Tokenisation and Compression of Java Class Files for Mobile Devices

Shawn Haggett

Bachelor of Information Technology (Honours)

Flinders University of South Australia

A Thesis Submitted for the Degree of Doctor of Philosophy

Flinders University

School of Computer Science, Engineering and Mathematics

Adelaide, South Australia

2012

(Submitted 18th February 2010)

Contents

Abstract **xviii**

Acknowledgements **xxi**

1 Background **1**

1.1 Introduction 1

1.2 Object Oriented Programming 1

1.2.1 Classes, Objects and Methods 2

1.2.2 Inheritance 2

1.2.3 Polymorphism 5

1.3 Polymorphic Call Dispatch 7

1.3.1 Selector Colouring 8

1.3.2 Virtual Function Tables 10

1.3.3 Row Displacement 11

1.4 Java 12

1.4.1 General Background 13

1.4.1.1 Write Once, Run Everywhere 14

1.4.1.2 Java Language Specification 15

1.4.1.3 Java Virtual Machine Specification 15

1.4.2 Class File Format 16

1.4.2.1 Header 17

1.4.2.2 Constant Pool 18

1.4.2.3 Flags & Inheritance/Interface Information 20

1.4.2.4 Fields 21

1.4.2.5 Methods 22
1.4.2.6 Additional Attributes 23
1.4.3 Usage of Constant Pool Entries 24
1.4.3.1 CONSTANT_Utf8 25
1.4.3.2 CONSTANT_Integer, CONSTANT_Float,
CONSTANT_String 27
1.4.3.3 CONSTANT_Long, CONSTANT_Double 27
1.4.3.4 CONSTANT_Class 27
1.4.3.5 CONSTANT_Fieldref 29
1.4.3.6 CONSTANT_Methodref 30
1.4.3.7 CONSTANT_InterfaceMethodref 30
1.4.3.8 CONSTANT_NameAndType 30
1.4.4 Linking in Java Class Files 31
1.4.4.1 Class References 32
1.4.4.2 Field References 32
1.4.4.3 Method References 33
1.4.4.4 Resolving Method References 35
1.4.4.5 Symbolic References 36
1.4.5 *invoke** Instructions 37
1.4.5.1 Constructors and *invokespecial* 39
1.4.5.2 Private methods and *invokespecial* 41
1.4.5.3 The SUPER keyword and *invokespecial* 41
1.4.5.4 Historic use of *invokespecial* 42
1.4.6 Usage of Java Instruction Set 43

2 Previous Work **48**
2.1 Introduction 48
2.2 Java Virtual Machine Implementations 49
2.2.1 Interpreter 50
2.2.2 Just-in-Time Compilation 50

2.2.3 Ahead-of-Time Compilation 52
2.2.4 Hardware Approaches 53
2.2.4.1 Co-processor 53
2.2.4.2 Java Processor 55
2.3 Optimisations for Java 56
2.3.1 Instruction Level Parallelism 56
2.3.2 Instruction Folding 57
2.3.3 Garbage Collection 58
2.4 J2ME 60
2.4.1 CLDC Specification 60
2.4.1.1 Differences to the Java Language Specification 61
2.4.1.2 Differences to the Java Virtual Machine Specification . . 62
2.4.2 The KVM 63
2.5 Java Card 64
2.5.1 The CAP File Format 64
2.5.2 Java Card's Virtual Method Tables 64
2.5.3 Handling of Interfaces 67
2.6 Squawk 69
2.6.1 Memory Restrictions 70
2.6.2 Bytecode Modifications 70
2.6.3 Method/field references 71
2.7 Compression of Java Class Files 72
2.7.1 Wire-Formats 72
2.7.1.1 Jar 73
2.7.1.2 Clazz 73
2.7.1.3 JAZZ 74
2.7.1.4 Pack 75
2.7.1.5 CAR 75
2.7.1.6 Generic Adaptive Syntax-Directed Compression 76
2.7.2 Interpretable formats 76

2.7.2.1 Compact Java Binaries for Embedded Systems 77

2.7.2.2 Java Bytecode Compression for Low-End Embedded Systems 78

2.7.2.3 Practical Java Card bytecode compression 79

2.7.2.4 Split-Stream Dictionary Program Compression 80

2.7.3 Runtime compression 80

2.7.3.1 Heap Compression 80

2.7.3.2 Energy savings through compression 81

2.7.4 Summary 81

2.8 Summary 82

3 Global Tokenisation of Class Files **83**

3.1 Introduction 83

3.2 Comparison to Java Card 84

3.3 Simple Tokenisation 85

3.3.1 Assigning Different Tokens to the Same Selector 87

3.3.2 Non-continuity of Token values 87

3.3.3 Binary Compatibility 88

3.3.4 Overview of Tokenisation Process 89

3.4 Method Groups 90

3.4.1 Creating Method Groups 92

3.5 Assigning tokens to Method Groups 97

3.6 Dealing With Static Methods 99

3.6.1 Binary Compatibility and Static Methods 100

3.6.2 Static Methods in Tokenised Classes 101

3.7 Descriptor File 102

3.8 Tokenisation of Fields 103

3.8.1 Static Fields 104

3.8.2 Fields in Interfaces 105

3.8.3 Non-static Fields 105

3.9 Libraries Used For Testing 106

3.9.1 CLDC API 107

3.9.2 MIDP API and Example Applications 108

3.9.3 Javolution 109

3.9.4 J2SE API 110

3.10 Global Tokenisation Efficiency 110

3.10.1 Token allocation efficiency 111

3.10.2 Virtual Method Table Size 112

3.11 Conclusions 113

4 Incremental Tokenisation of Class Files **114**

4.1 Introduction 114

4.2 Overview of Incremental Tokenisation Process 114

4.3 Difference to Simple Tokenisation 115

4.3.1 Same Methods, Different Tokens 116

4.3.2 Different Methods, Same Tokens 117

4.4 Extending Method Groups 119

4.5 Assigning tokens 122

4.6 Building Virtual Method Tables 123

4.6.1 Building Local Tables 125

4.6.2 Building Tokenised Tables 126

4.6.3 Generating an Encoded VMT 128

4.7 Issues with incremental tokenisation 130

4.7.1 Restriction of tokens in interfaces 130

4.7.2 Extending multiple existing tokenisations 132

4.7.3 Binary Compatibility 134

4.7.4 Static Methods 136

4.8 Incremental Tokenisation Efficiency 136

4.8.1 Token allocation efficiency 137

4.8.2 Virtual Method Table Size 138

4.9 Conclusions 141

5 Implementation of a VM 142

5.1 Introduction 142

5.2 Virtual Method Table Format 143

5.2.1 In Memory Representation 144

5.2.2 Alternate VMT Encodings 145

5.2.3 Size of Alternate Virtual Method Tables 146

5.3 Instruction Set Modifications 149

5.3.1 ldc, ldc_w and ldc2_w 150

5.3.2 getstatic/putstatic 151

5.3.3 getfield/putfield 154

5.3.4 invokevirtual 157

5.3.5 invokeinterface 159

5.3.6 invokestatic 161

5.3.7 invokespecial 164

5.4 Native Methods 165

5.4.1 Static Methods 166

5.4.1.1 java.lang.Class 166

5.4.1.2 java.lang.System 167

5.4.1.3 java.lang.Thread 168

5.4.2 Virtual Methods 168

5.4.2.1 com.sun.cldc.io.ConsoleOutputStream 169

5.4.2.2 java.lang.Class 169

5.4.2.3 java.lang.Object 170

5.4.2.4 java.lang.Runtime 171

5.4.2.5 java.lang.Thread 171

5.5 VM Constants 171

5.5.1 Implementing VM Constants 173

5.6 Correctness of Execution 173

5.6.1 Instruction set usage . . . 174
5.6.2 Method coverage . . . 174
5.7 Efficiency of Execution . . . 175
5.7.1 Test Application . . . 176
5.7.2 VM Implementations Tested . . . 178
5.7.2.1 KVM and KVM-Fast . . . 178
5.7.2.2 Tokenised VM . . . 180
5.7.3 Methodology . . . 180
5.7.3.1 Instruction Count . . . 181
5.7.3.2 Profiling Using gprof . . . 181
5.7.3.3 Profiling Using RDTSC . . . 182
5.7.4 Results . . . 183
5.7.4.1 KVM Instruction Count . . . 183
5.7.4.2 KVM gprof Timing . . . 184
5.7.4.3 RDTSC Overhead . . . 185
5.7.4.4 KVM RDTSC Timing . . . 187
5.7.4.5 KVM-Fast RDTSC Timing . . . 189
5.7.4.6 Tokenised RDTSC Timing . . . 189
5.7.5 Summary . . . 191
5.8 Conclusions . . . 194

6 Tokenised Class File Generation and Compression 195

6.1 Introduction . . . 195
6.2 Constant Pool Entries . . . 195
6.2.1 CONSTANT_Class . . . 197
6.2.2 CONSTANT_String . . . 198
6.2.3 CONSTANT_NameAndType . . . 199
6.2.4 CONSTANT_*ref . . . 200
6.3 field_info Section . . . 202
6.4 method_info Section . . . 203

6.4.1 access_flags . . . 205

6.4.2 Static Methods . . . 208

6.5 Attributes . . . 209

6.5.1 ConstantValue Attribute . . . 210

6.5.2 Code Attribute . . . 212

6.5.2.1 General Format of Java Instructions . . . 214

6.5.2.2 When Instructions Need to Change . . . 214

6.5.2.3 Process to Update Bytecode . . . 215

6.5.3 Exceptions Attribute . . . 216

6.5.4 InnerClass/Synthetic Attribute . . . 217

6.5.5 Debugging Attributes . . . 218

6.5.6 Deprecated Attribute . . . 218

6.5.7 StackMap Attribute . . . 219

6.5.8 VMT Attribute . . . 219

6.6 Code Compression . . . 220

6.6.1 Overall Code Compression for Global Tokenisation . . . 220

6.6.2 Overall Code Compression for Incremental Tokenisation . . . 223

6.6.3 Further Compression . . . 224

6.6.4 Constant Pool Usage . . . 224

6.7 Conclusions . . . 224

7 Conclusions & Future Work 226

7.1 Tokenisation . . . 226

7.2 Compression . . . 228

7.3 Key Contributions . . . 229

7.4 Future Work . . . 229

A Tokenised Class File Binary Format 231

A.1 Constant Pool . . . 233

A.1.1 field_info Section . . . 235

A.2 method_info Section . . . 236

A.3 Attributes 238

A.3.1 ConstantValue Attribute 239

A.3.2 Code 239

A.3.3 Exceptions Attribute 240

A.3.4 InnerClass Attribute 241

A.3.5 Synthetic/Deprecated 241

A.3.6 StackMap 242

A.3.7 VirtualMethodTable Attribute 242

B Descriptor File Binary Format 245

B.1 Method Group Entries 246

B.1.1 MethodGroupSingle 247

B.1.2 MethodGroupMulti 248

B.2 Class Entries 249

B.2.1 Field entries 251

B.2.2 Method entries 251

C Types of Native Methods in CLDC 1.1 API 253

C.1 Performance 253

C.1.1 java.lang.Double 253

C.1.2 java.lang.Float 254

C.1.3 java.lang.Math 254

C.1.4 java.lang.String 254

C.1.5 java.lang.StringBuffer 255

C.1.6 java.lang.System 256

C.2 JVM Interaction 256

C.2.1 java.lang.Class 256

C.2.2 java.lang.Object 257

C.2.3 java.lang.Runtime 257

C.2.4 java.lang.System 258

C.2.5 java.lang.Thread 258

C.2.6 java.lang.Throwable 259
C.2.7 java.lang.ref.WeakReference 259
C.3 IO 259
C.3.1 com.sun.cldc.io.ConsoleOutputStream 260
C.3.2 com.sun.cldc.io.ResourceInputStream 260
C.3.3 com.sun.cldc.io.Waiter 261

Bibliography **262**

List of Figures

1.1 Example of Inheritance in a GUI library . . . 3
1.2 Attempting to use the TextHandler class . . . 4
1.3 Example of multiple-inheritance . . . 4
1.4 Example Class hierarchy . . . 7
1.5 Selector Table Index . . . 8
1.6 Conflict Graph for Example Classes . . . 9
1.7 Selector Colouring . . . 9
1.8 Single Inheritance and Virtual Method Tables in C++ . . . 10
1.9 Multiple Inheritance in C++ . . . 11
1.10 Row Displacement . . . 12
1.11 Use of CONSTANT_Class entries in methods . . . 28
1.12 Example of a try/catch block . . . 28
1.13 Structure of a symbolic field reference . . . 32
1.14 Example of Classes in a Library . . . 37
1.15 Classes with over-riding . . . 38
1.16 Method using the SUPER keyword . . . 42
1.17 Compress Instruction usage in [53] . . . 45
1.18 DB Instruction usage in [53] . . . 45
1.19 Mandelbrot Instruction usage in [53] . . . 46
1.20 Queen Instruction usage in [53] . . . 46
1.21 Raytrace Instruction usage in [53] . . . 47

2.1 Inheritance Tree Including Tokens . . . 65
2.2 Virtual Method Tables . . . 66

2.3 Inheritance With Over-Riding 67
2.4 Inheritance With Interfaces 68
2.5 Representation of Class Names in Class Files 77

3.1 The same selector with a different token 87
3.2 Non-continuity of tokens 88
3.3 Example system classes 90
3.4 Example System with Method Groups Assigned 96
3.5 Example System After Assigning Tokens 99
3.6 Static method in a super-class 100
3.7 Adding new classes to an already tokenised system 102
3.8 Structure of a symbolic field reference 104

4.1 Work Flow for Incremental Tokenisation 116
4.2 Methods with different tokens 117
4.3 Methods with the same tokens 118
4.4 VMT with a conflict entry 118
4.5 Example of adding incremental class 120
4.6 Method Groups for the example system 121
4.7 Method Groups after simplification 122
4.8 Local tables for example classes 125
4.9 Virtual method tables for example classes 128
4.10 Types of VMT entries for tokenised binary class files. 129
4.11 Possible ambiguities with interfaces 131
4.12 VMT with ambiguous conflict entry 131
4.13 API with two custom libraries 134

5.1 Example of VMT usage 147
5.2 Size of Different Types of VMTs 147
5.3 get/put-static instruction implementation 154
5.4 get/put-field instruction implementation 156

5.5 Performing an *invokevirtual* . . . 158
5.6 Performing an *invokeinterface* . . . 161
5.7 Performing an *invokestatic* . . . 163
5.8 Histogram of Average Cycles/Call for Uncached *invokevirtual* Calls . . . 188
5.9 Histogram of Average Cycles/Call (Adjusted) for 32-bit Tokenised *invokevirtual* . . . 192
5.10 Histogram of Average Cycles/Call (Adjusted) for 64-bit Tokenised *invokevirtual* . . . 192

6.1 Standard CONSTANT_Class_info structure . . . 197
6.2 Tokenised CONSTANT_Class_info structure . . . 197
6.3 Tokenised CONSTANT_Array_Class_info structure . . . 198
6.4 Standard CONSTANT_String_info structure . . . 199
6.5 Standard CONSTANT_NameAndType_info structure . . . 200
6.6 Standard CONSTANT_*ref_info structure . . . 200
6.7 Standard field_info structure . . . 202
6.8 Tokenised field_info structure . . . 202
6.9 Standard method_info structure . . . 204
6.10 Tokenised method_info structure . . . 204
6.11 Example of code that performs an invokevirtual. . . . 205
6.12 Call to method var.m(int a, float b, Object c) . . . 206
6.13 Standard attribute_info structure . . . 209
6.14 Standard ConstantValue_attribute structure . . . 210
6.15 Tokenised ConstantValue1_attribute structure . . . 211
6.16 Tokenised ConstantValue2_attribute structure . . . 211
6.17 Standard Code_attribute structure . . . 212
6.18 Tokenised Code_attribute structure . . . 213
6.19 Standard Exceptions_attribute structure . . . 216
6.20 Tokenised Exceptions1_attribute structure . . . 217
6.21 Tokenised Exceptions2_attribute structure . . . 217

List of Tables

1.1 Results from Dixon *et al.*[28] 9
1.2 Constant Pool Entry Types 19
1.3 Flag Values used in Fields 21
1.4 Required attributes defined in the Java Virtual Machine Specification 24
1.5 Additional attributes defined in the Java Virtual Machine Specification 24
1.6 Places where CONSTANT_Utf8 entries are referenced 25
1.7 Places where CONSTANT_{Integer|Float|String} entries are referenced . . . 26
1.8 Places where CONSTANT_Class entries are referenced 28
1.9 Places where CONSTANT_Fieldref entries are referenced 29
1.10 Places where CONSTANT_NameAndType entries are referenced 31
1.11 Grammar for Encoding Field Types 33
1.12 Interpretation of *BaseType* 33

2.1 Comparison of Compression Schemes 82

3.1 Libraries used for each test case 107
3.2 Size of CLDC tests 108
3.3 Size of MIDP tests 109
3.4 Size of Javolution tests 109
3.5 Size of J2SE test 110
3.6 Number of tokens used during tokenisation 112
3.7 Usage of Virtual Method Tables 113

4.1 Number of tokens used in incremental tokenisation 138
4.2 Comparison to Global Tokenisation 139

4.3 Usage of Virtual Method Tables with Incremental Tokenisation 140
4.4 VMT Usage Compared to Global Tokenisation 140

5.1 Number of Virtual Method Table Entries 149
5.2 Usage of *putstatic* and *getstatic* instructions 153
5.3 Usage of *putfield* and *getfield* instructions 156
5.4 Usage of the *invokevirtual* instruction 158
5.5 Usage of *invokeinterface* instruction 160
5.6 Usage of the *invokestatic* instruction 162
5.7 Usage of the *invokespecial* instruction 164
5.8 Instruction Usage in Test Class 176
5.9 Timing Overheads Using RDTSC 186
5.10 *invokevirtual* Results Using RDTSC 188
5.11 KVM-Fast Results Using RDTSC 190
5.12 *invokevirtual* on the Tokenised Virtual Machine 191
5.13 Overall Summary of Adjusted Cycles/Call for all Tests 193
5.14 Execution times in clock cycles, taken from [76] 193

6.1 Standard constant pool entry type 196
6.2 Possible values for the *type* entry in a CONSTANT_Array_Class_info entry. . 198
6.3 Meaning of Bits in the *access_flags* component of *field_info* entries 203
6.4 *access_flag* values from the Java Virtual Machine Specification. 207
6.5 Additional *access_flag* values added to the tokenised VM. 208
6.6 Class File Attributes 210
6.7 Overall Size of class files (bytes) 221
6.8 Size of Descriptor Files (bytes) 222
6.9 Comparison to Pack format (using pack200 binary from Sun) (bytes) 222
6.10 Overall Size of class files and Descriptor file after Incremental Tokenisation (bytes) 223
6.11 Number of Constant Pool Entries by Type 225

A.1 Tokenised constant pool entry types 234
A.2 Possible values for the *type* entry in a CONSTANT_Array_Class_info entry. . 235
A.3 Meaning of Bits in the *access_flags* component of *field_info* entries 236
A.4 Additional *access_flag* values added to the tokenised VM. 237

Abstract

An object oriented language, such as Java, needs dynamic binding for method calls since the type of the target object will only be known at runtime. Desktop PCs have sufficient processor and memory resources that dynamic binding is not a significant bottleneck to performance. However, smaller devices such as mobile phones have much more limited resources, requiring efficient implementations. C++ makes use of dispatch tables (also called virtual function tables or just vtables) as a way of speeding up this dispatch. A given method call has an offset (or token value) associated with it, which is used as an index into the target object's vtable. The value stored in the vtable will be a pointer to the C++ function to be executed (similar to a function pointer in C). However, the multiple-inheritance support in C++ complicates this, often requiring a class to have a separate vtable for each super-class it extends.

Java Card (a reduced implementation of Java for smart cards) also uses virtual function tables. While Java does not have full multiple-inheritance as C++ does, it has the notion of an interface. Method calls are divided into two categories in Java, those where the declared type of the object is a class, and those where it is an interface. This allows for a form of multiple-inheritance without having multiple implementations for the same method. There are two different bytecode instructions for these, *invokevirtual* and *invokeinterface* respectively. In Java Card, only the *invokevirtual* instruction can be directly dispatched via the vtable. This leads to simpler vtables, but leaves the *invokeinterface* instruction to use a slower and more complicated dispatch mechanism.

This thesis presents a way to allocate tokens to methods such that both *invokevirtual* and *invokeinterface* can be dispatched via the same vtable and avoids the need of multiple tables as in C++. For tokenisation to succeed, all runtime dependencies must be present,

i.e. the class files for all libraries the application uses. These are needed to determine when methods do and do not need the same token values. An initial tokenisation scheme is presented, where the complete system must be tokenised as a single operation, that is, the application, any libraries it uses and the API. Next, this is extended to allow a new, previously unknown, set of class files to be added to an existing tokenisation (incremental tokenisation). For example, the first tokenisation could include the API and base libraries on a device, followed by a third-party library being added in the second pass and then an application can be added in the third pass. During each tokenisation the previous tokenisation does not need to be modified. This allows a device to be released with a tokenised Java system installed and then new applications can be developed, tokenised and released for that platform. The new application will operate as expected even though the tokeniser had no knowledge of the application at the time it tokenised the initial libraries.

The KVM (Kilobyte Virtual Machine) from Sun Microsystems is a reference virtual machine designed for mobile phone and other portable devices. It is shown that the vtable based dispatch can be between 3 to 45 times faster than the equivalent method dispatch in the KVM. The presence of vtables also removes the need for the symbolic information normally used for linking. The tokenisation concept was also carried further to apply to fields and the *getfield/putfield* and *getstatic/putstatic* instructions used to reference them. This allows for similar speedup and simplification when resolving these references. Further, removing the redundant linking information resulted in class files that were between 42 to 72 percent of their original size.

In mobile devices both processing power and memory can be in short supply. These resources are also limited by the amount of battery power available to run them, as faster processors and larger memories can require more power. This thesis therefore makes a significant contribution towards making Java code both faster to execute and smaller, two vital attributes for a language running on small, portable devices.

"I Shawn Haggett, certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text."

Candidate:

Shawn Haggett

Acknowledgements

Thanks to Professor Greg Knowles, my supervisor, for keeping me on track and for the guidance when I needed it. Also, thanks to (in no particular order): Graham Bignell, Darius Pfitzner, Tanya Bilka and all the staff at Flinders Uni for your help and support over the years. Finally, thanks to my parents, friends and family for putting up with me.

Chapter 1

Background

1.1 Introduction

This first chapter will begin with a general background on object oriented programming, then relating this to the Java language. This includes the various Java platforms, from desktop/server machines, down to Java Cards (a form of smart cards). Chapter 2 presents the current state of class file compression and optimisations for Java.

Chapters 3 & 4 present the tokenisation that has been performed on Java class files. Chapter 5 then discusses the compression that has been obtained and Chapter 6 brings this all together and explains how these modified class files are executed by a modified virtual machine. Finally Chapter 7 presents results to show that the tokenised classes are indeed faster to execute and smaller.

1.2 Object Oriented Programming

The term "Object Oriented Programming", or OOP, traces back to the Smalltalk language, created in the early 70's. Although Alan Kay has credited his previous work in mathematics, biology, the Burroughs 220 and 5000 systems, a system known as "Sketchpad" and finally the Simula language as all having lead up to the design for Smalltalk [52].

Wikipedia currently lists approximately 80 programming languages that have object oriented features [97]. This list includes common languages such as: Java, C++, C# and Objective-C, as well as many scripting languages such as: Perl 5, PHP, Python and Ruby.

More recently, Deborah Armstrong [10] made a review of the current literature to determine what should constitute the fundamental characteristics of OOP, coming up with 8 concepts: inheritance, object, class, encapsulation, method, message passing, polymorphism and abstraction.

1.2.1 Classes, Objects and Methods

A class defines what data will be stored (fields) and what operations (methods) an object will have. For example, an object to represent a Window within a GUI might have: a title for the window, a width and a height; all of which constitutes its data. Operations might include: "getTitle" to determine the current title of the window and "setTitle", to change the title of the window; along with corresponding operations for the width and height. The class will also define how to actually perform these operations (i.e. provide the code to be executed). In OO programming, method calls can be considered to be "message passing". One object will pass a message to another object, and based on the type of that message, a method will be selected and executed.

Creating an object of a class (known as "instantiation"), will involve allocating memory to store the object, and initialising the values of any fields in the object. In Java a special method called a "constructor" is run, to perform any initialisation required.

Another common feature found in OO languages is the ability to define "abstract" classes. An abstract class can specify the fields and methods of the class, but may omit some or all of the implementation for the methods. As such, objects cannot be made of an abstract class. The usefulness of this feature is not apparent without knowledge of inheritance, which is discussed in the next section.

1.2.2 Inheritance

Inheritance is one of the powerful features of OOP, allowing a sub-class (also called a child-class) to inherit its behaviour from a super-class (or parent-class). Common functionality is provided in the super-class, with sub-classes adding more specific functionality. An example of this would be GUI components, where the top of the hierarchy would consist of

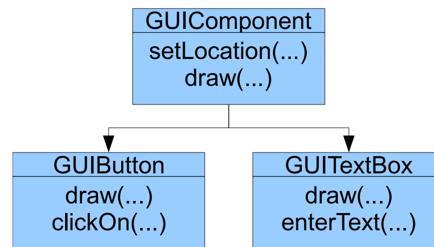


Figure 1.1: Example of Inheritance in a GUI library

a *GUIComponent* class. Methods needed by all types of GUI components, such as *draw()* to draw the component on the screen or *setLocation()* to position the element on the screen, would be declared in this class. Sub-classes of *GUIComponent* could then be made, such as *GUIButton* for a clickable button, or *GUITextBox* to allow text entry. Methods already present in the super-class are automatically available to objects of a sub-class (the methods are said to be "inherited" by the sub-class). Figure 1.1 shows the GUIComponent class with its two sub-classes, *GUIButton* and *GUITextBox*. All GUIComponents contain a method to draw them to the screen. The sub-classes "over-ride" the draw method, providing their own versions (the usefulness of this is presented in Section 1.2.3), while also providing behaviour specific to the type of class (in the case of *GUIButton*, a method to indicate the button was clicked on, for *GUITextBox*, a method to enter some text in the box).

Based on a languages support for inheritance, it can be described as supporting either single-inheritance or multiple-inheritance. In single inheritance, a class can have only one super-class, which limits flexibility. For example, if a library provided a class called TextHandler that could manage text (providing searching, find/replace and other features), this would be useful for implementing the GUITextBox from the previous example in Figure 1.1. In single inheritance it is not possible to extend both the GUIComponent and the TextHandler classes. Two possible solutions would be to either, make the GUIComponent class a sub-class of TextHandler or vice versa. These two scenarios are shown in Figure 1.2, however both of these have flaws. Firstly if GUIComponent extends TextHandler, then every GUIComponent would have the ability to handle text, which would not make sense for something like an image. In the other case, TextHandler could extend GUIComponent, but this means that all TextHandler objects are GUIComponents, however the class might

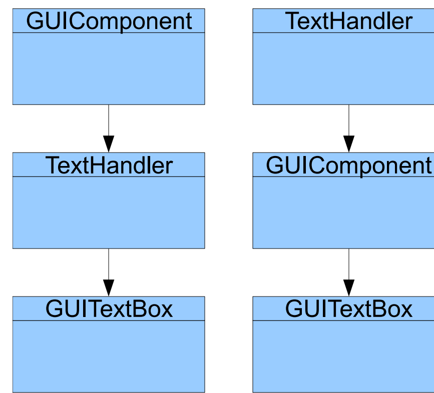


Figure 1.2: Attempting to use the TextHandler class

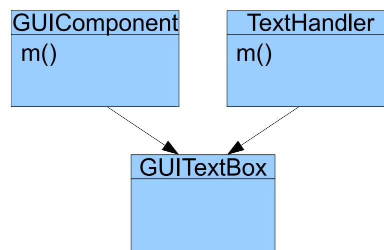


Figure 1.3: Example of multiple-inheritance

be used for in memory processing of text only, without the need for a GUI display. These solutions also assume that the GUIComponent and TextHandler classes can be modified. If these classes are both supplied by a library (either from the same library or two different ones), then it is quite possible the programmer will not be able to modify them.

Multiple-inheritance allows for a class to have more then one super-class, allowing the above scenario to be easily described. One common language to use multiple-inheritance is C++, but it comes with some drawbacks. For example, Figure 1.3 shows an example where the class GUITextBox has two super-classes, describing the desired situation, without all GUIComponents needing to be TextHandlers or vice versa. If GUIComponent and TextHandler were to have a method with the same name however, method calls can appear ambiguous. It depends on the language how these situations are resolved (in the case of C++, the declared type of an object is used to decide which of the methods to call).

Inheritance (whether single or multiple) can also be combined with abstract classes to form very powerful ways of representing data. An abstract class is one that can not be instantiated, but can contain common code and functionality for sub-classes to use.

For example, the GUIComponent class used above represents the generic component and probably should not be instantiated directly, rather sub-classes are used to provide specific functions (such as the GUIButton or GUITextBox classes). To prevent a programmer from creating GUIComponent objects, the class can be declared abstract, preventing it from being instantiated. In this way the programmer can have generalised classes, that can hold methods applicable to many different concrete classes, but protect against instantiating classes that do not have complete functionality.

An extension to the concept of abstract classes is that of abstract methods, which can only be found within an abstract class. An abstract method is one which specifies the name and parameters of the method and what it will return, but does not provide the implementation of the method. It is then up to any concrete class that extends the abstract class to provide an implementation for the abstract method. This allows the abstract class to guarantee that all sub-classes of it will have a given operation, but leave implementing it to the concrete classes.

Java makes use of a modified form of multiple-inheritance, one that involves the use of interfaces. An interface is like an abstract class, however it cannot define fields, and can only define abstract methods (the declarations in Java use the keyword *interface*, which implies all methods are abstract, therefore they do not include the *abstract* keyword). A class can therefore extend only one class (providing single-inheritance), but can extend ("implement" in Java terms) many interfaces. In Figure 1.3 there were two methods, both with implementations. However the lack of any implementation in an interface means this situation cannot arise in Java.

1.2.3 Polymorphism

Inheritance allows a variable declared as one type (e.g. GUIComponent) to contain any of that type's sub-types (e.g. GUIButton or GUITextBox). Since sub-classes can over-ride methods and change their behaviour, the same code can behave differently, depending on the type of object used. Applying this to the previous example in Figure 1.1, consider a class that needs to manage a GUI and draw the relevant parts to the screen. There are

Algorithm 1 Code using Polymorphism

```
GUIComponent[] components;
components[0] = new GUIButton();
components[1] = new GUITextBox();
...
foreach (GUIComponent gc in components) {
    gc.draw();
}
```

classes for managing specific types of content on the screen (so far, buttons for users to click on and text boxes for them to enter text into). This could be extended to displaying images, text, tabs, dialog boxes and many others. To ensure the GUI manager does not need to keep track of each of these different types separately, it can treat every object as its super-type, *GUIComponent*. From the perspective of the GUI manager, it will have a collection of GUIComponent objects and it will call the draw(...) method of each object to get the screen drawn.

Algorithm 1 shows some code using polymorphism. When the *draw()* method is called on each object, the appropriate method in each class will be called. The first call would result in the draw() method in the GUIButton class being called, and the second, the draw() method in the GUITextBox class would be executed. The important thing here is that in the loop the code does not need to differentiate, the draw() method is called the same way every time, with the runtime type of the object being used to determine which method to actually call.

When polymorphism is used, the target of a given method call cannot always be determined at compile-time. In the example in Algorithm 1, the same method call (i.e. "gc.draw();"), will result in different methods being executed, depending on the type of object stored in gc at the time. This means the target method can only be selected at run-time, once the type of gc is known.

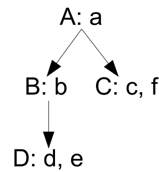


Figure 1.4: Example Class hierarchy

1.3 Polymorphic Call Dispatch

From the discussion in the previous section, polymorphic calls can only be dispatched at runtime, once the type of the receiving object is known. Since this dispatch mechanism is going to be used for every method call, it must be fast. Driesen [31] presents two extremes for dispatching methods, the first is Dispatch Table Search (DTS) and the second is Selector Table Indexing (STI). Four classes are used as a running example throughout this section: *A*, *B*, *C* and *D*, that form the class hierarchy shown in Figure 1.4. The lowercase letters denote the methods that are defined in that class.

In DTS, every class stores a list of the methods in that class, as well as a reference to the super-class. Object's contain a reference to the class which defines the object's type. To resolve a method call, the class reference in the object is used to get to the class and its list of methods. A search of that table is performed to find a match for the method being called. If no match is found, the search continues in the super-class. This continues recursively until either a match is found or there are no more super-classes. While this implements the semantics of an OO language, the runtime speed is unfeasibly slow, due to the large amount of searching required. The space requirements, however, are minimal.

On the other extreme is STI. Methods in the system are all assigned a token based on their selector (or method signature in Java), i.e. methods with the same signature will get the same token. A two dimensional lookup table is then created, with each class in the system as one dimension and every method selector as the second dimension. Each entry in the lookup table is a pointer to the target method (or null if the class does not implement that method). Figure 1.5 shows the resultant table for the example classes, where class *A* has the method *a*, class *B* inherits method *a* and also defines method *b* and so on. The coloured squares represent a pointer to a method implementation, while empty squares

Figure 1.5: Selector Table Index

are for when that class does not implement that method. Each class has been assigned a different colour purely for clarity in later diagrams. Resolving a method dispatch is then just a matter of using the object type and method selector to lookup the table. While STI is fast, the size of the table and the memory requirements for it will be large.

While neither DTS nor STI are very practical, they highlight the extreme cases. The ideal solution would be one that combines the best aspects of the two approaches, i.e. the small memory overhead of DTS and the efficient dispatch of STI.

1.3.1 Selector Colouring

Dixon *et al.* [28] discusses STI and proposes a method to reduce the size of the tables through the use of a technique called Selector Colouring (in object oriented languages a "selector" is the thing used to select a method, i.e. in Java, the method name and types of arguments). This was later applied to the dynamically typed Smalltalk language by Andre *et al.* [6]. Selector Colouring is similar to STI, however tokens are allowed to be aliased such that the same token value could represent multiple selectors, analogous to combining columns in Figure 1.5. By reducing the number of columns, the overall size, and therefore memory requirements, for the table are reduced. Two selectors can be aliased if they are never both used within any one class. For example, the selector *b* and the selector *c* are never used together in the same class, which can be seen visually in Figure 1.5 by the fact that the *b* column and *c* column could be overlaid without any overlapping entries, since the entries in the *b* column match up with "null" entries in the *c* column and vice versa. A "conflict graph" can be used to model this relationship, where every selector in the system becomes a node of the graph, while edges represent selectors that can not be aliased (i.e. two selectors that both appear within the one class, somewhere in the system). Figure 1.6 shows the resultant graph for the example classes. The task is then one of colouring

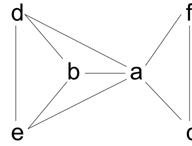


Figure 1.6: Conflict Graph for Example Classes

A	a			
B	a	b		
C	a	c	f	
D	a	b	d	e

Figure 1.7: Selector Colouring

the graph such that no two neighbouring nodes have the same colour (where a colour is analogous to a token value). Selector colouring leads on from previous work on register allocation using a similar graph colouring approach [18]. While graph colouring is an NP-Complete problem, it has been shown that approximations of the optimal solution can be found in a reasonable time [59].

Figure 1.7 shows the result of selector colouring for the example classes. The aliasing allows the size of the table to be reduced to only 4 columns, this being the largest number of methods a single class will accept. The heading across the top has been removed, since a particular column can be aliased to now represent several different methods. For example, the second column represents the “b” method in class *B* and *D*, but the “c” method in class *C*.

Table 1.1 reproduces the results presented by Dixon *et al.* [28]. Selector Colouring allows for a given token to be aliased to represent multiple different selectors. A selector, however, will always have the one token value representing it. The work presented in this thesis allows a selector to have more than one token value, thus giving greater freedom and token reuse than Selector Colouring alone. This work and results are presented in later chapters.

Selector Name Space	200	100	200	100
Number of Classes	50	50	100	100
Colours Used	27	23	33	26
Size of Table	1350	1150	3300	2600

Table 1.1: Results from Dixon *et al.*[28]

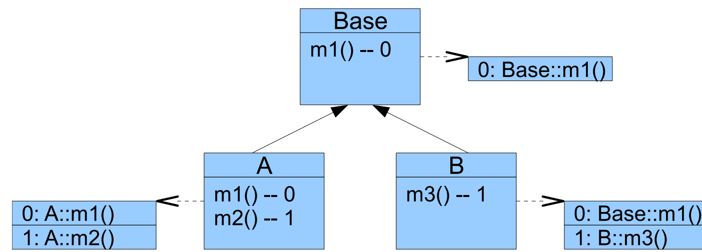


Figure 1.8: Single Inheritance and Virtual Method Tables in C++

1.3.2 Virtual Function Tables

In the STI scheme above, a single two-dimensional table is produced. A virtual function table (also referred to as virtual method tables or vtables) is similar to STI, except that instead of a global table, each class contains a one-dimensional table for the methods that can be called on objects of that class. The virtual function tables will be built at compile time by the compiler. The compiler will assign an offset in the virtual method table to each selector, which is the equivalent of assigning a token. The process of allocating tokens is different to STI however, by being performed first at super-classes, then moving down to sub-classes.

Figure 1.8 shows an example of single inheritance and the virtual method tables that would be built. The class *Base* has just the one method, which would be assigned token 0. The virtual method table shows the function that would be called for (using C++ syntax) “Base::m1()”, is the function m1() that was defined in the class *Base*. Class *A* then overrides the *m1()* method, as well as defining a second method. In the case of virtual method tables, overriding is as simple as changing the entry in the table to point to the new version of the function (as can be seen in Figure 1.8, with token 0 now referring to “A::m1()”). Class *B* shows a different case, where a new virtual function is added to the table by copying the *Base* class’s table then adding a new entry. The other important feature to observe here is that the two sub-classes (*A* & *B*) can use the same token, but for different functions. So far this scheme provides a constant speed lookup, similar to the STI approach above, but without a need for null entries and therefore less space overhead. This is true only when dealing with single inheritance.

C++ allows for a class to have more than one super-class, known as multiple-inheritance,

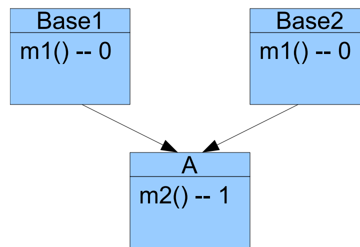


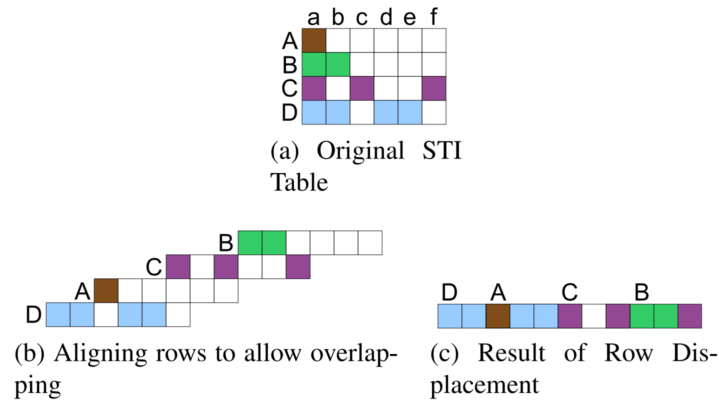
Figure 1.9: Multiple Inheritance in C++

as shown in Figure 1.9. This can create ambiguities at runtime as to which implementation of the *m1()* function should be called on an object of class *A*. To resolve this, two virtual method tables will be created, one for treating the *A* object like a *Base1* object, and the other for treating it like a *Base2* object. The decision on which table to use is then dependent on the declared type of the pointer that is used in the call. The first named super-class in the code is considered the "default" super-class, so a pointer of type *A*, would result in the *m1()* method in *Base1* being called, the same as if the pointer is a *Base1* type. However if the pointer is declared as pointing to a *Base2* type, then the method in that class would be called [38].

1.3.3 Row Displacement

Approaches such as Selector Colouring look to use STI for dispatch, but reduce the size of the selector table. In the case of Selector Colouring, this is through the aliasing of selectors such that they can use the same column in the selector table. Instead, Driesen [29, 30, 31] takes the approach of retaining the sparse STI tables, but storing them differently. Each row is taken and placed into a one-dimensional array, so that the used entries from one row fit into the unused ones in other rows.

Figure 1.10a shows the STI table produced for the example classes. Each row must be aligned so that none of its entries will overlap an entry from another class. Figure 1.10b shows such an alignment, where class *D* is the first class and the third entry in class D's row was a null, therefore class A's row has been fitted so that its one entry matches with the hole in class *D*'s row. Next is class *C*, placed so that its first entry overlaps the final entry from class D's row. Finally, class B's two entries have been placed into the gap within class



(a) Original STI Table

(b) Aligning rows to allow overlapping

(c) Result of Row Displacement

Figure 1.10: Row Displacement

C's row. These entries are then placed within a single "master array", as shown in Figure 1.10c.

The offset of a given class's row in the "master array" (i.e. the index of where the class's letter appears in Figure 1.10c) would need to be stored as part of the class information. Here the colours show which entries belong to which class, since the various rows have been overlaid. An additional check must be added to the dispatch process however, since a given class and selector reference which should result in a null entry could now point to an entry used by another class. For example, a call for method "b" on an object of class *A* should produce an error, however after row displacement, this would result in a reference to the method "d". Driesen presents several approaches to determine if a resolved method is actually valid or not.

1.4 Java

The Java language is found in many places, from the back-end of large, distributed applications (Java 2 Enterprise Edition), to tiny smart card devices (Java Card), although the term "Java" normally refers to the Java 2 Standard Edition (J2SE) designed for desktop computers. This is the direct descendant of a project that started at Sun back in 1991 [17], focused on the next generation of set-top boxes. Ultimately, the technology was never used for set-top boxes but the Sun engineers found a use for the programming language (then called Oak) on the Internet. Oak was a platform independent language, allowing small ap-

plications, named Applets, to be embedded into web pages. This language became what is known today as Java.

1.4.1 General Background

The original project at Sun Microsystems had been focused on the “next wave” of devices, and they had expected set-top boxes would be the place where new media and delivery systems would develop. However the Internet was gaining popularity for delivering text, graphics and video and Java would fit into this trend to provide “intelligence” to web pages. Java applets could be sent along with the page’s content and be executed by the client, providing animation, sound and interactivity not previously possible. By 1994 a prototype browser had been created that could execute these Java applets and this proved a huge success. In 1995 Java was officially announced to the world. Netscape Navigator was the primary browser at the time [12] and an agreement was made to include Java support.

Java applets were the beginning for what would become the Java 2 Standard Edition of Java, also known as J2SE. Aimed at desktop machines, with libraries providing support for GUIs, networking, file access and threading. Building on top of J2SE was the Java 2 Enterprise Edition, or J2EE, which added a framework and libraries for distributed applications, including: remote procedure calls, databases, managing data and its persistence and generating web pages. In this regard J2EE is a super-set of J2SE, by the addition of libraries and frameworks for developing distributed and web-based applications, but without removing or changing the J2SE virtual machine or API.

Heading in the other direction, Java 2 Micro Edition, or J2ME, is aimed at devices incapable of running the full J2SE. J2ME is further divided into two “configurations”; the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). The first is aimed at devices that typically run on mains power and have a stable network connection with potentially ample bandwidth, such as a set-top box or other fixed appliances. CLDC is aimed at smaller devices that may spend a large amount of their time operating on batteries and have only an intermittent network connection with possibly limited bandwidth, for example, mobile phones or PDAs. Both of these configurations have

their own API, however both can be seen as primarily a sub-set of the J2SE API. Due to the more limited nature of J2ME devices when compared to J2SE, the virtual machine is also slightly modified. The changes consist of: a simpler method of class file verification, a restriction on custom class loaders and simpler garbage collection.

