
  public int q0();
}

interface MyInterface15 {
  public int r0();
}

interface MyInterface16 {
```

```
  public int s0();
}

interface MyInterface17 {
  public int t0();
}

interface MyInterface18 {
  public int u0();
}

interface MyInterface19 {
  public int v0();
}

interface MyInterface20 {
  public int w0();
}

class MySuperClass {
  public int a0()  { return 0;}
  public int b0()  { return 0;}
  public int c0()  { return 0;}
  public int d0()  { return 0;}
  public int e0()  { return 0;}
  public int f0()  { return 0;}
  public int g0()  { return 0;}
  public int h0()  { return 0;}
  public int i0()  { return 0;}
  public int j0()  { return 0;}

  public int m0()  { return 0;}
  public int n0()  { return 0;}
```

```
  public int p0()  { return 0;}
  public int q0()  { return 0;}
  public int r0()  { return 0;}
  public int s0()  { return 0;}
  public int t0()  { return 0;}
  public int u0()  { return 0;}
  public int v0()  { return 0;}
  public int w0()  { return 0;}
}

class MyClassA
    extends MySuperClass
    implements MyInterface1, MyInterface2, MyInterface3,
            MyInterface4, MyInterface5, MyInterface6, MyInterface7,
            MyInterface8, MyInterface9, MyInterface10, MyInterface11,
            MyInterface12, MyInterface13, MyInterface14, MyInterface15,
            MyInterface16, MyInterface17, MyInterface18, MyInterface19,
            MyInterface20  {

  public int a0()  { return 0;}
  public int b0()  { return 1;}
  public int c0()  { return 2;}
  public int d0()  { return 3;}
  public int e0()  { return 4;}
  public int f0()  { return 5;}
  public int g0()  { return 6;}
  public int h0()  { return 7;}
  public int i0()  { return 8;}
  public int j0()  { return 9;}

  public int m0()  { return 10;}
  public int n0()  { return 11;}
  public int p0()  { return 12;}
```

```
  public int q0()  { return 13;}
  public int r0()  { return 14;}
  public int s0()  { return 15;}
  public int t0()  { return 16;}
  public int u0()  { return 17;}
  public int v0()  { return 18;}
  public int w0()  { return 19;}

  public void mya0() {System.out.println("0");}
  public void mya1() {System.out.println("1");}
  public void mya2() {System.out.println("2");}
  public void mya3() {System.out.println("3");}
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
        mya0(); mya1(); mya2(); mya3(); mya4();
        mya5(); mya6(); mya7(); mya8(); mya9();
        return 100;
  }
}

class MyClassB
    extends MySuperClass
    implements MyInterface20, MyInterface19, MyInterface18,
            MyInterface17, MyInterface16, MyInterface15, MyInterface14,
            MyInterface13, MyInterface12, MyInterface11, MyInterface10,
            MyInterface9, MyInterface8, MyInterface7, MyInterface6,
            MyInterface5, MyInterface4, MyInterface3, MyInterface2,
            MyInterface1  {
```

```
public int w0()  { return 119;}
public int v0()  { return 118;}
public int u0()  { return 117;}
public int t0()  { return 116;}
public int s0()  { return 115;}
public int r0()  { return 114;}
public int q0()  { return 113;}
public int p0()  { return 112;}
public int n0()  { return 111;}
public int m0()  { return 110;}

public int j0()  { return 109;}
public int i0()  { return 108;}
public int h0()  { return 107;}
public int g0()  { return 106;}
public int f0()  { return 105;}
public int e0()  { return 104;}
public int d0()  { return 103;}
public int c0()  { return 102;}
public int b0()  { return 101;}
public int a0()  { return 100;}

public void mya0() {System.out.println("0");}
public void mya1() {System.out.println("1");}
public void mya2() {System.out.println("2");}
public void mya3() {System.out.println("3");}
public void mya4() {System.out.println("4");}
public void mya5() {System.out.println("5");}
public void mya6() {System.out.println("6");}
public void mya7() {System.out.println("7");}
public void mya8() {System.out.println("8");}
public void mya9() {System.out.println("9");}
```

```
  public int  myb0() {
         mya0(); mya1(); mya2(); mya3(); mya4();
         mya5(); mya6(); mya7(); mya8(); mya9();
         return 100;
  }
}

public class VTest1 {
  public static void main(String[] args)  {
    VTest1 mytest = new VTest1();
    mytest.aaa(Integer.parseInt(args[0]));
  }

  public void aaa(int rounds) {
    MyClassA myA = new MyClassA();
    MyClassB myB = new MyClassB();
    MySuperClass classArray[] = new MySuperClass[2];
    classArray[0] = myA;
    classArray[1] = myB;