Moving from J2ME to even smaller devices, Sun has developed the Java Card specification which deals with applications destined to run on smart card hardware. Java Card defines the contexts that applications can run in, how applications are linked, and the file formats used. Java Card differs from the other Java versions in that it heavily modifies all levels of the Java system. The virtual machine has a reduced instruction set, modified linking and verification mechanisms. Only an extremely limited version of the Java API remains, with a few additions to handle the unique context of Java Card applications.

The rest of this section will focus on the details of the J2SE version of Java. J2SE is defined in two documents, the Java Language Specification and the Java Virtual Machine Specification. This gives the background for the later detailed discussion of J2ME and Java Card.

1.4.1.1 Write Once, Run Everywhere

One of the main features of Java is its “write once, run everywhere” nature. Java programs are written and compiled to an intermediate form called “bytecode”; an instruction set defined by Sun. A Java virtual machine is an interpreter for these bytecodes, and also provides services for garbage collection and threads. Most of the tools and libraries for Java are all implemented in Java, with the exception of a small amount of native code.

Sun provides standard libraries, known as the API, that will be present for all Java applications, irrespective of the underlying platform. The API provides a standard interface for accessing services such as I/O or graphical displays. However, for these libraries to be implemented, they have to, at some point, utilise libraries available on the host operating system, which is performed via a mechanism that allows a Java application to call a method/function written in non-Java, or “native”, code. The consistent Java interface is exposed for application programmers to use, while internally these native calls are used to implement the correct behaviour. This means that a Java application, once written and

compiled, can be shipped to any platform that has a compliant VM, and will execute and behave in the same manner[1].

The ability to call native code, as well as being used to implement the Java API, is available for the application programmer to utilise. An application programmer would write and compile this native code in another language, typically C. The Java code for the application will include the method header, along with the special identifier NATIVE, to indicate that the implementation for the method is provided via native code. Finally the virtual machine is instructed (typically via command-line arguments) how to find the required native libraries, which are then dynamically loaded and linked during program execution.

1.4.1.2 Java Language Specification

The Java Language Specification (JLS) [61] is the primary document that describes the Java language, including the grammar and related productions that define the syntax for Java source files. The syntax includes, how to define classes, interfaces, fields, constructors and methods as well as the various modifiers that can be applied to them (like PUBLIC or PRIVATE). Situations that must be caught by the compiler, generating a compile-time error, are also identified.

Also covered in the language specification is how inheritance and over-riding work. Simply put, it describes the precise syntax a Java source file must have, and the meaning of everything in that file.

1.4.1.3 Java Virtual Machine Specification

The Java Virtual Machine Specification (JVMS) [57] provides the details of the file formats used, and how a VM must function to correctly implement Java. While the JLS defines the meaning of statements, the JVMS describes how a virtual machine must behave to ensure that the correct semantics are maintained at runtime.

[1] There are some cases where platform dependent behaviour is exposed to the application programmer. One common example of this is in file paths, where a program must take note of what system it is running on, for example, generating “C:\path\to\a\file.txt” on Windows or “/path/to/a/file.txt” on unix/linux machines.

Some of the important details defined in the JVMS include the format for a binary class file, the compiled form of Java (Chapter 4 of the JVMS), and the meaning of and how to interpret each possible instruction (Chapter 6 of the JVMS). Other details include the general nature of the virtual machine, how to load and link class files, how to compile class files and how to handle multithreading and locking.

When combined, the the JVMS and JLS define the basis of how the Java language is written, compiled and executed. While this includes the general nature and semantics of how a VM must behave, no specific implementation details are discussed, leaving the virtual machine implementer free to include various optimisations, provided the virtual machine will load standard class files, and the observable result is the same. Such optimisations can include Just-In-Time compilation, "fast bytecodes" or hardware implementation/support. Section 2.3 covers Java optimisations in detail.

1.4.2 Class File Format

The format for binary class files is given in Chapter 4 of the Java Virtual Machine Specification [57] and provides the format and meaning of the structures that will be found inside a binary class file, as well as details on how to verify a class file as being correct.

One of the primary goals of a class file is that it be as self-contained as possible. To this end, all references to items outside of the current class are done via symbolic information stored in the class's constant pool. This symbolic information is given in the form of strings, either for the name of the item, or in some cases, a specially formatted string to define the number and type of arguments.

To avoid issues of endianness, a class file is considered to be a stream of 8-bit bytes. All larger quantities are built from reading two or four consecutive 8-bit bytes. In these cases the bytes are always stored in big-endian order[2], no matter what the underlying architecture supports. These values are always considered to be unsigned. The terms *u1*, *u2* and *u4* are used to refer to these unsigned one, two and four byte values respectively.

A class file can be broken up into several main sections: Header, Constant Pool, Flags

[2]Big-endian means the first byte in a multi-byte value is the most significant byte.

& inheritance/interface information, Fields, Methods and Additional Attributes. Each of these sections are explained in more detail below, using an example class to highlight the contents of each section. The class is a simple “Hello World” program, consisting of:

```
public class HelloWorld {
    private String line = "Hello World";
    public static void main (String args[]) {
        new HelloWorld().print();
    }
    public HelloWorld() {}
    public void print() { System.out.println(line); }
}
```

This class has been compiled with the Sun compiler “javac”, version 1.6.0_13, to a class file that is 563 bytes long. Then the class file has been turned into hexadecimal output using the Linux command, “od -t x1”, to produce output such as:

```
0000000 ca fe ba be 00 00 00 32 00 25 07 00 15 0a 00 01
0000020 00 16 0a 00 01 00 17 0a 00 09 00 16 08 00 18 09
0000040 ...
```

The first value is the address in the file (presented in octal), then remaining columns are each a byte of data, displayed in hexadecimal.

1.4.2.1 Header

The class file specification requires that every class file start with the same 4-byte value, when written in hexadecimal, this value is 0xCAFEBABE and is referred to as the “magic” value. It provides a simple sanity check that a given file or piece of data does appear to be a class file, before attempting to parse it further. Following the magic value is a 2-byte minor and a 2-byte major version number. These two numbers denote which version of the class file specification that this file conforms to. For a given Java version $1.k$, where $k \geq 2$, the

class file version number is $(44+k).0$[3]. These virtual machines will all support loading of classes with a version number between 45.0 and $(44+k).0$ inclusive. For example, a Java 1.4 virtual machine will be able to load classes with a version number between 45.0 and 48.0 inclusive [91]. The header is found in the first line of the hexadecimal output:

0000000 **ca fe ba be 00 00 00 32** 00 25 07 00 15 0a 00 01
0000020 ...

This corresponds to:

u4 magic = 0xCAFEBABE
u2 minor_version = 0
u2 major_version = 50

1.4.2.2 Constant Pool

The Constant Pool is stored as an array of variable length entries. Each entry has a 1-byte "tag" value to indicate the type of the entry. The possible tag values are given in Table 1.2. These entries fall into two general types: constant values from the source code and symbolic linking information. The Type column shows which category each type of entry falls into. The exception is the CONSTANT_Utf8 entry which stores string data encoded using UTF8. These can be referenced from a CONSTANT_String entry to provide a string that had been declared in the source code. However they are also referenced from the CONSTANT_Class and CONSTANT_NameAndType entries to provide symbolic links to classes/fields/methods.

All references to constant pool entries are done using an index. The next two bytes in the class file correspond to the size of the constant pool:

0000000 ca fe ba be 00 00 00 32 **00 25** 07 00 15 0a 00 01

The hexadecimal value of 25 is equivalent to 37 in decimal. Constant pool indexes always start at 1, and proceed up to the size - 1, giving 36 possible indexes in this case. All entries

[3]Recent Java version have been advertised as Java 5 and Java 6. These correspond to Java 1.5 and Java 1.6 respectively.

Table 1.2: Constant Pool Entry Types

Tag	Description	Type
1	CONSTANT_Utf8	Both
3	CONSTANT_Integer	Constant Value
4	CONSTANT_Float	Constant Value
5	CONSTANT_Long	Constant Value
6	CONSTANT_Double	Constant Value
7	CONSTANT_Class	Linking
8	CONSTANT_String	Constant Value
9	CONSTANT_Fieldref	Linking
10	CONSTANT_Methodref	Linking
11	CONSTANT_InterfaceMethodref	Linking
12	CONSTANT_NameAndType	Linking

take up one index, except for the CONSTANT_Double and CONSTANT_Long entries, which use 2 indexes. For example, a CONSTANT_Double entry at index 5, would mean that index 6 is unused with the next entry being referenced with index 7. In the example class, there are 36 valid indexes and in this case there are 36 entries in the constant pool since this class does not contain any CONSTANT_Double or CONSTANT_Long entries.

Following the two bytes for the size of the constant pool, are the entries in the constant pool, consisting of the next 360 bytes (which is about 63.9% of the file size). Each entry begins with a 1-byte "tag", indicating the type of the entry and therefore how to read the rest of the entry. For example, the first entry consists of the bytes:

0000000 ca fe ba be 00 00 00 32 00 25 **07 00 15** 0a 00 01

The first byte, "07", indicates this is a CONSTANT_Class entry, which means the next 2-bytes, "00 15", are an index into the constant pool to a CONSTANT_Utf8 entry giving the class's name. All constant pool entries, with the exception of CONSTANT_Utf8 entries, have a fixed size. Since a CONSTANT_Utf8 entry holds the contents of a string, the second and third bytes in the entry give the length of the entry, as in this case form the example class:

0000060 1e **01 00 04 6c 69 6e 65** 01 00 12 4c 6a 61 76 61

The tag value "01" indicates a CONSTANT_Utf8 entry and "00 04" indicates there are four bytes in this string. The final 4-bytes in the entry, "6c 69 6e 65" corresponds to the

character codes for the string “line”, the name of the field used in the source code above. Section 1.4.3 gives more details on the types of entries and the places they are used, while Section 1.4.4 gives more detail on how symbolic links are resolved.

All up, the constant pool for this example class contains: 4 CONSTANT_Class, 4 CONSTANT_Method, 2 CONSTANT_Field, 5 CONSTANT_NameAndType, 1 CONSTANT_String and 20 CONSTANT_Utf8 entries. CONSTANT_Utf8 entries are both the most prolific and have the longest length, which is why the constant pool accounts for over half the class’s size.

1.4.2.3 Flags & Inheritance/Interface Information

Each class file contains a 2-byte value, after the constant pool, which is interpreted as a set of 16 single bit flags. These flags are used to denote certain attributes of the class, for example, if it is declared PUBLIC, FINAL or ABSTRACT or if it is an interface instead of a class[4]. In the example file, this can be found at:

0000560 29 56 **00 21** 00 01 00 09 00 00 00 01 00 02 00 0a

In this case, the hexadecimal value 21 corresponds to the ACC_SUPER and ACC_PUBLIC flags. ACC_PUBLIC indicates this is a public class, while ACC_SUPER is a historical flag that will always be set in new class files[5].

Following this are references to *CONSTANT_Class* entries in the constant pool which denote the name of both this class and the super-class. Next is a variable length list of references to *CONSTANT_Class* entries, denoting the interfaces this class implements. This information is used to determine the inheritance information and can be seen here:.

0000560 29 56 00 21 **00 01 00 09 00 00** 00 01 00 02 00 0a

The “00 01” means index 1 in the constant pool represents this class, while “00 09” mean index 9 contains a reference to the super-class. The final two values, “00 00”, indicates the number of interfaces this class implements (none in this case).

[4]The compiled form of Java is called a “class file”, which can represent either a class or interface.

[5]The *invokespecial* instruction used to have a different name and behaviour. This flag was used to select if the instruction should use the new or old behaviour. More detail is provided in Section 1.4.5.4.

Table 1.3: Flag Values used in Fields

Flag Name	Value
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004
ACC_STATIC	0x0008
ACC_FINAL	0x0010
ACC_VOLATILE	0x0040
ACC_TRANSIENT	0x0080

1.4.2.4 Fields

This section details all the fields that exist within this class. For each field, this information consists of: 16 1-bit flags, Field Name, Field Type and Additional attributes. In the example class, this information can be found at:

0000560 29 56 00 21 00 01 00 09 00 00 **00 01 00 02 00 0a**
0000600 **00 0b 00 00** 00 03 00 09 00 0c 00 0d 00 01 00 0e

The initial "00 01" indicates there is only 1 field declared in this class (which corresponds to the "line" field declared in the source code above).

The flags consist of two 8-bit bytes. Seven of these flags are currently used, with the rest reserved for future use. Of these seven, 3 are used to represent the visibility of the field (PUBLIC, PROTECTED, or PRIVATE) and only one of these may be set. The remaining four indicate if the field is declared STATIC, FINAL, VOLATILE or TRANSIENT. Table 1.3 shows the list of these flags and the corresponding value that represents them (each value consists of only a single bit turned on). For the example class, this can be found in the bytes "00 02", which corresponds to just the ACC_PRIVATE flag, meaning the field was declared private.

The field name and type entries consist of unsigned 16-bit values, which form an index into the constant pool to a *CONSTANT_Utf8* entry. The name will be the string by which the field is known and therefore the value that a class must use in a symbolic reference to this field. Typically this will be the same name as was used for the field in the source code, although the specification does not require this. The type string is a specially formatted

string which denotes the data type that this field is declared to store, which could either be one of the primitive types, or in the case of an object type, the name of the class. The name and type strings, along with the name of the class, form the symbolic information required to reference a given field. In the example, this consists of the values "00 0a" and "00 0b", which corresponds to the indexes of the utf8 values "line" (which is the same entry examined earlier in Section 1.4.2.2) and "Ljava/lang/String;" in the constant pool.

The final data is the "00 00" values, indicating the number of attributes associated with this field (in this case, none). Attributes are an extensible mechanism for adding new information to class files, although any attributes not in Sun's Java Virtual Machine Specification can not modify the semantics of the class file. The only attribute specified for fields is the *ConstantValue* attribute, used to provide an initialisation value for the field. Section 1.4.2.6 gives more detail on the attributes feature of class files.

1.4.2.5 Methods

Next in the class file is the methods section, which stores details for every method declared in the class file. As with previous entries, the first two bytes denote how many methods there are (three in this case). The example file contains (the two byte counter and the first entry are highlighted):

0000600 00 0b 00 00 **00 03 00 09 00 0c 00 0d 00 01 00 0e**
0000620 **00 00 00 27 00 02 00 01 00 00 00 0b bb 00 01 59**
0000640 **b7 00 02 b6 00 03 b1 00 00 00 01 00 0f 00 00 00**
0000660 **0a 00 02 00 00 00 06 00 0a 00 07** 00 01 00 10 00

The entry for each method has a similar structure to the fields above, containing 16 1-bit flags, a name, descriptor and list of attributes. The bit flags are used to denote the same sort of information as for fields, as well as flags for the keywords NATIVE and SYNCHRONIZED. The flags represent methods with a native implementation or those requiring synchronisation for multithreading.

The name and descriptor fields both point to *CONSTANT_Utf8* entries in the constant pool. As for fields, the name is typically the name of the method used in the source code

(although the specification does not require this). The descriptor is a specially formatted string that represents the number and type of arguments and the return type of the method. These two values provide the "signature" for the method, and when combined with the class name, can be used by a class to symbolically reference this method. This information is contained in the first six bytes of the method's entry ("00 09 00 0c 00 0d" in the example above, consisting of two bytes for each of: the flags, the method name and the method descriptor). Therefore the name consists of the value "00 0c", which is the index for the string "main" in the constant pool, while the descriptor has the value "00 0d", which is the index for the string "([Ljava/lang/String;)V".

The next two bytes ("00 01") denote the number of attributes associated with the method. The attributes section allows for arbitrary data to be associated with the method. There are two attributes defined in the JVMS that a virtual machine must understand, the *Code* and *Exceptions* attributes. In the example the *Code* attribute is present, which supplies the bytecode and information needed to execute the method (such as maximum number of local variables, maximum size of the stack and exception handlers). The *Exceptions* attribute (which is not present in the example) indicates the list of checked exceptions that this method is declared to throw. Attributes are covered in more detail in Section 1.4.2.6.

1.4.2.6 Additional Attributes

The last part of a class file consists of variable length attributes, which provide an extensible mechanism for extra (and previously unknown) data to be added to the class file. The general format of these attribute entries is the same as the attributes associated with field and method entries earlier in the class file. The specification requires that this extra information cannot modify the semantics of a class file, however the information could be used, for example, to include additional debugging information or symbols. To differentiate the attributes, each attribute can be given a name, which can be any string and will be stored in the constant pool in a *CONSTANT_Utf8* entry. Each attribute also has a length associated with it, allowing a class loader (or other program that must read binary class files) to skip past unknown attributes and continue reading the file.

The specification defines nine attributes, whose names are considered to be reserved

Table 1.4: Required attributes defined in the Java Virtual Machine Specification

Attribute Name	Usage
ConstantValue	Denotes a value that a field must have when it is initialised.
Code	Provides the bytecode for a method and the relevant details needed to execute the method.
Exceptions	Lists the checked exceptions a method is declared as being able to throw.
InnerClasses	Used for the implementation of nested classes and interfaces.

Table 1.5: Additional attributes defined in the Java Virtual Machine Specification

Attribute Name	Usage
Synthetic	Denotes a class, field or method that does not actually appear in the source code.
LineNumberTable	Debugging information to map bytecode instructions back to the line in the source code they represent.
LocalVariableTable	Provides the names for the local variables inside methods to assist in debugging.
SourceFile	The source file this file was compiled from. Used for debugging.
Deprecated	Indicates a class, field or method that has been marked as deprecated. Used to issue warnings at compile time for code that is using deprecated items.

and can only be used as specified. Any other attributes found in a class file must be silently ignored by a virtual machine that does not know about the attribute. Of the nine specified, the four shown in Table 1.4 are required for the correct operation of a virtual machine. The ConstantValue attribute will only be present in the fields section of a class file, while the Code and Exceptions attributes will only be found in the methods section, finally the InnerClasses attribute will only be found in the additional attributes section at the end of a class file. The remaining five attribute types shown in Table 1.5 provide debugging and additional information which may be used by compilers or debuggers, but are not needed by the virtual machine for correct execution of the code.

1.4.3 Usage of Constant Pool Entries

The constant pool stores a large amount of information relevant to a class file, ranging from constant values that are used by the class, to the linking symbols needed for runtime

Table 1.6: Places where CONSTANT_Utf8 entries are referenced

Location	Description
all attributes	The name for the attribute, which determines its type.
InnerClasses attribute	The name of the inner class.
SourceFile attribute	The name of the source file.
LocalVariableTable attribute	The name and field type for each local variable.
method_info	The name and descriptor for a method in this class.
field_info	The name and type for a field in this class.
CONSTANT_String	To provide the literal string data for use in the program.
CONSTANT_NameAndType	The name and type for a symbolic reference, to either a field or method.
CONSTANT_Class	A symbolic reference to a class.

linking. These entries can be referenced from many different places in a class file, or even from other constant pool entries. This section will go through each type of entry, detailing what data it stores, and from where it can be referenced. Every constant pool entry will begin with a single unsigned byte which denotes the type of the entry. The size of the entry and meaning of the bytes is dependent on this type value.

1.4.3.1 CONSTANT_Utf8

Each of these entries stores a string, encoded using a slightly modified UTF-8 format. These strings are used for labelling parts of the class file, linking symbols and string literal data used in the program. Table 1.6 shows the places that can reference a CONSTANT_Utf8 entry.

Section 1.4.2.6 explained the use of "attributes" by the class file format to store arbitrary data. The type of a given attribute is defined by a string in a CONSTANT_Utf8 entry in the constant pool. Each attribute has an index to the relevant string, allowing custom attributes to use any name, so long as it is not one of the names used for the attributes defined in the JVMS. A virtual machine unfamiliar with these custom attributes will just ignore them.

The InnerClasses attribute is used to implement inner and nested classes. In this case

Table 1.7: Places where CONSTANT_{Integer|Float|String} entries are referenced

Location	Description
ConstantValue attribute	The initial value of a field.
Code attribute	To push a constant value onto the stack.

the string defines the original name of an inner or nested class.

SourceFile and LocalVariableTable attributes are used for debugging. The SourceFile attribute stores the file name of the source code that produced this file. This allows a debugger to relate the file being executed back to the source code that produced it. The LocalVariableTable is needed since local variables, unlike fields, do not have their names or types stored in a class file. Instead the compiler turns local variable references into a VM instruction involving a numbered slot in the frame's local variable table. Therefore the LocalVariableTable can be used by a debugger to relate these statements back to the relevant local variable in the source code.

The method_info and field_info sections are where the definitions of methods or fields in a class are stored. Each definition requires two references to Utf8 entries to provide the name and descriptor for a method and the name and type for a field.

The CONSTANT_String entries in the constant pool represent a string literal value that was used in the source code of the program. When the code needs to reference this value, it does so via a CONSTANT_String entry, which in turn points to a Utf8 entry to provide the actual string data.

The final place that can reference a Utf8 entry is the CONSTANT_NameAndType or CONSTANT_Class entry, which are used to specify symbolic references to other fields or methods. In particular the NameAndType entry specifies the name and type of a field, or name and descriptor of a method, by providing indexes to the two Utf8 entries for these strings. The Class entry provides an index to a single string, which is the name of a class. There are more details on how linking in Java works in Section 1.4.4.

1.4.3.2 CONSTANT_Integer, CONSTANT_Float, CONSTANT_String

CONSTANT_Integer, CONSTANT_Float and CONSTANT_String entries all hold a constant value that was used in the source code and can be referenced from the two locations shown in Table 1.7. The first is in a ConstantValue attribute, attached to a field. For example, if a field has been declared as:

```
private int i = 20;
```

Then the field entry will contain a ConstantValue attribute, which in turn contains a reference to a CONSTANT_Integer entry holding the value "20". If no such attribute exists for a field, the default value for the field's type will be used.

The other place these entries can be used is inside a Code attribute. In particular, the *ldc* and *ldc_w* instructions give an index to a CONSTANT_Integer, CONSTANT_Float or CONSTANT_String entry, the value of which is then pushed onto the stack. The only difference between the instructions is that the *ldc* instruction uses a single byte as the constant pool index, while *ldc_w* uses a two byte index.

1.4.3.3 CONSTANT_Long, CONSTANT_Double

CONSTANT_Long and CONSTANT_Double entries, like the CONSTANT_Integer and CONSTANT_Float, can be referenced from a ConstantValue attribute in a field or from a Code attribute. However the *ldc* and *ldc_w* instructions are only for 32-bit constant values. The 64-bit long and double values use the *ldc2_w* instruction instead.

1.4.3.4 CONSTANT_Class

A CONSTANT_Class entry contains the index of a Utf8 entry, which is the internal name for a class (Section 1.4.4.1 gives details on the internal format of class names). Table 1.8 contains a list of places that need to reference classes. In each case, a reference to a CONSTANT_Class entry is given, to indicate the required class. . The CONSTANT_{Fieldref|Methodref|InterfaceMethodref} entries all need to indicate the class that the referenced item will be found in, hence they will link to a CONSTANT_Class entry.

Table 1.8: Places where CONSTANT_Class entries are referenced

Location	Description
CONSTANT_Fieldref	The class the referenced field is in.
CONSTANT_Methodref	The class the referenced method is in.
CONSTANT_InterfaceMethodref	The class the referenced method is in.
Code attribute	Various instructions that need to reference classes.
Exceptions attribute	To list the exceptions a method can throw.
InnerClasses attribute	To list the inner and outer classes.

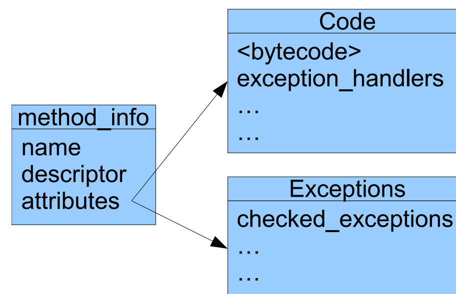


Figure 1.11: Use of CONSTANT_Class entries in methods

```
public void method() {
    someOperation();
    try {
        otherOperation();
    } catch (Exception e) {
        handleException(e);
    }
}
```

Figure 1.12: Example of a try/catch block

Table 1.9: Places where CONSTANT_Fieldref entries are referenced

Location	Description
Code attribute	For instructions that load/store data from/to fields.

Figure 1.11 shows the Code and Exceptions attributes which are both found as attributes of a method, and can both reference CONSTANT_Class entries. The Exceptions attribute will contain a list of indexes to CONSTANT_Class entries, indicating which checked exceptions the method could throw (as indicated by the *throws* keyword in the method declaration. The code attribute contains the bytecode that implements the method and an exceptions table, for implementing try/catch blocks. Figure 1.12 shows a method with a try/catch block in it. During execution of the method call “otherOperation()”, if an exception is thrown, execution will proceed to the *catch* block, resulting in execution of the “handleException(e)” line. Each exception handler in the Code Attribute indicates a range of bytecodes (corresponding to the source code that was covered by a *try* block), an exception type (via a reference to a CONSTANT_Class entry) and a target instruction. If that exception (or a sub-class of it) is thrown while executing any of the instructions in the given range, the virtual machine will jump to the target instruction (which is the code from the *catch* block).

References can also be found from within the bytecode in a Code attribute. The instructions *anewarray* and *multianewarray* for creating new arrays of objects, or multi-dimensional arrays of objects both reference a Class entry defining the type of objects the array will hold. The *new* instruction will create a new object, and references a Class entry to indicate the type of the object. Finally the *instanceof* instruction tests if an object on the top of the stack is an instance of the referenced class or *checkcast* to test if an object can be cast to the referenced type.

1.4.3.5 CONSTANT_Fieldref

This is a symbolic reference to a field, either in this class or in another class. Table 1.9 shows that these entries can be referenced from within the bytecode of a method, as found in a Code attribute. The *getfield*, *putfield*, *getstatic* and *putstatic* instructions are used to

load or store a value for a field in an object, or for a static field in a class. In either case, this entry is the symbolic reference to which field the data will be loaded from or stored into. More details on how this type of entry is used to perform linking in Java is given in Section 1.4.4.

1.4.3.6 CONSTANT_Methodref

CONSTANT_Methodref entries provide a symbolic reference to a method, either in this or another class. There are three instructions that can reference a Methodref entry: *invokestatic*, *invokevirtual and invokespecial* The *invokestatic*, *invokevirtual* instructions are for calling a static or non-static method respectively, while *invokespecial* provides special handling of super-classes, for implementing the "super" keyword. More details on how this type of entry is used to perform linking in Java is given in Section 1.4.4.

1.4.3.7 CONSTANT_InterfaceMethodref

All non-static method calls must be performed on an object, which will have a declared type (i.e. the type given for the variable in the source code that is holding the object). If the declared type of an object is an interface type, then the *invokeinterface* instruction is used, which contains a reference to a CONSTANT_InterfaceMethodref entry. When the declared type is a class, then the *invokevirtual* instruction (with a reference to a CONSTANT_Methodref entry) is used as described above, even if the method implements a method in an interface. More details on how this type of entry is used to perform linking in Java is given in Section 1.4.4.

1.4.3.8 CONSTANT_NameAndType

The Fieldref, Methodref and InterfaceMethodref entries all need to provide the name of the field or method, and the type or descriptor. Hence these entries are referenced from CONSTANT_Fieldref, CONSTANT_Methodref and CONSTANT_InterfaceMethodref entries, as shown in Table 1.10. More details on how this type of entry is used to perform linking in Java is given in Section 1.4.4.

Table 1.10: Places where CONSTANT_NameAndType entries are referenced

Location	Description
CONSTANT_Fieldref	Provides the name and type of the symbolic field reference.
CONSTANT_Methodref	Provides the name and descriptor of the symbolic method reference.
CONSTANT_InterfaceMethodref	Provides the name and descriptor of the symbolic method reference.

1.4.4 Linking in Java Class Files

Each class of a Java application is compiled into a separate file. To resolve the targets of method calls or field references Java uses three strings. The first of these is the name of the class that the target is in. Next is the name used for the field or method in the source file. Finally a descriptor, which for a method defines the types of arguments and return type, or for a field just the type of the field. By using strings, a library can be recompiled, providing no fields or methods are removed, change types or change behaviour, and all other applications will continue to function without needing to be recompiled themselves. Therefore a given class must also contain the strings to describe all the fields and methods that are declared in that class.

References to fields or methods are made via the constant pool, for example, an instruction to call a method will provide an index into the constant pool to a CONSTANT_Methodref entry, which in turn has references to other constant pool entries, ultimately providing the three strings mentioned above (class name, method/field name and descriptor). Figure 1.13 shows an example of a CONSTANT_Fieldref entry and the other constant pool entries involved. The CONSTANT_Fieldref entry provides the index of a CONSTANT_Class and a CONSTANT_NameAndType entry, which in turn provide the indexes of the three strings. A CONSTANT_Methodref or CONSTANT_InterfaceMethodref have the same form as the CONSTANT_Fieldref in the example. The types of constant pool entries was introduced in Section 1.4.3.

The following sections describe method and field references in more detail, along with the role that different constant pool entries play.

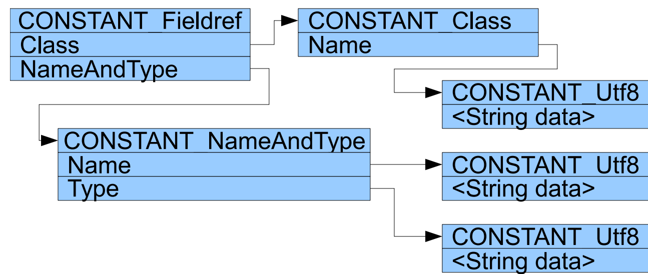


Figure 1.13: Structure of a symbolic field reference

1.4.4.1 Class References

Symbolic references to methods and fields both require a way to indicate the relevant class, so that a class loader may locate and load that class. The fully qualified name of the class, that is, not only the class name, but the package name that it belongs to, is used to provide this reference. For example, the *Thread* class has the fully qualified name "java.lang.Thread". However for historical reasons, the elements in a fully qualified class name are separated by '/' instead of the usual '.' when used internally in class files, meaning that while users would refer to the class as "java.lang.Thread", internally a class file would refer to it as "java/lang/Thread".

A CONSTANT_Class entry contains the 1-byte tag entry (to indicate the type of entry), followed by an unsigned 2-byte value to provide an index to a CONSTANT_Utf8 entry. The value of that entry is a string that contains the class's fully qualified name. Mapping from this string to the actual class file is then the job of a class loader and is implementation dependent.

1.4.4.2 Field References

There are three strings that are required for a symbolic reference to a field: class name, field name and field type. The format of these strings is defined in the Java Virtual Machine Specification [57]. The class name is described in the section above. The field name will be the string that was used to identify the field in the source code. Finally, the type string is a specially formatted string that is generated by the compiler to denote the type of value the field stores. Section 4.3.2 of [57] describes the syntax used to generate the field type string.

Table 1.11: Grammar for Encoding Field Types

Non-terminal	Production
FieldType	*BaseType*, *ObjectType*, *ArrayType*
BaseType	B, C, D, F, I, J, S, Z
ObjectType	L<*classname*>;
ArrayType	[*FieldType*

Table 1.12: Interpretation of *BaseType*

BaseType Character	Type	Interpretation
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
S	short	signed short
Z	boolean	true or false

Table 1.11 summarises the syntax for field type strings, which can consist of one of three main types: *BaseType*, *ObjectType* or an *ArrayType*. A *BaseType* represents one of the Java primitive types using a single letter. Table 1.12 shows the meaning of each of these letters. An ObjectType represents an object reference, where the "<classname>" part of the production is replaced with an internal class name, for example, a field declared as "Thread t;", would have the type "Ljava/lang/Thread;" (see Section 1.4.4.1 for a description of internal class names). An *ArrayType* represents an array reference and consists of a '[', followed by a *FieldType*, allowing arrays to contain primitive types, object references or even other arrays (as in the case of multi-dimensional arrays).

1.4.4.3 Method References

A method reference consists of three strings (class name, method name and descriptor), which are all stored in the constant pool. Each method defined in the class, as well as for all methods that are called from that class, will have their respective strings in the constant pool.

For each reference, the class name indicates the class that contains the method (the

format of class references was covered in Section 1.4.4.1). The method name will be the exact name used for that method in the source file. The method descriptor is a specially formatted string to show the number and type of arguments and the return type of the method. The syntax for these strings is given in Section 4.3.3 of [57] using the follow grammar:

MethodDescriptor:

(ParameterDescriptor) ReturnDescriptor*

Therefore a method descriptor will consist of a left parenthesis, zero or more *ParameterDescriptor* values, a right parenthesis then a *ReturnDescriptor*. A *ParameterDescriptor* will consist of:

ParameterDescriptor:

FieldType

While a *ReturnDescriptor* will be:

ReturnDescriptor:

FieldType

V

The 'V' represents the *void* type, while the *FieldType* production is detailed in Section 1.4.4.2, where it was used to encode the type of a field, and can represent either a primitive, object or array. For example, part of the contents of the java.lang.Object class consists of (the unimportant parts for this example have been replaced with "<...>"):

```
package java.lang;
class Object {
    <...>
    String toString() { <...> }
}
```

To provide a symbolic reference to the *toString* method above would require the strings: "java/lang/Object", "toString" and "()Ljava/lang/String;"; which consist of the class name, method name and method descriptor respectively. In this example the descriptor consists of empty parentheses, indicating the method takes no parameters, followed by "Ljava/lang/String;", indicating it returns a String object.

1.4.4.4 Resolving Method References

When a symbolic method reference is given, as described in Section 1.4.4.3, the reference must be resolved at runtime. Java differentiates between a method call that has a declared type which is an interface and one which is a class. This gives two types of method references, an *interface method reference* when the target method is in an interface type and a *method reference* when the method is not. If the declared type is a class, then an *interface method reference* is never used, even if the method happens to implement an interface method. The process for resolving the two types of references is very similar.

Chapter 5 of the Java Virtual Machine Specification [57] covers the broad topic of Loading, Linking and Initialising in the context of the virtual machine. The specific sections of interest to resolving method references are: Section 5.3 covers creation and loading of class files, Section 5.4.3.1 gives specific steps for resolving an unresolved class reference, Sections 5.4.3.3 and 5.4.3.4 give the specific steps for resolving a method reference and an interface method reference respectively.

For both interface method references and method references, the class name given in the reference is used to locate the class or interface with that name. If a matching class or interface cannot be found, then an error is thrown. The matching class is loaded, linked and initialised (if it has not already been). In the case of an interface method reference,

the resolved class must be an interface, else an error is raised. A recursive search is then made from this interface, up through any super-interfaces, until a matching method is found (i.e. one with the same name and descriptor). If no matching method can be found, then a *NoSuchMethodError* is thrown.

In the case of a method reference, the resolved class must be a class and not an interface. Then a similar recursive lookup is performed, this time searching from the class and up through any super-classes. If this fails, a recursive search is performed from the class up through any interfaces it implements and in turn their super-interfaces. If this search also fails to find a matching method, then a *NoSuchMethodError* will be thrown.

Resolution of method references are done during class linking to ensure that, for every method reference, a target method can be found, although due to polymorphism, this method will not always be the correct method to execute. When an instruction refers to a method via a method reference, a new lookup is performed then, to find the correct target method for that call. Which lookup algorithm is used depends on the type of instruction. Section 1.4.5 details the different instructions for performing method calls and their behaviour.

1.4.4.5 Symbolic References

Every Java class file uses symbolic references when referring to another class/method/field. In the case of virtual method calls, where polymorphism can be involved, the compiler cannot resolve the final target of a method call. Unlike virtual method calls however, static method calls can be resolved to a target method at compile time. Consider the classes shown in Figure 1.14. An application that uses this library might contain a reference such a method as:

```
Child.staticMethod();
```

During compilation the compiler can resolve that this will cause the Parent.staticMethod() to be called and could produce a class file with a reference directly to the Parent class. This would function as expected when executed. If the library was modified and a *staticMethod()* method added to the Child class without recompiling the application, the be-

```
public class Parent {
    public static int intField = 0;
    public static void staticMethod() {}
}
public class Child extends Parent {
}
```

Figure 1.14: Example of Classes in a Library

haviour will now be inconsistent. The application will still call the compiler resolved method Parent.staticMethod(), even though the source code would suggest that the Child.staticMethod() method should be called. This leads to hard to find bugs and binary inconsistencies.

This same situation also applies for static field references. Static field/method references are therefore always compiled with a symbolic reference to the type used in the source code. A similar situation can also appear for references to a super-class, i.e. when using the *super* keyword to call the super-class's version of an overridden method. Again the final target could be several classes up in the inheritance, but the symbolic reference must always be to the direct super-class.

Hence at runtime, any and all symbolic references can require searching through one or more classes to resolve. Any attempt to resolve them earlier will introduce differing behaviours when parts of a program are modified. Once the virtual machine has performed the expensive resolution the first time however, it can cache the resolved target. This avoids the expensive resolution if the same symbolic reference is used again.

1.4.5 *invoke** Instructions

Java bytecode (the compiled form of Java) has 4 different instructions for representing method calls, depending on the situation. These instructions are named: *invokevirtual*, *invokestatic*, *invokeinterface* and *invokespecial*, collectively referred to as the *invoke** instructions. All these instructions have one operand which provides (via the constant pool) a method reference for the instruction. A method reference consists of a class name, a method name and a method descriptor and is detailed in Section 1.4.4.3. The difference

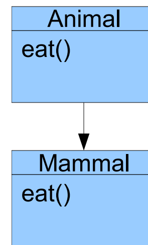


Figure 1.15: Classes with over-riding

between the 4 instructions is the type of method they are allowed to reference, and how the call is resolved.

The simplest instruction is the *invokestatic* instruction that is used for calling static methods. Since the target method is a static method, there is no object reference needed. The method found through the method resolution process will be the method to call and must be a static method, else an error is raised.

The *invokevirtual* instruction is used to perform virtual method calls. Being non-static, there must be an object reference on the operand stack and that object's class type is determined, which must be the same as, or a sub-class of, the declared type for this method call (i.e. the declared type of the variable in the source code, which was then stored in the symbolic method reference in the constant pool, as discussed in Section 1.4.4.3). Considering the classes in Figure 1.15, it could be possible to have the following code:

```
Animal a = new Mammal();
a.eat();
```

The symbolic reference for the *a.eat()* method call will consist of the class name: “Animal”, method name: “eat” and descriptor: “()V”. During linking, this will resolve to the *eat()* method in *Animal*, however at runtime, the object will be of type *Mammal*. This is valid, since *Mammal* is a sub-class of *Animal*. However the method resolved during linking is incorrect since *Mammal* has over-ridden it. Therefore, the *invokevirtual* instruction will cause a search, starting at the object's type, up through the inheritance structure, looking for a matching method. The *invokevirtual* instruction will be used for methods with public, protected and default levels of visibility. In particular, private methods are invoked using the *invokespecial* instruction.

For method calls where the declared type in the source code is an interface type, the *invokeinterface* instruction is used. Here the resolved method will be a method in an interface. The object reference on the stack is checked to find its class type (which must implement the resolved interface, or an error is raised). Then a recursive search is made from the class type, up through any super-classes, until a matching method is found. This process is very similar to *invokevirtual*, except that the resolved method will be in an interface instead of a class.

The final instruction, *invokespecial*, is used to call constructors, private methods and methods in a super-class. The instruction has the same symbolic method reference (via constant pool entries) as the other *invoke** instructions, which is resolved to method. If the method is an instance initialisation method (i.e. a constructor) or the class of the resolved method is not a super-class of the current class (the class this instruction is in), then the resolved method is executed, thus dealing with the constructor and private method cases. When implementing the SUPER keyword for calling non-constructor methods in the super-class, the resolved method will be in a super-class, and a modified version of the *invokevirtual* lookup procedure is used. All three of these cases are detailed, in the following sections.

1.4.5.1 Constructors and *invokespecial*

During compilation of a constructor, the compiler will add a call to the super-class's constructor at the start if one is not present. For example, a class might look like:

```
public class Parent extends Object {
public Parent() {
        }
}
```

However the constructor in this class will be compiled as if the code had looked like:

```
public Parent() {
```

```
    super();
}
```

This means the first instructions in a constructor always involves calling the super-class's constructor (except for java.lang.Object, which has no super-class).

During compilation constructors are also given special names. While in the source code a constructor must always have the classes name, when compiled the method will gain the name <*init*>. Since the '<' and '>' characters are not valid in Java method names, it is impossible for a non-constructor method to have this name. This allows the virtual machine at runtime to identify constructors and prevent them from being called with anything other than an *invokespecial* instruction.

A typical section of Java bytecode to create an object of the Parent class might look like:

```
NEW PARENT; //new object
DUP; //duplicate the object reference
ASTORE_1; //store the object reference
INVOKESPECIAL PARENT.<INIT>(); //call the constructor
```

While the call here would behave in the same fashion as an *invokevirtual* instruction, the same is not so in the Parent's constructor, which would look like:

```
ALOAD_0;
INVOKESPECIAL JAVA.LANG.OBJECT.<INIT>();
RETURN;
```

An *invokevirtual* instruction in this situation would just resolve the *<init>()* method in the Parent class again. When the resolved method is a constructor however, the *invokespecial* instruction will simply call the resolved constructor, instead of performing any further lookup. This ensures that the above reference to the Object class will cause that class's constructor to be executed.

1.4.5.2 Private methods and *invokespecial*

When the symbolic method reference resolves to a method that is not in a super-class of the current class (i.e. the class that contains the *invokespecial* instruction being executed), then the resolved method is the one that will be executed. This causes the call to behave in a non-virtual manner, which is essential when calling a private method. If a virtual method call were used, then a sub-class could over-ride a private method and hence intercept calls to it.

1.4.5.3 The SUPER keyword and *invokespecial*

The final case for an *invokespecial* instruction is when the resolved method is in a super-class of the current class and is not a constructor. This will trigger a lookup procedure similar to *invokevirtual*, however the search will start from the super-class of the current class, instead of the class of the object the method was called on. Consider the classes shown in Figure 1.16, where the m() method in the Child class will call the version in the super-class that it overrides (along with performing other tasks). This call would be compiled into Java bytecode similar to:

```
ALOAD_0; //load the this reference.
INVOKESPECIAL PARENT.M();
```

The symbolic reference for the *invokespecial* instruction will consist of the super-class (in this case, the class Parent). This will resolve to the m() method in the Parent class (however it could have been to a method high up the inheritance tree). This will meet the criteria (resolved method in a super-class and resolved method not a constructor) for a modified *invokevirtual* search process. A typical *invokevirtual* would start from the type of the object and search from that class up through the super-classes to find the first matching method. The same is done here, however the search will begin at the super-class of the current class (i.e. the one that contains this method, the class Child in this case). This ensures that the method will be in one of the super-classes, as the semantics of the *super* keyword require.

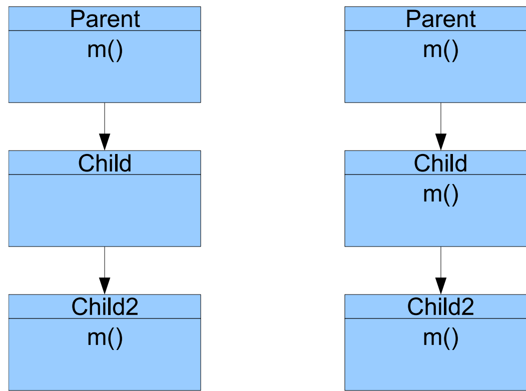


Figure 1.16: Method using the SUPER keyword

1.4.5.4 Historic use of *invokespecial*

In Java versions prior to 1.0.2, the *invokespecial* instruction was named *invokenonvirtual* and its behaviour was different. Instead of performing a search, it would always execute the resolved method (i.e. execute it in a non-virtual manner). Using this mechanism to call methods in a super-class is not reliable if the super-class is modified at a later date. This is the same as the example above for the *invokestatic* instruction and the classes shown in Figure 1.14. If non-static methods were used, and an application extends the Child class with a call using the super keyword, then an *invokenonvirtual* instruction would have to indicate the method in the Parent class. When a new version of the library is released with a new matching method in Child, the method would not be called as expected.

To provide for backwards compatibility, an extra bit-flag in a class file is provided to select the behaviour of the *invokespecial* instruction. The flag appears in the *access_flags* component of the class file, just after the constant pool, which contains 16 single bit flags. Only 5 of these flags are used, the rest are reserved for future use. One of these flags, named "ACC_SUPER", controls the behaviour of the *invokespecial* instruction. Early Java compilers would always set this flag to 0, and as such cause the *invokespecial* instruction to behave the same as the historic *invokenonvirtual* instruction. Newer compilers will always set the flag to 1, causing the newer behaviour to be used[6]. The work presented herein is targeted at versions of Java newer than 1.0.2 and therefore the old behaviour will not be

[6] It was observed that the Sun Compiler would not set the ACC_SUPER flag in the case of an interface (even though Sun's specification says all compilers should set the flag). Since an interface never contains any code however, the setting of the ACC_SUPER flag is irrelevant.

considered and is mentioned here only for completeness.

1.4.6 Usage of Java Instruction Set

The goal of this thesis is to tokenise the *invoke** instructions so that their implementation will be possible in hardware. During the creation of the tokenisation scheme, field instructions for loading and storing values in fields will also be considered. Kent measures the instruction usage of several applications in his thesis [53]. A histogram for each test in Kent is presented to visually show the number of times each instruction was used. These tests included:

Compress "Modified Lempel-Ziv method (LZW) finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly" [53]. Instruction usage shown in Figure 1.17.

DB "Performs multiple database functions on a memory resident database. It reads in a 1 MB file which contains records with names, addresses and 24 phone numbers of entities and a batch file which contains a stream of operations to perform on the records in the database" [53]. Instruction usage shown in Figure 1.18.

Mandelbrot "Generates a 320x240 picture of the mandelbrot set with a maximum iteration of 2000 for each pixel in the graph" [53]. Instruction usage shown in Figure 1.19.

Queen "A programming solution for the n-queens problem. It uses a tree-parsing approach of recursively placing pieces, but trimming away incorrect solutions at the first sign of failure" [53]. Instruction usage shown in Figure 1.20.

Raytrace "A raytracer that works on a scene depicting a dinosaur" [53]. Instruction usage shown in Figure 1.21.

Each figure has been divided into several sections, to denote the types of Java instructions, these consist of:

Data Manipulation Instructions for loading, storing or converting data, either primitive types or object references, including loading and storing values between local variable slots and the operand stack and loading and storing values in arrays.

Arithmetic Instructions for arithmetic operations.

Branching / Return Instructions for branching (i.e. if statements, logical comparisons, goto) and for returning from function calls.

Field / Invoke Instructions for loading/storing values in fields and for invoking methods.

Misc. Remaining instructions, includes creation of objects and arrays, throwing exceptions, checking the type of an object and thread synchronisation.

The loading and storing of values and arithmetic operations map very closely to instructions commonly present in hardware processors and is therefore already efficient to implement. However, the field and invoke instructions do not map so cleanly onto a hardware implementation, yet as shown here, those instructions see common use. Implementation of these instructions via microcode or trapping to software will represent a very large overhead. This strengthens the justification for focusing on the optimisation of the *invoke** instructions in this thesis, to achieve an efficient and direct execution of these instructions.

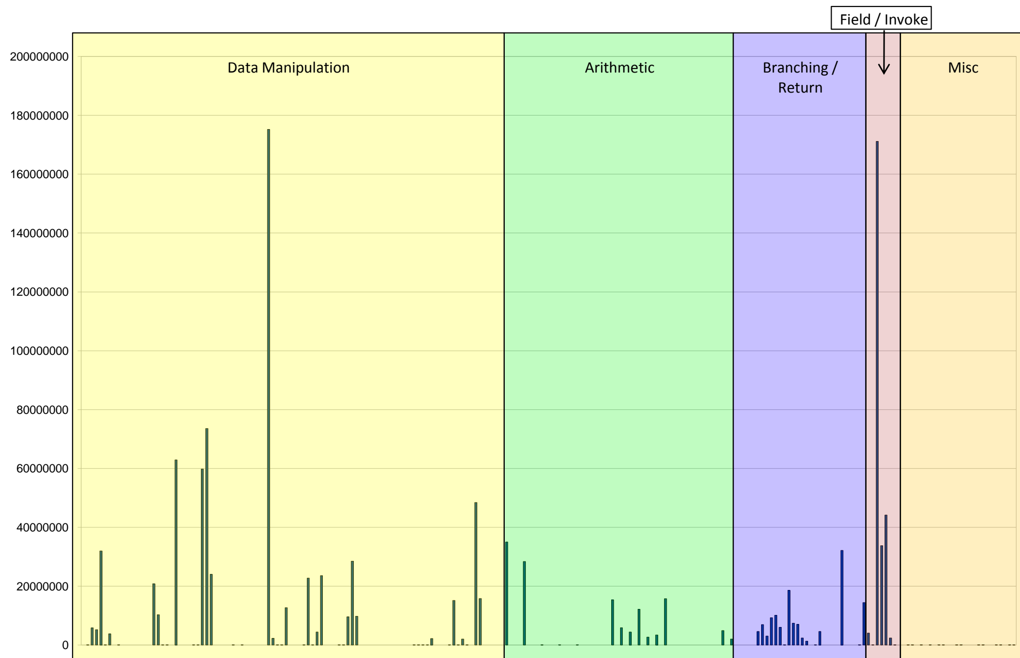


Figure 1.17: Compress Instruction usage in [53]

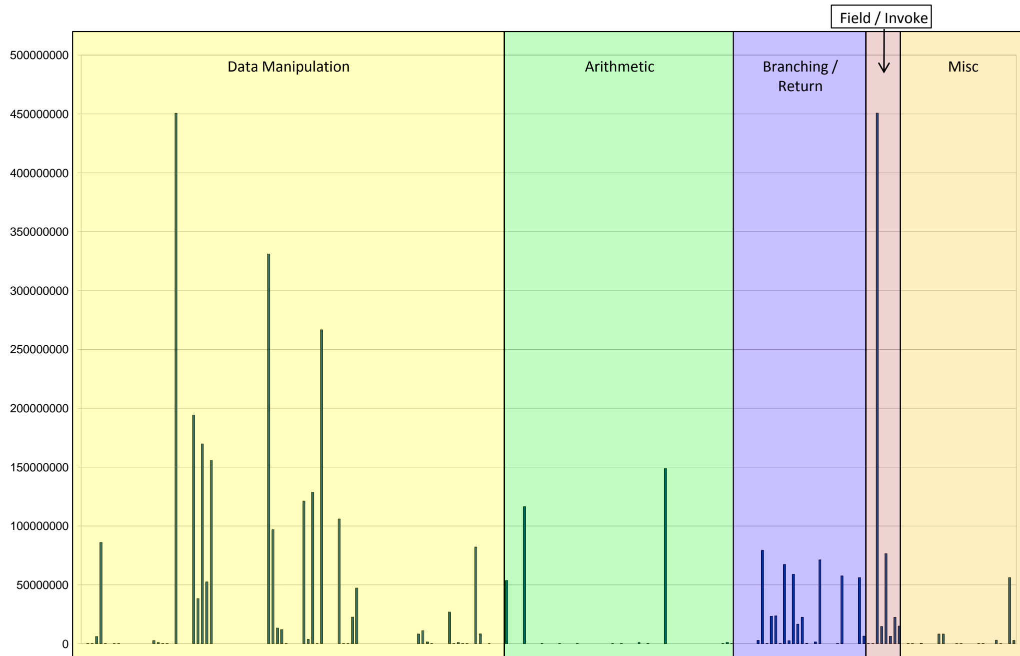


Figure 1.18: DB Instruction usage in [53]

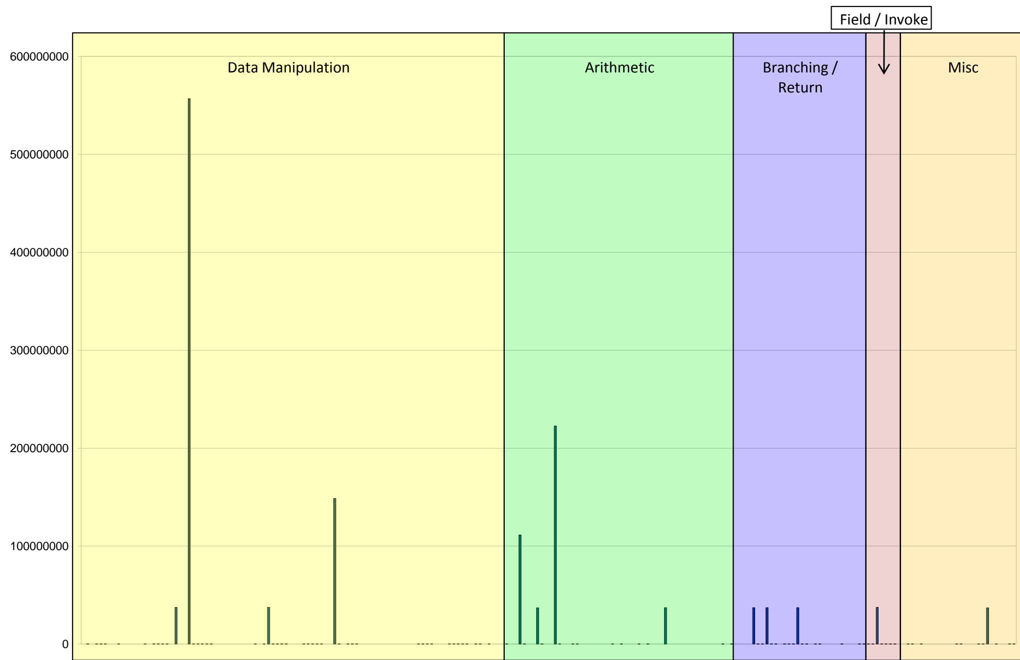


Figure 1.19: Mandelbrot Instruction usage in [53]

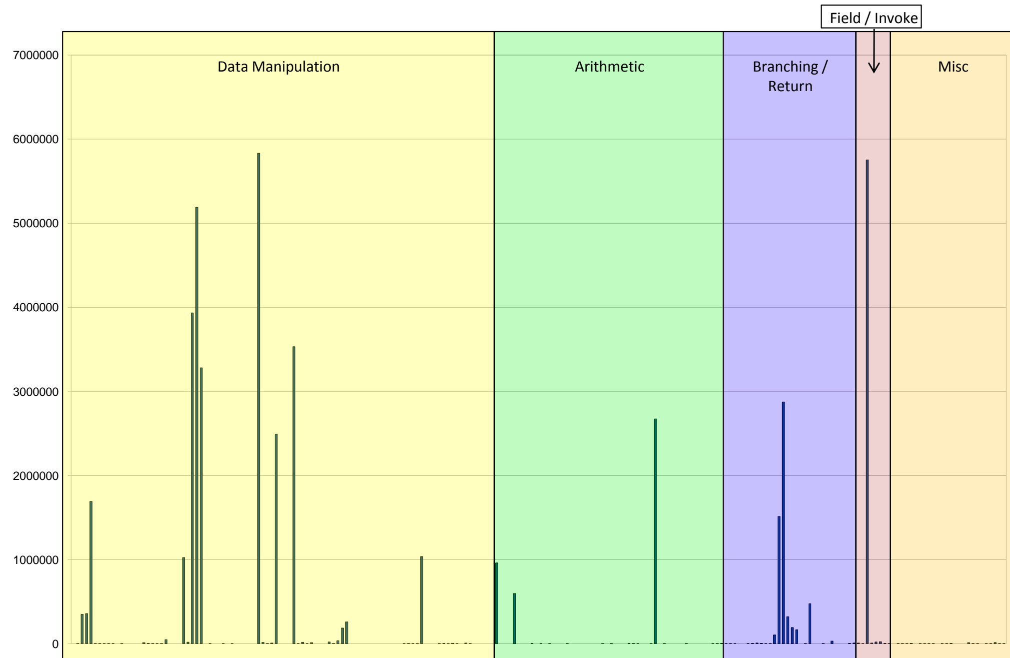


Figure 1.20: Queen Instruction usage in [53]

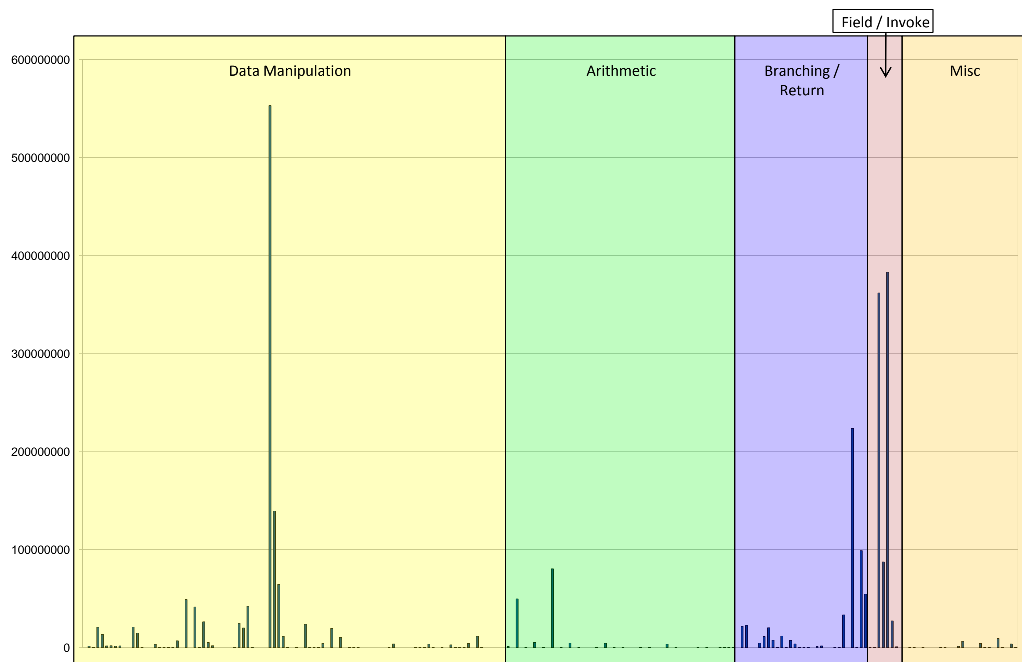


Figure 1.21: Raytrace Instruction usage in [53]

Chapter 2

Previous Work

2.1 Introduction

This chapter presents background and related work that is relevant to the new work that will be presented in later chapters. Firstly there is some general background on various approaches to implementing Java virtual machines and general types of optimisations applied to Java.

Following on from the general discussion, some specific Java implementations are examined, namely, Java 2 Micro Edition (J2ME), Java Card and Squawk. J2ME is designed to run on low-power or embedded devices and is used as the target for the tokenisation work presented later in this thesis. Following J2ME is a description of Java Card and how method calls are handled in the Java Card environment, in particular, the handling of interface method calls. The approach used in Java Card to performing method linking is used as a starting point for the tokenisation scheme presented in this thesis. Another virtual machine, Squawk, is also examined and how that virtual machine performs method calls. These sections provide background and discuss the issues with dispatching interface method calls in these systems, which is the core problem this thesis addresses.

In addition to the tokenising of class files to improve method calls, this thesis also presents the resultant compression that is achieved in class files. Therefore, this chapter also covers previous work on compression of class files for comparison and also to cover additional techniques that could be used to further improve compression.

2.2 Java Virtual Machine Implementations

There has been a lot of research on how to optimise Java and therefore how to best implement a virtual machine. These efforts can be categorised into four broad types [67]:

- Software interpreter - The virtual machine is implemented entirely in software. This is easy and quick in terms of implementation and debugging, however the software interpreter adds a large runtime overhead.
- Just-in-Time (JIT) compilation - In addition to, or instead of, a software interpreter, the Java bytecodes are compiled into native instructions. This removes some of the interpreter overhead, such as instruction decoding, allowing for faster execution than a software-only approach. However, the compiler execution time is part of the application's execution time, meaning only minimal compiler optimisations can be used. As well, the need to load the Java bytecode representation of a method, as well as a compiled version, leads to larger memory requirements than a software-only approach.
- Offline compilation/Ahead-of-Time (AOT) compilation - Similar to JIT, but done offline before the application is executed, meaning the cost of compilation is a one off cost, instead of being incurred at each application run. However, this can only be used in limited scenarios and care must be taken if some of the class files change, for example, if an application is updated. Can also incur a larger storage cost, if both the bytecode and compiled versions of method must be stored.
- Hardware VM - By implementing most or all of the virtual machine in hardware, the memory and power requirements can be minimised. This also avoids the overheads of the other approaches, providing the fastest solution, but with the highest development costs.

2.2.1 Interpreter

An interpreter is the simplest and cheapest way to implement a virtual machine, being written entirely in software. There are many examples of Java interpreters, such as the KVM and the J2SE runtime from Sun Microsystems (although in the case of the later, it also supports JIT, detailed in the next section). Typically, such an interpreter is run on top of some form of underlying operating system and consists of a loop that will fetch, decode and execute Java bytecodes. Also implemented in software will be features such as threading, garbage collection and a call stack, possibly with the use of libraries provided by the operating system (such as native threading support).

Due to the software nature of the virtual machine, performance is typically low. To execute a single Java instruction will by definition require at least several native instructions to fetch, decode and execute a bytecode. Complex bytecodes, such as invoking a new method, can take hundreds of native instructions to execute.

2.2.2 Just-in-Time Compilation

One of the overheads of an interpreter based virtual machine is the need to fetch and decode instructions in software, meaning by definition it will take at least several native instructions to execute a single Java instruction, even for something simple like an add instruction. By compiling the Java instructions into the processor's native instruction set, this overhead can be removed. If the compilation takes place before the application is executed, it is said to be ahead-of-time (covered in the next section), if instead, it happens at runtime just before the code is needed, it is said to be Just-in-Time (JIT).

Some compilers can also offer optimisation of the code during the compilation process, typically at the expense of longer compilation. However, it is important that the JIT compilation be as quick as possible since time spent compiling becomes part of the total execution time of the application. A JIT compiled method will typically be quicker to execute than interpreting the same method, and this time gain is realised each time the method is called, while the JIT compilation cost is only realised once, when the method is compiled. With enough repeated calls, the gains will eventually out-weigh the costs. However,

a short-lived application could incur the high cost of compiling a method, but very few calls to that method, resulting in an overall greater execution time than a pure interpreted approach. How many times a method needs to be called to reach this "break-even point" will depend on several factors, the speed of the JIT compiler, the speed-up of the compiler vs interpreted method and number of times the method will be used. The efficient implementation of JIT has therefore been the focus of a large amount of research effort [3, 16, 23, 50, 49, 51, 56, 54, 69, 88, 89, 95].

There are several different approaches to a JIT capable virtual machine, with [68] describing three broad categories:

Native-Only The virtual machine will only ever execute native code, therefore all Java code must be loaded and compiled before it can be executed. A call to a new method will result in a "pause" as the method is loaded and compiled, although future calls will happen quickly. For long-lived applications this allows the "investment" of compiling the code to give a large benefit over the application's life. However, a short-lived application is unlikely to see much benefit before it completes.

Interleaved The virtual machine will attempt to predict which methods will be used and interleave the compilation with the applications' execution. This avoids the lengthy pauses of the first solution, since the method should be compiled by the time the application calls it, which will make the application seem more responsive to users. Compared to the first type of JIT, interleaved compilation will disperse the compilation operations to prevent less noticeable pauses in the application, but overall it will spend as much time compiling as the first solution. A robust method is also needed to predict which methods will be used and when. One solution is to use a virtual machine that can execute native code or via an interpreter, allowing methods that are not compiled to be interpreted, which leads onto the final type of virtual machine.

Selective-JIT The virtual machine will interpret all methods initially, until some critical threshold is reached. The most commonly used methods will then be compiled, with future calls using the native versions, allowing the virtual machine to target its compilation efforts at the methods that should give the biggest improvement.

> The difficulty is in selecting an appropriate algorithm that is both quick to run and can select the methods that will give the biggest improvement. If implemented correctly, short-lived applications can avoid compilation overheads by simply being interpreted, while longer running applications will get compiled and therefore benefit from the improvement of compiled code.

The above deals with choosing when a method should be compiled to native code, however, the next challenge is to provide a compilation process that is both quick and provides highly optimised code. While analysis and optimisation of code are common place in regular compilers, it comes at the price of slower compilation. In the case of JIT, compilation time is part of the applications' overall running time, so any optimisations done to the code need to "pay for themselves" (i.e. the reduction in runtime because of the optimisation has to be greater than the time spent performing the optimisation).

Java bytecodes are designed for a stack based processor, while most hardware is register based. Presented in [98] is an approach to translate stack operations to a register form using "pseudo-registers", that is, an infinite number of arbitrary registers. Then the use of these pseudo-registers is analysed and they are assigned to real hardware registers. If not enough registers are available, code is added to store the contents of some registers to memory and load them back in again later. This is referred to as "spill code".

In all cases, JIT is providing a memory/speed trade-off, that is, by using more memory to store native versions of methods, greater execution speed can be obtained. How appropriate this trade-off is depends on the type of system. A desktop PC, with large amount of memory, can afford to store native code. A mobile device, with limited memory, may not have sufficient memory to store both the Java and native code.

2.2.3 Ahead-of-Time Compilation

Ahead-Of-Time (also called offline compilation) is similar to JIT, however compilation is done before execution of the program, with the native code stored to disk (or some form of permanent memory, such as flash). Since compilation is only done once, it does not affect the application's runtime. Instead there is a need for increased storage, since all methods

must be stored in their native form, even ones that are rarely or never used. Care must also be taken so that any changes to an application's classes is reflected in the compiled code.

The advantage, however, is that a given method only needs to be compiled once and can then be used many times. As well, since compilation is a once-off task, more time can be spent to highly-optimise the produced code.

2.2.4 Hardware Approaches

Previous implementations have focused on the use of software, however another common way to improve Java performance, especially in embedded devices, is with hardware support. Schoeberl [76] classifies Java hardware into either co-processors, which assist a general purpose processor, or a dedicated Java processor, which replaces the general purpose processor entirely. The following section covers co-processor designs, followed by a section on dedicated Java processor designs.

2.2.4.1 Co-processor

In a co-processor design, a device will contain a general purpose processor and alongside that, some form of Java co-processor. The operating system and "native" applications will run on the general purpose processor, with the Java co-processor bypassed (either powered off completely or powered on but inactive). When a Java application is executed, the co-processor will fetch Java bytecodes from memory and produce a stream of instructions which are fed to the general purpose processor. By maintaining the general purpose processor, existing operating systems and libraries can be utilised to implement the Java environment, reducing development overheads while providing a boost in speed.

Some examples of co-processor designs include:

Hard-Int Standing for "Hardware-Interpreter", since it will fetch bytecodes and translate them into instructions for a RISC processor, much the way a software interpreter or JIT compiler does [73, 74]. Short-lived client applications are identified as the target for this work, since long-lived applications will recoup the initial overhead of JIT compilation (and any optimisations performed to the code in this step). However, a

short-lived application will expend a large effort to compile code, only to have the code used very few times. By performing the JIT step in hardware, there is a smaller overhead to the translation.

DELFT-Java A processor running a custom instruction set which consists of a super-set of Java features (with features such as IO added) [42]. While it is intended for the processor to support many high-level languages (such as C or C++), the instruction set has also been tailored to suit the execution of Java bytecodes, making the translation from Java bytecodes to native instructions simple. The translation itself is performed in hardware.

Jiffy Consists of an FPGA alongside a general purpose processor [1]. The FPGA is used to implement a JIT compiler, translating Java bytecodes to the processors native instructions. By reducing the time taken to compile the Java bytecodes, the overall benefit for JIT is increased.

Hardware-Compiled Hariprakash *et al.* [44] describe a similar device to above, where a front-end pipeline will compile the bytecode instructions to a register-based instruction set, then cache those instructions for a backend processor to execute. They anticipate that such a design could greatly reduce the overhead of JIT compilation, from hundreds of cycles to only a few. However the authors also note that such a hardware compiler would require reasonable amount of hardware resources to make it feasible. For high-performance scenarios, such as servers, this could be acceptable, however extra hardware on mobile devices would lead to greater power requirements and therefore shorter battery life.

JA108 The JA108 is a commercial product from Nazomi Communications [65]. It sits between the main processor and memory, translating Java instructions into the processors native instructions. The JA108 has a bypass mode to allow other instructions (i.e. those the processor executes natively) to be read from memory without interference.

2.2.4.2 Java Processor

Unlike a co-processor, a Java processor is designed specifically to be the main processor in a system, and hence executes Java instructions directly as there is no general purpose processor. A Java system, be it an interpreter or a co-processors, would normally rely on an underlying operating system to provide some features, such as access to IO or threading support. Since Java instructions are the only instructions that can be executed, the operating system and support libraries must all be written in Java themselves.

Examples of Java processors include:

aJ-100 Made by aJile Systems, the aJ-100 is a direct execution processor for Java [4]. Supports direct execution of Java bytecodes, multithreading and support for having multiple instances of the virtual machine running at once.

IM1101 A Java based processor made by Imsys Technologies [46]. Provides support for native execution of Java as well as C, C++ or assembler code. Internally uses microcode to implement the various instruction sets.

JOP Presented in Schoeberl's thesis [76]. Provides a hardware Java processor aimed at real-time systems. Instructions are implemented via either direct execution or microcode, with a few special instructions added to support activities normally provided by the operating system. All IO control and hardware drivers are implemented in Java.

Jazelle ARM processors can include a feature known as Jazelle [9]. Previously ARM processors had supported two instruction sets, the regular ARM instruction set with 32-bit instructions and the Thumb instruction set with 16-bit instructions. The Thumb instruction set was a subset of the ARM instruction set, only including the commonly used instructions. The reduced instruction width of Thumb instructions results in smaller code size, ideal for embedded or otherwise constrained devices. The Jazelle extension allows the processor to enter a new mode where it will fetch and execute Java bytecodes directly. The implementation consists of a combination of direct execution, microcode and finally traps into software for the most complex instructions.

picoJava Developed by Sun Microsystems, there were two versions, the picoJava-I and picoJava-II cores. These can execute some bytecodes directly, while the more complex ones are implemented in either microcode or via traps to software to emulate the instruction.

2.3 Optimisations for Java

There has been a large body of work in both academic and commercial fields aimed at optimising various aspects of Java, some of which have been discussed in the previous section on virtual machine implementations. The techniques presented in this section focus on optimising the execution of Java bytecodes (thorough instruction level parallelism and instruction folding) and improving garbage collection.

2.3.1 Instruction Level Parallelism

In modern microprocessors, optimisations at the instruction level are important for performance. A modern processor needs to begin executing (issue) as many instructions as it can every clock cycle. For example, the latest Core architecture from Intel allows up to 5 instructions to be issued per clock cycle, and up to 4 to be retired [47]. Multiple instructions can therefore be "in-flight", that is, part way through execution. Since modern processors include multiple arithmetic logic units (ALUs) and floating point units (FPUs), multiple instructions can be run on each of these units in a given clock cycle. If one instruction stalls, such as while waiting for a memory read, then the other in-flight instructions can make use of FPU and ALU resources. Even if some instructions take multiple clock cycles to finish, as long as the CPU continues to issue more than 1 instruction per cycle, then its average throughput will be greater than 1 instruction per cycle, even if some instructions take many cycles to complete.

Java interpreters typically execute one bytecode at a time, so if one bytecode stalls, the whole interpreter stalls waiting for that bytecode to complete. Scott and Skadron [78] examine ways to parallelise Java bytecodes, in much the same way as modern processors

parallelise instructions. For the SPECjvm98 [93] suite they find a large amount of potential parallelism, up to an average 19.8 bytecodes that can be executed in parallel. This is only a measure of the potential for parallelism however, an implementation might not be able to actually take full advantage of it.

Radhakrishnan *et al.* [72] examine the picoJava-II core from Sun Microsystems [90] and attempt to optimise it. The picoJava-II core already includes instruction folding (covered in the next section), however Radhakrishnan *et al.* improve its performance, allowing the processor to decode Java bytecodes into the processor's microcode instructions at a faster rate. Since the processor is being supplied with instructions at a faster rate, these instructions are then parallelised to increase the overall throughput. The result is a 10% to 14% improvement in execution cycles. With some additional work, false dependencies between instructions can also be removed, allowing even more parallelism, with a reported additional 10% increase in performance.

Achutharaman *et al.* [2] take a higher level approach. Instead of looking to parallelise small sections of instructions, they analyse bytecodes for sequences that leave the operand stack in the same state at completion as it was in the beginning (i.e. a set of instructions that will load some values, operate on them, then store the results). This sequence of bytecodes, termed a "bytecode trace", can then be executed in parallel with other such traces. The authors propose a novel processor architecture to support execution of multiple traces on independent stacks. The result was a speedup of between 9% and 28%.

Ebcioglu *et al.* [32] also make use of parallelism. However, their proposal is closer to a JIT solution, where the JIT code can make use of parallelism rather than parallelism at the Java instruction level.

2.3.2 Instruction Folding

Since Java is designed as a stack machine, all arithmetic instructions operate on values on, and store their results to, the operand stack. This leads to many common code sequences. For example, the line: *"a = b + c;"*, would become: *iload_0*, *iload_1*, *iadd*, *istore_2*, in bytecode. These instructions correspond to:

1. Push local variable 0 onto the operand stack.
2. Push local variable 1 onto the operand stack.
3. Pop two values from the operand stack, add them and push the result back onto the operand stack.
4. Pop a value from the operand stack and store it into local variable 2.

Sun Microsystems introduced instruction folding in their picoJava-I [66] and picoJava-II [90] processors. For example, the picoJava-I core could detect the combination of "*iload_1, iadd*" above and combine the two instructions into the one cycle, avoiding the extra push and pop operations shown above. For common operations, such as addition, the number of cycles needed can be greatly reduced. With good instruction folding, contention for the stack can also be reduced, allowing greater parallelism of instructions [72].

Another approach is to combine instruction folding with a Java bytecode to native code converter [36, 33, 37]. During the translation, bytecodes are classified into one of several types, based on how they use the operand stack. These instructions can then be folded together before producing instructions for the processor to execute. Yiyu *et al.* [99] also present a similar instruction classification system where, after classifying instructions, they are folded together to produce instructions in their own language called *jHISC*. These *jHISC* instructions are then fed to a processor for execution.

The common theme among all these implementations is the use of a hardware component in the processor pipeline to implement the instruction folding. Bytecodes are fetched into an instruction cache, from where the folding unit will read them and attempt to perform folding.

2.3.3 Garbage Collection

Objects are created by a program during execution and Java allocates memory for these on the heap. It is then up to the garbage collector to determine when a program can no longer reach a given object, and to reclaim that memory. Therefore, the application programmer can allocate memory as needed (through the creation of objects) and does not

need to manage the release of that memory, leaving such work to the garbage collector. Since the garbage collector is a core part of the Java environment, it must operate without impairing the overall performance and responsiveness of the virtual machine and therefore, the application. There are many different approaches to garbage collection.

Chen *et al.* [22] show how the garbage collector can affect the power consumption of a device. Therefore the correct choice of garbage collector is important for devices that primarily run on batteries, such as mobile devices.

Stichnoth *et al.* [86] present an approach that allows garbage collection to occur at any instruction during execution and built on similar work that had been done by Diwan *et al.* [27]. Typically garbage collection will only occur when an object is created and needs to be allocated from the free space in the heap. If enough space can not be found, the garbage collector will attempt to free some, meaning the current thread will always be executing the *new* instruction. However, in a multi-threaded application, while the triggering thread will be at a *new* instruction, other threads could be at any type of instruction. By adding GC maps, to tell the garbage collector where to find object references, garbage collection could happen efficiently, regardless of the current state of the threads.

Garbage Collection in general (as it applies to much more than just the Java language) is a very large area of research in computer science. Ideally garbage collection should be invisible to the application/user. That is, execution should never have to be stalled for a long period of time to allow garbage collection to take place. However, this is often very hard to guarantee. Fuhrmann *et al.* [41] presents an incremental garbage collector aimed at real-time embedded Java applications, where strict timing constraints are required. Commonly a garbage collector will cause the entire system to pause for an unknown length of time as it frees unused memory. Instead, an incremental garbage collector will run in small increments of a known maximum length, therefore avoiding the need to stop applications for an unknown length of time.

2.4 J2ME

The Java 2 Micro Edition (J2ME) is designed for use on devices where it is not feasible to run the full Java 2 Standard Edition (J2SE). This includes devices from set-top boxes or other embedded appliances, down to hand held battery powered devices such as mobile phones or pagers. To accommodate such a wide variety of devices, J2ME is further divided into two main “configurations”. These are the Connected Device Configuration (CDC), for larger more powerful devices, and the Connected Limited Device Configuration (CLDC), for smaller devices. These configurations provide basic services and interfaces, but not a working system. Instead there are a series of “profiles” that can be added on top of these configurations. Each profile is aimed at a given type of device or situation, but there may be more than one profile implemented on a given device. Therefore the configuration only provides the common services that that system will require, leaving the specific functionality up to the profile.

One example of a profile is the Mobile Information Device Profile (MIDP). This profile resides on top of CLDC and is intended for use on mobile devices such as mobile phones or PDAs. The increasingly common Java games that can be found on modern mobile phones are in fact written using the MIDP and CLDC libraries. In this case, CLDC provides the basic library, while it is MIDP that provides the higher level graphics and interface abilities to interact with the user. The focus of this work is on these libraries and the associated virtual machine. The next section covers CLDC in more detail.

2.4.1 CLDC Specification

The standard J2SE specification is in two parts:

- The Java Language Specification [61] describes the language itself. This includes the grammar of Java source files, and the semantics of that grammar as well as threads and binary compatibility.

- The Java Virtual Machine Specification [57] describes the virtual machine itself, the format that binary class files must take and how to load and execute them. It also

describes the bytecode instruction set that is used in class files and how the virtual machine must behave.

CLDC[63] is a subset of the functionality provided by J2SE and is written as a list of differences to the J2SE specification.

2.4.1.1 Differences to the Java Language Specification

There are only two differences between the Java Language [61] and the CLDC specifications: the CLDC garbage collector does not call the finalize() method on objects and CLDC has less Error classes.

Firstly J2ME does not support finalisation of classes. In J2SE the class *Object* has a finalize() method that will be called by the garbage collector when the object can no longer be reached and is about to be removed. By overriding this method a programmer can create objects that will perform some action just prior to being removed by the garbage collector, such as releasing resources or locks held by the object. However, the finalize() method is allowed to pass a reference for the object back to other parts of the program, making it reachable again and this causes the garbage collector to be more complex. Once an object has been marked for garbage collection, the finalize() method will be called, then another check has to be made to ensure the object is still not reachable. J2ME removes this complexity and no application can expect that finalisation will be available.

The second difference between CLDC and the Java Language Specification is in terms of the EXCEPTION and ERROR classes that are included in the CLDC API. The difference between an exception and an error is that an application might be able to recover from an exception, but not from an error. An exception could be, for example, attempting to convert the string “abcd” into an integer. If the string was a user input, then the application can recover by printing a message to ask for new input. An example of an error would be when the application attempts to use a class, but the virtual machine can not find the class to load, and the application has no way of recovering. CLDC supports all the exceptions from J2SE, except *asynchronous exceptions*, however since the application can not recover from errors, only a small subset of the error classes are required in the CLDC specification. The

errors included are: *java.lang.NoClassDefFoundError* to indicate that a class file could not be found, *java.lang.OutOfMemoryError* to indicate the virtual machine has run out of memory and *java.lang.VirtualMachineError* to indicate a general error in the virtual machine. If a J2ME device implements more of the Error checks from J2SE, then it must either throw the super-class of the J2SE Error class that is in the CLDC, or it must halt in an implementation specific way.

2.4.1.2 Differences to the Java Virtual Machine Specification

The main difference between the CLDC and the J2SE virtual machine specifications is that CLDC does not allow for custom class loaders. In J2SE the application programmer may provide their own implementations of classloaders that are queried by the virtual machine when a class is needed. While J2SE has a very extensive security mechanism, CLDC does not. In particular, the class file verifier has been simplified, because of the need for CLDC to run on smaller, less powerful devices than J2SE, and therefore custom classloaders have been disallowed so that the system can remain secure. Instead, CLDC allows only one classloader, the bootstrap loader, which is built into the virtual machine and cannot be altered or overridden by an application. As well, the implementer of the virtual machine must ensure that this classloader's lookup order for classes cannot be altered, so that it will always load the real versions of the API and systems classes, not malicious copies.

CLDC does not support daemon threads (the virtual machine will terminate if daemon threads are the only threads running in the system) and thread groups. Since J2ME devices do not offer a large amount of processing power, an application is not expected to use a large number of threads and therefore not require thread groups to manage them. Management of threads is left entirely to the application programmer.

There is also considerable difference in how verification of class files is carried out. All class files that will execute on a CLDC device are required to contain a *StackMap* attribute associated with every method that also has a *Code* attribute. These attributes are defined in Appendix 1 of the CLDC Specification [63] and indicate what types should be on the stack at certain points during the method's execution. The stack map is generated off-device, and allows the on-device verifier to be much simpler in terms of space and time

complexity. The specification requires that class files contain StackMap attributes, however it is implementation specific as to whether the reduced verification algorithm using the StackMaps, or the full J2SE verification is performed.

2.4.2 The KVM

Sun provides a reference implementation of a virtual machine for CLDC, the Kilobyte Virtual Machine, or KVM. Its name is derived from the fact that it is designed to run with only a few tens or hundreds of kilobytes of memory, but to still be a complete implementation of a CLDC virtual machine [62]. The KVM has been implemented in C, with a view to porting to many different devices and systems.

One of the features of the KVM is a "ROMizer" tool. Normally devices such as a mobile phone will require that the CLDC and MIDP libraries always be present, so downloaded applets can be run. These libraries can therefore be placed in ROM, since they are not likely to change and it gives added security as an attacker cannot make modifications. The ROMizer tool that comes with the KVM is used to simulate this storing of libraries in ROM, which is usually referred to as romizing the library. In the case of the KVM the ROMizer functions by converting a given set of class files into a C source file that can then be compiled and linked into the rest of the KVM. However, it does not convert the bytecodes into native code, rather just stores the class information into the KVM binary, removing the need to dynamically load the romized classes. Burgaard *et al.* [15] performed an analysis of the KVM where the system classes were romized and again where they were not and compared the memory usage. Just loading the system classes used 90,228 cells of heap space, vs 5,468 cells when the classes were romized, however no analysis of why this is the case was given. It could be assumed the additional heap usage was simply the space needed to store the loaded classes. The romized version would have that memory as part of the KVM binary itself and therefore would not show up as heap usage, however no comparison was given for binary sizes. Even so, a class file contains linking information in the form of strings, which must be compared to other strings during runtime. Hash tables or similar data structures can improve the speed of these string matching operations, but at

the cost of additional memory, possibly accounting for some of the extra memory usage in the dynamically allocated case.

2.5 Java Card

Java Card is more restricted than J2ME, primarily due to the limited hardware on which Java Card is designed to operate. The class file format used by J2SE and J2ME is not practical for Java Cards, instead a new format was introduced called a CAP (Converted APplet) file. A CAP file uses fixed length tokens to identify each method, instead of the variable length strings used in class files (and described in Section 1.4.4).

2.5.1 The CAP File Format

The most obvious difference between class files and CAP files is that a CAP file contains an entire package. A package in Java can consist of any number of individual class files. The constant pool for a single class can make up more than 50% of that class's size, while many of the entries in a class will also be present in other classes in the same package. By combining all the class files into a single CAP file, hence having a single constant pool for the package, the redundancy present in class files can be greatly reduced. Similar techniques are used by compression schemes, such as JAZZ [14], to reduce redundancy.

The other significant difference is the allocation of tokens to methods instead of the UTF8 strings that are used in class files to identify methods and fields. More detail on this can be found in the next section.

2.5.2 Java Card's Virtual Method Tables

J2SE uses UTF8 strings to identify methods, necessitating a search for a matching string, starting in the target class and moving up through super-classes until a match is found. Instead, Java Card makes use of tokens to identify methods, which simplifies the method lookup process. The class hierarchy will form a tree structure with *Object* at the top. Figure 2.1 shows a simple inheritance hierarchy (in Java Card, *Object* only contains the *equals*

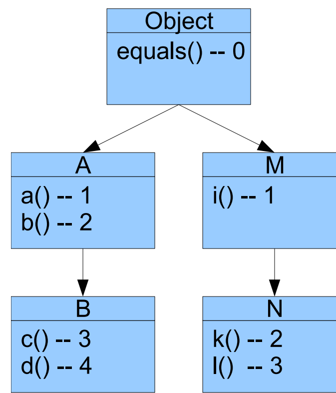


Figure 2.1: Inheritance Tree Including Tokens

method). Included in this figure are the token values each method would have. As can be seen, the tokens start at 0 in the class *Object* and increase independently down each branch of the tree.

An "inheritance list" is defined as a sequence of classes from a leaf class in the inheritance tree, up through its respective super-classes until *Object* (or vice-versa, starting at *Object* and traversing to each leaf class). Therefore, Figure 2.1 contains two inheritance lists, the first: *Object* –> *A* –> *B*, and the second: *Object* –> *M* –> *N*. It is safe to re-use the same token values, so long as the re-use is in a different inheritance list to any previous use, i.e. tokens 1, 2 and 3 are used for different methods in each of the inheritance lists above. Since tokens are assigned starting at *Object* and continuing down each branch, the re-use condition will always be met. Replacing the UTF8 strings with tokens trades an expensive string comparison for a cheap integer comparison, but still requires searching up through super-classes until the target method is found. To avoid this search, Java Card uses a virtual method table approach similar to C++ [40]. However, unlike C++, Java does not have a 'virtual' keyword, rather all methods in Java are virtual.

Each class's virtual method table will contain, at each offset in the table (i.e. method token), an index into the method component of the CAP file to where that method is defined. Figure 2.2 shows the previous example with virtual method tables (VMTs) added. The VMT for class *Object* consists of one entry, whose value is a pointer to the equals() method[1]. For a sub-class, the parent's table can be copied and any new entries added. For

[1]C++ style notation has been used for method pointers, consisting of: "<class>::<method>", to clearly

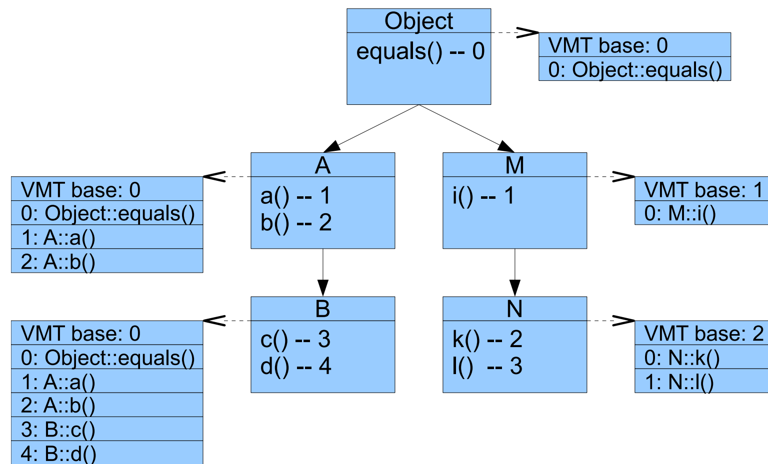


Figure 2.2: Virtual Method Tables

class *A* the two entries for the methods *a()* and *b()* have been added to the table, leaving the existing pointer to *Object::equals()*. The same process is applied again for class *B*. Copying the super-class's table each time leads to a large amount of duplicated data (as can be seen in the VMTs for classes *A* and *B*)". Class *M* shows a different style of VMT, where the duplicated data from class *Object* has been left off. The "base" value of the table (shown at the top of each table in the figure) indicates the starting value for that table, i.e. entry 0 in class *M*'s table should have occurred at index 1 in the VMT, therefore is has a base of 1.