    MyInterface1  in1;  MyInterface2  in2;  MyInterface3  in3;
    MyInterface4  in4;  MyInterface5  in5;  MyInterface6  in6;
    MyInterface7  in7;  MyInterface8  in8;  MyInterface9  in9;
    MyInterface10 in10; MyInterface11 in11; MyInterface12 in12;
    MyInterface13 in13; MyInterface14 in14; MyInterface15 in15;
    MyInterface16 in16; MyInterface17 in17; MyInterface18 in18;
    MyInterface19 in19; MyInterface20 in20;

    long times[] = new long[10];

    for (int i = 0; i < times.length; i ++) {
      long result = 0;
      long time_1 = System.currentTimeMillis();
```

```
      for (int j = 0; j < rounds; j ++) {
        MySuperClass ob = classArray[j%2];
        in1  = (MyInterface1) ob;   in2  = (MyInterface2) ob;
        in3  = (MyInterface3) ob;   in4  = (MyInterface4) ob;
        in5  = (MyInterface5) ob;   in6  = (MyInterface6) ob;
        in7  = (MyInterface7) ob;   in8  = (MyInterface8) ob;
        in9  = (MyInterface9) ob;   in10 = (MyInterface10) ob;
        in11 = (MyInterface11) ob;  in12 = (MyInterface12) ob;
        in13 = (MyInterface13) ob;  in14 = (MyInterface14) ob;
        in15 = (MyInterface15) ob;  in16 = (MyInterface16) ob;
        in17 = (MyInterface17) ob;  in18 = (MyInterface18) ob;
        in19 = (MyInterface19) ob;  in20 = (MyInterface20) ob;

        result += ob.a0(); result += ob.b0(); result += ob.c0();
        result += ob.d0(); result += ob.e0(); result += ob.f0();
        result += ob.g0(); result += ob.h0(); result += ob.i0();
        result += ob.j0(); result += ob.m0(); result += ob.n0();
        result += ob.p0(); result += ob.q0(); result += ob.r0();
        result += ob.s0(); result += ob.t0(); result += ob.u0();
        result += ob.v0(); result += ob.w0();
      }
      long time_2 = System.currentTimeMillis();
      times[i] = time_2 - time_1;
      System.out.print(result);
    }
  }
}
```

C.5 Virtual Invocation: Test Case v-2

The objective of the test case v-2 is to get the execution time of virtual methods, which are all normal calls. It is very similar with the test case 2 (for interface invocation), except that every method is invoked upon a class reference, instead of an interface reference.

```
interface MyInterface1 {
  public int a0();
}

interface MyInterface2 {
  public int b0();
}

interface MyInterface3 {
  public int c0();
}

interface MyInterface4 {
  public int d0();
}

interface MyInterface5 {
  public int e0();
}

interface MyInterface6 {
  public int f0();
}

interface MyInterface7 {
  public int g0();
}

interface MyInterface8 {
  public int h0();
}

interface MyInterface9 {
```

```
  public int i0();
}

interface MyInterface10 {
  public int j0();
}

interface MyInterface11 {
  public int m0();
}

interface MyInterface12 {
  public int n0();
}

interface MyInterface13 {
  public int p0();
}

interface MyInterface14 {
  public int q0();
}

interface MyInterface15 {
  public int r0();
}

interface MyInterface16 {
  public int s0();
}

interface MyInterface17 {
  public int t0();
```

```
}

interface MyInterface18 {
  public int u0();
}

interface MyInterface19 {
  public int v0();
}

interface MyInterface20 {
  public int w0();
}

class MySuperClass {
  public int a0()  { return 0;}
  public int b0()  { return 0;}
  public int c0()  { return 0;}
  public int d0()  { return 0;}
  public int e0()  { return 0;}
  public int f0()  { return 0;}
  public int g0()  { return 0;}
  public int h0()  { return 0;}
  public int i0()  { return 0;}
  public int j0()  { return 0;}

  public int m0()  { return 0;}
  public int n0()  { return 0;}
  public int p0()  { return 0;}
  public int q0()  { return 0;}
  public int r0()  { return 0;}
  public int s0()  { return 0;}
  public int t0()  { return 0;}
```