When a VMT has a base value of zero, it will contain an entry for every method that the class accepts, allowing method dispatch to occur in constant time, for example, calling token 0 on an object of type *B* would require reading index 0 of class *B*'s VMT, giving the pointer to the method. When a VMT has a base value that is greater than zero, the required entry might not be found in the current class's table, i.e. calling token 1 on an object of type *N*, instead requiring searching in a super-class. At run time, the operation: *method_token - base*, can be performed to give the real index to use in that class's VMT, with a negative value indicating a search is required in the super-class's VMT. Therefore, calling token 1 on an object of type *N* (with a base value of 2), would evaluate '1 - 2', giving an answer of -1, indicating the super-class's VMT needs to be used. Class *M* has a base value of 1, *method_token - base* gives a value of 0 in this class, so the 0th entry in Class *M*'s VMT

denote which version of a method is being referred to. This is needed, for example, when a method is over-ridden, to denote which class's version of the method is being referred to.

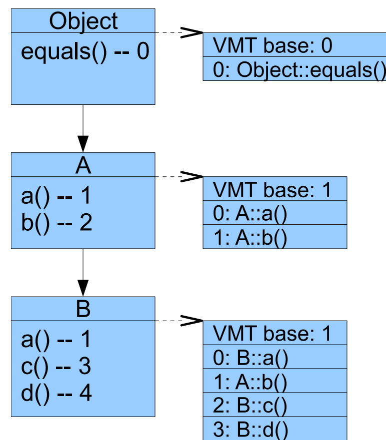


Figure 2.3: Inheritance With Over-Riding

contains a pointer to the target method.

This means that instead of searching each class's list of methods, the VM can perform a single subtraction to decide if the given method is in this class or not, reducing redundancy at the cost of a slightly slower (but still much better than linear search) method dispatch time. However, special consideration has to be taken when overriding methods. Consider Figure 2.3, where class *B* overrides the *a()* method. Class *B* cannot have a base of 3, since this would cause the VM to miss the over-ridden version of *a()*, instead it must now have a base of 1. Java Card requires that in this case the class *B* should copy the entry from class *A* for method *b()* into its own virtual method table. This will result in a small amount of redundancy being introduced back into the virtual method tables, but is better than each class having a complete copy.

2.5.3 Handling of Interfaces

In Section 2.5.2, VMTs were built only taking into account the class inheritance tree (i.e. ignoring interfaces). Since a given interface can be implemented by multiple classes, and a given class can also implement multiple interfaces, the tree in Figure 2.1 becomes a more general graph, as shown in Figure 2.4.

As discussed in Section 2.5.2, an inheritance list consists of a sequence of classes from *Object* to a leaf node of the inheritance tree (i.e. in Figure 2.4 there are two inheritance lists: *Object*->*A*->*B* and *Object*->*M*->*N*). When considering only inheritance, the method A::a()

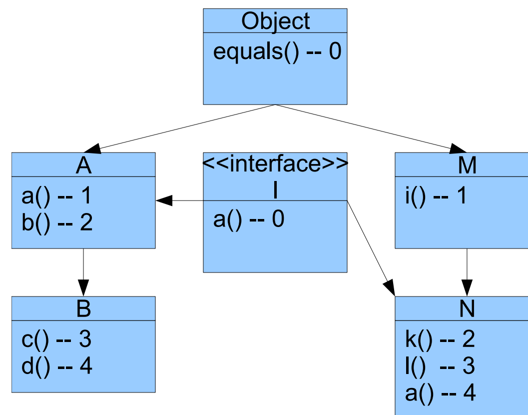


Figure 2.4: Inheritance With Interfaces

and N::a() are unrelated, leading to the methods receiving different tokens. However, these two methods are related when considering interfaces (due to the interface *I*). Extending the tokeniser to be aware of both interface and inheritance relationships, instead of just inheritance, is a non-trivial task. Instead, methods in an interface are assigned tokens which are unique only within that interface. A class which implements an interface must then provide a mapping from the tokens used within the interface to those used within that class, i.e. class *A* has a mapping from 0 to 1, while class *N* has a mapping from 0 to 4.

The addition of interfaces and the corresponding mapping from interface method tokens to the tokens used in a class require different semantics during execution depending on the declared type of the object involved (i.e. the type used in the source code). If the declared type of an object is a class, then the *invokevirtual* instruction will be used, if the declared type is an interface, then the *invokeinterface* instruction will be used (the different types of invoke instructions was introduced in Section 1.4.5). Each instruction will include: an object reference (via the operand stack), a token (via the constant pool), and a reference to the class or interface the method was declared in. An *invokevirtual* will involve the following steps:

1. Use the object reference to find its class type and hence the virtual method table for that class.

2. Use the token value in the method call as an index into the virtual method table, giving a pointer to the target method.

3. Execute the method.

However, for an *invokeinterface*, a mapping must also be done from the interface method token to the correct method token for this class, therefore requiring the following steps:

1. Use object reference to find its class type, then search from that class up to find the class that implements the target interface.

2. Use the mapping table in that class to map from the token in the method call to the correct token value for this inheritance branch.

3. Perform an invokevirtual for the new token value.

The need to first map from the interface method token to the correct method token means that *invokeinterface* is inherently slower than an *invokevirtual* in Java Card.

2.6 Squawk

Squawk is another virtual machine from Sun Microsystems and it is described in [79, 81, 82, 83]. While Squawk is designed to run J2ME programs, it differs from the KVM in that it is written in Java. The team credit the inspiration for implementing a Java virtual machine in Java to two other projects. Firstly, Squeak [85], a virtual machine for Smalltalk and Klien [84, 94], a now defunct project at Sun Microsystems, both of which were written in the language they implemented.

Implementing a virtual machine in the language it interprets presents an obvious recursive problem. In Squawk this is solved by having the very low levels of Squawk complied into native code. In particular, the interpreter, garbage collector and native code are written using a common subset of Java and C. In this way the interpreter can be run and debugged as a Java application first, then converted and run as a native program for speed. Due to a few syntactical differences, a small converter is needed to turn the code into valid C code, which can then be compiled as normal.

2.6.1 Memory Restrictions

The Squawk virtual machine is tailored to small devices, therefore it uses a split-vm approach similar to Java Card (which is covered in Section 2.5). Here the class files are pre-processed off the device, so that only light-weight verification/linking needs to be done by the device when installing them, hence moving some of the burden of loading/verifying/linking class files to the pre-processor machine. A converter application is used to read a set of class files and produce a "suite" file, containing everything needed to install those classes on a device. Design of the suite file is tailored to devices with limited memory, for example, linking of a given suite can be done in a single pass over the suite, as opposed to a standard virtual machine that would have to load each class independently, with several passes over each class.

Typical target systems will have a tripartite memory structure, consisting of ROM, non-volatile memory (NVM) and RAM. The virtual machine is aware of the different regions and will prevent memory references from a more permanent memory to a less permanent, for example, something stored in ROM or NVM cannot have a reference to something in RAM. However the inverse is allowed, i.e. RAM can have a reference to something NVM or ROM. Also to limit memory usage, the immutable class information (e.g. method bytecodes) will be stored in NVM or ROM, with only the mutable information (for example static variables) being stored in RAM.

2.6.2 Bytecode Modifications

Optimisations have been made to bytecodes including minimising the size of some operands, resolving references and simplifying verification. The reduction in operand sizes adds some extra restrictions (such as limiting the number of static fields in a class to 256), however it allows for a greater density of code. The same approach has been used in other architectures such as the ARM Thumb [8]. Squawk also defines that local variables have an implicit type, meaning that typed load and store instructions are not needed. The typed load/store instructions have been replaced with load/store instructions with implicit local variable numbers which do not need operands, further reducing code size.

An extra restriction is placed on bytecodes that could trigger garbage collection, requiring that only the operands for that bytecode may be on the operand stack when the instruction is executed. Along with typed local variables, this means that object references within a given method's frame will always be in the same place, making the job of finding object references that need updating very easy for the garbage collector.

2.6.3 Method/field references

The symbolic method and field references have to be resolved by the virtual machine before they can be used. J2SE virtual machines would perform these tasks during class loading, or would delay them till a given reference was used, then cache the result to avoid having to resolve the reference again. Suite files in Squawk are designed so that symbolic reference can, once resolved, be overwritten with the resolved address. However, since applications are stored in slow-to-write NVM, this will limit performance during initial execution or whenever previous unused code branches are encountered. These references are instead resolved as part of the conversion/installation process. In the case of a field reference, it will become an offset to where the field is within an object, and in the case of a method, it becomes an index into a virtual method table (covered in more detail in Section 1.3.2). The ordering of data in a suite file is such that symbolic linking information is presented first, then references can be resolved as they are received and written to NVM, saving the need to make a second pass to resolve references after installation.

Virtual method references are resolved as described above, however interface method references require special handling, as symbolic references to interface methods are not provided via the same virtual method table (similar to in Java Card, presented in Section 2.5.3). For each interface that a class implements, the class must have a table to map from the offset (or token value) used within the interface to the value (or rather, VMT offset) used in that class [79]. The performance of the *invokeinterface* instruction is therefore limited to always be slower than an *invokevirtual*, since *invokeinterface* must first consult the lookup table to map to the correct virtual method token for the given class, then perform the equivalent of an *invokevirtual* for that token value. Limiting the performance of interfaces

to be slower than the equivalent code without interfaces does not encourage their use by programmers.

This thesis presents a solution to this, such that the dispatch of an *invokeinterface* instruction would be equivalent to an *invokevirtual* instruction, allowing them to share a common implementation for both instructions.

2.7 Compression of Java Class Files

Each class or interface in Java will be compiled to an individual file. All but the most basic applications will require more than one class, requiring a way to distribute applications without needing to transmit many small files. The standard approach is through the use of a Jar (Java ARchive) file, which uses the Zip file format to store and compress many files into one file, while still maintaining the individual file names and directory structure within the Zip file. Research has been done on other packaging schemes for Java (and applications in general), with Ernst *et al.* [39] defining two main types of compression that can be applied to application binaries, “wire-formats” and “interpretable-formats”. A wire-format is one which aims to reduce size at all cost to ease distribution across networks but cannot be directly executed. An interpretable-format typically has less compression, but can be executed directly instead of needing to be unpacked or decompressed first. As well as the above two formats defined by Ernst *et al.*, a third technique is also considered, that of runtime compression, with the aim of compressing application data (i.e. objects on the heap) during runtime. The following sections look at the work that has been done in each of these three areas.

2.7.1 Wire-Formats

Wire-formats are designed to be as small as possible for transmission of data, while not requiring that it be directly executable by the Virtual Machine, which is useful when the data must be transmitted across slow and/or intermittent network connections. The side-effect is that the code must be extracted or converted in some fashion before it can be

efficiently executed. Below are examples of wire-formats.

2.7.1.1 Jar

Jar derives its name from Java ARchive, in that a Jar file is a collection or archive of several Java class files and is the standard format for Java [60]. Jar files can be created with or without compression and use standard zip compression. A Jar file is literally just a Zip file that contains all the class files and some optional additional information. Jar does not exploit any knowledge about the class file structure, instead just compressing the class files as if they were any other binary file. Jar will compress each file individually, this reduces compression efficiency, but allows a file to be decompressed independent of the rest. While technically a Jar file could be considered an interpretable format, since it contains just standard class files that can be executed, it has been included here since it can include compression and hence decompression before execution.

2.7.1.2 Clazz

Horspool *et al.* [45] presented the Clazz file format which uses knowledge of the class file format to reorder the contents and to encode different parts of the file with different techniques. Unlike Jar, which when uncompressed will give identical files to the original class files, Clazz does not, but they will be semantically equivalent. The primary space saving in Clazz is in the constant pool, as this is very often the largest part of a class file. Here Horspool *et al.* apply an ordering to the entries in the constant pool, as typically they are in a random order. This ordering allows them to firstly save space by not needing to use as many bits to encode the structure of the constant pool, since this can be partially implied by the ordering. Also the compression techniques used rely on finding repeating patterns in the data. By ordering the entries similar entries will be closer together and the compressor will more efficiently identify patterns, resulting in better compression.

The result of this approach was a class file that was significantly smaller than one that was simply ZIPed, such as those in a Jar file. A Clazz file cannot easily be interpreted by the Virtual Machine due to many offsets and other values in the file being coded as *delta*

values or as differences from previous values. Instead the Virtual Machine would need a customised class loaded that knew how to decompress the Clazz format into an in memory executable version.

2.7.1.3 JAZZ

The Jazz file format is presented by Bradley *et al.* [14] as a replacement to the Jar format. Jazz archives achieve better compression for a group of class files than the same files compressed using Clazz or stored in a Jar archive. Jazz uses similar approaches to Clazz, but operates on a collection of class files rather than on individual files, allowing it to remove a lot of the redundancy that is common between class files. For example, a class that defines a method will have that method's name and descriptor, as strings, stored in its constant pool. Any classes that wishes to call that method will also need the name and descriptor strings in their respective constant pools, leading to many files having the same strings in them. A Jar file will compress each file within the archive individually, meaning the compressor can not take advantage of these often repeated strings. A Jazz file introduces a single constant pool shared by all classes in the archive, and hence these common strings will only be stored once.

The Jar format has the advantage that individual class files can be loaded very easily, which is important as most Virtual Machines will only load a class file when it is needed. In Jazz, some of the information in each class file has been combined, for example with the global constant pool, making it harder to extract individual classes from the archive. Bradley *et al.* do not provide details in their paper, but suggest that it is possible to perform individual file loading from Jazz, provided the class loader has cached some of the decoded information. They also indicate that they would need to add some extra structures to Jazz to make this efficient, but give no indication of how this will affect compression. At present they suggest that the Jazz format must be decompressed upon receipt to individual class files or into a Jar format.

2.7.1.4 Pack

Pugh [71] proposes another format for storing a collection of class files, referred to as the "packed" or "pack" format in his paper, consisting of an encoding to represent a set of class files, which is then compressed with gzip to give the final pack archive. The design principles are similar to those of the Jazz format above, and it is also designed as a replacement for Jar files. However, Pack differs in its implementation to Jazz, with Pugh reporting significantly better compression. In a Pack file, classes will share common information (i.e. with only one constant pool for all classes in the archive), and information is arranged to allow gzip to perform well when compressing the file.

The Pack format requires that the class files be decompressed from the archive in the same order in which they were compressed, with output written to a Jar file or into individual class files, which can then be loaded by a VM as needed. A typical VM will use lazy class loading, that is, it will only load a class file once it reaches a point that it requires the class to be loaded (such as making an object of the class). However, Pugh also suggests the idea of eager class loading, where the VM will load all of the classes from the archive, allowing the archive to be read, and all classes loaded, with a single pass. In the case of an application where it is likely that the Virtual Machine will need all the classes in the archive this could improve class loading. However, cases such as a library, where only some of the classes are used, will result in loading of unnecessary classes. Pugh has acknowledged these constraints, as his intent was to create a wire-format where bandwidth was the most precious resource, not compression/decompression time or memory constraints.

An implementation of the pack algorithm has been incorporated into Java distributions in for form of the 'pack200' and 'unpack200' binaries (or 'pack200.exe' and 'unpack200.exe' on Windows) and the 'javax.pack.*' package, which is specified in Java Specification Request (JSR) 200 [92].

2.7.1.5 CAR

Antonioli presents the CAR (Class ARchive) format [7] as another replacement for Jar files. While CAR does not perform better than the packed format from Pugh, based on

size, Antonioli suggests that space saving is not the only metric, but also the speed and binary size of the decompressor.

Antonioli suggests the binary size of the decompressor is important as it would need to be transmitted with the compressed application. Therefore, the space saving that the compressor achieves must be better than the size of the decompressor itself, otherwise there is no reduction in transmission size. If the decompressor is already present, however, (or provided as an additional application and only downloaded once), then the size of the decompressor, and hence its transmission size, is amortised over all applications downloaded to the device.

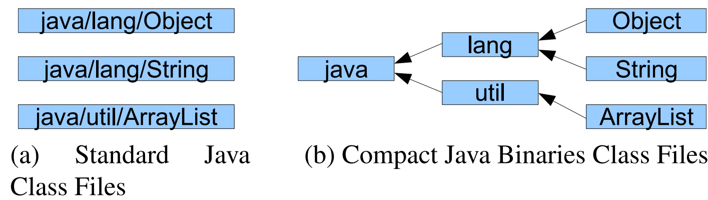
While compression is a once-off operation, decompression will need to be run before every execution if the device only stores the CAR version of a package. This could be true for a device that needs to limit the memory used to store applications, therefore storing them in the compressed form. However, the intention of a wire-format is to reduce the transmission size of the application, not the storage size.

2.7.1.6 Generic Adaptive Syntax-Directed Compression

Stork *et al.* [87] present a more generic approach to compression. Specifically they look at parsing the source code for an application to develop an abstract syntax tree (AST). This represents all the required information from the source code (minus comments, formatting and some variable names), and is commonly the first step of compilation. They then compress the AST form of the program. At the delivery end, the code must be decompressed, then compiled. This approach could apply to any language, and has the benefit of being more platform neutral than compiled binaries (to such an extent that the underlying language is portable).

2.7.2 Interpretable formats

Interpretable formats are designed so that they can be executed directly without needing to be extracted first. These formats will be larger than a wire-format as they can not make use of the more dramatic transformations of a wire-format.



(a) Standard Java Class Files

(b) Compact Java Binaries Class Files

Figure 2.5: Representation of Class Names in Class Files

2.7.2.1 Compact Java Binaries for Embedded Systems

Rayside *et al.* [75] propose reducing the redundancy in the constant pool through reuse of strings, specifically in class names. All class names in the constant pool are fully qualified (they consist of the full package name followed by the class name), for example, the *Object* and *System* classes both exist within the *java.lang* package, resulting in the strings 'java/lang/Object' and 'java/lang/System' to represent them[2]. The package structure forms an implicit tree structure, for example, in the J2SE API, the top level packages are 'java' and 'javax', each with numerous sub-packages (such as 'swing', 'lang', 'util', 'math', etc.) which, in turn, can have further sub-packages. By using this property of class names, each string is broken up into individual package/class name strings, with each string having a link to the next component in the full name. For example, Figure 2.5a shows the standard Java class names for three classes, while Figure 2.5b shows the format proposed in [75]. Each class name gives just the name for the class, with a pointer to the package the class is in, each package in turn has a pointer to its super-package. The top-level package ('java' in this example) only needs to be stored once, instead of the three duplicate copies in Figure 2.5a. The more classes within the same package, the greater the opportunity for space saving.

The type of each constant pool entry is implied by ordering the contents by type and recording the number of each type, saving the one byte tag value for every entry. Further, this ordering allows some of the references between entries in the constant pool to become implicit. Rayside, *et al.* conclude that these modifications to the constant pool would only affect the linking stage of the class file when it was loaded by a virtual machine and there

[2]For historical reasons Java source code uses a full stop character to separate the package names, i.e. 'java.lang.System', while internally (within binary class files) the slash character is used, i.e. 'java/lang/System'.

should be little or no impact on the runtime performance.

The final change presented in [75] is to the Code attribute of methods, which stores the bytecodes that implement the method. Each bytecode will consist of 1 byte to identify the opcode (i.e. which instruction to run), followed by zero or more bytes of operands for the instruction. Importantly, most operands fall into one of two types, indexes into the constant pool (e.g. for method references or constant values) and offsets within the bytecode array (e.g. for branching). Instead of encoding the bytecodes as a single array containing opcodes and operands, two arrays are used, one for the opcodes and the other for the operands. The opcodes array is compressed using a Huffman algorithm. However, some pairs of opcodes can often appear together, so pairs of opcodes are encoded, rather than individual opcodes. For the operands' array, most constant pool indexes can be reduced from 2 bytes to 1 byte, since most classes do not have more than 256 entries in their constant pool. Offsets within the bytecode array, such as branching instructions, must encode how many bytes forward or backward to jump within the array, which previously included both the opcode and operand values. Due to the split nature of these new arrays, offsets only need to encode the number of instructions to jump (i.e. the number of bytes in the opcode array, since each instruction is one byte), allowing many of the offsets to be reduced from 2 bytes to 1 byte. However, there is no analysis presented in [75] of how these bytecode transformations would effect the runtime performance of the application.

2.7.2.2 Java Bytecode Compression for Low-End Embedded Systems

Clausen, *et al.* [24] examine an interpretable compression format tailored specifically to Java Card. Java Card makes use of CAP files (described in Section 2.5.1), instead of class files. In a standard class file it is the constant pool which accounts for the largest amount of space, however in a CAP file it is the bytecodes which are the biggest component. Clausen, *et al.* estimated that bytecodes could account for approximately 75% of the memory footprint of an application [24].

Clausen, *et al.* has exploited the fact that opcodes can appear in common sets of 2 or 3, which are replaced with a single 'macro opcode'. Each bytecode is encoded as a single byte, therefore giving 256 possible values, but not all of these are used by the standard Java

Card Virtual Machine (JCVM), with the remaining values assigned to be macros. A special table is used to record the real instructions that comprise a macro and a modified JCVM can then consult this table whenever it encounters one of these macros. The result was that on average a package was reduced to 85% of its original size, with a speed penalty of between 2% and 30% for looking up the macros [24].

2.7.2.3 Practical Java Card bytecode compression

Extending the idea of macro opcodes, Bizzotto & Grimaud [13] look at the practicalities of using them. The compression algorithm is too large and resource intensive to run on a Java Card. Therefore it needs to run off-card, but the resulting CAP file must be verifiable once it is sent to a card or else Java Card's security model will be broken. The solution proposed by Bizzotto & Grimaud [13] adds a *compress component* to an otherwise standard CAP file. Since the CAP file is still in the standard format, with just the addition of an extra component, the normal Java Card verifier can test the standard parts of the CAP file for safety, then an addition verifier determines the safety of the compress component. Finally, the card can use the information in the compress component to efficiently generate the macro/compressed version of the CAP file.

Bizzotto & Grimaud also examine different strategies for allocating macros. There are 69 opcode values not used by the Java Card specification. Of the many available opcode pairs the best ones to use will depend on the applications being compressed. These allocations may be either local to a specific package or global across all devices. The local approach would allow an application to carry its own custom macro table, which would allow for the allocation of macros optimised for that application. The global approach would mean that the macros may not be the optimal set for a given application, however the macro table can be directly coded into the JCVM, making macro lookups quicker. To compensate, several different global tables could be used, with the application selecting the best.

2.7.2.4 Split-Stream Dictionary Program Compression

While not specifically targeting Java, Lucco also examines the compression of program instructions [58]. Program code is compressed with a granularity of basic blocks, allowing a basic block of the code to be decompressed on its own. Lucco suggests this can be coupled with a JIT compiler (see Section 2.2.2 for more details about JIT), with each block being decompressed, then JIT compiled before decompressing the next, and so on, hence reducing memory requirements.

Using x86 code on a desktop PC, Split-Stream Dictionary compression resulted in a 47% compression ratio and a 6.6% overhead on runtime. Since code is decompressed and JIT compiled before execution, a cache of already compiled code is maintained in memory. As pressure grows for RAM, less of the JIT compiled code can be kept and if code is needed again, it will be decompressed/compiled again. With a buffer to hold JIT code only 0.4 times the size of the original x86 code, Lucco reports a 99.3% hit rate for function calls already being present in the buffer. As the memory was constrained further, the hit rate showed a gradual decline and a corresponding increase in runtime overhead, allowing a graceful degradation in performance as the demand for RAM increases.

2.7.3 Runtime compression

While interpretable formats deal with compressing the application code, data such as objects on the heap remain unaffected. Runtime compression deals with compressing this runtime data, to allow an application to run in a smaller memory footprint.

2.7.3.1 Heap Compression

When trying to run larger applications on smaller devices, memory limitations can become a primary concern. While compression of code and other class data can help limit the amount of memory needed to store the application, there still remains the issue of runtime data size. This can also become an issue on devices that use non-volatile memory, such as EEPROM, for storing class data, and separate RAM for heap data. Chen, *et al.* [20] proposed a garbage collector that allows objects on the heap to be compressed when memory

demand is high. This would allow an application to run with less RAM than its nominal memory footprint or alternatively to have more applications running concurrently.

The price is a slight performance penalty during runtime to allow for compression and decompression. Although in some rare applications, the additional memory that is freed results in less garbage collection during the applications life and therefore an overall increase in performance.

2.7.3.2 Energy savings through compression

For battery operated systems, energy consumption can become a critical aspect of the system. Memory devices will leak a small amount of current whenever they are powered on, even if they are not performing operations. This led Chen at al. [21] to investigate the use of compression for read-only memory. Data such as the VM code, and API class data can be compressed offline, then stored in read-only memory on device. Since energy leakage is a function of memory size, being able to use smaller memories for the read-only data results in less power consumption. A "scratch-pad memory" (SPM) resides between the processor and main memory, in much the same way as a cache. When an address is requested, if the relevant block is in the SPM, it is read from there, else the block is brought in from main memory. In the case of the read-only memory, a hardware decompressor will decompress the data on the fly as the block is placed into the SPM.

Additional energy is required to run the hardware decompressor and SPM, however energy use is reduced due to the smaller read-only memory. Overall this resulted in an energy saving in a simulated environment.

2.7.4 Summary

The work presented later in this thesis covers tokenisation of class files, which will also results in compression, and can be best categorised as an interpretable format. While the tokenised format presented in this thesis does not compete directly with existing wire-formats or runtime compression, they are presented as possible directions that could be used to improve the compression gained from tokenisation. Table 2.1 provides a summary

Table 2.1: Comparison of Compression Schemes

Scheme	% vs Class	% vs Jar (comp)
Jar [60]	53-73	100
Clazz [45]	35-38	–
Jazz [14]	22-29	44-60
Pack [71]	10-18	20-34
Car [7]	15-40	32-73
Generic Adaptive Syntax-Directed [87]	–	13-39
Compact Java Binaries [75]	75	50-55
Java Bytecode Compression [24]	80[a]	–
Practical Java Card bytecode [13]	68[a]	–
Split-steam dictionary [58]	37-58[b]	–

[a] This number only takes into account the bytecodes, not the additional parts of a class file.
[b] These results were for compressing x86 instructions only.

of results from the wire-format and interpretable format schemes that were presented in the previous sections. Numbers indicate the percentage of the original size, for either uncompressed class files or compressed JAR files, and have all been taken from the respective author's original work.

2.8 Summary

This chapter has presented background information covering the general approaches to implementing virtual machines, some specifics about the J2ME, Java Card and Squawk implementations, then finally some discussion of the various approaches for compressing Java class files.

The work presented in this thesis focuses on J2ME and the production of method lookup tables. The approach is based on that used in Java Card, but without the need for additional lookup tables in the presence of interfaces. In addition to improving method dispatch, the lookup tables allow a large volume of string information (previously used for linking) to be removed from the class files, resulting in much smaller files.

Chapter 3

Global Tokenisation of Class Files

3.1 Introduction

As stated earlier, one of the goals of this thesis is to produce tokenised (essentially pre-linked) class files using lookup tables (or virtual method/function tables) for each class. References to a method then become an index (or token) into this table. Previous approaches have over-heads in the presence of multiple-inheritance, either requiring duplicate tables (C++) or additional lookup tables (Java Card). Instead, this work's aim is to tokenise the methods in such a way that a single dispatch table can be used, irrespective of the declared type of the object (be it a class or an interface). Stating this goal another way, *invokevirtual* and *invokeinterface* will share the same implementation, with dispatch done through a single *virtual method table* per class. Starting with the dispatch tables used by Java Card for the *invokevirtual* instruction, some extra constraints must be added to allow the *invokeinterface* instruction to use the same dispatch process as *invokevirtual*:

1. Interface methods will be tokenised the same as any other method in the system (instead of interface-only tokens).

2. All methods that implement an interface method, irrespective of class, must use the same token as the interface method.

A direct result of tokenising the class files is that much of the symbolic information present (in the form of strings) will not be needed at runtime, resulting in less memory requirements

for execution, less overheads from not having to link class files and additionally less storage space by removing the strings.

3.2 Comparison to Java Card

Java Card can be considered to have two "name spaces" when it comes to token values, the first for methods declared in classes, the second for methods declared in interfaces. Classes will always (as the result of always having only one super-class) form a tree structure, with the *java.lang.Object* class as the root, and tokenisation of these methods starts from the root and proceeds down the tree. Each class in the tree will be allocated increasing token values, starting from the largest token used in the super-class. For a given class, C, which contains n methods, if the largest token used in the super-class of C is m, then tokens would be allocated to a class by:

1. For a class with n methods in it, the token values $m+1$ to $m+n$ would be allocated to each of the methods in turn (for $n=0$, no tokens are allocated).

2. For each sub-class of this class, repeat this process.

The tokenisation process is started from the *java.lang.Object* class with $m=0$. The tree nature of class inheritance means that the used tokens are always in a contiguous block, with every slot in the table corresponding to a method that can be called on an object of that class[1]. Having full dispatch tables is not a requirement for Java Card, but results from the tokenisation algorithm. Ideally, the tokenisation algorithm presented in this thesis should have full dispatch tables, as this will reduce the space needed to store the tables. Additionally, the incremental assignment of tokens via the inheritance tree in Java Card means that methods within a given class will have token values greater than those used in any super-classes (with the exception of over-ridden methods, which use the same token as the previous version of the method). A class can then store only this last, changed, part of

[1]This is with an O(1) dispatch time, where every class has an entry in its virtual method table for every method that could be called on it. Although as mentioned in Section 2.5.2, the tables can be smaller at the cost of an O(n) dispatch time, based on the number of super-classes that need to be searched to find the correct table.

the dispatch table (this is optional in Java Card and is described in detail in Section 2.5.2). If the tokenisation algorithm in this thesis can maintain increasing token values the same as Java Card, then it will be possible to use the same style of dispatch table, further reducing the space needed to store these tables.

The methods in an interface are allocated tokens independent of the tokens allocated to methods in a class. When a class implements an interface I, it must contain a table which maps from the method token for each method in I to the token value used for the implementing method (for more details on how Java Card handles interfaces, see Section 2.5.3). The tokenisation scheme presented in this thesis will assign the same tokens to interface methods as to methods in classes. By requiring methods in interfaces and classes to have the same token "name-space", the need for these extra interface tables will be removed. Instead an *invokeinterface* instruction would be dispatched using the same dispatch table and process as would an *invokevirtual* instruction for the same token.

3.3 Simple Tokenisation

This chapter assumes that a complete system is being tokenised, i.e. all the class files for the API, optional or third party libraries and applications. If new classes are to be added to the system, then the existing tokenisation is discarded and a new tokenisation performed on this new set of class files. The following chapter will discuss the extension of this technique to allow tokenisation to be incremental, i.e. to allow a new application to be added to an existing system, without changing any of the existing tokenised classes.

The simplest approach to tokenisation would be for each unique method selector to be allocated a unique token, leading to the number of tokens in the system being dependent on the number of unique method selectors. While interface methods will have the same token as the methods that implement them, (since they will have the same selector), the allocation of token values will have little relation to the where the methods are declared. In Java Card, tokens are allocated while moving down the branches of class inheritance, meaning that any given class will always use tokens within a contiguous range from 0 to the number of methods in that class and all super classes. In the proposed scheme, however,

tokens are allocated to method selectors, so for any given class, it will be unlikely that the methods in that class have token values in a contiguous range, leading to large dispatch tables with very few of the entries being used.

Previous approaches to tokenisation, such as Dixon *et al.* [28], have reduced the size of the dispatch tables through the use of selector colouring. While two selectors that appear in the same class cannot share the same token, if two selectors will never need to be in the same dispatch table, then they can be assigned the same token. Selector colouring uses graph theory and the well understood problem of graph colouring to produce a near optimal allocation of tokens such that the smallest number of tokens are allocated. However, the range of tokens likely to appear in any given class is still not restricted in this approach, rather the total number of tokens is reduced. While this will reduce the size of the dispatch tables, they are still likely to contain a large percentage of unused entries.

Another approach is to maintain the sparse lookup tables, but encode them differently to reduce their size [31]. Each row (or column) of the lookup table is taken as a one dimensional array, then overlapped so that unused entries in one row match up with used entries in another, resulting in a master array with only one dimension. Each row of the original two-dimensional array is therefore stored at a different offset in the master array. While reducing the number of unused entries, there is an added overhead of determining if a value actually belongs to the class in question, or if the entry was empty and another class's entry has been stored over the top. Previous work on dispatch tables was discussed in detail in Section 1.3.

In previous approaches, there has been a one-to-one mapping of method selectors to tokens. Dixon *et al.* presented a many-to-one approach, with multiple method selectors mapping to one token, allowing for more flexibility in token allocation and hence better space saving. The next step is to allow a many-to-many relationship, i.e. allowing different methods with the same selector to be assigned different tokens.

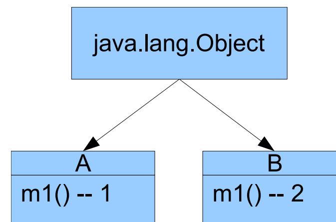


Figure 3.1: The same selector with a different token

3.3.1 Assigning Different Tokens to the Same Selector

There are situations where a given selector (a method's name and descriptor in Java) may be assigned different token values in different classes, without compromising the correctness of the tokenisation. Figure 3.1 shows a case where class *A* and *B* both inherit from *Object*. Both classes contain a method, *m1()* in this example, that is not contained in the *Object* class. Therefore, even though both methods have the same name and descriptor, it is not possible for the m1() method from class B to be called on an object of type A or vice versa.

By allowing more flexibility in which token values can be assigned to methods, a given class can use token values within a smaller range (ideally with a contiguous sequence of tokens, as in Java Card), allowing dispatch tables to be as small as possible. Tokenisation therefore becomes a problem of determining when two methods with the same selector must have the same token and when they can be different. After determining which methods need the same token, they should be allocated with the aim of minimising the average size of the dispatch tables (i.e. by keeping the token values as contiguous as possible).

3.3.2 Non-continuity of Token values

The ideal solution for tokenisation would be for any given class, if it needs n entries in its dispatch table, to use the next n token values, starting from the largest token used in the super-class. Because Java Card ignores interfaces for assigning tokens, it can guarantee this criteria is always met. However, with the added complexity of interfaces, it becomes impossible to guarantee this condition.

Figure 3.2 shows a case where a hole in one of the dispatch tables will be required. In this case the method m1() will be allocated some token, X, while the method m2() will

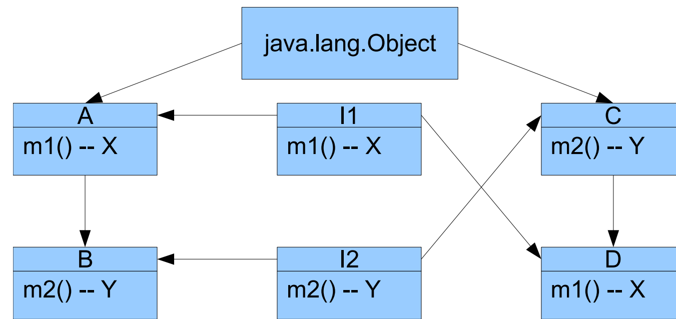


Figure 3.2: Non-continuity of tokens

be allocated the token Y. Since X and Y must appear in the same dispatch tables (i.e. in classes B and D), then X must not equal Y, and therefore, either $X < Y$ or $X > Y$. If $X < Y$ then class C will be using token Y, but not the smaller value X, although class D's dispatch table will fill in this gap. If $X > Y$, then class A will use token X, even though the smaller value Y is not in use. Therefore, it can never be guaranteed that all dispatch tables will be completely full for every possible set of class files.

These holes, when they exist, will not affect the runtime of the system. Assuming the classes have all been through the Java verification process, then method calls can only be to methods that exist. Thus after tokenisation, all method calls will be to tokens that were allocated within the target class and will therefore be to entries in the dispatch table that are non-null. Runtime protections can still be added so that if a token does reference one of these holes, or null entries, then a runtime error will be produced.

The ideal solution will not always be reachable (except for cases with fairly simple use of the Java interface mechanism), however, the goal is still to minimise the extent to which null entries are present in the dispatch tables.

3.3.3 Binary Compatibility

There are certain situations where changing a given Java class and recompiling it without recompiling other classes can lead to an incompatibility. For example, if a method that was once public is changed to private and only the single class is recompiled, then other classes could still contain references to the once public method. These (and other) problems will be detected at runtime when the target method is found to be private and therefore not

visible. The fact that the tokenisation is recreated whenever any classes are changed implies that the tokenised files will remain compatible, assuming there are no changes that would normally break binary-compatibility. So if the standard class files are compatible, then the re-tokenisation will update the references to be correct.

3.3.4 Overview of Tokenisation Process

A tokenisation process has been described where a many-to-many mapping will be allowed between method selectors and tokens. The process to implement this tokenisation will consists of several steps:

1. Open and parse each class file to build details of each class, such as: a complete inheritance tree, interface relationships (super/sub-interfaces and which classes implement which interfaces) and the methods in each class.
2. Produce "method groups".
3. Assign tokens.
4. Each class is now re-read, one at a time, and using the information in memory, all method/class references are updated to use the token information. Unneeded information (such as symbolic string data) is removed and the tokenised file written out.
5. The meta-data used during the tokenisation is written out to a "descriptor file".

While the initial pass reads every class file, not all the contents of the class file are loaded, to reduce memory requirements. Later after tokenisation is complete, each class is completely loaded into memory, updated, then saved to disk. In this way only one class file needs to be in memory at any given time. Step 2 and 3 are where the main part of the tokenisation work occurs. "Method groups" are used to denote a group of methods, which require the same token, allowing the tokeniser to distinguish between methods that have the same selector, but do not require the same token.

To illustrate the process, consider the simplified source code presented in Algorithm 2 to be the entirety of a Java system requiring tokenisation. First the source files would be

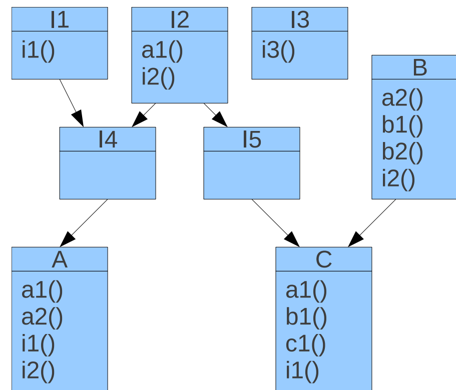


Figure 3.3: Example system classes

compiled using the standard Java compiler to produce binary class files. Next, these files would be read by the tokeniser to produce an in-memory representation of the interface and inheritance relationships as shown in Figure 3.3. The following sections will use this example to give more detail on the creation of method groups and how tokens are allocated.

3.4 Method Groups

To implement a many-to-many tokenisation scheme, it is necessary to determine which methods will require the same token and which will not. A "method definition" for this purpose is defined to be a method in a class or interface. It will consist of the class or interface in which it is defined and the method's name and descriptor. There may or may not be an implementation associated with the method (i.e. abstract methods in classes and methods declared in interfaces will not have an implementation). Through loading and examining the complete set of class files for a given system, the complete set of method definitions can be determined.

Next, a "method group" is defined as a group of method definitions that require the same token. Two methods will require the same token when either:

1. One of the methods over-rides the other, or

2. One of the methods is defined in an interface, and the other implements it.

In the simplest case, a method group would consist of exactly one method. Method groups

Algorithm 2 Source for example system

```
interface I1 {
    i1(){}
}
interface I2 {
    a1()
    i2()
}
interface I3 {
    i3()
}
interface I4 extends I2, I1 {
}
interface I5 extends I2 {
}
class A implements I4 {
    a1()
    a2()
    i1()
    i2()
}
class B {
    a2()
    b1()
    b2()
    i2()
}
class C extends B implements I4 {
    a1()
    b1()
    c1()
    i1()
}
```

containing more than one method would occur when there is over-riding and/or interfaces present. However, in all cases, the methods within a method group must always share the same method signature (i.e. method name and types of arguments), since a method can never over-ride or implement a method with a different signature.

The first step in creating a method group is to examine all the classes and interfaces and load the method definitions from each. During this process, the super/sub-class, super/sub-interface and interface implementation information must be maintained to allow for easy traversal of both the inheritance tree, as well as the graph of interfaces and implementing classes.

3.4.1 Creating Method Groups

Each method will need to be assigned a method group, therefore, the approach used is to traverse the inheritance tree, starting at *java.lang.Object*. For each class, each method in the class will be assigned a method group and a search made for other methods that need to be in the same group. After all methods in the current class are considered, the algorithm will recurse into all sub-classes of the current class. Considering the example system in Figure 3.3, the algorithm would start in *java.lang.Object* (which is not shown), which in this example has no methods. Both class *A* and *B* are implicitly sub-classes of *java.lang.Object* and therefore would be searched next (the exact order is not important, but for this example they will be considered alphabetically).

As each class is visited, any methods not already in a method group will be assigned a new one, and a search made from that point in the graph to find other methods to include in the group. When talking of searching, either in classes or interfaces, the terms "up" and "down" will be used for simplicity. In the case of a class, searching up refers to visiting the super-class, while searching down refers to visiting all sub-classes. Likewise for interfaces, with the exception that interfaces may have multiple super-interfaces. This describes searching either within the interface graph or within the inheritance tree. At places where, when searching interfaces, an interface is implemented by a class, or when searching classes, the class implements an interface, then the search must also transition

from classes to interfaces or vice-versa. After any such transition, there could be instances of the target method above the current position. Therefore, the algorithm needs to, after any such transition, search up to find any instances of the target method, then travel down from each place where an instance is found, to find any lower places that over-ride that same method. This, "search up, assign down" approach means the algorithm will find the highest place where a given method is used, then travel down finding all instances of the method being over-ridden or places where classes implement an interface method.

The above results in an algorithm with four basic actions, two for classes and two for interfaces. In the case of either classes or interfaces, the first action is to search upwards for instances of the target method, then secondly, to search back down from each found instance and assign all matching methods to the current method group. It is only during the assignment process that connections between interfaces and classes (i.e. interface implemented by a class, or a class that implements an interface) would be considered, triggering a search to start in the other. This algorithm described here is presented as pseudocode in Algorithm 3. The *searchClass(...)* and *searchInterface(...)* methods are called with the name and descriptor of the target method and the method group that instances of the target method should be assigned to. The *assignClassmethodGroup(...)* and *assignInterfaceMethodGroup(...)* methods are both used once an instance of the target method is found, to actually perform the assignment of each instance tot he method group, and to also perform the transitions between searching classes and searching interfaces. It is quite common for this algorithm to fold back on itself, and therefore the *lastSearchedFor(...)* method will return true if the previous call to that method was for the same name and descriptor, preventing infinite recursion. The *containsMethod(...)* method will return true if the class or interface contains a method with the given name and descriptor, while *assignMethod(...)* will add the instance of the method in that class or interface to the given method group.

To better understand this algorithm, consider the classes found in the example Java system in Figure 3.3. Starting from *java.lang.Object* (not shown in the figure), there are no methods, so the process continues with class *A*. The first method encountered is *a1()* and as this method does not currently have a method group, a new one is created. The *searchClass* method will find no super-classes that contain the *a1()* method, however the present class,

Algorithm 3 Search functions for creating method groups

```
searchClass(name, descriptor, methodGroup) {
    if (superClass != null)
        superClass.searchClass(name, descriptor, methodGroup);
    if (containsMethod(name, descriptor))
        assignClassMethodGroup(name, descriptor, methodGroup);
}
assignClassMethodGroup(name, descriptor, methodGroup) {
    if (lastSearchedFor(name, descriptor)) return;
    if (containsMethod(name, descriptor))
        assignMethod(name, descriptor, methodGroup);
    for (each subclass C)
        C.assignClassMethodGroup(name, descriptor, methodGroup);
    for (each implemented interface I)
        I.searchInterface(name, descriptor, methodGroup);
}
searchInterface (name, descriptor, methodGroup) {
    for (each superinterface I)
        I.searchInterface (name, descriptor, methodGroup);
    if (containsMethod(name, decriptor))
        assignInterfaceMethodGroup(name, descriptor, methodGroup);
}
assignInterfaceMethodGroup (name, descriptor, methodGroup) {
    if (lastSearchedFor(name, descriptor)) return;
    if (containsMethod(name, decriptor))
        assignMethod(name, descriptor, methodGroup);
    for (each subinterface I)
        I.assignInterfaceMethodGroup(name, descriptor, methodGroup);
    for (each implementing class C)
        C.searchClass (name, descriptor, methodGroup);
}
```

A, does contain the method, causing the algorithm to proceed to *assignClassMethodGroup*. At this point the *a1()* method in class *A* is assigned to the method group. There are no sub-classes to search in this case, however there is an interface, *I4*. Therefore, the *search-Interface* method is used to recursively search up through the two super-interfaces. In this case the *a1()* method is found in the *I2* interface, causing the *assignInterfaceMethodGroup* method to be called. This then proceeds to search through all sub-interfaces, and at interface *I4*, to attempt to call *searchClass* on class *A* again, an example of where the algorithm will fold back on itself. Since class *A* has already been searched, it will be ignored, and the search would continue down through interface *I5* to class *C*. Searching up from class *C*, no higher up instances of *a1()* are found, leaving the instance in class *C* to be added to the method group. At this point the search is exhausted and completes, having added three methods to the method group.

The second method encountered in class *A* is the *a2()* method. In this case a search is made through the interface, but since the method is never found in any interfaces the search terminates with the *a2()* method in a method group by itself. In particular, the *a2()* method in class *B* will not be discovered.

Next the method encountered in class *A* is is *i1()*. Again, the same search process is used, however in this case the *i1()* method is discovered in the interface *I1*. The *assignInterfaceMethodGroup* is therefore called, resulting in the interface method being added to the same method group and a search made from that point down. In this case the search will result in the algorithm folding back on itself when it attempts to call *searchClass* on class *A* again. Importantly however, it will not reach class *C* and therefore not find the *i1()* method declared there. This is correct as the classes *A* and *C* do not share an inheritance relationship, and class *C* does not implement the interface *I1*.

The final method in class *A* is the *i2()* method and again a similar search process is performed that will first find the method in class *A* itself, then search up through the interfaces to find the instance in interface *I2*. Since the method is found in *I2*, the algorithm moves back down from there searching for other instances of the method to be added to the method group. While no new instances are found during the downward search through the interfaces, the algorithm does encounter class *C*, causing *searchClass* to begin there. This

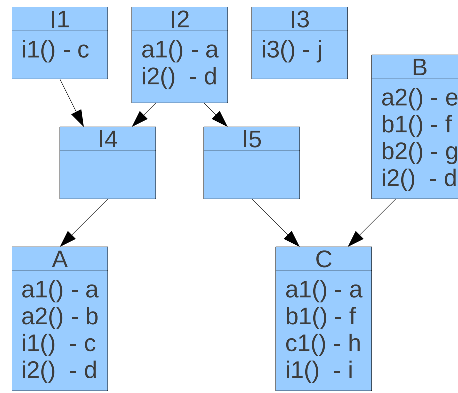


Figure 3.4: Example System with Method Groups Assigned

is an example of the need to always begin by searching up when moving from interfaces to classes or vice versa, since in this case class *C* does not contain the target method, but its super-class, class *B* does.

Similarly, the process continues for each method in class *B*, assigning method groups to each method (except for method *i2()* since it was added to a method group earlier), then moving on to find the last remaining methods in class *C*. Finally, after all classes have been searched, a final pass must be done over all interfaces to ensure every interface method was assigned a token, e.g. interface *I3* and the *i3()* method in the example would have been missed, since it is not implemented anywhere. For global tokenisation, this is not as important, since the interface is not implemented anywhere, it will never be used and if classes are added later that do implement it, the tokenisation will be re-performed. However, this becomes more important for the next chapter, when incremental tokenisation is introduced, and classes may be added later that implement the interface without these current classes being re-tokenised.

The final result of running this algorithm is shown in Figure 3.4, where letters are used to indicate the method group each method belongs too. Letters are used so as not to confuse the method groups with tokens, which have not been assigned yet, and is discussed in the next section.

3.5 Assigning tokens to Method Groups

With method groups in place, tokens will be assigned to a method group, rather then individual methods, since every method in a group, by definition, requires the same token. Even though tokens are assigned to method groups, it is useful to refer to a method as being "tokenised" if the method group that contains that method has had a token assigned, even if the assignment happened via another method in the method group. Likewise, a class can be referred to as "tokenised" if all the methods within the class are tokenised.

As discussed in Section 3.3.2, the ideal solution would be for the token values in each class to be contiguous (i.e. a given table will, as much as possible, have all its entries used), which would result in the smallest possible dispatch tables. The other constraint was that tokens in a sub-class should, as much as possible, be greater than the tokens used in the super-class, which allows the possibility of only storing the last, changed, section of the dispatch table.

Since sub-classes should have higher token values than super-classes, the tokenisation process should start at the root of the inheritance tree with the *java.lang.Object* class. The algorithm contains a list of classes currently having tokens assigned (which initially only contains *java.lang.Object*) and the current token value (initially 0). While the list contains classes, an attempt is made to assign the current token value to each class, then any class with is now fully tokenised is removed and all its sub-classes are added to the list, then finally the token value is incremented and the process repeats until the list is empty. The complexity arises when attempting to assign a token to a given class, in which case each un-tokenised method is consulted to determine if it can have the token assigned to it.

To attempt to assign a token to a class, an attempt is made to assign the token to each un-tokenised method in the class until one succeeds (or more correct, to each un-tokenised method group). For a method group to be able to use a given token, a search must be made for any possible conflicts, that is, cases where assigning the given token to this method group would result in a class whose virtual method table contained two different methods with the same token. Since virtual method tables are constructed only in classes, interfaces do not need to be considered in this process. For the method group being considered, iterate

through each method contained in the method group and therefore to the class the method is contained in. For each of these classes, search recursively both up and down the inheritance tree to find any places where the target token is already used. If any case where the token is already in use is found, the token can not be used and assignment fails, otherwise, there are not conflicts and the assignment is successful.

Applying this to running example, Figure 3.4 shows the example system with method groups in place. Starting from *java.lang.Object*, there are no methods, so all the sub-classes are added to the list of current classes. An attempt is then made to assign token 0 to one of the methods in class *A*. The first method in class *A* without a token is *a1()*, so each method in the group is checked to see if it can use the token. Since this is the first assignment, there are not any conflicts anywhere and the assignment succeeds. The same attempt is made to assign token 0 to class *B*, and again the first method in this case is *a2()*, which is the only method in method group *e*. As such, class B is searched to see if it can use token 0, and no conflicts are found. However, when search class *B*'s sub-classes, it is discovered that class *C* contains a method already using token 0, causing this assignment to fail. This highlights the need to search both up and down the inheritance tree before assigning a token. All classes currently being processed (*A* and *B*) have now been considered, so token 0 has now been assigned everywhere it can be, additionally, both classes still have untokenised methods, so they both remain on the list.

Next an attempt is made to assign token 1 to class *A*, and the assignment succeeds on method *a2()* and likewise in class *B*, where the other *a2()* method is also assigned token 1. The fact that both *a2()* methods have been assigned the same token in this case is coincidental.

The process repeats again, this time attempting to assign token 2 to both classes, again succeeding in both cases resulting in method group *c* and *f* both getting assigned token 2 and both classes remaining to be processed. Next an attempt is made to assign token 3 to class *A*, which succeeds in assigning it to the method *i2()* and therefore method group *c*. When attempting to assign token 2 to class *B* next, the only un-tokenised methods left are *b1()* and *b2()*, however assignment to both of these fail because token 2 is already in use in class *B* due to the *i2()* method. At this point, all methods in class A now have tokens, so

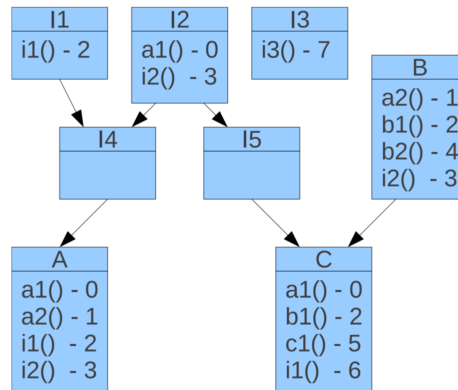


Figure 3.5: Example System After Assigning Tokens

the class is removed from the list of classes current be processed and all its sub-classes (of which there are none) are added.

The process continues, assigning token 4 to the last method in class *B*, then tokens 5 and 6 to class *C*. Figure 3.5 shows the example system with the token values displayed next to each method. Any remaining methods in interface that do not already have tokens, must be methods that are never implemented any, e.g. method *i3()* in the example. While the above does not mention it, a final pass is made over interfaces to assign token to any of these methods. As mentioned earlier, this is not important for global tokenisation, since these methods are never implemented, and therefore will not be used. However, it becomes relevant for the next chapter when introducing incremental tokenisation.

There will be cases where a class will not be able to use a given token value, resulting in a 'hole' in the class's virtual method table, e.g. class *B* in the example does not use token value 0. This 'hole' will be an entry for the unused token that would map it to a null value. Provided the original class files were well formed, calls to these null tokens should never occur. Such events could be guarded against by the virtual machine when performing method invocations, causing an Error to be thrown if they arise.

3.6 Dealing With Static Methods

Static method calls do not carry the same complexity as virtual method calls and have their own instruction, *invokestatic*. The simplicity is due to there being no object reference,

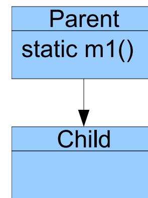


Figure 3.6: Static method in a super-class

rather the *invokestatic* instruction will include a method reference (class name, method name and method descriptor), which will always resolve to the same method (assuming the classes are not changed and recompiled).

While the call will always resolve to the same method, the reference might not be to the exact class that contains the method, but could be to a sub-class. The reason for this is to maintain binary compatibility between class files in situations where otherwise incorrect behaviour would result. The next section covers binary compatibility in detail, followed by how static methods are represented in tokenised classes.

3.6.1 Binary Compatibility and Static Methods

Method references supplied to the *invokestatic* instructions are the same as for all other invoke instructions (a class name, method name and method descriptor). Depending on how the code is written, this can lead to cases where the target method is not in the class specified in the method reference. Consider the classes in Figure 3.6. A perfectly legal call would be:

```
Child.m1();
```

This in turn would be compiled to an *invokestatic* instruction, with a class reference of *Child* and the *m1()* method. At runtime the *invokestatic* instruction must search the *Child* class, upon finding no matching method, search the *Parent* class and find the method. Adding extra complexity to the lookup process seems redundant, when the compiler could have resolved the method call to point to the class *Parent*. However, having the compiler resolve references introduces issues with binary compatibility.

Assume the above method call exists within an application that will not be recompiled. If the Child class were modified, so that is now contains a static method also called m1(),

and then recompiled, the above reference in the application should call this new method in the *Child* class. However, if the compiler had modified the reference in the application to point to the *Parent* class (where the target method was), then the application will continue to call the static method in *Parent*, even though a method clearly exists in *Child* and the method call clearly states it wants to call that method. While a recompile of the application would fix this situation, the designers of Java have chosen to retain the method reference as it appeared in the original source code, so that later changes to other classes (such as the addition of a method to the *Child* class) will still produce the expected results.

3.6.2 Static Methods in Tokenised Classes

In the above discussion, a single class could be modified without changing any others, therefore care had to be taken to maintain the expected behaviour of an application. Since tokenisation is essentially a pre-linking of a given set of classes, and will be entirely recomputed if any class changes, it is reasonable to resolve the static method references to their final target method, removing the need to perform searches of classes to find matching methods at runtime. Additionally, since runtime binding is not needed, static methods will not need to be allocated tokens as part of the tokenisation process for virtual methods. Therefore, static method references will be resolved to the class that contains the target method, meaning that static method tokens only need to be unique within a single class.

As a result, static methods are stored separately from non-static methods within a tokenised class file. For each class, static methods are allocated tokens from 0 to $n-1$, where n is the number of static methods in that class. For each *invokestatic* instruction, the method reference is used to resolve the final target method (which might not be in the referenced class, as discussed above), then the instruction is updated to include the class token and static method token for the resolved method. Once installed onto a device, these references could then be resolved completely to the static method's memory location, further improving the performance of static method calls.

By tokenising static methods separately, they do not need tokens in the virtual method tables, meaning the tables can be smaller.

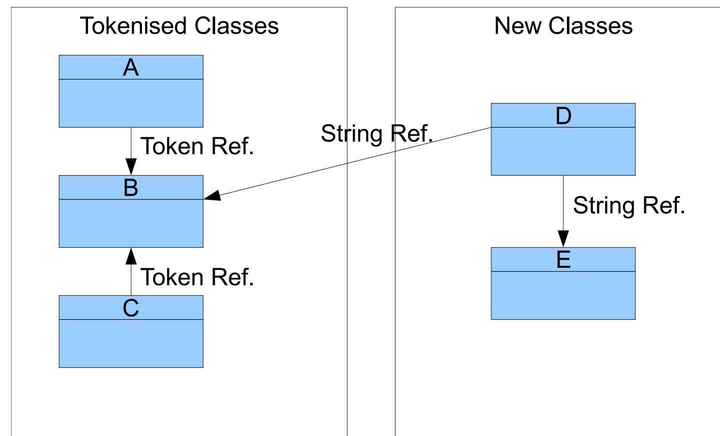


Figure 3.7: Adding new classes to an already tokenised system

3.7 Descriptor File

The tokenisation scheme presented in this section shows how it is possible to use virtual table dispatch, even in the presence of interfaces, thus allowing the *invokevirtual* and *invokeinterface* instructions to share an identical implementation. While the analysis to determine method groups is complex, the actual allocation of tokens becomes simple. If some additional applications or libraries are to be installed after tokenisation has occurred, all the classes on the device need to be re-tokenised. Therefore, the tokeniser is required on the device.

When considering the set of class files on the device, there will be an internally consistent linking between all these files using only the token data. Therefore, the symbolic string data originally used for linking purposes is not required at runtime. However, if new class files are added to the system, then these new files will contain references to the existing classes in string form. Figure 3.7 shows a case where classes A, B and C are already on the device and tokenised, with references between (and within) these classes all using tokens. While the new classes D and E will reference each other using the standard string form, as well as any references to existing classes (i.e. to the API classes). For a new tokenisation to be performed, these string references must be resolved, meaning the string names for classes and methods within the already tokenised files, must be kept on the device.

Therefore, the output of the tokeniser will consist of the tokenised class files, as well as a “descriptor file”, which stores the symbolic string data from the tokenised class files.

Using the information in the descriptor file, it is possible to reverse the tokenisation on a class file and recover the original string names for classes and methods, which will be required when adding new classes to the system.

3.8 Tokenisation of Fields

Fields in Java are referenced in much the same way as methods, with a symbolic reference via strings in the constant pool that consist of a class name, field name and field type. Therefore, tokenising of these field references was investigated during class tokenisation and the process was found to be trivial. This section presents the issues relevant to tokenising fields.

There are four instructions used to load and store values in fields, these are: *getstatic* & *putstatic* for loading and storing values in static fields, and *getfield* & *putfield* for non-static fields. Examining the standard version of these instructions, as defined in the Java Virtual Machine Specification [57], they are found to be very similar. All the instructions are followed by an index into the constant pool to define which field they are referencing. The *get* instructions both result in the loaded value being pushed onto the stack, while the *put* instructions both remove a value from the stack. The main difference is that the non-static instructions will also consume an object reference from the stack.

The constant pool reference, provided as an operand to the instruction, is an index to a CONSTANT_Fieldref structure as defined in Section 4.4.2 of the Java Virtual Machine Specification [57]. Figure 3.8 shows how a fieldref entry provides the required symbolic information. The fieldref entry points to two intermediate entries, the CONSTANT_Class and CONSTANT_NameAndType. These in turn point to the final three CONSTANT_Utf8 entries, which provide the class name, field name and field type, in much the same way a method reference contains a class name, method name and method descriptor.

The two types of fields, static and non-static, are handled separately. A static field is contained within a class, while a non-static field will be within an object. The following section will look at how often field references are used, to justify the importance of also optimising field instructions. Following that, first static, then non-static fields are examined

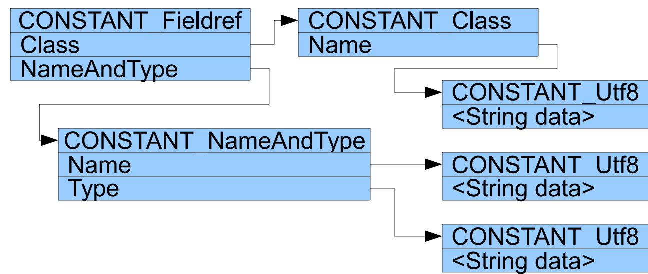


Figure 3.8: Structure of a symbolic field reference

and how tokens can be allocated to them. While these sections cover allocation of tokens, Chapter 5 covers implementing a tokenised virtual machine. In particular, Section 5.3.2 covers the get/put-static instructions and Section 5.3.3 covers the get/put-field instructions.

3.8.1 Static Fields

A static field is a global variable, in that there is only ever one instance of the field in the virtual machine. This means there is no need to allocate space for a static field in each object of the class. The space to store a static field only needs to be allocated once, when the class is loaded. Static field references will always contain a class, however this may not be the class that actually holds the static field. As discussed in Section 1.4.4.5, standard class files must be able to be recompiled individually and still maintain the expected behaviour. As such, references to static methods and static fields can sometimes be to a class that does not actually contain the method or field. During tokenisation, the converter will perform the same search the virtual machine would normally have to perform to find the class that contains the target field. The field reference will then be updated to contain the class token for the final target class.

Now a static field's token need only be unique for the class that contains the static field. When the class is loaded and the static fields are initialised, the tokens can be used to create a table of fields for that class. Any get/put-static instructions can use the token value to efficiently lookup the table of static fields. This has been combined with a modification to the get/put-static instructions to include the class and field tokens into the bytecode operands. More details on the instructions are given in Section 5.3.2.

3.8.2 Fields in Interfaces

Java allows fields to be declared in an interface. At first it seems the compiler will allow the declaration of non-static (although they must be final) fields in an interface. Section 9.3 of the Java Language Specification [61], however, states that "Every field declaration in the body of an interface is implicitly *public*, *static*, and *final*", even if the field does not explicitly list these qualifiers (although it can).

While at first glance it appears that fields in interfaces can be non-static, and therefore will need to be handled with the other non-static fields, this is not the case. Instead the fields can be tokenised and referenced as defined above. This is an important point, as it greatly simplifies the following section handling non-static fields. Not only this, but the fields must be final. While in standard Java, this must still result in a *getfield* instruction, so that if the interface is recompiled and the values changed, other class will read the new values. However during tokenisation there is no reason why these final values cannot be included into classes as constant values.

3.8.3 Non-static Fields

The non-static fields for a class are the fields that will exist in every object of that class. When a class extends a super-class, all the super-classes non-static fields will continue to exist, while any new ones will need to be added. This means that for an object of a class C. Entries need to exist in order to hold the value of every non-static field in C, as well as every non-static field from the super-class of C and so on all the way up to java.lang.Object. This means that field tokenisation must now take into account the field tokens that have been allocated in the super-class. However as stated in the section above, interfaces will never have non-static fields, meaning only the inheritance tree of class files needs to be considered, rather than the graph of interface relationships.

Allocating tokens in this tree structure can be accomplished with a simple algorithm, starting at java.lang.Object and allocating increasing tokens down each branch of the tree. This is the same sort of token assignment as used for methods in Java Card (discussed in Section 2.5). This also guarantees that if the total number of non-static fields for a class and

all its super-classes is n, then the field tokens will always be in the range of 0 to $n-1$. These tokens can then be used to find the offset to the field with an object. Detailed information about the implementation of *getfield* and *putfield* is in Section 5.3.3.

3.9 Libraries Used For Testing

Several libraries and applications were used throughout testing and will therefore be used through this thesis. Tests included:

- conversion tests, checking the efficiency of token allocation and virtual method table size,
- compression tests, to check the level of compression obtained,
- execution tests, to ensure that tokenised files were still executable, and
- testing the minimum size required for certain data types (such as instruction op-codes).

This chapter will only be examining the conversion tests in relation to the global tokenisation that has been presented, the other tests will be presented in later chapters.

Table 3.1 lists each test case used and where the files were sourced from. The CLDC, MIDP and J2SE tests all used packages available from Sun Microsystems. The Javolution [25] tests consisted of a third-party open-source library designed to support real-time applications, which also contained a benchmark suite. The test case names in Table 3.1 are used throughout the rest of this thesis when referring to different test cases.

The following sections discuss each of the libraries in more detail. Details for each test case include:

1. Test Case - The name used in this thesis to refer to the test case.
2. Classes - The number of classes found in the test case (includes concrete and abstract classes, but not interfaces).
3. Interfaces - The number of interfaces found in the test case.

Table 3.1: Libraries used for each test case

Test Case	Source	Description
CLDC1.0	CLDC 1.0.4 Reference Implementation	The CLDC 1.0 API.
CLDC1.1	CLDC 1.1 Reference Implementation	The CLDC 1.1 API.
CLDC1.1M	CLDC 1.1 Reference Implementation	The CLDC 1.1 API with reduced native methods.
MIDP	MIDP 2.0 Reference Implementation	The MIDP 2.0 API.
MIDPExamples	MIDP 2.0 Reference Implementation	Example applications using the MIDP 2.0 API.
Javolution3	Javolution 3.7.10	Real-time library and benchmark suite.
Javolution5	Javolution 5.2.5	Real-time library and benchmark suite.
J2SE	J2SE 1.4.2_17 JDK	J2SE API classes.

4. Methods - The number of method entries found, these may or may not have code associated with them. In particular, abstract methods, methods declared in interfaces or methods declared NATIVE are included in this number.

5. Methods with code - The number of method entries that also have bytecode associated with them.

6. Unique Selectors - The number of unique method selectors found in the test case. A method's selector consists of the method's name and the method's descriptor. There will typically be less unique selectors than methods, since some methods will reuse selectors, i.e. over-ridden methods or common method names, such as a *size()* method.

3.9.1 CLDC API

The CLDC 1.0.4 and 1.1 reference implementations from Sun were used. From each package the corresponding source files that implement the CLDC API were extracted. Each of these were independently compiled to form the CLDC1.0 and CLDC1.1 test cases.

Table 3.2: Size of CLDC tests

Test Case	Classes	Interfaces	Methods	Methods with code	Unique Selectors
CLDC1.0	99	14	793	656	354
CLDC1.1	85	13	625	524	316
CLDC1.1M	85	13	626	532	316

An additional version was also created, referred to as CLDC1.1M, used for execution tests in Chapter 5. The CLDC1.1 library includes several native methods, either for integration with the virtual machine or for performance reasons. To simplify implementation of a custom virtual machine, these native methods were, where possible, implemented in Java.

Table 3.2 shows the size of each test case, with CLDC1.0 slightly larger than CLDC1.1. The counts are nearly identical for the CLDC1.1 and CLDC1.1M packages, except for one additional method and more methods with implementations in the CLDC1.1M package as a result of replacing the native method prototypes with Java code.

3.9.2 MIDP API and Example Applications

The Mobile Information Device Profile provides services and libraries in addition to CLDC for applications on devices such as mobile phones or PDAs. Services in MIDP include application support (referred to as MIDlets), application life-cycle, GUI access and interaction. Significantly, this library is underpinned by many native methods.

The MIDP distribution from Sun Microsystems also includes several example applications. The classes for these example apps were separated from the MIDP API classes and formed their own test case. There are 13 example applications of varying complexity (from the equivalent of "Hello World", to a simple Towers of Hanoi game).

The MIDP library was significantly larger then the CLDC library, as shown in Table 3.3. Since the CLDC library is the "lowest common denominator" for the varied profiles it supports, it consists mostly of just framework, while the MIDP library provides implementation for many features (such as providing a complete networking stack). As well, the MIDP library adds extra functionality in the form of GUI support and user input. Finally,

Table 3.3: Size of MIDP tests

Test Case	Classes	Interfaces	Methods	Methods with code	Unique Selectors
MIDP	196	43	2305	1893	1264
MIDPExamples	74	0	547	547	317

Table 3.4: Size of Javolution tests

Test Case	Classes	Interfaces	Methods	Methods with code	Unique Selectors
Javolution3	262	52	1971	1629	781
Javolution5	367	62	2522	2072	966

the example applications are fairly large, with nearly as many classes as the CLDC library itself. However, none of the applications are complex enough to include interfaces, instead consisting entirely of classes and with all methods having implementations associated with them.

3.9.3 Javolution

The Javolution library [25] is an open-source library to provide support for real time applications and includes time-deterministic versions of many of the API base classes (such as the util / lang / text / io / xml packages). Two versions of this library were used for testing, version 3.7.10 and the newer 5.2.5. The Javolution library was chosen for three reasons:

1. Provides a runnable application (in the form of a benchmarking suite).
2. Can run on many platforms, including CLDC 1.0 & 1.1 without requiring any other libraries (most importantly, MIDP support is not required).
3. Is implemented purely in Java.

This allowed the Javolution library to serve as a complex library/application that could easily be executed to check for correct operation. As shown in Table 3.4, the 5.2.5 version contained 367 classes, which is more then the CLDC1.1, MIDP and MIDPExamples combined.

Table 3.5: Size of J2SE test

Test Case	Classes	Interfaces	Methods	Methods with code	Unique Selectors
J2SE	9075	1035	74313	65345	26717

3.9.4 J2SE API

While the main focus of the work is on J2ME, the API from J2SE was also as a very large scale test, with 9,075 classes and 1,035 interfaces, as shown in Table 3.5. Such a large number of classes proved a useful stress test, especially for checking the required sizes of some data structures. The 1.4 version of the J2SE API was used as J2ME and in turn the converter, does not support the extra features added in Java 5[2], such as generics.

In order to have a complete set of class files (since for every class, the converter must be able to load every other class that it references), several Jar files from the distribution had to be combined. The Java 1.4.2_17 version of the Java runtime environment for Linux was downloaded from Sun Microsystems and the JAR files containing the relevant class files were extracted from the distribution. In particular, the JAR files used were: rt.jar, charsets.jar, jce.jar and jsse.jar. All class files were extracted and placed into a single directory and form the classes used for the J2SE test case.

3.10 Global Tokenisation Efficiency

To metrics used to examine the efficiency of global tokenisation is the number of tokens used to tokenise all methods and the size of the dispatch tables. Results against these metrics are compared to previous techniques that have been used to generate dispatch tables, which were presented in Section 1.3. Testing was done using the packages described above, with some tests consisting of 2 or 3 of the packages being combined before tokenisation. The combinations tested consisted of the following cases ('+' indicates the packages were combined, then tokenised):

[2]Java version numbering changed between Java 1.4 and Java 5. Java 5 is equivalent to Java 1.5 in the old versioning scheme.

- CLDC1.0
- CLDC1.0 + MIDP
- CLDC1.0 + MIDP + MIDPExamples
- CLDC1.1
- CLDC1.1M
- CLDC1.1M + Javolution5
- CLDC1.1M + Javolution3
- J2SE

3.10.1 Token allocation efficiency

To avoid having large dispatch tables with many blank entries (i.e. for token values the class does not accept), the token allocation needs to ensure each class has a contiguous (or as close to as possible) range of tokens. As discussed in Section 3.3.2, there will be cases when null entries must be used within some dispatch tables.

While Java Card can produce dispatch tables which are always full (i.e. no null entries), they require secondary tables to dispatch interface methods. Section 1.3 discussed previous approaches to building dispatch tables.

In analysing the performance of the global tokenisation scheme, the number of tokens required to tokenise each test cases is first examined. Table 3.6 shows for each test case, the total number of methods (with or without code associated with them), the number of unique selectors (i.e. unique pairs of method name and descriptor) and the number of tokens required to tokenise that test case. The final column shows the number of tokens used by the selector colouring algorithm described by Dixon *et al.* [28] (and previously discussed in Section 1.3.1).

Selector colouring uses the same or slightly less token values than the tokenisation algorithm presented in this thesis. However, the overall size of the dispatch tables is more

Table 3.6: Number of tokens used during tokenisation

Test Case	Total Methods	Unique Selectors	Used Tokens	Selector Colouring
CLDC1.0	793	354	83	83
CLDC1.0+MIDP	3098	1492	101	101
CLDC1.0+MIDP+MIDPExamples	3645	1754	116	115
CLDC1.0 + Javolution3	2764	1060	84	83
CLDC1.1	625	316	44	44
CLDC1.1M	626	316	44	44
CLDC1.1M + Javolution5	3148	1204	74	74
J2SE	74313	26717	599	587

important than the number of tokens used, as it is the size of the tables that defines amount of memory required. Therefore, the next section examines the compactness of the dispatch tables.

3.10.2 Virtual Method Table Size

For any given class, the number of used (i.e. non-null) entries in the class's dispatch table is a fixed number, defined by the number of methods that can be called on the class. Therefore, the overall size of the dispatch tables, and therefore their memory requirements, for a system will depend on how many null entries are needed in the dispatch tables.

Table 3.7 shows the number of used entries in the Virtual Method Tables (VMTs) for each test case. The *Total VMT Entries* and *% Used* columns show the number of entries produced by the tokenisation algorithm presented in this chapter and therefore what percent of the virtual method tables were actually used (i.e. not null). The final column shows the percentage of the tables that were used when the selector colouring algorithm from Dixon *et al.* was used.

While Table 3.6 above showed that selector colouring could sometimes use less tokens than the approach in this thesis, Table 3.7 shows that the virtual method tables become very sparse, with just over 50% used in the best case. The final column shows the relative increase in the size of the dispatch tables in the case of Selector Colouring.

Examining the virtual method tables that resulted from both approaches, the reason

Table 3.7: Usage of Virtual Method Tables

Test Case	Used VMT Entries	Total VMT Entries	% Used	Selector Colouring % Used	Increase In Size (%)
CLDC1.0	2159	2179	99.08	32.36	306.19
CLDC1.0 + MIDP	7694	8271	93.02	35.55	262.04
CLDC1.0 + MIDP + MIDPExamples	10570	11998	88.10	38.39	229.48
LDC1.0 + Javolution3	7691	7976	96.43	30.86	312.47
CLDC1.1	1481	1487	99.60	50.62	196.75
CLDC1.1M	1482	1488	99.60	50.65	196.64
CLDC1.1M + Javolution5	8495	8800	96.53	33.97	284.17
J2SE	381163	586989	64.94	12.50	519.48

for such a large difference is due to the order of token allocation. While the tokenisation approach in this thesis attempts to allocate tokens in increasing order down through subclasses, selector colouring attempts to minimise tokens used. While, selector colouring can use less tokens, the average range of token values used in a given class is often much wider, therefore requiring many more null entries.

3.11 Conclusions

The tokenisation presented in this chapter will result in class files that are smaller in size (due to the removed symbolic linking strings) and faster to execute (due to the use of virtual method tables). The next chapter will detail an extension to the tokeniser, allowing incremental tokenisation. Chapter 6 then discusses the compression of class files and the tokenised class file format in detail. Chapter 5 details the modifications required to a virtual machine to allow it to execute tokenised class files natively. All the work presented in this thesis is then brought together in Chapter 7, which presents results to show the reduction in class size, the increase in execution speed and that tokenised applications can still be correctly executed.

Chapter 4

Incremental Tokenisation of Class Files

4.1 Introduction

The tokenisation presented in the previous chapter allowed for virtual method dispatch, via virtual method tables, even in the presence of interfaces. However, the tokenisation must be applied to all class files on a given device, in a single pass. A more practical approach would be able to apply the tokenisation incrementally, allowing new applications to be added to an already tokenised device, without needing to retokenise existing class files. This chapter will cover the complexities and changes required to allow such incremental tokenisation to take place.

4.2 Overview of Incremental Tokenisation Process

When performing incremental tokenisation, knowledge is needed of the existing classes, methods and the relationships between them. However, these classes are not going to be modified and as all the required information can be found in the descriptor file the converter does not need access to the class files themselves. In particular, the descriptor file will include details of each class (the class name, token and flags to indicate if it is abstract and/or an interface), details of each method (which class its in, name and descriptor strings, method token) and the contents of each method group.

Section 3.3.4 gave the overall process followed for global tokenisation. For incremental tokenisation the process is very similar, however an addition step is added at the beginning

to load the existing descriptor file. The incremental tokenisation process now consists of:

1. Load the existing descriptor file, and recreate the data structures to represent classes/methods/method groups. This represents the completely tokenised system as it currently exists (i.e. without the new classes added).

2. Now load the new classes and add them to the existing meta-data. This gives a full set of classes and associated meta-data, some of which has been tokenised, and some which is new.

3. Perform method-group creation again. During this process, existing method groups must be taken into account, and conflicts handled when they arise.

4. Assign tokens to those methods which still do not have tokens.

5. For the new classes, one at a time, read them into memory, update them with token information and save the resulting tokenised file back to disk.

6. Output the new descriptor file, which will contain the meta-data for the entire system (that is, the classes that were described in the original descriptor file as well as the new classes just added).

While steps 2 to 6 of the incremental process correspond to the 5 steps presented in Section 3.3.4, each step is slightly different, since some classes are already tokenised (i.e. when creating method groups and assigning tokens, the previously tokenised methods will already be in method groups and have tokens).

4.3 Difference to Simple Tokenisation

The simple tokenisation scheme presented in the previous chapter does a good job of minimising the token values used, but at the cost of needing be performed as a global operation. The next goal was to allow the above process to be applied in an incremental fashion, allowing the API to be tokenised, then applications tokenised without needing to modify the API. Figure 4.1 shows an example of the process to install an application on a phone.

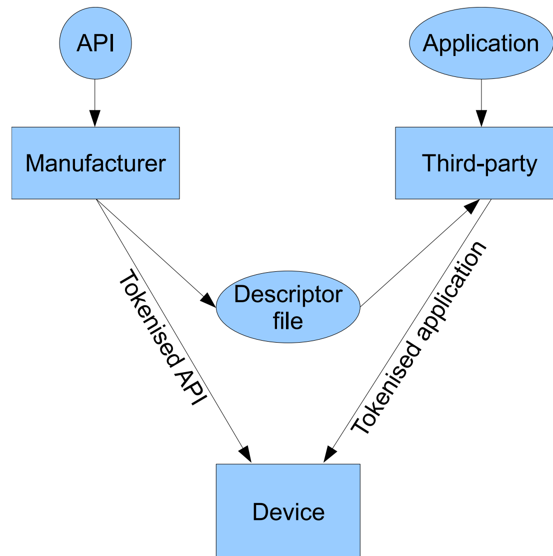


Figure 4.1: Work Flow for Incremental Tokenisation

Initially a manufacturer will produce a tokenised version of the API which is installed on devices. Next, developers download an SDK from the manufacturer that includes a descriptor file and the converter. The descriptor file describes the relationships between the name/descriptors used to identify methods and the tokens that are now used on the device. The third-party can use the converter, along with the descriptor file, to tokenise their application such that it will be executable on the device.

During global tokenisation, all relationships between classes/interfaces/methods could be discovered and taken into account. However, during incremental tokenisation, the converter has no knowledge of what classes or interfaces might be added later, leading to the converter making sub-optimal decisions. In particular, there are two types of problems that can arise as the result of new relationships being created between previously unrelated classes. Following are descriptions of these two specific problems. Section 4.4 then deals with how these problems can be overcome.

4.3.1 Same Methods, Different Tokens

The "same methods, different tokens" problem arises when two previously unrelated methods (which have the same name and descriptor) were assigned different token values, but through the addition of new classes, have now become related. Figure 4.2 shows an exam-

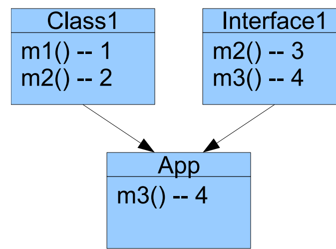


Figure 4.2: Methods with different tokens

ple of the "same methods, different tokens" problem. Here the method *m2()* in Class1 has the token value 2, while the method *m2()* in the interface *Interface1* has token 3. Since the *App* class was not present during the previous tokenisation, these methods were not in the same method group and therefore have different tokens, which was perfectly valid. With the addition of the *App* class however, the two methods now require the same token.

The goal of the incremental scheme is to allow *App* class to be tokenised without needing to modify the existing *Class1* or *Interface*. To resolve this problem, both token values, 2 & 3 can be used to reference the same method. The virtual method table in the *App* class will therefore have tokens 2 & 3 both point to the appropriate version of *m2()* (in this case, the one in *Class1*).

4.3.2 Different Methods, Same Tokens

The second problem arises when two different methods never appear in the same class, and therefore have been assigned the same token. However, the dispatch table of a new class must use the same token to call two different methods. Figure 4.3 shows a case where a new application class has extended an existing class and also implemented an existing interface. In this case the two existing entities were not previously related, therefore the same token (2) has been used to represent both *m2()* and *m3()*, depending on the context. At runtime, a request to dispatch token 2 for the *App* class is ambiguous, since it could refer to *m2()* from *Class1* or the implementation of *m3()* defined in *Interface1*. Since both *Class1* and *Interface1* can not be modified to avoid this conflict, extra steps must be taken at runtime to resolve the ambiguity.

The different methods, same token problem always involves exactly one class in the

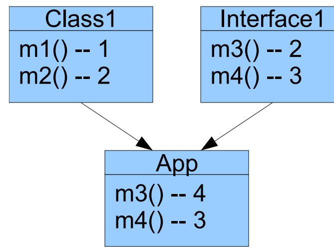


Figure 4.3: Methods with the same tokens

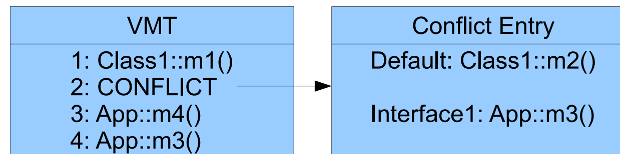


Figure 4.4: VMT with a conflict entry

existing system (since any class will only ever extend one other class in Java), and 1 or more interfaces (since zero would not result in a conflict). This property is important for resolving the situation.

There are two cases that can occur at runtime; firstly, an object of type *App* is stored in a variable that is of type *App* or one of its super-classes. In this case, a method call on that object would use the *invokevirtual* instruction. The second case is where the object of type *App* is stored in a variable that is an interface type (i.e. *Interface1*). This would result in a method call using the *invokeinterface* instruction.

Solving this problem involves adding a special “conflict entry” to the virtual method table (VMT) at the entry for token 2, which will consist of a “default” entry, and 1 or more “interface” entries. Figure 4.4 shows the virtual method table for the above example. For token values 1, 3 and 4, they map directly to the appropriate method. Due to the conflict with token 2 it now references a special conflict entry. When an *invokevirtual* instruction uses token 2, the default value in the conflict entry is always used (calling the method *m2()* in *Class1* in this case). The conflict entry also includes a table of interface entries, which will consist of an entry for each interface that introduced a conflict and which method should be called instead of the default. In this example there is only the one interface entry, for *Interface1*. For example, consider the following code snippet:

```
Interface1 i = new AppClass();

i.m3();
```

The above code will be compiled to an *invokeinterface* instruction for token value 2 (the token of the *m3()* in *Interface1*), and will include the class token for *Interface1*, to indicate the declared type. During execution, token 2 results in the conflict entry being found, a search is then made of the interface entries looking for *Interface1*. Upon finding a match, the virtual machine will use the pointer from that interface entry (resulting in the *m3()* method in the *App* class being called). If no matching entries were found for the given interface, the virtual machine will assume that interface did not introduce any conflicts and use the default entry.

Both the “same methods, different tokens” and “different methods, same tokens” problems can occur in the same class, and affect the same methods. However, the solution to each problem is not mutually exclusive. Therefore, both solutions can be applied, even to the same method, without introducing any further constraints.

4.4 Extending Method Groups

Figure 4.5 shows an already tokenised system consisting of a class and two previously unrelated interfaces, and a new class that is being added. The “same methods, different tokens” problem can be seen where the *i2()* method uses token 3 in class *A* and token 4 in interface *I2*. Also, the “different methods, same token” problem can be seen where interface *I1* uses token 1 for the method *i1()*, but class *A* uses token 1 for method *a1()*. This example will be used throughout this section and the following ones to show the process for extending the method groups, assigning tokens and finally building virtual method tables.

Global tokenisation had the concept of a “method group”, which consists of a list of one or more methods that require the same token and the token assigned to them. However, the difference in the incremental case is that for a given method group, some of the methods may already have a token assigned (or even multiple different tokens), while others are still in need of a token. Since a method group as used in global tokenisation can only have

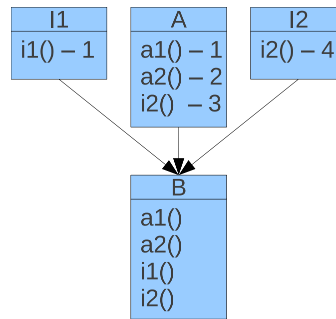


Figure 4.5: Example of adding incremental class

a single token assigned to it, they are considered a "single-method group". A new type, called a "multi-method group", is defined for incremental tokenisation. A multi-method group does not contain a token or list of methods itself, rather it contains a list of two or more single-method groups. A multi-method group with only one single-method group would be redundant, and can be represent by the single-method group on its own. This allows the representation of cases where a group of methods would ideally have the same token, but due to conflicts, end up with several different tokens. For example, to represent the *i2()* method from class *A* and interface *I2* in Figure 4.5, there would be two single-method groups with the token values 3 and 4 respectively. Both of these would then be contained within a multi-method group (due to the addition of class *B* which now requires the two methods to be related).

Section 3.4.1 described the process of creating method groups during global tokenisation and consisted of a single pass over all the classes and interfaces to create the method groups. Incremental tokenisation requires a two-pass process. The first pass is identical to global tokenisation, however each time a group of methods that require the same token is identified, some methods will already have method groups (and therefore a token assigned), while other methods may be new and require a token. Multi-method groups are created in these cases, with already tokenised methods in single-method groups with their token, and newly added methods in another single-method group yet to be assigned a token. This first pass builds the information needed by the tokeniser to be able to determine which tokens are used where. The second pass then performs a simplification, attempting to move any untokenised methods into an already tokenised one in the same multi-method group (if one

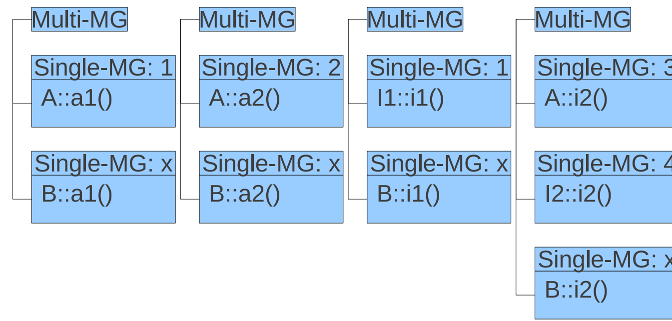


Figure 4.6: Method Groups for the example system

exists).

With the concept of single and multi-method groups defined, consider the process of creating method groups in the incremental case. The tokeniser uses the same search technique as described in Section 3.4, which produces single-method groups, however some or all of the methods may already have a method group assigned. For example, the tokeniser could start by examining class *A* and method *a1()*, where it will find the instance in class *A* and the one in class *B*. During the first stage of method group creation the tokeniser doesn't attempt to determine if an untokenised method (i.e. *a1()* in class *B*) can use a given token (i.e. token 1 used by the *a1()* method in class *A*). This is because the tokeniser does not yet have complete knowledge of which methods require the same tokens and which do not, and therefore can't know if a given token can be freely used in a given class or note. Therefore, a new multi-method group is created containing two single-method groups, one with a token already, one without. The process continues and will result in four multi-method groups, one for each of the methods: *a1(), a2(), i1()* and *i2()* as shown in Figure 4.6. For the *a1(), a2()* and *i1()* multi-method groups, there will be two single-method groups, one with a token and one for the newly added method in class *B*. In the case of the *i2()* method, there will be three single-method groups, one for each of the existing tokens and a final one for the untokenised method in class *B*.