```
  public int u0()  { return 0;}
  public int v0()  { return 0;}
  public int w0()  { return 0;}
}

class MyClassA
        extends MySuperClass
        implements MyInterface1, MyInterface2, MyInterface3,
            MyInterface4, MyInterface5, MyInterface6, MyInterface7,
            MyInterface8, MyInterface9, MyInterface10, MyInterface11,
            MyInterface12, MyInterface13, MyInterface14, MyInterface15,
            MyInterface16, MyInterface17, MyInterface18, MyInterface19,
            MyInterface20  {
  public boolean b = false;

  public int a0()  { if(b) return 0;  else return myb0();}
  public int b0()  { if(b) return 1;  else return myb0();}
  public int c0()  { if(b) return 2;  else return myb0();}
  public int d0()  { if(b) return 3;  else return myb0();}
  public int e0()  { if(b) return 4;  else return myb0();}
  public int f0()  { if(b) return 5;  else return myb0();}
  public int g0()  { if(b) return 6;  else return myb0();}
  public int h0()  { if(b) return 7;  else return myb0();}
  public int i0()  { if(b) return 8;  else return myb0();}
  public int j0()  { if(b) return 9;  else return myb0();}

  public int m0()  { if(b) return 10;  else return myb0();}
  public int n0()  { if(b) return 11;  else return myb0();}
  public int p0()  { if(b) return 12;  else return myb0();}
  public int q0()  { if(b) return 13;  else return myb0();}
  public int r0()  { if(b) return 14;  else return myb0();}
  public int s0()  { if(b) return 15;  else return myb0();}
  public int t0()  { if(b) return 16;  else return myb0();}
```

```
  public int u0()  { if(b) return 17;  else return myb0();}
  public int v0()  { if(b) return 18;  else return myb0();}
  public int w0()  { if(b) return 19;  else return myb0();}

  public void mya0() {System.out.println("0");}
  public void mya1() {System.out.println("1");}
  public void mya2() {System.out.println("2");}
  public void mya3() {System.out.println("3");}
  public void mya4() {System.out.println("4");}
  public void mya5() {System.out.println("5");}
  public void mya6() {System.out.println("6");}
  public void mya7() {System.out.println("7");}
  public void mya8() {System.out.println("8");}
  public void mya9() {System.out.println("9");}
  public int  myb0() {
         mya0(); mya1(); mya2(); mya3(); mya4();
         mya5(); mya6(); mya7(); mya8(); mya9();
         return 100;
  }
}

class MyClassB
        extends MySuperClass
        implements MyInterface20, MyInterface19, MyInterface18,
            MyInterface17, MyInterface16, MyInterface15, MyInterface14,
            MyInterface13, MyInterface12, MyInterface11, MyInterface10,
            MyInterface9, MyInterface8, MyInterface7, MyInterface6,
            MyInterface5, MyInterface4, MyInterface3, MyInterface2,
            MyInterface1  {
  public boolean b = false;

  public int w0()  { if(b) return 119;  else return myb0();}
  public int v0()  { if(b) return 118;  else return myb0();}
```

```
public int u0()  { if(b) return 117;  else return myb0();}
public int t0()  { if(b) return 116;  else return myb0();}
public int s0()  { if(b) return 115;  else return myb0();}
public int r0()  { if(b) return 114;  else return myb0();}
public int q0()  { if(b) return 113;  else return myb0();}
public int p0()  { if(b) return 112;  else return myb0();}
public int n0()  { if(b) return 111;  else return myb0();}
public int m0()  { if(b) return 110;  else return myb0();}

public int j0()  { if(b) return 109;  else return myb0();}
public int i0()  { if(b) return 108;  else return myb0();}
public int h0()  { if(b) return 107;  else return myb0();}
public int g0()  { if(b) return 106;  else return myb0();}
public int f0()  { if(b) return 105;  else return myb0();}
public int e0()  { if(b) return 104;  else return myb0();}
public int d0()  { if(b) return 103;  else return myb0();}
public int c0()  { if(b) return 102;  else return myb0();}
public int b0()  { if(b) return 101;  else return myb0();}
public int a0()  { if(b) return 100;  else return myb0();}

public void mya0() {System.out.println("0");}
public void mya1() {System.out.println("1");}
public void mya2() {System.out.println("2");}
public void mya3() {System.out.println("3");}
public void mya4() {System.out.println("4");}
public void mya5() {System.out.println("5");}
public void mya6() {System.out.println("6");}
public void mya7() {System.out.println("7");}
public void mya8() {System.out.println("8");}
public void mya9() {System.out.println("9");}
public int  myb0() {
       mya0(); mya1(); mya2(); mya3(); mya4();
       mya5(); mya6(); mya7(); mya8(); mya9();
```

```
        return 100;
  }
}

public class Test2 {
  public static void main(String[] args)  {
    Test2 mytest = new Test2();
    mytest.aaa(Integer.parseInt(args[0]));
  }

  public void aaa(int rounds) {
    MyClassA myA = new MyClassA(); myA.b = true;
    MyClassB myB = new MyClassB(); myB.b = true;
    MySuperClass classArray[] = new MySuperClass[2];
    classArray[0] = myA;
    classArray[1] = myB;