Once the first pass is complete, the tokeniser can then attempt to simplify the method groups, now that it has complete knowledge of which methods need the same tokens. The same process that was presented in Section 3.5 for checking if a given method can use a given token without conflict is used here. For example, in the case of the *a2()* method, the

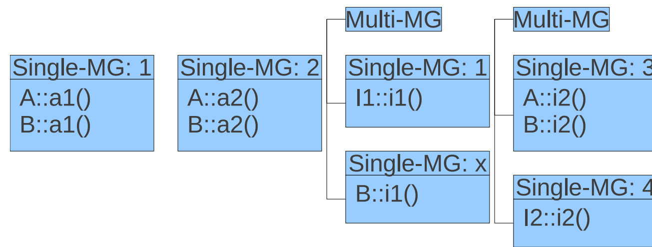


Figure 4.7: Method Groups after simplification

instance in class *B* does not have a token, so the same process as in global tokenised is used to find that token 2 is not used anywhere else. At this point the *B::a2()* method can be moved into the already tokenised method group along with the *A::a2()* method. The now empty single-method group is removed. Since the multi-method group now contains only one single-method group, it is also removed, leaving just the single-method group containing both methods. In the case of the *B::a1()* method, it would appear that it couldn't use token 1 since that is in use by *I1::i1()*. However, since *B::a1()* is overriding *A::a1()*, that takes precedence and multi-method group can be simplified. This means that the *B::i1()* method can't be simplified, since token 1 is already in use, so must remain. For the final multi-method group, the *B::i2()* method can be combined with either of the existing tokens, since both 3 and 4 are not used anywhere else. This results in the simplified method groups shown in Figure 4.7.

4.5 Assigning tokens

After method groups have been updated and simplified, tokens must be allocated to any remaining single-method groups that do not yet have tokens. The process is the same as for the non-incremental tokenisation, detailed in Section 3.5, except for the case of multi-method groups and those method groups which already have tokens. Any method groups that already have tokens are those that were tokenised during a previous run of the tokeniser. Since incremental tokenisation can not change any tokens that were previously allocated, the method group is considered tokenised.

A multi-method group is considered tokenised only when all of its sub-groups have tokens. A multi-method group will always have at least one set of methods with an existing

token and up to one set of methods without a token. During the simplification of method groups, the multi-method group will have been tested to see if untokenised methods could use any existing tokens, therefore any remaining untokenised methods will require a new token. Section 3.5 described assigning tokens to method groups for global tokenisation. If a method is a part of a multi-method group, then the token allocation is attempted for the group of untokenised methods within the multi-method group. If there are no untokenised methods, then the method group is considered tokenised.

In the case of the example, there was only the one method group left untokenised after simplification, as shown in Figure 4.7. Since tokens 1 to 4 are already in use, the next available token, 5, will be assigned.

4.6 Building Virtual Method Tables

After all methods have been assigned tokens, Virtual Method Tables (VMTs) can be built for each class file. When installed to a device, the VMT will contain pointers to target methods. However, since the tokeniser outputs individual class files, absolute offsets can not be given to methods outside the current class. The tokeniser will instead output a VMT which contains symbolic references to the target methods. During installation, these symbolic references can be converted into absolute offsets for a given device. The symbolic reference will include a class token to indicate the relevant class, and a method token. Unlike method references via a class's constant pool, the class token will always indicate the class that contains the target method. Since a class can only ever have one method with a given token, a class token/method token pair will always identify a single unique method.

The final VMT can be considered an array with three possible types of entries: null, single and multi entries. The null entries indicate a token value that can never be used, while single and multi entries denote the single- and multi-method groups respectively. The multi-method groups resulted from the "same methods, different tokens" problem discussed in Section 4.3.1. This situation is resolved by having multiple entries in the one virtual method table point to the same method. The "different methods, same tokens" problem discussed in Section 4.3.2 occurred when two or methods with different selectors

were required to have the same token. To resolve this situation, the virtual method table entries in this case must differentiate between the selector gained via inheritance (the so called "default" entry used for the *invokevirtual* instruction), and the ones gained via interfaces (and used by the *invokeinterface* instruction). This requires building the virtual method tables in several steps. The follow sections discuss the process in more detail, but as an overview, the process consists of:

1. Local tables are built. For a class, this consists of the super-class's table plus any methods in the current class. For an interface, the table just contains the methods in that interface. An entry consists of just a name/descriptor pair and the index at which it is placed, indicating that that token value should call a method with that name/descriptor. At this point there will be no conflicts in the tables. This identifies the "default" entries that should be used, should conflicts occur later.

2. Combine the local tables for each class with those from any implemented interfaces to produce a tokenised table. Any conflicts will become apparent at this point when, for the same token, the class's local table has a different name/descriptor than an interface's table does. These will be converted into a conflict entry. Additionally the name/descriptor pairs are resolved during this process to the final target method.

3. Using the tokenised table, generate an encoded VMT attribute where each entry is a class token/method token pair, which denotes the exact class and method that would need to be executed (since many of these target methods could be in other classes). This encoded form is then stored in the binary class file as an attribute.

4. When the class is loaded by the VM, use the class token/method token pairs to resolve the target method and produce a table with direct links to the target methods (which could be performed during installation on a device, during VM startup or in a lazy manner as entries are used).

A	B	I1	I2
1: a1()	1: a1()	1: i1()	1:
2: a2()	2: a2()		2:
	3: i2()		3:
	4:		4: i2()
	5: i1()		

Figure 4.8: Local tables for example classes

4.6.1 Building Local Tables

A class or interface's local virtual method table will only contain entries for methods in that class or interface, where each entry will consist of the method's name and descriptor strings. The local table provides the first stage of building virtual method tables, indicating for a given token (index into the VMT), the name and descriptor of the method to call, but not which class that target method is in. In the case of multi-method groups, there will be more than one token. In this case, the token for the single-method group that contains the method from this class will be used when building local virtual method tables.

To build a given class's local table, the super-class's table is needed, therefore construction of tables must begin at *java.lang.Object* and proceed down via the inheritance pathways to each sub-class in turn, once its parent is complete. In the case of interfaces, an empty table is always used as the starting point. For each class, the following process is used:

1. Copy the parent's table (or for *java.lang.Object* or an interface, start with an empty table).

2. For each method in the current class, find its token (for methods with more than one token, i.e. in a multi-method group, take the token for the individual group that contains that particular method).

3. Insert the method's name/descriptor pair into the local table at the given token offset. If an entry already exists at that offset, it will always have the same name/descriptor (since it would be over-riding a method in a super-class).

Figure 4.8 shows the result of applying this process to the example classes. Each interface only has a single entry for their respective methods. Similarly, for class *A*, only the

two methods defined in that class are included. For class *B*, the two entries from class *A* are inherited, then the two new methods defined in class *B* are included. Since the table in class *B* does not take into account interfaces yet, the conflict in token 1 is not yet apparent.

4.6.2 Building Tokenised Tables

Tokenised virtual method tables will combine the class's local table with those from any interfaces the class implements, while also resolving the name/descriptor strings from the local table to the final target method and hence a class token and method token pair. The class/method token pair is used a symbolic reference to the exact method to call. When installed to a device, these entries can be updated to more direct references, such as a memory offset of where the target method resides in memory.

To begin, the methods in the class's local table are resolved to their targets. To resolve each entry in the local table, a search is made starting in the current class (the one that the virtual method table is being built for) and searching for a method with a matching name and descriptor. If not found, the search is performed recursively in the super-class, until a match is found (and one will always be found, or the entry would not have been in the local method table). For the local tables generated in the previous section, shown in Figure 4.8, resolving the methods is trivial. For both the *A* and *B* classes, all the methods in their local table are present in the class itself. If class *B* had inherited methods that it didn't override, then those entries would be resolved to the appropriate method in the super-class. If the "different methods, same tokens" problem presented in Section 4.3.2 is encountered for any tokens in the class's virtual method table, then these current entries will become the default value in the conflict entry.

The next step is for any classes that implement interfaces to merge the interface's local table into its own. For each entry in an interface's table, the entry for the corresponding token in the class's table will be check. There are three possible outcomes:

1. The entry in the class's VMT is null. The "same methods, different tokens" problem has occurred and the target method currently has a different token value pointing to it. The existing entry is left, and the name/descriptor from the interface's table is

used to resolve the target method, which is added to the class's VMT at this index.

2. There is already an entry present, but it has the same name/descriptor as the interface's entry. Then a call for that token value will execute the correct method and nothing else is needed (this is the ideal case when neither of the "same methods, different tokens" or "different methods, same tokens" problems have occurred).

3. There is already an entry present, but it has a different name/descriptor. The different methods, same token problem has occurred, which means this entry must behave differently when an object is declared to be of this interface type and a conflict entry is needed in the VMT. The default entry will be the current entry, while an extra conflict line will be added for this interface type, pointing to the method found after resolving the name/descriptor from the interface's table.

For the running example, only class *B* implements any interfaces. Taking the local tables as shown in Figure 4.8, class *B* will need to be merged with both *I1* and *I2*. When merging *I1*, a conflict is discovered, since class *B* uses token 1 for the *a1()* method, however interface *I1* uses it for the *i1()* method. This causes the entry in class *B*'s virtual method table to be converted to a conflict entry. The default entry will be the current method (i.e. the *a1()* method), while a conflict line will be added to indicate that if an object is being treated as an *I1* type, token 1 should call the *i1()* method. Finally, the table from interface *I2* is also merged. In this case the additional entry for the *i2()* method fits into the unused slot at token 4. The final virtual method tables are shown in Figure 4.9. The effects of the "different methods, same tokens" problem can be seen in the conflict entry for token 1 in class *B*. While the effect of the "same methods, different tokens" problem can be seen in the case of tokens 3 and 4 both referring to the same method. Virtual method tables are only used by objects of a given class. Since interfaces can not be used to create objects (rather an object of a class is created, which implements the interface), they do not require virtual method tables. At this point the virtual method tables for the example system of have been generated and all that remains is to encode them into the binary class files.

A
1: A::a1()
2: A::a2()

B	
1: default:	B::a1()
I1:	B::i1()
2: B::a2()	
3: B::i2()	
4: B::i2()	
5: B::i1()	

Figure 4.9: Virtual method tables for example classes

4.6.3 Generating an Encoded VMT

With the virtual method tables generated, each class will now have a VMT that consists of a class/method token pair to identify the target method (or in the case of conflicts, the default method and alternate methods for the interfaces that introduced conflicts). The final stage for the tokeniser is to encode this information into the tokenised binary class files, which requires encoding references to target methods. While direct method references could be stored for methods within the current class, methods within another class require symbolic references. A class token and a method token can uniquely identify any method within the system (since a class can not have two methods with the same token), therefore each method reference is encoded as a class token/method token pair. The installer or virtual machine for a device can resolve these references at install time, class load time or at runtime as entries are used, potentially updating them to direct references to the target method, depending on the design of the system.

Since objects can only be made of class types, it is only classes that require virtual method tables (in particular, interfaces do not require a VMT). To avoid overhead in interfaces, the VMT is added as an attribute at the end of a binary class file. The encoded VMT will contain a count for the number of entries, followed by that many entries. These entries will correspond to the entries in the VMT from index 0 and up, with three types of entries: null, single and conflict. Each entry in the table (null, single or conflict entry) will have the general form:

```
VMTEntry {
    u1 type;
}
```

```
VMTEntryNull {
    u1 type = 0;
}
VMTEntrySingle {
    u1 type = 1;
    u2 class_token;
    u2 method_token;
}
VMTEntryConflict {
    u1 type = 2;
    u2 default_class_token;
    u2 default_method_token;
    u2 conflict_count;
    {
        u2 interface_token;
        u2 class_token;
        u2 method_token;
    } conflicts[conflict_count];
}
```

Figure 4.10: Types of VMT entries for tokenised binary class files.

Where the *type* value will denote which of the three types the entry is. Figure 4.10 shows the three specific entries that can be present: Null, Single or Conflict. An 8 bit value for *type* is very inefficient for representing only 3 possible values, however all class files are encoded as a sequence of 8 bit words, making this the smallest possible choice. The high-order 6 bits could be used for future use to encode new information if required. A Null entry is used to denote a gap in the VMT, therefore it only requires a type. The Single entry is for a regular VMT entry and contains the class and method tokens. The Conflict entry is used to represent conflicts and will contain the default class/method tokens, as well as a variable length list for each conflict line.

The encoded VMT entries are stored within a VMT attribute, details of which are given in Section 6.5.8, while full details of the binary file format used for tokenised files is in Appendix A. Section 5.2 discusses how the virtual machine will make use of these encoded VMTs at runtime.

4.7 Issues with incremental tokenisation

Section 4.3 described both the “same methods, different tokens” and “different methods, same tokens problems” and their solution. Special “conflict” entries were added to the virtual method table to solve the different methods, same token problem. However, the addition of conflict entries introduces a new ambiguity which is discussed in the next section. Following that, is an examination of trying to extend multiple tokenisations, binary compatibility issues and finally the tokenisation of static methods.

4.7.1 Restriction of tokens in interfaces

The format used for conflict entries in a virtual method table can result in ambiguities. Consider Figure 4.11, where class *A* has extended interface *I3*, which in turn extends two other interfaces: *I1* and *I2*, and the methods have been allocated the tokens X, Y and Z as shown. The virtual method table for class *A* is shown in Figure 4.12. An *invokevirtual* for token value X will resolve correctly, as will an *invokeinterface* call for type *I1* or *I2* and

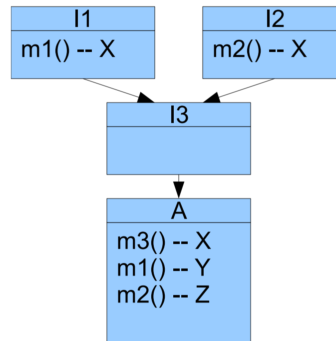


Figure 4.11: Possible ambiguities with interfaces

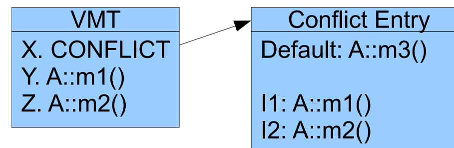


Figure 4.12: VMT with ambiguous conflict entry

the token value *X*. However, an *invokeinterface* call, with a declared type of *I3,* makes it impossible to know if *m1()* or *m2()* should be called.

The first solution would be to disallow the addition of an interface such as *I3* during an incremental tokenisation. However, this would be very restrictive for programmers, removing the ability to extend interfaces. The ambiguity only arises because two methods with different selectors (name/descriptor) have been allocated the same token. If this can be avoided, then there would be no problem.

Another solution is to restrict the use of tokens within interfaces. Once a given token value has been used for a given selector in an interface, then no other interface may use that token for a different selector. In other words, once the token *X* had been assigned to the method *m1(),* then it could only ever be used in an interface for a method called *m1().* A different, and unrelated, *m1()* method in another interface may use a different token, maintaining some flexibility in the allocation of tokens. However, this constraint will require at least enough tokens for each unique selector used in an interface, which is a relatively large number. For example, with out this restriction the CLDC 1.1 API and Javolution 5.2.5 library can be tokenised with 74 unique tokens (from Table 3.6). By constraining the use of tokens in interfaces however, the same classes then require 328 unique tokens, primarily because of the 322 unique selectors used in interfaces. Such a

large increase in token values will result in very large, and therefore very sparse, virtual method tables, making this solution less than ideal.

The final solution is to ensure an *invokeinterface* instruction will never point to the sub-class (or sub-interface) of the target method (i.e. the interface *I3* in this case). During tokenisation, a method reference to either *m1()* or *m2()* and the type *I3*, will be resolved to the type of *I1* or *I2* respectively (i.e. the interface that actually declares the method). After modifying the method reference, the ambiguity described above can not occur, since a method call will match one of the present conflict lines in the virtual method table's conflict entry.

Section 3.6.2 discussed how static method references will be updated to always point to the class which contains the target method (instead of possibly referencing a sub-class). The changes here are similar, updating a interface method call to reference the exact interface that declares the method, rather than one of the sub-interfaces. These updates can break the semantics of the class file, but only if the interfaces are updated without updating the classes that use them. However, tokenisation requires that if any classes/interfaces change, then the tokenisation must be re-performed, meaning the interface method references would be resolved again, hence finding any changes. Therefore, the modification of interface method references does not add any extra constraints on top of what the process of tokenisation has already added.

4.7.2 Extending multiple existing tokenisations

The tokenisation takes as input a set of standard class files and an optional descriptor file, and will produce a set of tokenised class files and a descriptor file. If a descriptor file is given as input into the tokeniser then all information from that descriptor file will also be in the output to the new descriptor file, along with information about the newly added classes, meaning descriptor files can only ever gain information. This approach was a design decision to simplify the implementation of the tokeniser. An alternate approach would involve each descriptor file containing a reference to a parent descriptor file. An incremental tokenisation will then produce a new descriptor file which describes the newly

added classes and a reference to the previous descriptor file the tokenisation was based upon (obviously a tokenisation that does not build upon an existing one will not need such a reference).

Figure 4.13 shows a situation where a device manufacturer has tokenised the API for a device and released the descriptor file that describes that API. Two other vendors have then created additional libraries (*Library1* and *Library2*) for the device, each one tokenising their library on top of the existing API. Each tokenisation process has produced a set of tokenised class files, as well as a descriptor file containing the metadata for the API classes and the respective library's classes. If yet another vendor now wishes to create an application (or another library) that will require both Library1 and Library2, they require a single descriptor file that can describe all of the API, Library1 and Library2, but such a file does not exist. If the tokeniser were to read more than one descriptor file, several problems would arise:

1. The API classes (and any other classes both tokenisations are based on) will be described in each file. Provided the API classes were identical for both tokenisations, this will not be an issue, however any change (such as a new version of the API was used for one tokenisation) would lead to conflicts with classes added/removed or methods within a class added/removed.

2. Since each tokenisation has been done independently, reuse of tokens is likely, especially class tokens.

3. If the the same files have been tokenised in each descriptor but at different levels (i.e. descriptor 1 results from tokenising: API -> Library1, and descriptor 2 results from tokenising: API -> Library2 -> Library1), then the tokens used for the same classes are likely to be radically different in each descriptor file.

These problems make it impossible to combine two or more descriptor files in their current form. So while a given descriptor file can have many incremental tokenisations performed against it, each tokenisation will create a new and unique descriptor file. Each new descriptor file can in turn have many new tokenisations performed using it as the base,

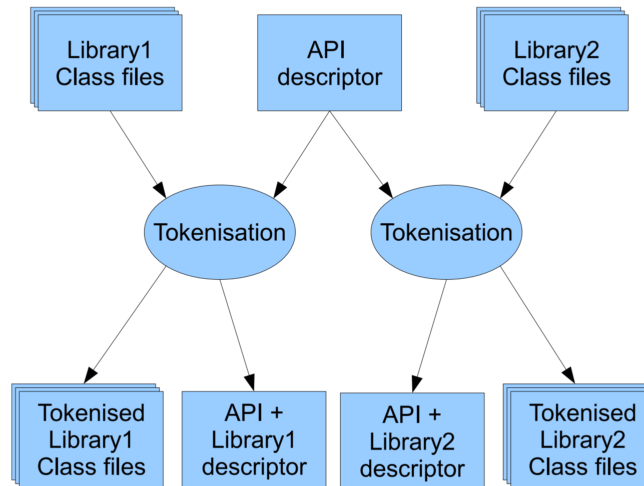


Figure 4.13: API with two custom libraries

and so on, however at no point can these various descriptor files be merged back together. In this fashion the descriptor files would form a tree-like structure, similar to class files, with each descriptor file able to extend only one other descriptor file.

The only way to overcome the problem presented above, where an application needs both Library1 and Library2 would be to acquire both libraries as standard class files (i.e. as a Jar file which most Java applications or libraries are distributed as), then tokenise both libraries in the one operation, extending the existing API tokenisation. Another approach is to tokenise Library2 on top of the API+Library1 tokenisation, or vice versa. Therefore, this limitation only affects cases where libraries are only distributed in tokenised form, if the compiled class files are available, then developers are free to tokenise the library as a part of their application, or as their own library package shared by several applications. The standard distribution mechanism for Java applications and libraries is a Jar file containing class files, and this is all that is needed for someone to tokenise that library and make use of it.

4.7.3 Binary Compatibility

In the previous chapter on global tokenisation, the tokenised files were all regenerated if any class files were changed (including addition or deletion of classes), meaning there were no additional binary compatibility constraints added by global tokenisation. Therefore,

binary compatibility was only broken in cases where the modifications also broke binary compatibility for the standard class files.

In incremental tokenisation however, the system consists of various collections of classes, each collection having been tokenised independently and most of them being built on top of a previous tokenisation. If a given package is retokenised, then any one of several changes could cause breakages:

1. Adding methods will result in the new methods using tokens which have probably also been used in the packages tokenised on top of this one, introducing conflicts that did not previously exist.

2. Removing methods is not possible as this breaks binary compatibility in standard class files, since other classes may try to call the now missing method. The same is true for classes.

3. Adding classes will result in the use of a class token that would have been used to identify classes in the packages tokenised on top of this one (since class tokens are allocated in increasing order), presenting an ambiguity about which class the token refers to.

4. Adding or removing non-static fields would result in all non-static fields after that one changing position within the object, making load/store references for those fields wrong.

While standard class files can have methods or classes added without breaking binary compatibility, the use of tokens prevents even that. It is still possible that the implementation of existing methods could be changed (i.e. to fix bugs), so long as no methods/classes/non-static fields are added or removed. However, the tokeniser would also need to know that such an update had occurred, and attempt to create a compatible package, but such functionality has not yet been implemented. Therefore, the present tokenisation approach has the limitation that a given package can only be used with the exact version of the package that it was tokenised against. If a library is retokenised, then all other packages that use that library must also be retokenised.

4.7.4 Static Methods

Section 3.6 covered static methods in the case of global tokenisation. In standard class files, a static method reference might actually be to the sub-class of where the target method is, to maintain the correct semantics if individual classes are changed and recompiled without recompiling the whole system. The classes that are not recompiled could otherwise be left referencing the incorrect method.

Global tokenisation changed these references to always point to the class that contained the target method. Since any changes to any of the classes required all classes to be retokenised, such references would be recomputed during the new tokenisation. Incremental tokenisation allows some packages to be tokenised without changing others. However, as the previous section discussed, if a given package is retokenised, then all packages that make use of it must also be retokenised, which will cause the tokeniser to resolve the new target method. Therefore, static methods can still be resolved to the target class, even with incremental tokenisation, without introducing any additional binary compatibility constraints.

4.8 Incremental Tokenisation Efficiency

A range of test cases were used to test incremental tokenisation. These used the same libraries as the global tokenisation tests, however combining them in different combinations of global and incremental tokenisation. The notation used for the test cases consists of package names (from those given in Section 3.9) and either a plus (+) or minus (-) sign when there is more than one package. A plus sign indicates the two packages were converted in a single step, while a minus sign indicates that the left side was tokenised first, then an incremental tokenisation performed to add the right side. In this notation the plus sign has higher precedence. For example, "CLDC1.0 - MIDP + MIDPExamples" would indicate that first the CLDC1.0 API was tokenised and saved. Then the the MIDP API and the MIDP example applications were combined and tokenised using the descriptor file produced from the first tokenisation.

The test cases consisted of[1]:

[1] Entries marked with a * are the same as the global tokenisation versions. These packages are included

- CLDC1.0*
- CLDC1.0 + MIDP*
- CLDC1.0 + MIDP - MIDPExamples
- CLDC1.0 - MIDP
- CLDC1.0 - MIDP + MIDPExamples
- CLDC1.0 - MIDP - MIDPExamples
- CLDC1.0 - Javolution3
- CLDC1.1*
- CLDC1.1 - Javolution3
- CLDC1.1M*
- CLDC1.1M - Javolution3
- CLDC1.1M - Javolution5

These where chosen to give a good blend of different test cases. The CLDC1.0 + MIDP case is the likely way a device would be shipped. Then the MIDPExamples are tokenised on top of this library, much like third party applications being added to a device. Also, the CLDC1.0 - MIDP - MIDPExamples tests multiple levels of incremental tokenisation.

4.8.1 Token allocation efficiency

The first thing to consider is if the number of tokens used has been significantly changed by incremental tokenisation. Table 4.1 shows the number of tokens that were used for the corresponding global case and the number used for the incremental case. It can be seen that in almost all cases, the number of tokens used was almost identical. In the case CLDC1.0 - Javolution3 test case, tokenisation of CLDC1.0 had allocated tokens up to 83,

---

here since they form the basis for the incremental tokenisations.

Table 4.1: Number of tokens used in incremental tokenisation

Test Case	Tokens Used For Global	Largest Method Token
CLDC1.0 + MIDP - MIDPExamples	116	115
CLDC1.0 - MIDP	101	101
CLDC1.0 - MIDP + MIDPExamples	116	116
CLDC1.0 - MIDP - MIDPExamples	116	115
CLDC1.0 - Javolution3	84	83
CLDC1.1 - Javolution3	68	68
CLDC1.1M - Javolution3	68	68
CLDC1.1M - Javolution5	74	74

while the incremental tokenisation of Javolution3 only required token values up to 68. The tokens for the new methods all used values that were within the range already used during the tokenisation of CLDC1.0. Also of note is that for several tests, incremental tokenisation used one less token than for global tokenisation. Examination revealed that during incremental tokenisation a conflict was found, hence requiring the use of a conflict entry. Since all class files are available during global tokenisation however, the conflict is detected and an additional token value used, avoiding the need for the conflict entry.

Apart from these minor differences, a similar number of tokens were used for incremental tokenisation as were for global tokenisation, showing that incremental tokenisation has not greatly affected the allocation of tokens.

4.8.2 Virtual Method Table Size

While the number of tokens used for incremental tokenisation was very similar to the number used in global tokenisation, the size of the resulting VMTs must also be considered. Figure 4.2 shows the number of VMT entries in each of the test cases. The VMT Entries column gives the number of entries for that tokenisation only (i.e. in a incremental tokenisation, only the VMT entries for the classes that were tokenised), while the Total VMT Entries column is for the sequence of conversions. For example, the CLDC1.0 - MIDP value of 8176 is the number of entries in MIDP (5997), plus the number in the CLDC1.0 library (in this case 2179). The final column shows the number of entries used for the cor-

Table 4.2: Comparison to Global Tokenisation

Test Case	VMT Entries	Total VMT Entries	Total Entries in Global
CLDC1.0 + MIDP - MIDPExamples	3652	11923	11998
CLDC1.0 - MIDP	5997	8176	8271
CLDC1.0 - MIDP + MIDPExamples	9724	11903	11998
CLDC1.0 - MIDP - MIDPExamples	3652	11828	11998
CLDC1.0 - Javolution3	6010	8189	7976
CLDC1.1 - Javolution3	5770	7257	7334
CLDC1.1M - Javolution3	5798	7286	7363
CLDC1.1M - Javolution5	7393	8881	8800

responding global tokenisation case. In all but two cases incremental tokenisation resulted in less VMT entries, representing more compact tables and therefore smaller files. In principle, the smaller tables represent a good result, however, only if a large number of conflict entries have not been added, and so the types of entries used is examined next.

Table 4.3 shows for each test case, the number of VMT entries actually used (i.e. ignoring null entries) and the number of conflict entries. While the slightly lower number of entries in total compared to global tokenisation is the result of conflict entries, the number of conflict entries is quite low, especially when compared to the number of classes and methods in each test case. Therefore the need for conflict entries in incremental tokenisation is small, adding a very small overhead when compared to global tokenisation. Finally, Table 4.4 shows a comparison for the percent of VMT entries that were non-null after incremental tokenisation compared to global tokenisation. While there is a small amount of variance, as a result of the addition of conflict entries, and the slight changes in token usage, there is no significant change in the efficiency of token allocation.

Not only is incremental tokenisation possible, but the efficiency of token allocation and size of virtual method tables are hardly affected. There is a slight overhead in having to deal with conflict entries, however, there were very few of these, for the codebase that was tokenised. In return packages no longer need to be tokenised on a device, reducing the memory and processing requirements needed in the device.

Table 4.3: Usage of Virtual Method Tables with Incremental Tokenisation

Test Case	VMT Entries	Used VMT Entries	Conflict VMT Entries
CLDC1.0 + MIDP - MIDPExamples	3652	2876	12
CLDC1.0 - MIDP	5997	5535	2
CLDC1.0 - MIDP + MIDPExamples	9724	8411	2
CLDC1.0 - MIDP - MIDPExamples	3652	2876	12
CLDC1.0 - Javolution3	6010	5540	1
CLDC1.1 - Javolution3	5770	5552	4
CLDC1.1M - Javolution3	5798	5555	4

Table 4.4: VMT Usage Compared to Global Tokenisation

Test Case	Used % (Inc.)	Used % (Global)
CLDC1.0 + MIDP - MIDPExamples	88.65	88.10
CLDC1.0 - MIDP	94.1	93.02
CLDC1.0 - MIDP + MIDPExamples	88.8	88.10
CLDC1.0 - MIDP - MIDPExamples	89.36	88.10
CLDC1.0 - Javolution3	94.02	96.43
CLDC1.1 - Javolution3	96.91	95.87
CLDC1.1M - Javolution3	96.58	95.55
CLDC1.1M - Javolution5	95.65	96.53

4.9 Conclusions

The previous chapter showed that it was possible to tokenise methods even in the presence of interfaces, thus simplifying the implementation of the *invokeinterface* and *invokevirtual* instructions. However, the global tokenisation required complete knowledge of all classes and interfaces in the system to be able to correctly allocate tokens. This chapter has presented a modified tokenisation scheme so that it can be performed in incremental steps, with each tokenisation adding more classes/interfaces to an existing set, without needing to change any of the already tokenised classes. Since the tokeniser can not know what future additions might be made, some ambiguities can arise, but these situations can be resolved with minimal overhead.

Furthermore, the incremental tokenisation does not reduce the efficiency of produced virtual method tables, with incrementally built tables being of similar size and composition to those generated with global tokenisation. The need for conflict entries in the virtual method tables was also minimal, with a worst case of only 0.42% of the used entries being conflict entries.

With the addition of incremental tokenisation, the tokeniser is no longer required on the device, instead the API or libraries are installed on the device and a descriptor file, which describes those classes already on the device, is provided to developers. Developers can then tokenise their application in such a way that they can use the existing classes on the device, without needing to modify the existing classes. After a developer has tokenised the application, it is then distributed to the device in tokenised form. Since the tokeniser will be run on standard PC level hardware (i.e. on a developers workstation), rather than on the mobile device, the tokeniser does not need to deal with the processor/memory restrictions of a mobile device. So the requirements for the mobile device are reduced, while still providing the performance of virtual method table based dispatch for the mobile device.

The following chapter will cover the generation and format of tokenised class files and the compression that is gained through tokenisation. After that, the implementation of a virtual machine capable of executing tokenised classed files is presented, followed by an analysis of results and final conclusions.

Chapter 5

Implementation of a VM

5.1 Introduction

To test the tokenisation, a Virtual Machine was implemented that could understand the tokenised class file format. This allowed tokenised code to be executed directly and prove that the tokenised programs were still correct. The goal for this virtual machine was that it:

- Implement a functional sub-set of Java instructions
- Load and use the Virtual Method Tables and other features of tokenised class files
- Support at least the CLDC library and applications written to use CLDC

The KVM [62] was considered for this purpose as it already provides an implementation of a CLDC virtual machine. However, it is a production level VM and incorporates many optimisations which are not required for testing, and thus complicate the process of modifying it to use tokenised class files. Therefore, it was easier to start from scratch and code a simplified VM in Java to use for testing.

Performance or absolute completeness of the instruction set was not considered important. To this end, the virtual machine lacks some features, such as multithreading or the synchronisation mechanisms for threads. Support for MIDP is not provided, as this would require implementation of the native methods to implement a GUI. The lack of MIDP support limits the applications that can be executed, but allowed for a quicker development of the virtual machine. Since the test VM was being implemented in Java, many features of the

underlying host VM (such as garbage collection) were already present, again simplifying the implementation.

Initially, a virtual machine was developed to load, link and execute standard Java class files and was used to debug the interpreter loop, instruction set and to ensure a usable VM design before proceeding. A branch was then made from the existing codebase to develop a tokenised VM, capable of loading, linking and executing tokenised class files. To support tokenised files, changes had to be made to the class loader, objects and some of the instruction set.

The class loader had to be modified to identify classes via their class token, instead of the usual string name (which also allowed a simplified lookup table for classes, via their class token). Objects were also modified so that each object contained a class token, to indicate which class it was an object of. When combined with the class table, this allowed efficient access from an object to the class structure, which is needed when performing some operations on objects (such as checking if an object can be cast to another type).

The following sections detail the implementation of the virtual method table, changes that were made to Java instruction set, the handling of native methods, constant values required by the VM and finally the correctness and efficiency of the tokenised virtual machine.

5.2 Virtual Method Table Format

The Virtual Method Table is used at runtime to find the method structure that represents the target method of a method call. A method call consists of a method token, which is an index into the Virtual Method Table, where the entry is a reference to the method to call. Since only concrete classes can be used to create objects, and virtual methods are methods within an object, only concrete classes require virtual method tables. While methods in abstract classes might be the target of a virtual method call, the object for the call will have been created from a concrete class, and therefore, the concrete class's virtual method table will point to the target method.

At present an additional step is needed during class loading, since the binary class for-

mat still uses a form of symbolic reference in the virtual method table, which is a side-effect of having classes stored in individual class files. Even if all the classes from a given tokenisation run were placed into a single output file, there can still be references to methods from previous tokenisations, preventing absolute addresses from being used. The test VM performs this linking at class load time, however, a device should perform this linking during installation. Once the installer has placed the classes into memory, the location of all methods will be knowable and the a final VMT can be constructed with absolute memory addresses.

5.2.1 In Memory Representation

There are two types of entries that can be in a virtual method table, as discussed in Section 4.6, a simple entry which points to a single target method and a multi-entry which can have several target methods, depending on the type of call. A simple entry just requires an offset to the appropriate method structure for the target method. A multi entry is more complex, containing a default method reference and a list of class token/method reference pairs, giving a multi entry a variable length. The test VM represents VMT entries via objects and therefore the VMT itself is an array of objects. While an object array allows a token value to be used as an index, it adds a level of indirection, as the entry in the array is actually a reference to the object, stored elsewhere in memory.

On a device, it is more likely to store the contents of each entry in an array structure, removing the need for the object references. However, the variable size of conflict entries will remove the ability to quickly calculate the location of a given index, therefore fixed size entries would be preferred. Making the entries large enough to hold the largest conflict entry will result in extremely large and mostly empty tables. Therefore, the virtual method table would include fixed sized entries, with each entry consisting of either a method reference (for single entries) or a reference to a conflict entry, adding one extra layer of redirection for the rare case of a conflict entry.

5.2.2 Alternate VMT Encodings

One way to reduce the sizes of virtual method tables is to not include the full table in every class. For example, Java Card reduces the size of dispatch tables by not including the first part of the table that has not changed (this is discussed in depth in Section 2.5.2). Four possible table encodings were considered:

Complete Tables Every class contains the complete virtual method table for all methods it can call. These produce the largest tables, but only the one table will need to be consulted during method dispatch.

Java Card Tables Using the same approach as Java Card, any entries at the start of the table that have not changed from the parent class will not be included. One extra field is needed to indicate the index the current table corresponds to (i.e. the table's base value). If a target token is too small for a given table, then the parent class's table needs to be consulted, adding to the dispatch time.

Java Card Tables (ignoring default constructor) A default constructor is found in most classes, meaning it is often over-ridden and also has a very small token value (it is the first method in the java.lang.Object class, where token assignment starts from, resulting in the default constructor being assigned token value 0). If index 0 has changed in the VMT, then the Java Card approach requires the complete table to be included in the class. If the default constructor was handled in some different fashion, then it can be ignored when determining the smallest changed token value.

Aggressive Java Card Tables The previous two approaches reduce the size of the VMT by leaving off the start of the table that has not changed from the parent class's table, because in Java Card, the start is the only part that can be unchanged. However, with the addition of interfaces, it is possible for a super-class to have had some larger token values that do not change in the sub-class. Aggressive Java Card Tables therefore considers the end of the table, leaving any entries off the end that have not changed from the super-class's table.

An example is presented in Figure 5.1 showing three classes and the token values used in each class. Class A contains two methods with token values 0 and 1, giving a table size of 2. Class B extends Class A and contains four methods which have, due to conflicts not shown, been assigned the token values 0, 5, 6 and 7, leaving values 2, 3 and 4 unused and giving a table size of 8. Finally class C adds methods 2, 3 and 4, while also over-riding 5 and 6, also giving a table size of 8. The white squares denote a VMT entry that was used in the super-class but not overridden, for example, entry 1 in the C class would still point back to the method in class A.

Figure 5.1 is shown again four times, for each of the four encoding schemes presented above, in Figure 5.2. Part (a) shows the complete tables, with a total of 18 entries (and is unchanged from the initial figure). Part (b) shows the Java Card style tables, where Class C has 2 less entries at the start of the its table, since they have not changed from Class B's table. While this saves 2 entries, three more would need to be added to store the base value for each of the three tables (although in practise, with more classes, the overall effect is a reduction in size). Part (c) shows a larger improvement, since token 0 (the default constructor) is being handled specially, Class B can ignore it and only needs to start its table at token 5. The overall size is therefore reduced to 11 entries, plus 3 for the base values, giving a total of 14 entries. The final variation in Part (d) shows the Aggressive Java Card Tables, where the unchanged entry at the end of Class C's table can also be left off, reducing the size by a further one entry.

While none of these alternate encodings were implemented in the virtual machine, they were examined to see how they would affect the size of the virtual method tables and what extra steps would be needed to implement them. Results on the effect of these alternate tables are presented in the next section.

5.2.3 Size of Alternate Virtual Method Tables

In the previous section, several different approaches to encoding the virtual method tables was given, which allowed for a given table to have less entries, at the expense of having to search in a super-class's table if the appropriate entry was not found. These encodings

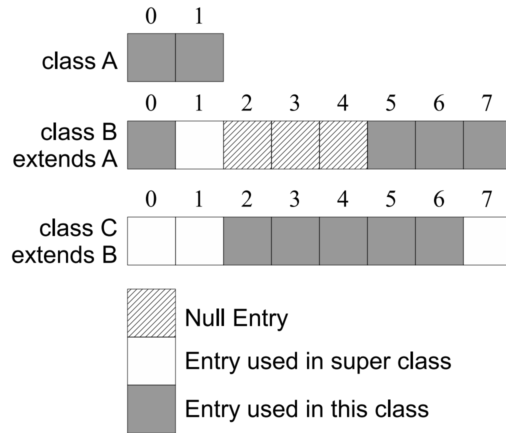


Figure 5.1: Example of VMT usage

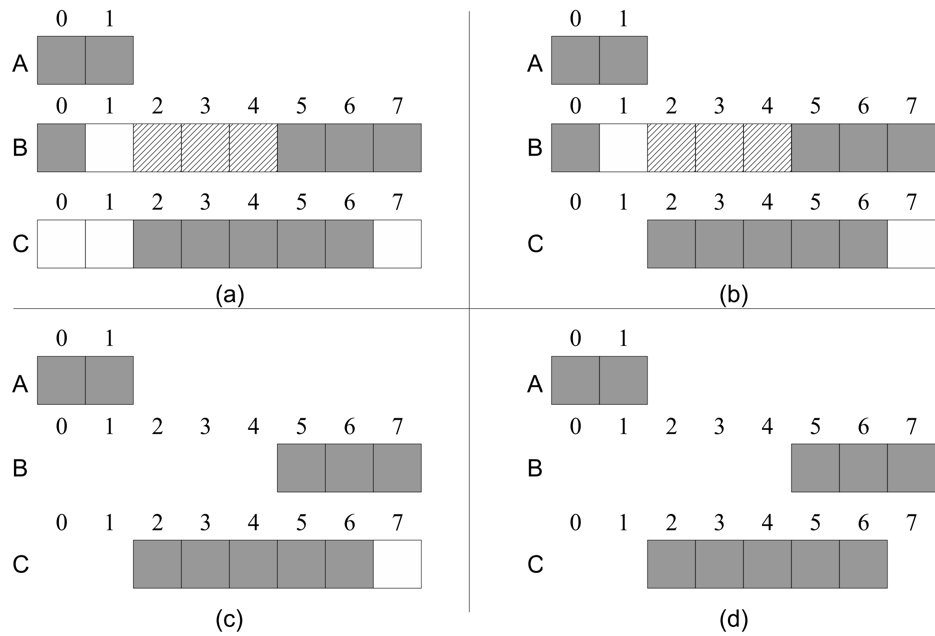


Figure 5.2: Size of Different Types of VMTs

present a time vs space trade-off, between the complexity of the lookup process and the size of the tables.

To know if these trade-offs would be worthwhile, analysis of the test packages originally presented in Section 3.9 was done to determine how much space could be saved. For the virtual method table in a given class, there are four values that are important for determining the size of the virtual method table, these are:

- a is the smallest token used by a method in the current class.
- b is the smallest token used by a method other than the default constructor, in the current class.
- c is the largest token used by a method in the current class.
- d is the largest token used in this or any of the super-classes.

For the four possible encodings presented in Section 5.2.2, the starting and ending values of the virtual method table would be:

Name	Starting Value	Ending Value	Uses super-class's Tables
Complete Tables	0	d	no
Java Card Tables	a	d	yes
Java Card Tables (default constructor)	b	d	yes
Aggressive Java Card Tables	b	c	yes

The final column in the table denotes how many of the tables need to be searched at runtime. For Complete Tables, only the one table ever needs to be searched, if no result is found, then an error is produced. For all the others, if no result is found in the first table, the search must be performed in the super-classes table, continuing recursively until either the result is found, or there are no more super-classes, in which case an error is produced.

Table 5.1 was produced by analysing the test packages to determine the above values for each class. The use of Java Card Tables provides only a small saving in the number of

Table 5.1: Number of Virtual Method Table Entries

Test Case	Complete Tables	Java Card Tables	JC Tables (ignoring def. constructor)	Aggressive JC Tables
CLDC1.0	2159	1983	1172	995
CLDC1.0+MIDP	8271	6861	4753	4376
CLDC1.0+MIDP+ MIDPExamples	11998	9567	6955	6501
CLDC1.0+ Javolution3	7976	6578	4071	3408
CLDC1.1	1487	1370	808	643
CLDC1.1M	1488	1371	809	644
CLDC1.1M+ Javolution5	8800	7060	4255	3426
J2SE	586989	469359	369646	333810

entries because overriding of the default constructor is common, and any class that does this will require the complete virtual method table. If the default constructor is ignored however, then a significant drop is noticed in the number of entries used, but with the added complexity of needing to handle the default constructor separately. Moving to the Aggressive Java Card tables only has a small increase over the Java Card Tables ignoring the default constructor.

Moving from left to right in Table 5.1 shows a decreasing number of entries being used, therefore saving space, at the cost of increasing complexity to implement. While the available space savings have been examined, the most appropriate encoding will depend on the target device and the relative cost/performance trade-offs for that device.

5.3 Instruction Set Modifications

This section covers modifications that have been made to the Java bytecode instruction set (or simply just, “bytecodes”). All Java code is compiled to bytecode, which is the instruction set for a virtual processor, typically implemented in software as a virtual machine. These modifications have happened for one of two reasons, firstly, as a direct result of tokenisation and secondly, to provide shorter versions of some instructions.

The changes are only to the *ldc** instructions for loading constant values from the con-

stant pool, *get* and *put* instructions for loading/storing values in fields and the *invoke** instructions for performing method calls. The following sections detail the changes to each of these instructions.

5.3.1 ldc, ldc_w and ldc2_w

These three instructions load a constant value from the constant pool. For the *ldc* and *ldc_w* instructions, the value will be one of: integer (CONSTANT_Integer), float (CONSTANT_Float) or string (CONSTANT_String), while the *ldc2_w* instruction will load a double width constant value, i.e. a long (CONSTANT_Long) or double (CONSTANT_Double). The "_w" at the end of the instruction indicates it is a wide instruction, i.e. it has a two byte operand, which serves as an unsigned index into the constant pool, while the *ldc* instruction only has a one byte operand. The *ldc2_w* instruction does not have a "narrow" version of the instruction, unlike the *ldc_w* instruction. The standard formats for these instructions are:

ldc:	18	*index*	
ldc_w:	19	*indexbyte1*	*indexbyte2*
ldc2_w:	20	*indexbyte1*	*indexbyte2*

The only change to these instructions is in the case of a reference to a CONSTANT_String entry. Since a CONSTANT_String entry contains an index to a CONSTANT_Utf8 entry for the string data, the instruction will be updated to point directly to the CONSTANT_Utf8 entry, removing the need for the CONSTANT_String entry and saving 3 bytes for each such entry. While the instructions will still have the same form, tokenisation greatly reduces the number of constant pool entries, causing the indexes for the remaining entries to be smaller. In some cases where an *ldc_w* instruction was used, it will now be possible to use an *ldc* instruction, thereby shrinking the code. Section 6.5.2.3 details the steps needed to update the bytecodes, as shortening or lengthening instructions is not a trivial matter.

The tokeniser leaves *ldc** instructions with references to the constant pool, which therefore must be resolved by a tokenised virtual machine, representing a layer of indirection during execution. The indirection is not helped by the fact that constant pool entries are

of a variable size, preventing a simple calculation to find a given index, as can be done with fixed sized entries. Possible alternatives to the current indirection and having to find a variable sized item in the constant pool include:

Lookup table The constant pool reference could be used as an index to a table of fixed-size memory references, giving the location of each constant pool entry and avoiding the need to find an entry in a list of variable sized entries.

Separate constant tables Placing the 32-bit and 64-bit constant values into their own tables with fixed sized entries, allowing the index to be used to directly calculate the location of a required entry.

Inlining The constant value itself could be placed within the instruction as an operand, at the cost of making the instruction larger. Depending on how many places use the same constant value, this could make the overall size smaller or larger for a class.

The pros and cons of each of these approaches would be highly dependent on the characteristics of the hardware being used.

5.3.2 getstatic/putstatic

The *getstatic* and *putstatic* instructions are used to read values from, or store values into, static fields. In standard class files these instructions have the form:

178/179	*indexbyte1*	*indexbyte2*

The first byte value is 178 for *getstatic* and 179 for *putstatic*. The two *indexbyte* values are combined to form an unsigned 16-bit value, which is an index into the constant pool to a CONSTANT_Fieldref structure which provides the symbolic field reference. Symbolic field references were described in Section 1.4.4.2.

A CONSTANT_Fieldref entry normally contains two indexes to a CONSTANT_NameAndType and a CONSTANT_Class entry, which in turn contain references to three CONSTANT_Utf8 entries for the class name, field name and field type. Given the tokenisation of fields (described in Section 3.8), the utf8 string entries are no

longer needed to reference a field, instead being replaced with two tokens, a class token and a field token, each consisting of a 16-bit value. Given the relatively small and fixed sized nature of the tokens, it would be wasteful to store them in the constant pool. Even if the instructions were updated to point to a constant pool entry which contained both tokens, that would require a 16-bit entry for the constant pool reference and an 8-bit entry for the tag indicating the type of constant pool entry, to reference 32-bits worth of data, which is a 75% overhead. Given that, the instructions were updated to hold the two tokens as operands, which removed the need for indirect constant pool references and simplified the implementation of the instruction in the virtual machine.

Instructions such as the *ldc* instruction described in the previous section have a standard form with an 8-bit constant pool reference and a wide form with a 16-bit constant pool reference. The same approach is applied to the class and field tokens, providing shorter instructions when only 8-bit tokens are required, but also supplying wider forms so the full 16-bit token values can still be used. To examine the impact of this, the same libraries that were used for testing earlier (introduced in Section 3.9) were examined to see what sized instructions would be required. There are two token values (class token and field token) and two possible sizes for each token (8-bit or 16-bit), giving four possible combinations. A given combination is denoted by two numbers, first giving the class token size, then the field token size, i.e. 16,16 is a 16-bit class token and 16-bit field token, 16,8 is a 16-bit class token and an 8-bit field token and so on. Table 5.2 shows how many times each of the four lengths was needed for both the *putstatic* and *getstatic* instructions.

The results show that in all but a few cases an 8-bit field token is sufficient, only needing a 16-bit field token for the relatively large J2SE API, while 16-bit class tokens are needed much more often. For static fields, the field token is only unique within a single class, meaning the maximum size token will depend on the maximum number of static fields within a single class. Further examination showed that for the J2SE test case, the largest static field token was 426 in one class, with the next largest was only 54. With so few uses of 16-bit tokens, it would seem that many of these fields were not used, however examination of the class with 426 static fields showed that most of them were declared *final*. When code references a final static field, the compiler adds that value into the class's constant pool and

Table 5.2: Usage of *putstatic* and *getstatic* instructions

	putstatic				*getstatic*			
Package	16,16	8,16	16,8	8,8	16,16	8,16	16,8	8,8
CLDC 1.0	0	0	0	60	0	0	0	112
CLDC 1.1M	0	0	0	56	0	0	0	150
MIDP	0	0	81	149	0	0	248	737
MIDP examples	0	0	95	0	0	0	247	29
Javolution 3	0	0	44	121	0	0	88	322
Javolution 5	0	0	91	134	0	0	149	293
J2SE	2	0	9466	334	8	0	25404	4118

uses an *ldc** instruction to load it, rather than loading it from the field[1].

The results in Table 5.2 indicate that an 8,8 sized instruction would see common use (and is the same size as the standard instruction), while 16,8 sized instructions would also be required. The 8,16 sized instruction is redundant and would only complicate the instruction decoder with an additional instruction. For maximum compatibility, the 16,16 instruction should also be available so as not to impose any additional constraints on developers.

With the above results in mind, the following tokenised instructions were defined for the *getstatic* instruction:

178	*ctok*	*ftok*

213	*ctokH*	*ctokL*	*ftok*

215	*ctokH*	*ctokL*	*ftokH*	*ftokL*

For *putstatic* the forms are:

179	*ctok*	*ftok*

214	*ctokH*	*ctokL*	*ftok*

216	*ctokH*	*ctokL*	*ftokH*	*ftokL*

Each box represents a single byte, where *ctok* is a class token and *ftok* a field token. In the case of a 16-bit token, there is a high and low value (i.e. *ctokH* and *ctokL* for a class

[1]There is the case of small values, where the *iconst_** or similar instructions can be used to push the value directly, without the need for a constant pool reference.

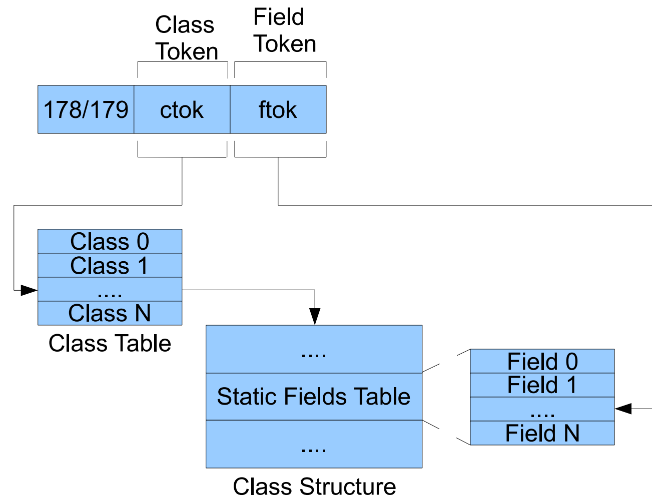


Figure 5.3: get/put-static instruction implementation

token), which if read together form the 16-bit token. In terms of space, the first instruction is the same size as the standard version, while the second instruction is one byte longer and the third two bytes longer. However, the need to load entries from the constant pool has been removed, allowing those entries to be removed from the constant pool and also significantly simplifying the implementation of the instructions.

Figure 5.3 shows how the tokens are used at runtime by the virtual machine. The initial byte is either 178 for *getstatic* or 179 for *putstatic*. The next byte provides the unsigned class token and the final byte the unsigned field token. The class token is an index into the class table which provides the structure to describe that class, including the table of static field values. The field token value is an offset into the static fields table to find the field in question. The only difference between the two operations is that *getstatic* will load the value from the field and push it onto the stack, while *putstatic* will pop a value off the stack and store it into the field. The wider forms of the instruction (with opcode 213-216) behave exactly the same way, just with a 16-bit class and/or field token value.

5.3.3 getfield/putfield

The *getfield* and *putfield* instructions are similar to the *getstatic* and *putstatic* instructions, in that they load and store values in a field that is part of an object, instead of a static field. The standard instructions have the format:

180/181	*indexbyte1*	*indexbyte2*

The *getfield* instruction for loading values has the byte value 180, while the *putfield* instruction for storing values has the value 181. The next two bytes form an index into the constant pool to a CONSTANT_FieldRef entry, the same as for the static field instructions, described in Section 5.3.2, except resolving the field reference in this case will give a non-static field. The operand stack will contain an object reference, whose type must be the same as, or a sub type of, the class in the field reference. For *getfield*, the object reference is popped from the stack, the value from the named field is read and pushed back onto the stack. For *putfield*, a value and an object reference is popped from the stack, with the value stored into the named field in the object.

These non-static fields were tokenised such that for all the fields in a given object, each field has a unique token (originally discussed in Section 3.8), allowing the field token to be used as an offset within an object. The static field instructions required a class reference to identify which class contained the field, however, since an object reference will always be on the stack, a class reference is not needed by these non-static instructions.

As with the static field instructions above, some consideration was given to the required size of the field token, either 16-bits or 8-bits. For static field tokens, the maximum required size was dependant on the number of static fields within the one class. For non-static fields, the token size will depend on the number of fields in an object, which consists of the non-static fields in the class, plus those in all super-classes.

The test libraries introduced in Section 3.9 were analysed to see how often a 8 or 16-bit field tokens were needed. Table 5.3 shows the results, with even the relatively large J2SE API never having an object with more than 256 fields. In particular, the largest field token that was required was 149. Therefore, an instruction with a 16-bit operand would result in unnecessary complexity, as an instruction with only an 8-bit operand is sufficient for all the test cases. For the rest of this discussion, an 8-bit operand version is considered, however, a 16-bit instruction could be added easily if support was needed for larger objects.

Since an 8-bit operand is sufficient, the tokenised instruction has the form:

180/181	*ftok*

Table 5.3: Usage of *putfield* and *getfield* instructions

	putfield		*getfield*	
Package	16	8	16	8
CLDC 1.0	0	454	0	941
CLDC 1.1M	0	271	0	877
MIDP	0	2128	0	6845
MIDP examples	0	864	0	3086
Javolution 3	0	1099	0	2883
Javolution 5	0	1127	0	3194
J2SE	0	48875	0	128007

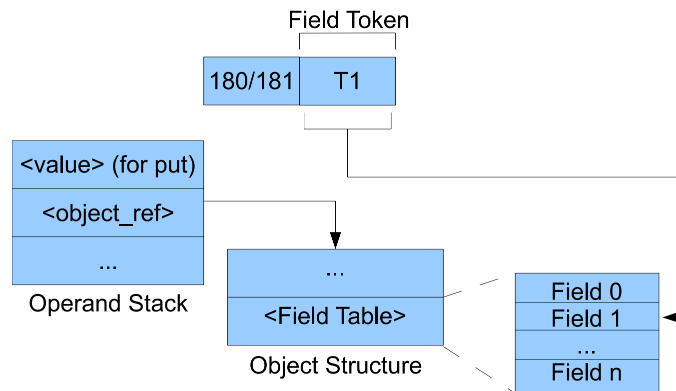


Figure 5.4: get/put-field instruction implementation

The single operand gives the field token. Figure 5.4 shows the implementation for these instructions. The object reference is popped from the operand stack (for a *putfield* instruction the value is popped first), which gives access to the object structure in memory, including the table of fields. The field token from the instruction can then be used to find the appropriate entry in the field table, which is the location where the value of that field is stored and is therefore read or written depending on the instruction.

The tokenised field instructions greatly reduce the complexity required to implement the instruction, while also removing the need for the symbolic linking data in the constant pool and made the instruction one byte shorter. Even if a wider instruction was added, to allow for objects with a large number of fields, a significant space saving is realised through the removal of the constant pool entries and in the shorter instruction.

5.3.4 invokevirtual

An invoke virtual instruction in a standard class file consists of three bytes:

182	*b1*	*b2*

The first byte has the value 182 to indicate this is an invokevirtual instruction, while the b1 and b2 bytes are used as an unsigned 2-byte index to the constant pool to find the method to be invoked. Since method references are now made using a token, the data in the constant pool, and therefore also the reference, are no longer needed. The constant pool reference is replaced by the method token for the target method and a *nargs* values, giving the number of arguments for the method.

When a method call is performed, the object for the call is pushed onto the stack, followed by any arguments. To dispatch the method call, the VM needs to access the object reference and therefore needs to know how many arguments there are. In a standard class file, the number of arguments is encoded in the descriptor string for the method. Since the strings have been removed during tokenisation, the *nargs* value allows the VM to know how many of the values on the stack are needed for the method call.

A method token is a 16-bit value, however, the full 16-bit value will not always be needed, as small token values could be represented with an 8-bit operand, leading to two *invokevirtual* instructions, one for an 8-bit token and one for a 16-bit token. Having additional instructions requires a more complicated instruction decoder, but offers shorter instructions and therefore more compact code. Using the test libraries introduced in Section 3.9, each instance of an *invokevirtual* instruction was examined to see if it needed a 16- or 8-bit token. Table 5.6 shows the results for each package. In all but the relatively large J2SE test case an 8-bit operand was sufficient to hold every method token used in an *invokevirtual* instruction. The presence of an 8-bit *invokevirtual* instruction is very important, as it reduces the size of compiled code. Depending on the target device, the need for two instructions can be avoided if the *invokevirtual* instruction is limited to only an 8-bit version, at the cost of limiting the possible size of the codebase.

For the purposes of testing, both the 16-bit and 8-bit versions of the instruction were defined with the following forms:

Table 5.4: Usage of the *invokevirtual* instruction

Package	16-bit	8-bit
CLDC 1.0	0	997
CLDC 1.1M	0	681
MIDP	0	4631
MIDP examples	0	2928
Javolution 3	0	2594
Javolution 5	0	3299
J2SE	13196	160957

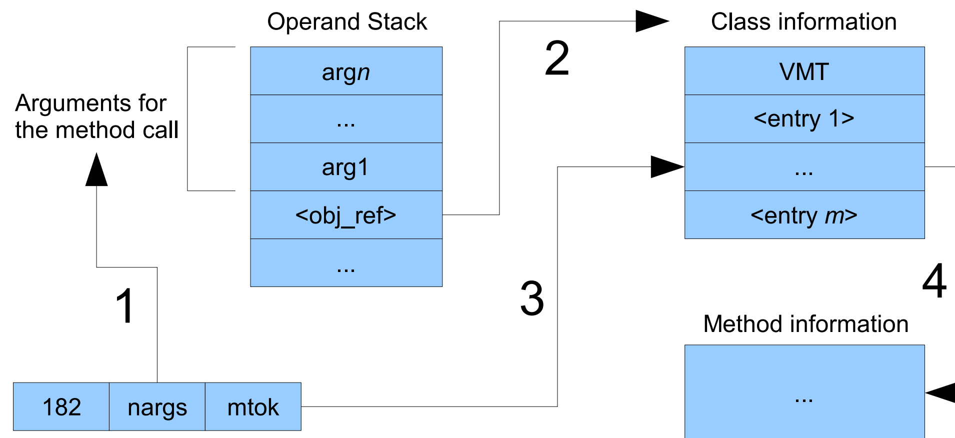


Figure 5.5: Performing an *invokevirtual*

182	*nargs*	*mtok*

206	*nargs*	*mtokH*	*mtokL*

Regardless of the size of the method token operand, the process needed to execute an *invokevirtual* instruction is the same and is shown in Figure 5.5. The numbered lines in the diagram correspond to the following steps:

1. The *nargs* value is used to find the object reference on the stack that this method is being called on.

2. The object reference contains an index to the class information for that type of object, which, in turn, contains the Virtual Method Table for that class.

3. The *mtok* value is used as an offset into the VMT. If this entry is a simple entry, then the value in the entry will be the method reference. If it is a multi entry, then the default value will be the method reference, since this is an *invokevirtual*.

4. The method reference provides the location of the information required for execution of this method, including: the bytecode, maximum size of the stack, maximum number of local variables and information on exception handlers. Using this information and the *nargs* value from step 1, the VM can create a new frame for the method and transfer execution.

5.3.5 invokeinterface

The *invokeinterface* instruction is used when the declared type in the source code is an interface type. In a standard class file, this instruction has the form:

185	*b1*	*b2*	*count*	0

Where 185 is the byte value to indicate an *invokevirtual* instruction, the *b1* and *b2* bytes form an unsigned 2-byte value index into the constant-pool, the *count* is the number of arguments to the method (which can also be derived from the descriptor string) and the final byte is always 0[2]. The constant pool reference gives (via references to other constant pool entries) the name of the interface, the method name and method descriptor.

A tokenised *invokeinterface* instruction needs at least as much information as an *invokevirtual*, that is, a *nargs* value and a method token. While this information is enough to dispatch most *invokeinterface* instructions, if a conflict entry is found, then the virtual machine also requires the class token of the interface (conflict entries were introduced in Section 4.3.2). Therefore, the tokenised *invokeinterface* will require three values: *nargs*, *ctok* and *mtok* for the number of arguments, class token and method token respectively. Just as with *invokevirtual*, the test packages were examined to see the required size for each token. The size of the instruction is given by two numbers, the size of the class token (either 8 or 16 bits), and the size of the method token (either 8 or 16 bits). Table 5.5 shows the results of tokenising the test classes. Interestingly, the 8,16 and 16,8 instructions were never needed, implying that these can be left out to simplify the instruction decoder. The 8,8 instruction gives a compact instruction (being one byte shorter than the standard instruction) and is sufficient for nearly all instances. The 16,16 instruction was needed for

[2]The Java Virtual Machine Specification [57] notes that the redundant *count* byte and final 0 byte remain for historical and compatibility reasons.

Table 5.5: Usage of *invokeinterface* instruction

Package	16,16	8,16	16,8	8,8
CLDC 1.0	0	0	0	29
CLDC 1.1M	0	0	0	14
MIDP	0	0	0	297
MIDP examples	0	0	0	205
Javolution 3	0	0	0	347
Javolution 5	0	0	0	300
J2SE	338	0	0	6301

the J2SE test case, so was maintained for compatibility, giving the following two forms for the tokenised version of *invokeinterface*:

185	*nargs*	*ctok*	*mtok*

212	*nargs*	*ctokH*	*ctokL*	*mtokH*	*mtokL*

To dispatch the *invokeinterface* instruction, the initial lookup process is the same as steps 1-3 for an *invokevirtual* instruction, as shown in Figure 5.5. However step 4 becomes slightly more complicated with one of 3 possible outcomes:

1. The VMT entry is a simple one, in which case the method reference is used, same as an invokevirtual.

2. The VMT entry is a multi entry, however there is no associated value for the *ctok* from the instruction. In which case the default method reference is used.

3. The VMT entry is a multi entry and there is an associated value for the *ctok* from the instruction. In this case the associated method reference is used, and the method call continues.

For case 1, the process will be the same as shown in Figure 5.5 and the *ctok* value in the instruction will not be used. Figure 5.6 shows the result for cases 2 or 3 above, where the VMT entry is not just a simple entry. Here the list of class token values in the VMT entry are searched and if an entry has the same value as the *ctok* value, then the method reference from that entry is used. If no matching entries can be found, then the default entry is used.

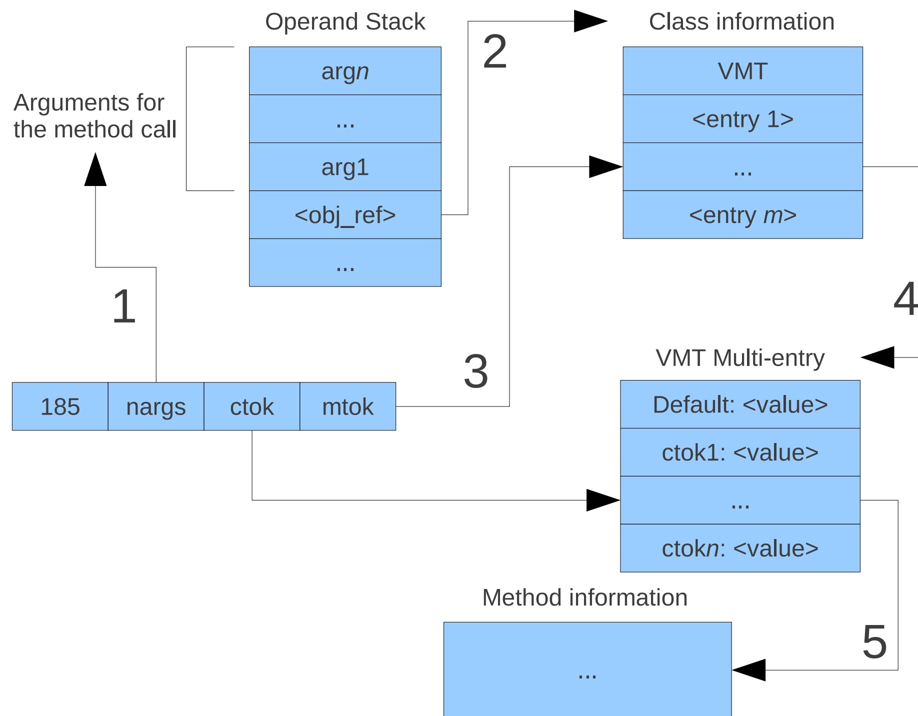


Figure 5.6: Performing an *invokeinterface*

5.3.6 invokestatic

An *invokestatic* instruction is used for calling methods that have been declared as STATIC in the source code. A standard class file uses an instruction with the form:

184	*b1*	*b2*

An *invokestatic* instruction, like the other invoke* instructions, uses two bytes to form an unsigned 2-byte value that gives an index in the constant pool, which provides the usual three strings for class name, method name and method descriptor.

Section 3.6 discussed static methods in tokenised class files. Because the converter will resolve a static method call to the class that contains the target method, the tokens for static methods only need to be unique within the class that contains the method. Therefore, a class token and method token are sufficient to uniquely identify any target method in the system. The same test libraries (introduced in Section 3.9) were used to again see how large the class and method token values should be.

Table 5.6: Usage of the *invokestatic* instruction

Package	16,16	8,16	16,8	8,8
CLDC 1.0	0	0	0	247
CLDC 1.1M	0	0	0	286
MIDP	0	0	413	1015
MIDP examples	0	0	247	151
Javolution 3	0	0	215	1139
Javolution 5	0	0	392	792
J2SE	0	0	25940	9420

Figure 5.6 shows how often each sized token was needed. While 16 bit class tokens were common, 16 bit method tokens were never required, with the largest used token being 115, used in the J2SE API. A limit of 255 static methods in any one class seems to be a suitable limit, although to maintain flexibility a 16,16 version of the instruction has also been defined.

An *invokestatic* instruction requires the same information as an *invokeinterface* instruction, namely, *nargs*, *ctok* and *mtok*. Allowing for different lengths of the *mtok* and *ctok* values, gave three forms for a tokenised *invokestatic* instruction:

184	*nargs*	*ctok*	*mtok*

204	*nargs*	*ctokH*	*ctokL*	*mtok*

211	*nargs*	*ctokH*	*ctokL*	*mtokH*	*mtokL*

Unlike virtual or interface methods, STATIC method do not require runtime binding of the target method, therefore the information contained in an *invokestatic* instruction is sufficient to resolve the call. The *nargs* value is required to know how many of the values currently on the stack are arguments to the method. In standard class files, the number of arguments was available by parsing the descriptor string for the method. The *ctok* value (or in the case of the second version, the *ctokH* and *ctokL*, which are combined to form a single 16-bit value), is used to find the class that will contain the method. While the *mtok* value is then used to find the relevant method within that class. Figure 5.7 shows the steps involved, which consist of:

1. Use the *nargs* value to collect the arguments for the method call.

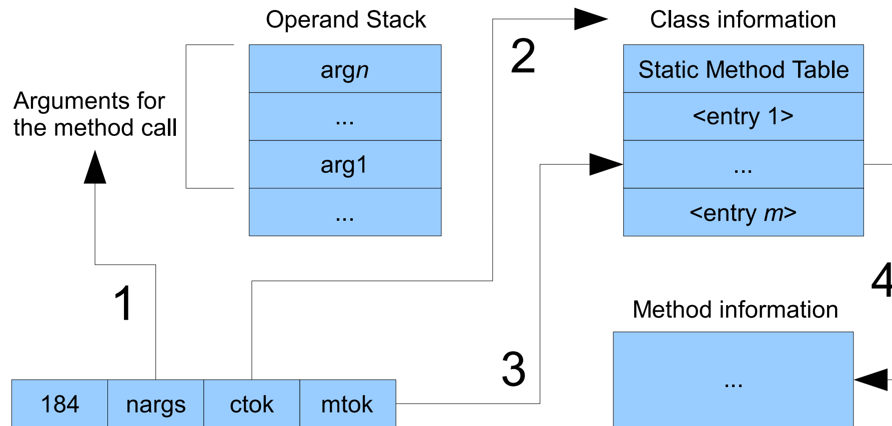


Figure 5.7: Performing an *invokestatic*

2. Use the *ctok* value as an index into the class information, giving the structure for that class, which includes the table of static methods.

3. Use the *mtok* value as an index in the table of static methods.

4. The entry from the static method table will give the reference to the method content, which contains the information and instructions needed for the method.

The test virtual machine has been implemented this way, resolving the target method each time an instruction is executed. Depending on the device that class files are stored on, it is possible to simplify the dispatch process by storing a direct reference to the method information in the instruction, similar to “fast” bytecodes used in other virtual machines. Either at install time or at runtime, the target method for a given *invokestatic* instruction is resolved and the instruction updated with a reference to the target method. If installed applications are not moved within memory, this reference could be as direct as the memory address of the target method information found in step 4 above. The feasibility of such updates will also depend on the size required for the memory reference, since changing the length of instructions can be non-trivial, requiring branching instructions to have their branch offsets updated. If such a system were implemented, it would be desirable to limit the tokeniser to always produce *invokestatic* instructions of the same length, allowing the virtual machine or installer on the device to replace those instructions easily.

Table 5.7: Usage of the *invokespecial* instruction

Package	16,16	8,16	16,8	8,8
CLDC 1.0	0	0	0	656
CLDC 1.1M	0	0	0	531
MIDP	0	0	488	1921
MIDP examples	0	0	290	535
Javolution 3	0	0	266	1285
Javolution 5	0	0	747	1177
J2SE	1978	0	61763	23568

5.3.7 invokespecial

The *invokespecial* instruction has three roles to perform: calling class initialisation methods (constructors), calling private methods and calling methods in the super-class (i.e. implementing the *super* keyword). The format in a standard class file for this instruction (just as with the other invoke instructions) is:

183	*b1*	*b2*

With the two operands, *b1* and *b2* forming a 16-bit unsigned offset into the constant pool which results in a symbolic method reference.

To differentiate between the uses of the *invokespecial* instruction, the resolved method reference is examined. If the resolved method is in the current class, or is to a constructor, then that is the method to be called. This covers the case of a private method (which would be in the current class) and constructors. If the resolved method is instead in a super-class of the current class, then the third behaviour is assumed, and a new search is made from the current class's super-class, to find a matching method.

Table 5.7 shows how often each sized token was used by the test classes. While 16-bit method tokens were relatively rare, only being used by the J2SE library, 16-bit class tokens are reasonably common. Given that the 8,16 version of the instruction was not needed at all, it can be left out, with the 16,16 version being available if such a situation arises, giving three tokenised versions as follows:

183	*nargs*	*ctok*	*mtok*

203	*nargs*	*ctokH*	*ctokL*	*mtok*

210	*nargs*	*ctokH*	*ctokL*	*mtokH*	*mtokL*

The process for the tokenised *invokespecial* is identical to the tokenised *invokevirtual*, up until step 4. After resolving the method reference in step 3, a test must be made to determine which of the 3 possible behaviours is being used. The method reference will be to a structure that contains details about the method, including a flag to indicate if it is a constructor or not and if this flag is set, then the constructor calling behaviour is used and the resolved method is the one to execute. If the flag is not set, but the resolved method is in the same class as the currently executing method (i.e. the one that contains the *invokespecial* instruction), then the private method call behaviour is being used, and again, the resolved method is the one to execute. Finally, if neither of these are true, then the third behaviour is being used, meaning a call to a method in the super-class. Using the currently executing method, the VM finds the class it belongs to and from there, the class structure that represents the super-class. The virtual method table from the super-class is then used to dispatch the *mtok* value as if it were an *invokevirtual*.

5.4 Native Methods

The CLDC library requires native methods to implement some features, therefore, a virtual machine must implement these native methods. Only the CLDC 1.1 library was used for executing programs. Since some methods appeared to have been declared native purely for performance reasons (with the Java version of the method present but commented out in the source code), the library was modified to use the Java version of these methods, hence reducing the number of native methods to be implemented. The full list of native methods in the CLDC 1.1 API can be found in Appendix C. A minimal set of these methods was implemented to allow for the execution of the Javolution benchmarks (more details on the libraries used for testing in Section 3.9). There were both static and virtual native methods that needed to be implemented.

In all cases, the virtual machine would check the target of each method call to see if it had the native flag set. If so, there were two general dispatch methods, one for static

methods and one for virtual methods. The arguments for the native call would be gathered from the stack and passed to the dispatch method. Using the class and method tokens, the appropriate implementation is selected and executed. The following details which methods needed implementation and how these were implemented. Each section lists the relevant classes and below that, the prototype for each native method.

5.4.1 Static Methods

There were 5 static methods that needed implementing for execution to succeed. These were:

- java.lang.Class
  - Class forName(String)
- java.lang.System
  - void arraycopy(Object, int, Object, int, int)
  - long currentTimeMillis()
  - String getProperty0(String)
- java.lang.Thread
  - void sleep(long)

5.4.1.1 java.lang.Class

The *forName(String)* method allows a program to call the method with any arbitrary string and have the system attempt to load the class with that name (if it has not already) and return a Class object that represents the class. An application can then query that object to find out various information about the class or call the default constructor to create an object of the class, allowing for a very limited form of reflection.

The forName method causes a problem for a tokenised virtual machine, as the string class names have been removed during tokenisation. Either applications must be prevented

from using this method, or the string data for class names must remain in the tokenised files. Even if applications are prevented from using this method, the API classes make use of it in some situations for internal operations. To allow this method to function correctly, the converter was modified to output a VMConstants.java file, which is a valid Java source file that contains, as fields, the tokens assigned to certain classes and methods. The VM Constants file is detailed in Section 5.5.

5.4.1.2 java.lang.System

The system class had three static methods that needed implementation. The first, arrayCopy, is a performance method, provided to allow efficient copying of large amounts of data from one array to another. While it is provided purely for performance reasons, implementing this method in Java code was not possible, as its array arguments are declared to be of type *Object*, allowing any array types to be passed into the method. It is difficult to implement in Java, having to cast the objects to the appropriate type of array. Therefore a VM level method was used to move the data between the two array objects, which is efficient in the VM since it can take advantage of the internal format of array objects.

The *currentTimeMillis()* method will return the number of milliseconds that have elapsed since midnight at the start of 1st January 1970, and is used by any application that wishes to know the date or for timing. Since the VM is implemented in Java, this method is implemented by querying the underlying VM's version of this method for the current value.

The final method, *getProperty0*, is a private method used to implement the getProperty(String) method in System. This is used to query system properties that will have been initialised by the VM, including the version of the VM, class paths, the working directory, the users home directory and other platform specific information. It was found that the Javolution library, while it called this method, could function if this method did not return anything. The method contract allows for the method to return *null* if there is no property with the given name and therefore this method always returned *null*.

5.4.1.3 java.lang.Thread

The sleep method causes the current thread (i.e. the one that calls the method) to sleep for the given number of milliseconds, after which it becomes runnable again and must wait for the scheduler to select it. The Javolution library calls this method between tests, just after calling System.gc(), to allow time for the garbage collector to run. The sleep method was implemented by just calling through to the host VM's version of Thread.sleep(long).

5.4.2 Virtual Methods

There were 10 native virtual methods that were used. These consisted of:

- com.sun.cldc.io.ConsoleOutputStream
  - void write(int)
- java.lang.Class
  - String getName()
  - boolean isInstance(Object)
  - boolean isInterface()
  - Object newInstance()
- java.lang.Object
  - Class getClass()
  - int hashCode()
- java.lang.Runtime
  - void gc()
- java.lang.StringBuffer
  - String toString()

- java.lang.Thread
  - void setPriority0(int)

5.4.2.1 com.sun.cldc.io.ConsoleOutputStream

This class is used to implement the standard output from the application. While on some devices, standard out might not make sense (such as a mobile phone with a GUI interface, since there is not normally a console for standard out to go to). However, the API will take anything written to standard output (or standard error, which is just an alias for standard out in CLDC) and for each character, call *write(int)* to output one character at a time. In the test VM, each call to this method results in the parameter being written to a file called std.out in the VM's working directory. When running the Javolution benchmarks, standard out is used to report the test results, and therefore used to verify correct operation.

5.4.2.2 java.lang.Class

This class represents a class file that has been loaded by the virtual machine, allowing a limited form of reflection, such as an application querying classes at runtime that were unknown at compile time. The static *forName(String)* method is used to request the virtual machine return a Class object to represent the class with the given name. These virtual methods can then be used on such a Class object:

- String getName()
- boolean isInterface()
- boolean isInstance(Object)
- Object newInstance()

The *getName()* method will return the name of the class this object represents. Given the removal of the string data, the getName method was implemented to return the class's token as a string instead, which was sufficient for running the tests. If the functionality

of getName was required, then the names of each class would need to be included on the device.

The *isInterface()* method allows an application to query if the class object represents a class or an interface.

The last two methods are for interacting with objects of the class. Firstly, *isInstance(Object)* can determine if the given object is actually an instance of this class. The *instanceof* keyword can be used in source code, rather than an explicit call to this method, however, the *instanceof* keyword is compiled to the *instanceof* instruction, which requires the class type to be known at compile time. The *isInstance(Object)* method allows the test to be performed for any class type. The final method is *newInstance()*, which will create a new object of this class using the default constructor (CLDC does not support calling a constructor that takes parameters via reflection). This is the reflection implementation of the *new* keyword.

All these operations involve interaction with either the class loader or with objects at a low level, therefore requiring knowledge of the VM's implementation, which can only be done in native methods.

5.4.2.3 java.lang.Object

The root of the inheritance tree, as such this class contains methods that every Java object will contain. Two of these methods have native implementations:

- Class getClass()
- int hashCode()

The *getClass()* method is used to return the Class object that represents the type of this object, which requires knowledge of the structure of objects and interaction with the class loader. The *hashCode()* method is used to return a hash value for the object. There is a very specific contract on how this method must behave, the returned hash code must be unique within the system and must not change for a given object during that objects life. To since objects within the test VM were represented using objects in the underlying host VM, the native implementation just returned the hash code of the underlying object.

5.4.2.4 java.lang.Runtime

The Runtime class provides access to information and controls for the runtime of the VM, and underlies some of the System classes features. In this case, the *System.gc()* method will call the native *gc()* method in this class. Since the *Runtime.gc()* method should cause the virtual machine to perform garbage collection, it must be native, so it can interact with the garbage collector. Since the test VM was implemented in Java, there is an underlying Java virtual machine which runs the test virtual machine. The test VM implements this method by simply calling the *gc()* method of the underlying VM.

5.4.2.5 java.lang.Thread

The Thread class allows an application to interact with and create threads of execution within the VM. The only method used was *setPriority(int)*, which is used to change the priority of a thread. A private native method, *setPriority0(int)*, is used to interact with the VM and do the actual change of priority. Since the test VM was not multi-threaded, this method was just ignored.

5.5 VM Constants

The virtual machine needs to interact with some classes, such as creating an exception object or accessing the internals of string objects, and therefore needs to know what tokens were assigned. In particular, class tokens are needed for the following classes:

- java.lang.Class
- java.lang.ClassCastException
- java.lang.ClassNotFoundException
- java.lang.Error
- java.lang.InstantiationException
- java.lang.Object

- java.lang.String
- java.lang.Throwable

As well as the class tokens, the field tokens are also needed for the following three fields in the java.lang.String class:

- value
- offset
- count

The class tokens are needed by the virtual machine as it must interact with these classes and be able to generate instances of them at runtime. For example, the VM is required to throw various exception/error classes in certain situations, such as an ArithmeticException from the divide bytecode if the divisor is 0, and therefore must be able to instantiate objects of these classes. While the Class, Object and String classes must be instantiated by the VM in certain situations and as such the VM must know what token each of these classes received.

The three fields in the String class are needed because the VM must interact with String objects on a very low level. For example, when creating an exception object, the VM must first create a string object with the error message, then pass that object to the exception's constructor. The VM achieves this by creating a string object and setting the internal fields of the object to the appropriate values. The inverse operation is also required where a String object in the application needs to have the string it represents extracted. Internally a String object consists of a character array holding the characters, as well as an offset of where the string starts in that array and a count of how many characters are actually in the string.

As well as the above, the class and method tokens are needed for all the native methods. When a method call that is marked as native is encountered, the only thing known about the method is the class token for the class it is in and the method token. Therefore, these values are used by the VM to dispatch to the correct implementation for the method.

5.5.1 Implementing VM Constants

To leave flexibility in the converter, the tokens that the VM needs to know are not restricted at conversion time, allowing the converter maximum flexibility to allocate tokens as it sees fit. However, so the VM can execute the resulting tokenised classes, an addition file is output from the converter, called “VMConstants.java”, which is a correctly formatted Java source file that defines one class: *VMConstants*. The VMConstants class contains a number of fields, each one representing the token for the various classes, fields and methods. The VMConstants file is compiled into the VM, which then references the appropriate field when needed.

Compiling constant values into the VM is not extensible, but was sufficient for testing purposes. The constant values were only required for classes in the CLDC library and updating of this library on a real device would be quite rare. If such an update took place, it would be done by the device manufacturer, allowing them to update the virtual machine and libraries together. Further libraries/applications will not need to extend or modify these constant values.

5.6 Correctness of Execution

Two working virtual machines were created, capable of executing either standard or tokenised class files respectively. The standard virtual machine provided verification of the implementation, the code for which was then modified to allow execution of tokenised files. For testing, the CLDC1.1M and Javolution5 libraries were used. Specifically the Javaloution5 library had been incrementally tokenised on top of the CLDC1.1M library.

The Javolution package was executed using Sun’s J2SE virtual machine, to observe correct execution. While the Javolution library is intended as a real-time library for Java applications, it includes a benchmark feature, which exercises various parts of the library, therefore output was in the form of timing information for each test. The Javolution package was then executed using the implemented virtual machine to execute the standard class files and the tokenised class files respectively. All runs completed normally, with the only

output variation consisting of the timing information in the benchmarks, indicating that the virtual machine was functioning correctly.

Further analysis was done to ensure that the execution was non-trivial and that a suitably large volume of code had actually been executed. Firstly, it is ensured that a reasonable amount of the instruction set had been used and secondly, that these instructions had been from a large set of methods, not just a few methods called many times. The following sections present the analysis of the instruction set usage and method coverage.

5.6.1 Instruction set usage

Each Java instruction consists of a single byte, allowing for 256 different possible opcodes. Of these, Sun specify 201 instructions (using the values 0-201 inclusive, except for 186[3]). There are three reserved values for special use, 202, 254 and 255 and the remaining values are reserved for future use. The three reserved values cannot appear in a valid class file, however they may be used by the interpreter or inserted at runtime. The opcode 202 indicates a breakpoint, i.e. for use by a debugger to insert breakpoints into code, while 254 and 255 are left to the VM implementer to define their action.

When the Javolution 5.2.5 library was executed on the virtual machine, it used 121 of the 201 valid instructions, leaving 80 instructions unused. The primary concern for testing was that the 4 invoke instructions be shown to operate correctly. For the 4 instructions, invokevirtual was executed approximately 2.9 million, invokestatic 1.2 million, invokeinterface 0.5 million and invokespecial 0.4 million times. At 2.9 million uses, invokevirtual was the forth most used instruction during execution. The program ran as expected and produced its expected output, showing that the tokenised class files were executed successfully.

5.6.2 Method coverage

The CLDC1.1M and Javolution5 packages have a total of 452 classes, which contain 3148 methods (including methods without implementations). Since the virtual machine uses lazy

[3]The opcode 186 was used historically, but is considered invalid in modern class files.

class loading, only 199 of these classes actually got loaded during execution. These loaded classes contain a total of 1450 method definitions, 1387 of which have code attached to them.

Running the Javolution benchmarks resulted in the execution of 700 unique methods, which constitutes about 22.2% of all the methods in the CLDC1.1M and Javolution5 packages combined (or nearly 50% of the methods actually loaded). Complete method coverage was not expected, since the Javolution benchmark code is not designed as a thorough testing suite. Even if the benchmarks exercised all code in the Javolution library, there are many classes and methods in the CLDC API that would not get used, such as the framework for handling IO and network connections. As well, there are exception and error classes, that will not get used in the course of normal execution.

A significant variety of methods were executed, indicating that tokenisation produced class files that were executed successfully.

5.7 Efficiency of Execution

As well as testing the correctness of execution, the efficiency of execution is also examined. Since the target of this work is J2ME, the logical virtual machine for comparison is the Kilobyte Virtual Machine (KVM) from Sun Microsystems [62]. In particular, a Java Card virtual machine is not considered because open-source implementations are not available that could be easily instrumented and because the Java Card VM is significantly different to J2ME. Java Card is targeted at embedded chips with extremely limited processors (even the 'int' data type is optional), and uses a fundamentally different file structure to larger Java relations (e.g. J2ME). The goal is to examine the application of Java Card like tokenisation on J2ME systems, hence the focus on testing against a J2ME virtual machine.

Since the test virtual machine discussed above was not optimised for speed, it could not be used for comparison. Instead, a small implementation of just the method lookup process was implemented to test the complexity of the procedure. Since only the performance of the *invokevirtual* instruction was being examined, a Java application was used that consisted mainly of *invokevirtual* instructions. The following sections describe the testing process in

Table 5.8: Instruction Usage in Test Class

Instruction Name	Setup	Test Case	Total Executions
aload_0	1	111,111,110	111,111,111
aload_1	1	0	1
astore_1	1	0	1
dup	1	0	1
return	3	111,111,111	111,111,114
invokevirtual	1	111,111,110	111,111,111
invokespecial	2	0	2
new	1	0	1

detail.

In particular, a Java Card virtual machine is not considered for

5.7.1 Test Application

An application that makes heavy use of the *invokevirtual* instruction is needed to test each virtual machine. An ideal test would be an application that consists entirely of *invokevirtual* instructions. However, parameters must be pushed onto the stack and the *return* instruction must accompany any method calls, making this impossible. Loops were avoided to minimise any extra instructions, resulting in a class that consisted of 8 methods, named run100000000, run10000000, run1000000, and so on, down to run10. Each method called its smaller version ten times, i.e. run100000000 contained ten calls to run10000000, run10000000 contained ten calls to run1000000 and so on. Finally the run10 method would call a ninth method, noop, ten times. The noop method did nothing other than return. Since the aim was to test the *invokevirtual* instruction, the above methods were all non-static. Finally a main method was added to the class, that would create an object and call the run100000000 method once.

The result was an application that, when executed, would perform a total of 111,111,110 method calls. To verify the instructions used, the class was compiled and executed with the test virtual machine, which contained instrumentation to count the use of each instruction. The results of this are shown in Table 5.8.

A few additional instructions get used by the application. If the first line in the main

method is examined, which consists of: "Main m = new Main();", it is compiled to the bytecode sequence: *new, dup, invokespecial, astore_1*. These instructions first create an uninitialised object, duplicates the reference on the stack, calls the default constructor on it (consuming the first object reference), then finally stores the remaining object reference into local variable 1. The default constructor for a class, when not specified, will call the default constructor of the super-class (in this case, java.lang.Object), which will use the instructions: *aload_0*, *invokespecial* and *return*, which loads the *this* reference, calls the super constructor and returns. The default constructor in Object just performs a *return*. The next line in the main method is: "m.run100000000();", which produces an *aload_1* instruction to load the stored object reference from earlier, then an *invokevirtual* to call the method. A final *return* from the main method is also needed when execution finishes. These instructions account for the "setup" for the test and are shown in the Setup column of Table 5.8.