    MyInterface1  in1;  MyInterface2  in2;  MyInterface3  in3;
    MyInterface4  in4;  MyInterface5  in5;  MyInterface6  in6;
    MyInterface7  in7;  MyInterface8  in8;  MyInterface9  in9;
    MyInterface10 in10; MyInterface11 in11; MyInterface12 in12;
    MyInterface13 in13; MyInterface14 in14; MyInterface15 in15;
    MyInterface16 in16; MyInterface17 in17; MyInterface18 in18;
    MyInterface19 in19; MyInterface20 in20;

    long times[] = new long[10];

    for (int i = 0; i < times.length; i ++) {
      long result = 0;
      long time_1 = System.currentTimeMillis();
      for (int j = 0; j < rounds; j ++) {
        MySuperClass ob = classArray[j%2];
        in1  = (MyInterface1) ob;   in2  = (MyInterface2) ob;
```

```
        in3  = (MyInterface3) ob;   in4  = (MyInterface4) ob;
        in5  = (MyInterface5) ob;   in6  = (MyInterface6) ob;
        in7  = (MyInterface7) ob;   in8  = (MyInterface8) ob;
        in9  = (MyInterface9) ob;   in10 = (MyInterface10) ob;
        in11 = (MyInterface11) ob;  in12 = (MyInterface12) ob;
        in13 = (MyInterface13) ob;  in14 = (MyInterface14) ob;
        in15 = (MyInterface15) ob;  in16 = (MyInterface16) ob;
        in17 = (MyInterface17) ob;  in18 = (MyInterface18) ob;
        in19 = (MyInterface19) ob;  in20 = (MyInterface20) ob;

        result += ob.a0(); result += ob.b0(); result += ob.c0();
        result += ob.d0(); result += ob.e0(); result += ob.f0();
        result += ob.g0(); result += ob.h0(); result += ob.i0();
        result += ob.j0(); result += ob.m0(); result += ob.n0();
        result += ob.p0(); result += ob.q0(); result += ob.r0();
        result += ob.s0(); result += ob.t0(); result += ob.u0();
        result += ob.v0(); result += ob.w0();
      }
      long time_2 = System.currentTimeMillis();
      times[i] = time_2 - time_1;
      System.out.print(result);
    }
  }
}
```

Bibliography

[1] *The Apache XML Project.* 2001. `http://xml.apache.org/xerces-j/`.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, 2000.

[3] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, 1999.

[4] Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David Grove, and Derek Lieber. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, 2001.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[6] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 52–64, 2000.

[7] David Brownell. *SAX2.* O'Reilly, 2002. `http://www.saxproject.org/`.

[8] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, and Harini Srinivasan. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Conference on Java Grande*, pages 129–141, 1999.

[9] Tom A. Cargill. Controversy: The case against multiple inheritance in C++. *USENIX Computing Systems*, 4(1):69–82, 1991.

[10] Frank M. Carrano and Janet Prichard. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*, chapter 4. Addison-Wesley, 2001.

[11] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *Languages and Compilers for Parallel Computing*, pages 35–52, 1999.

[12] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 21–31, 1999.

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 1: Basic Architecture. 2004. `http://www.intel.com/design/pentium4/manuals/`.

[14] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, second edition, 1991.

[15] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[16] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[17] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software — Practice and Experience*, 30(3):199–232, 2000.

[18] Johnray Fuller. *Red Hat Linux Reference Guide*, chapter 1. 2003.

[19] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 27–40, 2001.

[20] Motorola Incorporation. *PowerPC Microprocessor Family: The Programmer's Reference Guide.* 1995.

[21] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification.* Java Series. Addison-Wesley, second edition, 2000.

[22] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Java Series. Addison-Wesley, second edition, 1999.

[24] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[25] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, 1987.

[26] Bill Venner. *Inside the Java Virtual Machine*, chapter 5. MCGraw-Hill, second edition, 1999.

[27] World-Wide Web Consortium (W3C). *Document Object Model (DOM) Level 2 Core Specification*. 2000. `http://www.w3.org/TR/DOM-Level-2-Core/`.

[28] World-Wide Web Consortium (W3C). *XML Schema Part 0: Primer*. 2001. `http://www.w3.org/TR/xmlschema-0/`.

[29] World-Wide Web Consortium (W3C). *XML Schema Part 1: Structures*. 2001. `http://www.w3.org/TR/xmlschema-1/`.

[30] World-Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes*. 2001. `http://www.w3.org/TR/xmlschema-2/`.

[31] Jim Waldo. Controversy: The case for multiple inheritance in C++. *USENIX Computing Systems*, 4(2):157–171, 1991.