The Test Case column shows the instructions used during the execution of the run* methods. The number of *aload_0* and *invokevirtual* instructions are the same, since before each invoke, the object reference for "this", stored in local variable 0, must be pushed onto the stack. The apparent imbalance between invokes and returns occurs because there is no invoke instruction used to start the call to the main() method, however there is a return instruction. It appears in this section since the initial call into the run100000000() method is counted as setup, and the return from it as part of the test case.

The virtual machine used to verify these numbers shortcuts some of the start-up procedure of a normal virtual machine, such as the KVM. The KVM performs more initialisation of some of the API classes before it even calls the main method in the application, which lead to slightly more uses of the *invokevirtual* instruction, up to 111,111,172. The exact number of calls was not important for the test however, just the amount of time each call takes, and the testing took into account the few extra instructions in the KVM.

5.7.2 VM Implementations Tested

The KVM was used as an example of a current J2ME virtual machine for comparison against the tokenised dispatch presented in this thesis. Since the KVM is designed for a variety of hardware, some features can be enabled or disabled at compile time. Of relevance to testing are the "fast" bytecodes, which are used to improve performance of the *invoke** instructions. The KVM was therefore tested with fast bytecodes both turned on and turned off, giving three implementations for testing: KVM, KVM-Fast and Tokenised. The following sections describe each in detail.

5.7.2.1 KVM and KVM-Fast

To perform an *invokevirtual*, the following steps must be performed by the KVM:

1. Resolve the symbolic reference from the constant pool, which involves:
   (a) Use the operand as a constant pool index, to load a method reference.
   (b) Use the entries in the method reference to load other constant pool entries, resulting in all the data needed for a symbolic method reference.
   (c) Use the symbolic data to find the target class (loading and linking the class if it has not happened yet).
   (d) Use the symbolic data to find the target method within that class (or possibly within a super-class).
2. Use the number of arguments from the method details found from the constant pool to get the object reference from the parameters on the stack.
3. Find the dynamic target for the method call, which is a repeat of step 1 part (d), except using the class type of the object, rather than the class from the symbolic reference.
4. Push a new stack frame and populate it.
5. Begin execution of the new frame.

Since tokenisation only speeds up the method lookup procedure, testing was focused on determine how long it took from when the instruction started to be processed until the target method had been found (steps 1 to 3 above). The time to push a new stack frame and transfer control will be similar no matter what method lookup procedure is used.

The KVM includes two important caching mechanisms to speed up *invokevirtual* instructions. The first caching occurs during step 1 above, when using the symbolic data in the constant pool to find a matching method. After first resolving the target method for that constant pool entry, the method is cached. Future references to the same constant pool entry will result in the cached method, therefore only needing to perform part (a) of the above sequence before moving onto step 2. The second caching mechanism involves replacing the *invokevirtual* bytecode with a special *invokevirtual_fast* bytecode, which caches the final method that was called, for a given object type. In future if the same object type is on the stack, then the steps 1 to 3 can all be skipped, and the cached method reference used. If the object type is different, then steps 1 and 2 can still be skipped, however step 3 still has to be performed, as the object may have a different implementation for the target method.

The constant pool caching feature is always present in the KVM, however, fast bytecodes require that the bytecodes can be modified during execution, which is not possible on some platforms, for example, when the bytecodes are stored in read-only memory. Therefore, support for them can be turned on or off via compile time flags. The combination of constant pool caching and fast bytecodes means there are three possible cases for an *invokevirtual* instruction:

1. *invokevirtual* performing the full constant pool lookup.

2. *invokevirtual* where the constant pool lookup has been cached.

3. *invokevirtual_fast*.

When timing the KVM, the type of invokevirtual performed is also taken into account. Therefore, the KVM (with fast bytecodes disabled) will have a measurement for case 1 and 2, while the KVM-Fast will measure all 3.

5.7.2.2 Tokenised VM

While the test virtual machine does make use of tokens and the virtual method table, it was not written to have an optimised method dispatch system and is therefore not suitable to use for performance testing. Therefore, an implementation of the method dispatch procedure for tokenised code was written in C, using memory structures to represent the various structures a virtual machine needs to maintain, such as class files, method pointers and the virtual method table. An image was created that represented the test application, including an entry for the class, with pointers to the classes virtual method table and method structures to represent each of the run*() methods in the class.

To correspond to the method resolution steps being timed in the KVM, a C function was written that accepted a method token and object as parameters and would return the target method as a pointer. The function was used to resolve the same method calls in the same order as they are present in the test Java application.

During compilation the optimiser was turned off ("-O0" option to GCC) to ensure none of the lookup process was optimised out. Also, the resolved method was used to run a loop to simulate execution of the resolved method, which adds additional code between each method dispatch, to remove the effect of processor caching and more closely emulate the workload that would be expected in a production virtual machine.

5.7.3 Methodology

Three different approaches were used to time each of the virtual machine implementation. First, an approximation of the runtime was found by counting the number of instructions used. Performance was then tested using "gprof", the GNU Profiler, and finally, the code was instrumented using the RDTSC (ReaD TimeStamp Counter) x86 instruction to read the processors time stamp.

Testing was done on an AMD Opteron 244[4] based, dual processor system with each processor running at 1800 MHz. To minimise noise, only one benchmark was ever run at a time, leaving the second processor free to handle other system events and while no

[4]Specifically the C0 stepping of the "Sledgehammer" model.

other users were on the system. The operating system was Red Hat Enterprise Linux AS release 3 (Taroon Update 2), which is a 64-bit operating system, allowing testing of 64-bit and 32-bit versions of applications. The KVM however, was not written to support 64-bit and as such will not compile in 64-bit mode. Therefore, testing of the KVM was limited to 32-bit only, however the tokenised test was run in 32 and 64-bit modes for comparison. The only difference between 64-bit and 32-bit was during compilation, with the use of the "-m32" or "-m64" flag to gcc, to request a 32-bit or 64-bit binary respectively.

The following sections detail how each timing mechanism was used.

5.7.3.1 Instruction Count

To gain an initial approximation, the code was executed in the GNU Debugger (gdb), using the step-instruction feature. The code was compiled with debugging symbols and an appropriate breakpoint set, then the step command was used to step through individual x86 instructions until the target code had completed execution.

Counting instructions in this manner is very slow and labour intensive, therefore limiting the number of samples that can be gathered. However, this provided an initial estimate of execution time, which was compared to later results to check for consistency.

5.7.3.2 Profiling Using gprof

The GNU Profiler (gprof) allows an application to be executed and profiling information generated. The instrumentation done by gprof will ensure exact values for the number of calls to a given function. As well, function runtime is given by sampling the currently executing function at given intervals, and then using a statistical approach giving an approximation of the functions runtime [43].

The KVM has its interpreter loop separated into two functions, FastInterpret and SlowInterpret. The FastInterpret function contains the main loop and the implementation for the "common" bytecodes, while the SlowInterpret function contains the implementation for the "uncommon" bytecodes. This approach is taken by the KVM to reduce the amount of code in the switch statement used to decode instructions, so compilers can optimise it

easier. The *invokevirtual* instruction was the only instruction used that was implemented via SlowInterpret. Since gprof only provides information about the execution of functions, not arbitrary sections of code, the timing information for the SlowInterpret function is the most accurate data that can be obtained using gprof. Ideally the measurement should cover just the time needed to resolve a method call, however, the SlowInterpret function includes other overheads, such as pushing the new frame for the method, reporting the method call for debugging and finally transferring control to the new frame.

5.7.3.3 Profiling Using RDTSC

The most precise method for timing is the RDTSC (read timestamp counter) instruction, which was added to the Pentium processor and allows a program to read the number of clock cycles that have occurred since the processor was last reset. The behaviour of the counter varies depending on brand (Intel or AMD) and on processor version. On all AMD processors and earlier Intel processors, the counter would increment on each processor clock cycle, giving an accurate count of processor clocks, but an inaccurate measurement of real time if the processor has changed clocked speeds (i.e. for power saving or thermal management). Newer Intel processors have the counter update at a rate equivalent to the maximum clock speed of the processor in all except for a few situations [48], hence giving a better measurement of real time, but not of the number of processor cycles. Since testing was performed on an AMD based server system, RDTSC will give an accurate count of processor cycles.

Benchmarking was done by adding an RDTSC instruction before and after the code to be timed, with the difference between the two values giving the duration of the code in question. Modern processors do not guarantee the order in which a set of instructions will be executed, rather instructions can be executed out-of-order (provided the final result is as if they were executed in order). Out-of-order execution is a problem when trying to measure a short sequence of instructions, as the initial RDTSC instruction may not get executed until some of the instructions to be timed have already been completed. The processor must be forced to finish all previous instructions, including the RDTSC instruction, before execution on the code needing timing. Typically, instructions known as a "serialising instruc-

Algorithm 4 Timing Harness

```
RDTSC(); // Get the current time stamp value
CPUID(); // Ensure that the rdtsc instruction is finished
<code to be timed>
CPUID(); // Ensure the timed code has completely finished.
RDTSC(); // Get the finish time for the code.
```

tion" can be used, that is, an instruction that cannot execute until all previous instructions are complete. In particular, the CPUID instruction, which provides details about the CPU is one such instruction. Some additional instructions are also required, such as getting the values returned from the RDTSC instruction (which will be in the EDX and EAX registers) and storing them for later. To implement the timing and the serialising, two functions were written, both using inline assembly. The first, RDTSC(), would use the RDTSC instruction and return the 64-bit timestamp value, while the second, CPUID(), would execute the CPUID instruction. Using these functions leads to the pseudocode shown in Algorithm 4. The timing harness will add a small amount of overhead, since after the initial RDTSC instruction, the value must be stored and the CPUID instruction then stalls the pipeline to ensure that the timestamp value is stored before continuing with the code requiring timing.

5.7.4 Results

The first two results cover the instruction counting and gprof timing of the KVM. These were used as initial approximations to verify that later results were within the range expected. Following these are the more accurate timings of the KVM and tokenised VM, performed using the RDTSC instruction.

5.7.4.1 KVM Instruction Count

To get an initial approximation of the number of instructions it should take to perform a single invokevirtual instruction, gdb was used, with a break point set at the beginning of the method resolution process. Once the program had stopped, the step instruction feature of gdb was used to step through individual x86 instructions until the target method had been identified and the KVM was ready to perform the call. The fast bytecodes feature was

disabled for this test. Several executions were observed, to cover cache hits and misses for the constant pool lookup.

In the case of a cache miss, somewhere between 700 and 1100 instructions were used. Continuing execution until a cache hit, the number of instructions was much lower, being approximately 200. Both method lookups were observed to use linear search when searching for a method within a class, which makes the number of instructions dependent on the size of the method tables and relative position of the target method. Due to the labour intensive nature of this approach, only 2 executions of each case (cache hit and cache miss) were observed and therefore do not represent reasonable sample size. However, these counts provide a rough estimation to verify that later results seem reasonable.

5.7.4.2 KVM gprof Timing

The SlowInterpret function was reported as being called 111,111,360 times. The extra instructions are the result of extra initialisation performed by the KVM, before it begins execution of the application's main method. These few extra calls will have a statistically insignificant impact on the total execution time of the SlowInterpret function. The test Java application was run on the KVM a total of 3 times, giving a total execution time of 20.69 seconds for the SlowInterpret function,or an average of approximately 62 nanoseconds for each call to SlowInterpret and hence to execute a single invokevirtual. However, this time includes the entire method call process, including pushing stack frames.

When taking into account the clock speed of the test machine (1800MHz), 111 nanoseconds would be equivalent to 112 clock cycles. By stepping through with the debugger earlier, it was found that the complete function required approximately 200 instructions (in the case of a cache hit, which is by far the most common). Given the super-scalar nature of modern processors, throughput of 2 instructions per clock cycle does not seem unreasonable. However, gprof measures run-time of function by sampling the currently executing function at a fixed interval of 0.01 seconds. The cumulative time used by all functions (as reported by gprof), was only approximately 72% of the runtime reported by the Linux 'time' command, indicating that gprof likely 'missed' some of the program's execution time, due to the very short runtime of the functions involved. If this were the case, then the

real number of cycles per invoke would be greater than the 112 cycles found using gprof.

5.7.4.3 RDTSC Overhead

Since the RDTSC timing harness will add some overhead to the code being timed, the test harness was executed with no code inside, to measure the size of this overhead. The test was run on the dual processor AMD system as a 32-bit application and then again as a 64-bit application. Testing was also repeated on another machine in the same cluster, with no significant variation. The results presented below are for the first machine.

The results of these tests are presented in Table 5.9. Parts (a) and (b) show the number of runs and the percent of runs that took the given number of cycles to complete for 32-bit and 64-bit modes respectively. Parts (c) and (d) then present the summary, with the minimum, median, average, maximum and standard deviation for the number of cycles it took to execute the timing harness.

Both tests resulted in a fairly tight clustering of results, however the 64-bit executables gave far more consistent results. For the 64-bit test, 99% of executions of the timing harness were complete in 132 cycles. Executions that took only a few cycles more than 132 can be accounted for as effects of the pipeline and/or cache, while larger values are likely the result of an interrupt or other operating system interference. While the standard deviation is quite high for the 64-bit test, this can be accounted for by the few, but extremely large, outliers. Therefore, it can be concluded that, in the case of a 64-bit executable, the overhead is 132 cycles, with the other measured values being the result of noise in the system.

The results for the 32-bit test are not as clear as those for 64-bit. The immediate observation is that for 32-bit, 132 cycles seems to be an unobtainable number. With only a handful of results between 132 and 135, it would seem 136 cycles is the best time for a 32-bit binary, 4 cycles slower than for 64-bit. The distribution of results is also different, with the majority of results distributed between four values: 136, 138, 140 and 142 cycles. This distribution was explored further by executing the same test case again on the same test machine and on a second identical machine. While the distributions between the four key values of 136, 138, 140 and 142 cycles changed, those values still accounted for the majority of the executions. Therefore, by excluding the larger times as aberrations

Table 5.9: Timing Overheads Using RDTSC

(a) 32-bit

Cycles	# of runs	% of runs
132	0	0.00
133	0	0.00
134	1	0.00
135	1	0.00
136	34,429,237	34.43
137	28,495	0.03
138	16,116,784	16.12
139	17,900	0.02
140	34,929,562	34.93
141	37,497	0.04
142	14,040,439	14.04
143	33,917	0.03
>143	366,167	0.37

(b) 64-bit

Cycles	# of runs	% of runs
132	98,971,031	98.97
133	143,681	0.14
134	42,592	0.04
135	32,585	0.03
136	95,570	0.10
137	55,098	0.06
138	14,294	0.01
139	48,847	0.05
140	43,687	0.04
141	86,605	0.09
142	16,660	0.02
143	293,207	0.29
>143	156,143	0.16

(c) 32-bit Summary

Min Value:	134
Median Value:	138
Average Value:	139.5215
Max Value:	174,434
StDev	112.8356

(d) 64-bit Summary

Min Value:	132
Median Value:	132
Average Value:	133.8219
Max Value:	172,483
StDev	159.4441

caused by timer interrupts or other operating system interference, it can be concluded that the timing overhead for 32-bit will consist of between 136 and 142 cycles. Averaging all the data collected after multiple runs on on both of the testing machines resulted in a value of 141.218 cycles, still within the 136 to 142 bracket.

The evidence suggests a 64-bit binary would provide the most accurate timing, however the KVM cannot be compiled in 64-bit mode, requiring the KVM to be executed in the slightly slower and noisier 32-bit mode. During later testing, similar noise within the system can be expected, as was present during this testing, resulting in the occasional large value. Therefore, the average values found will be used as the expected overhead in later testing. These values are: 141.218 cycles for 32-bit binaries and 133.8219 cycles for 64-bit binaries.

5.7.4.4 KVM RDTSC Timing

For the first test, fast bytecodes were disabled, leaving only the constant pool caching mechanism. References to items in the constant pool could either result in a cache miss, requiring an expensive lookup for that item, or a cache hit. Therefore, the results consisted of two timing values, one for a cache hit (cached) and one for a cache miss (uncached). To gather a large sample size, the KVM was executed a total of 500 times, with the results being recorded for each run. Table 5.10 summarises the results in each of these two cases. Since the test application calls a small number of methods a large number of times, there are only 27 uncached method calls. The remaining 111,111,145 calls all occur with the constant pool entries having been cached. In both tables, the Total Cycles column shows the total accumulated cycles for all the calls, while the next column shows the average number of cycles needed per call. The final column, Adjusted Cycles/Call, shows the average number of cycles that was really taken once the overhead of the timing code is removed, in this case 141.218 as found in Section 5.7.4.3. Figure 5.8 shows the histogram graph for the 500 runs.

These results show that the complete constant pool lookup and creation of the cache entry adds approximately 500-700 cycles to the execution time, for a total time of 720-960 cycles for an uncached call. Although, the small number of uncached calls limits

Table 5.10: *invokevirtual* Results Using RDTSC

(a) Times for a cached *invokevirtual*

Cached - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	42,691,182,381	384.221	243.003
Average	43,744,893,928	393.704	254.182
Median	43,419,992,203	390.780	251.258
Max	45,276,032,029	407.484	266.266
StDev	619,721,794	5.557	

(b) Times for an uncached *invokevirtual*

Uncached - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	23324	863.85	722.63
Average	24089	892.18	752.66
Median	24042	890.44	750.92
Max	29701	1100.04	958.82
StDev	416	15.39	

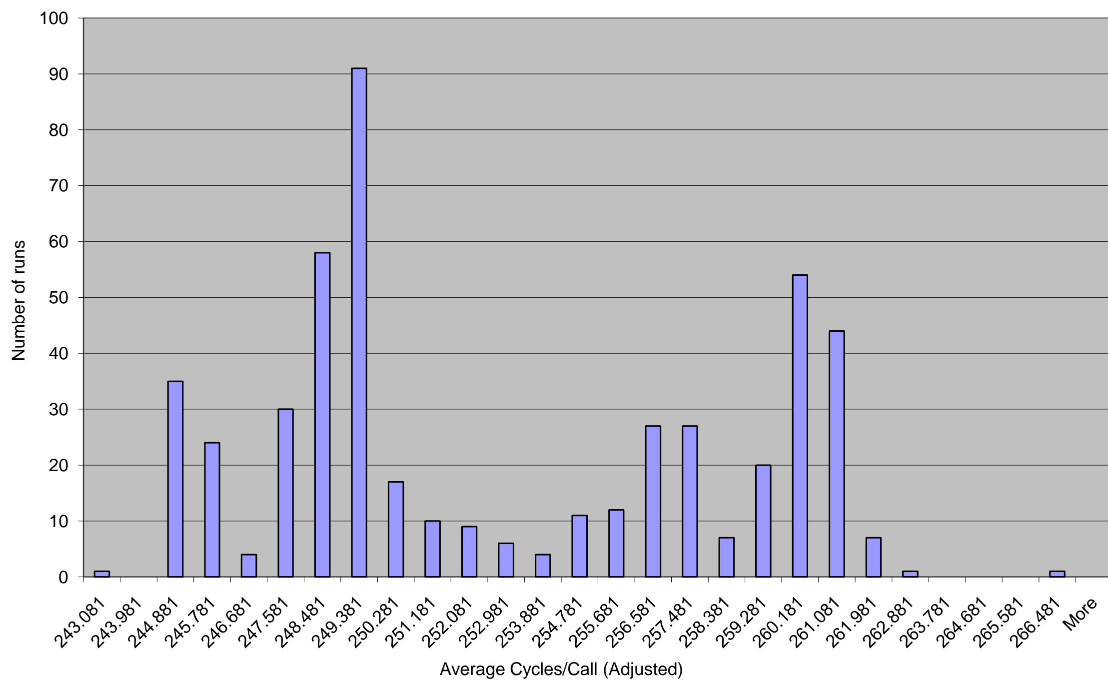


Figure 5.8: Histogram of Average Cycles/Call for Uncached *invokevirtual* Calls

the confidence that can be placed in these values. For a cached *invokevirtual,* the time taken was 243 to 266 cycles per call. The initial instruction counts from Section 5.7.4.1 had estimated 200 instructions for a cached *invoke* and 700-1100 for an uncached, which corresponds to the findings here. The grpof testing gave an estimate of 112 cycles for a cached *invoke*, however that number is likely to be too small, since gprof did not account for all of the applications runtime.

5.7.4.5 KVM-Fast RDTSC Timing

The second test involved compiling the KVM with support for fast bytecodes. The additional *invokevirtual_fast* method was also instrumented with the same RDTSC timing harness and the same test application was then executed a total of 500 times. The results are summarised in Table 5.11.

As can be seen, the *invokevirtual_fast* bytecode is significantly faster, taking on average only 73 cycles to execute. The fast bytecode works by caching the type of the object and final target method inline in the instruction. On later executions, if the object for the method call has the same type as the cached entry, then the target method from the cache can be used without any lookups. However, an application with a polymorphic call site will find the cached entry does not always match, resulting in a call similar to a cached *invokevirtual*. For the test application there was only ever the one object, therefore the *invokevirtual_fast* bytecode always had the correct target, never needing to do these extra lookups. Compared to the previous test, the cached *invokevirtual* time is slower by about 100 cycles, which is consistent with the extra work needed to replace the instruction with the *invokevirtual_fast* instruction.

5.7.4.6 Tokenised RDTSC Timing

Two versions of the Tokenised test were run, one in 32-bit mode the other in 64-bit. As with the RDTSC Overhead testing, the only difference between the two tests was the use of either the “-m32” or “-m64” flag to GCC. The summary of the times can be seen in Table 5.12. When comparing the 32-bit and 64-bit times, again the 32-bit seems to be slightly

Table 5.11: KVM-Fast Results Using RDTSC

(a) Times for a cached *invokevirtual*

Cached - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	38,374	491.974	350.756
Average	40,898	524.336	383.118
Median	40,707	521.878	380.660
Max	46,796	599.949	458.731
StDev	1,207	15.475	

(b) Times for an uncached *invokevirtual*

Uncached - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	25,632	949.333	808.109
Average	26,792	992.300	851.081
Median	26,740	990.389	849.170
Max	28,697	1,062.852	921.630
StDev	549	20.332	

(c) Times for *invokevirtual_fast*

Fast Invoke - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	23,794,664,921	214.152	72.934
Average	23,851,908,225	214.667	73.449
Median	23,838,514,508	214.547	73.328
Max	23,925,270,503	215.328	74.109
StDev	39,078,227	0.352	

Table 5.12: *invokevirtual* on the Tokenised Virtual Machine

(a) Times for *invokevirtual* (32-bit)

32-bit - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	17,768,138,285	159.913	18.694
Average	17,788,735,390	160.098	18.880
Median	17,789,276,036	160.103	18.885
Max	17,815,377,609	160.338	19.119
StDev	10,992,164	0.099	

(b) Times for *invokevirtual* (64-bit)

64-bit - System 1	Total Cycles	Cycles/Call	Adj. Cycles/Call
Min	17,005,485,975	153.049	11.830
Average	17,119,602,484	154.076	12.857
Median	17,116,518,483	154.048	12.830
Max	17,224,280,656	155.018	13.800
StDev	54,784,798	0.493	

slower, taking on average 6 more cycles, which is consistent with the discrepancy noticed when testing the RDTSC overhead. The distribution of the results can be seen in Figure 5.9 for the 32-bit and Figure 5.10 for the 64-bit.

5.7.5 Summary

Taking the results from the previous tests results in the summary shown in Table 5.13. It can be seen that for the KVM cached and uncached implementations, the version with fast bytecodes was significantly slower. This can be explained be the extra overhead of having to replace the bytecode with the *invokevirtual_fast* instruction.

The slowest three times (KVM-Fast Cached, KVM Uncached and KVM-Fast Uncached) all only had a very small number of samples compared to the others. In the case of the KVM, most invokes were of the cached type, while for the KVM-Fast, the most common was the invoke_fast. Although the small number of samples reduces the confidence in these values, it is obvious that they are significantly slower than the tokenised implementation.

These results have shown that the tokenised invoke calls are significantly quicker to

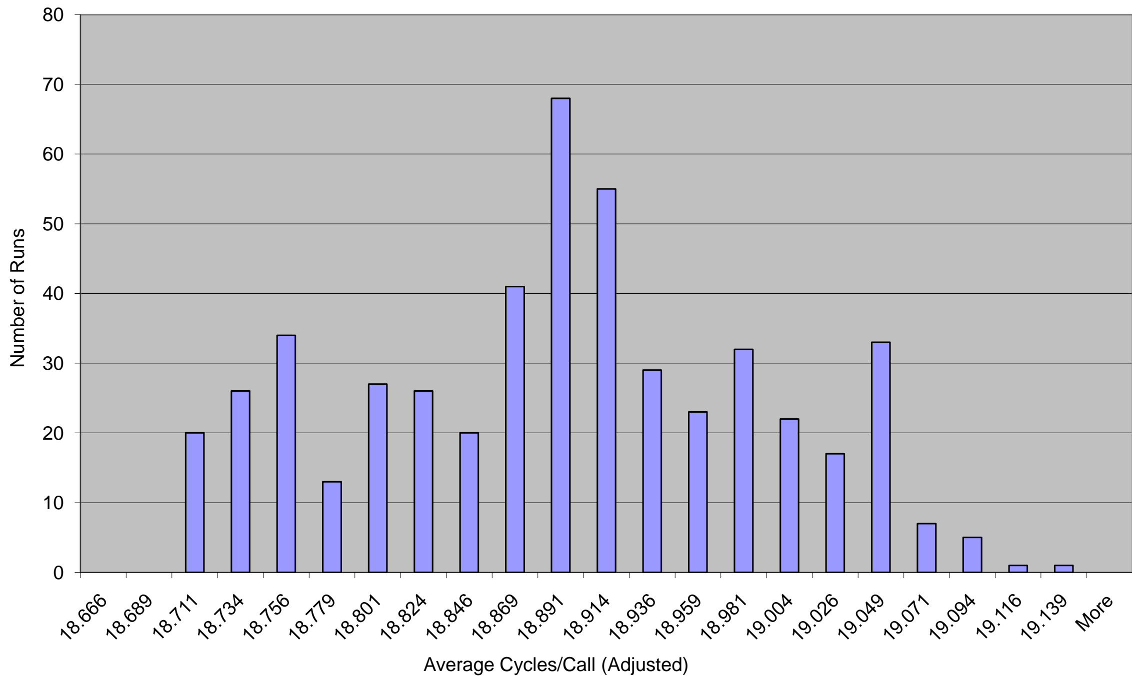


Figure 5.9: Histogram of Average Cycles/Call (Adjusted) for 32-bit Tokenised *invokevirtual*

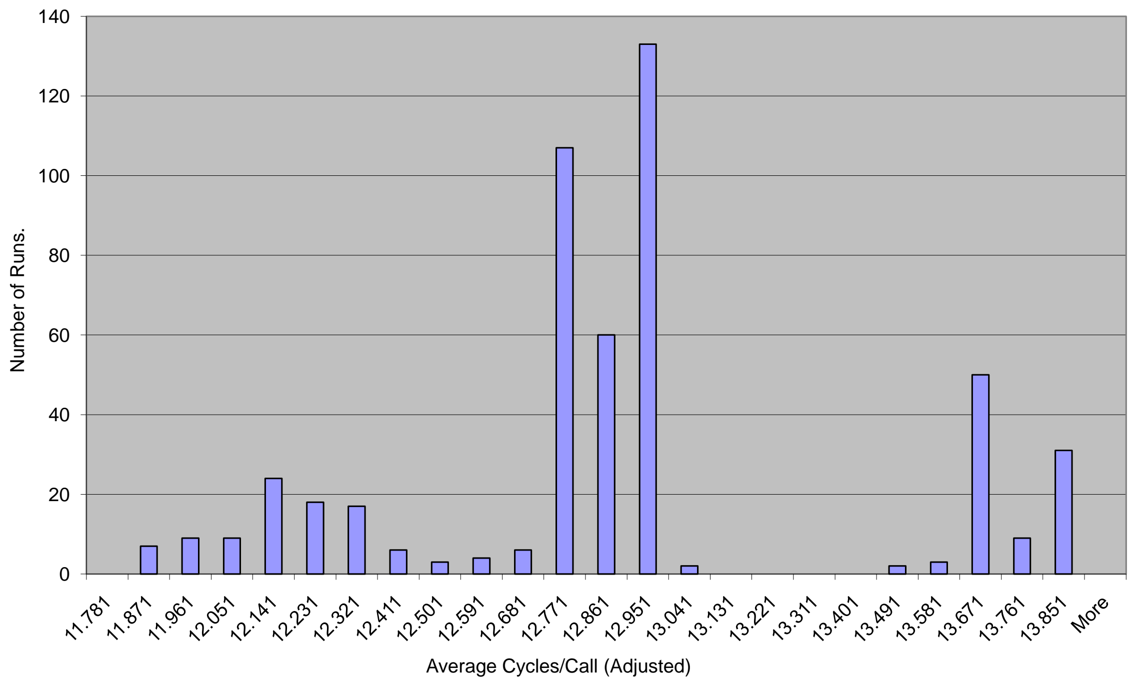


Figure 5.10: Histogram of Average Cycles/Call (Adjusted) for 64-bit Tokenised *invokevirtual*

Table 5.13: Overall Summary of Adjusted Cycles/Call for all Tests

VM	Min	Average	Median	Max
Tokenised	18.69	18.88	18.89	19.12
KVM-Fast invoke_fast	72.93	73.45	73.33	74.11
KVM Cached	243.00	254.18	251.26	266.27
KVM-Fast Cached	350.76	383.12	380.66	458.73
KVM Uncached	722.63	752.66	750.92	958.82
KVM-Fast Uncached	808.11	851.08	849.17	921.63

	JOP	leJOS	TINI	Komodo	JStamp	SaJe	Xint
invokevirtual	128	4,759	6,495	384	349	112	182
invokeinterface	146	5,094	6,797	1617	531	148	193

Table 5.14: Execution times in clock cycles, taken from [76]

execute than even the fastest KVM instructions. Also, the fast bytecodes in the KVM require the ability to modify the bytecodes during execution, so they must be stored in a writable medium, i.e. not in ROM. Being able to store the API classes into ROM can make a device both cheaper and potentially more secure, since an attacker cannot modify the classes. Tokenisation still allows for quick dispatch times, without needing to modify the code at runtime, therefore still allowing it to be stored in ROM.

In a thesis produced by Martin Schöberl [76] he examines the execution times for various Java virtual machine implementation. Table 5.14 shows the times reported for *invokevirtual* and *invokeinterface* from [76]. No details are reported on how much of the invoke process this represents or how these values were obtained and it is assumed these involve the entire invoke process (i.e. from the start of interpreting the invoke instruction, until the next bytecode is ready to start), making comparisons with the results found here difficult. Since the measurement of the KVM has given results that appear to be in line with other virtual machine implementations as reported in [76]), it would suggest that the measurements presented here are reasonable.

5.8 Conclusions

The implementation of a basic virtual machine has allowed for the execution of tokenised class files, which produced the same results as executing the standard class files on the standard virtual machine, showing that the tokenised classes can still be executed correctly. Analysis of the test application also showed it was non-trivial, using a significant portion the classes and the Java instruction set. Not only can tokenised classes be executed correctly, but testing has shown them to be between 4-45 times faster than the KVM for resolving the target method.

Other improvements that result from a tokenised virtual machine include:

- The *invokevirtual* and *invokeinterface* instructions can share nearly all of their implementation (with *invokeinterface* only being different in the rare case of a conflict entry in the virtual method table).

- Removes the need to update bytecodes on-the-fly with “fast” versions, which testing showed adds 100 or more cycles to the execution of the instruction each time these entries are made and also requires the bytecodes be stored in a writable medium.

- Removes the dependence on the constant pool from the *invoke** implementation, removing the need for mechanisms such as caching of constant pool entries used by the KVM to speed up these operations (although currently loading constant values could still benefit from cached constant pool values).

Chapter 6

Tokenised Class File Generation and Compression

6.1 Introduction

The previous chapters have presented the tokenisation scheme used to allocate tokens to methods and fields in class files. These are then stored in tokenised class files, a modification of the existing class file format. During the creation of this new tokenised file format, compression opportunities have also been identified. The original goal when creating the class file format was to create files which are self-contained and easily extensible, but this resulted in larger files. Since tokenisation tightly couples a set of class files, each file no longer needs to be self-contained, allowing better use of space within the files.

The following sections detail the changes that have been made to the class file format and is presented in the same order as entries are found in a class file, as defined in the Java Virtual Machine Specification [57]. Most of the changes result from using numeric tokens for linking and therefore the removal of the old linking symbols. The remainder of the changes have been done to either simplify the class files, or to reduce their size.

6.2 Constant Pool Entries

The constant pool is not just used to store constant values needed by the program, but also all the linking symbols. There are several different types of entries that can occur in the

Table 6.1: Standard constant pool entry type

Tag	Description	Standard	Tokenised
1	CONSTANT_Utf8	X	X
3	CONSTANT_Integer	X	X
4	CONSTANT_Float	X	X
5	CONSTANT_Long	X	X
6	CONSTANT_Double	X	X
7	CONSTANT_Class	X	X
8	CONSTANT_String	X	
9	CONSTANT_Fieldref	X	
10	CONSTANT_Methodref	X	
11	CONSTANT_InterfaceMethodref	X	
12	CONSTANT_NameAndType	X	
13	CONSTANT_ArrayClass		X

constant pool, with the type of each entry denoted by a 1 byte "tag" value at the start of the entry. Table 6.1 shows the various types of entries and if they are found in standard class files, tokenised class files or both. Some of the standard entries are no longer needed, due to tokenisation, while others are modified or shortened.

The CONSTANT_Utf8 entries are used to store string data, including strings used within the program and those used for linking. While constant values used by the program must remain, those used for linking are no longer needed, providing a large space saving.

The CONSTANT_Integer, CONSTANT_Float, CONSTANT_Long and CONSTANT_Double entries are all used to store a constant value of the given primitive type. For the BYTE, CHAR, and SHORT types, they will be stored in a CONSTANT_Integer entry. Again, since these are constant values used by the program at runtime, they cannot be removed, and are maintained in the same format as for standard class files.

The following sections cover the remaining types of entries and the changes that have been made to them.

```
CONSTANT_Class_info {

    u1 tag = 7;
    u2 name_index;

}
```

Figure 6.1: Standard CONSTANT_Class_info structure

```
CONSTANT_Class_info {

    u1 tag = 7;
    u2 token_value;

}
```

Figure 6.2: Tokenised CONSTANT_Class_info structure

6.2.1 CONSTANT_Class

In Java, a reference to a class and to an array class both use a CONSTANT_Class entry, which in turn will reference a CONSTANT_Utf8 entry with the symbolic string to represent the class or array. For a class, the string is the class's fully qualified name, while an array's name will start with the character '[' (which can not be part of a class's name) and consists of a special syntax to indicate the type of values the array stores. A CONSTANT_Class entry in a standard class file has the structure given in Figure 6.1. The *tag* value is used to differentiate the types of the entries, with the value 7 indicating a CONSTANT_Class entry, and the *name_index* is the index of the UTF8 entry in the constant pool. The tokenised CONSTANT_Class_info entry is the same as before, however the *name_index* value has changed meaning and now represents the token value, giving the structure in Figure 6.2.

A tokenised class will not contain strings as classes are referenced via a token value. To represent arrays, a new type of entry was created, since a single token could not contain enough information to define the array, and there are too many possible types of arrays to allocate tokens to them all.

To represent an array, a new CONSTANT_Array_Class_info entry has been created, as shown in Figure 6.3. The *tag* value of 13 is used, as this is not in use in standard class files.

```
CONSTANT_Array_Class_info {
    u1 tag = 13;
    u1 type;
    u2 dimensions;
    u2 class_token;
}
```

Figure 6.3: Tokenised CONSTANT_Array_Class_info structure

Table 6.2: Possible values for the *type* entry in a CONSTANT_Array_Class_info entry.

Value	Description
1	Primitive *boolean* type.
2	Primitive *char* type.
3	Primitive *byte* type.
4	Primitive *short* type.
5	Primitive *int* type.
6	Primitive *long* type.
7	Primitive *float* type.
8	Primitive *double* type.
9	Object reference type.

The *type* entry is used to describe the type of values stored in the array, which can consist of one of the Java primitive types, or an object reference. Table 6.2 shows the possible values for the *type* entry. The *dimensions* value indicates the number of dimensions in the array, i.e. 1 is a standard array, while 2 would be an array with 2 dimensions and so on. This value must be >0. Finally, the *class_token* value gives the types of objects that are stored in this array if the *type* entry is 9 (i.e. the array stores objects of some type), otherwise the array will store primitive values and the *class_token* value must be 65,535.

6.2.2 CONSTANT_String

A CONSTANT_String entry can be referenced from either a *ConstantValue* attribute or the *ldc* and *ldc_w* instructions. In the first case, the *ConstantValue* attribute will be associated with a field, and indicates that the field needs to be initialised with a string value (see Section 6.3 for a discussion about fields, and Section 6.5.1 for information about the Constant-

```
CONSTANT_String_info {
    u1 tag = 8;
    u2 string_index;
}
```

Figure 6.4: Standard CONSTANT_String_info structure

Value attribute). The *ldc* and *ldc_w* instructions are used to push a reference to the string onto the evaluation stack (Section 5.3.1 provides more information on the modifications to the instruction set). The Java Virtual Machine Specification defines a CONSTANT_String entry with the structure given in Figure 6.4. The *tag* value of 8 indicates this is a CONSTANT_String entry, while the *string_index* will be the index in the constant pool of a CONSTANT_Utf8 entry that contains the actual string data.

Since a CONSTANT_String entry exists only to provide an index to a CONSTANT_Utf8 entry, it has been removed and instead the 3 locations (ConstantValue attribute, ldc instruction and ldc_w instruction) that can reference a CONSTANT_String entry, have been updated to reference the appropriate CONSTANT_Utf8 entry. This has three benefits, firstly it saves 3 bytes for each CONSTANT_String entry, secondly it reduces the number of entries in the constant pool and finally it reduces the amount of indirection.

The side-effect of using less values in the constant pool is a space saving in other locations. Some instructions that reference constant pool entries, such as the *ldc* instruction, use a single byte value when the referenced index is <256. If there are more entries in the constant pool, then the longer *ldc_w* instruction with a two byte index will be needed. Therefore, minimising the number of entries in the constant pool leads to space savings in the bytecode for that class's methods, due to having smaller index values.

6.2.3 CONSTANT_NameAndType

A symbolic method or field reference requires three strings in a standard class file, the class name, the method or field name and the type (in the case of a method, the type is the number and type of parameters). Such a reference is represented by a CONSTANT_*ref entry in

```
CONSTANT_NameAndType_info {

    u1 tag = 12;
    u2 name_index;
    u2 descriptor_index;

}
```

Figure 6.5: Standard CONSTANT_NameAndType_info structure

```
CONSTANT_*ref_info {

    u1 tag;
    u2 class_index;
    u2 name_and_type_index;

}
```

Figure 6.6: Standard CONSTANT_*ref_info structure

the constant pool (covered in the next section), however the method/field name and type are provided by a CONSTANT_NameAndType entry, which has the form given in Figure 6.5. The *name_index* value is the the index of a CONSTANT_Utf8 entry, giving the name of the method or field. The *descriptor_index* value is the index of another CONSTANT_Utf8 entry, giving the descriptor of the method or field.

6.2.4 CONSTANT_*ref

There are different types of entries used for symbolic references to fields, methods or interface methods, namely, the CONSTANT_Fieldref, CONSTANT_Methodref and CONSTANT_InterfaceMethodref. These entries are referenced from certain bytecode instructions, and in the case of a CONSTANT_Fieldref, from *getfield* and *putfield* for reading or writing to a field in an object and *getstatic* and *putstatic* for reading or writing a static field. A CONSTANT_Methodref is referenced from the *invokevirtual*, *invokespecial* and *invokestatic* instructions, and a CONSTANT_InterfaceMethodref from an *invokeinterface* instruction. In all cases, the bytecodes for these instructions will include an operand which is an index to the relevant entry in the constant pool. All three entries have the form given

in Figure 6.6.

There are three symbols involved in resolving a symbolic reference: Class name, Field or method name and Field or method descriptor. For example, an invoke instruction must reference the target method, and the three symbols are obtained through several layers of indirection from the instruction. The invoke instruction (or any other structure referencing a field, method or interface method) contains an index to the constant pool to a CONSTANT_*ref entry (where the exact type of entry depends on if it is a field, method or interface method being referenced). The *class_index* value in the CONSTANT_*ref entry provides the index of a CONSTANT_Class entry, which in turn contains the index of a CONSTANT_Utf8 entry that contains the class name, thus providing the first symbol. Next, the *name_and_type_index* value provides the index of a CONSTANT_NameAndType entry, which in turn contains the index of two CONSTANT_Utf8 entries, thus providing symbols two and three. However, after tokenisation, none of these string values are needed, as they are replaced by tokens.

In the case of a static field, only the class token and field token are needed, and for a non-static field, just the field token. CONSTANT_Fieldref entries are only referenced from *getfield, putfield, getstatic* and *putstatic* instructions. Since the tokens are the same size as a constant pool reference, it makes sense that they are included directly in the bytecode as operands to the above instructions (details on instruction modifications are in Section 5.3). With the required tokens stored as operands to bytecodes, CONSTANT_Fieldref entries are no longer needed in tokenised class files and are therefore removed.

Similarly, the four invoke instructions (*invokevirtual, invokespecial, invokestatic* and *invokeinterface*) also now include the relevant tokens as operands, again removing the need for constant pool entries and reducing indirection. Therefore none of the CONSTANT_*ref entries are now needed in tokenised class files. In turn, since CONSTANT_NameAndType entries were only referenced from CONSTANT_*ref entries, they also have been removed, along with any strings they used.

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.7: Standard field_info structure

```
field_info {
    u2 access_flags;
    u2 token;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.8: Tokenised field_info structure

6.3 field_info Section

Following the constant pool is the *field_info* section, which lists information for each field declared in the class file. The standard field section is shown in Figure 6.7. The *access_flags* entry provides 16 bit-flags, of which only seven are used in the specification, with the remaining nine reserved for future use. The seven used flags consist of: public, private, protected, static, final, volatile and transient. Of these seven only some combinations are legal (i.e. a field can not be both *public* and *private* at the same time).

The next two entries, *name_index* and *descriptor_index,* both provide an index to a CONSTANT_Utf8 entry in the constant pool, giving the symbolic name used to reference the field and the type of values stored in the field. Finally, additional attributes can be associated with the field, with the Java specification allowing three such attributes: ConstantValue, Synthetic and Deprecated (more details on attributes are in Section 6.5).

Since tokenisation removes the name and descriptor strings, these two entries are re-

Table 6.3: Meaning of Bits in the *access_flags* component of *field_info* entries

(a) Bit Map For *access_flags* Entry

Bit	Usage
0	Public
1	Private
2	Protected
3	Static
4	Final
5	Reserved
6	Volatile
7	Transient
8-11	Type
12-15	Reserved

(b) Meaning of Type Value For Fields

Value	Type
0	Boolean
1	Char
2	Byte
3	Short
4	Int
5	Long
6	Float
7	Double
8	Object

placed with a single token value, resulting in the structure given in Figure 6.8 for tokenised class files. With the removal of the descriptor, the virtual machine would be unable to determine the type of values stored in a field and therefore some form of type information must be added. Instead of adding an extra value, and using extra space, the unused bits in the *access_flags* are used. There are nine possible types that must be represented requiring the use of a 4-bit field.

The bit-map for the *access_flags* field is given in Table 6.3a. Bits 0 to 7 are unchanged from standard class files, with the type information added into bits 8-11. The 4-bit type value is read as an unsigned value. Table 6.3b shows the mapping from values to types. For fields that store objects, the type of class is not stored, just that the field will be holding an object reference of some type.

6.4 method_info Section

Following the fields section there is a method section, containing a *method_info* structure for every method defined in the class. The standard *method_info* structure in a class file is very similar to a *field_info* entry and is given in Figure 6.9. The *access_flags* field consists of a sequence of bit-flags, which is discussed in detail in Section 6.4.1. The name and

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.9: Standard method_info structure

```
method_info {
    u2 access_flags;
    u2 token;
    u2 arg_count;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.10: Tokenised method_info structure

descriptor entries retain the same purpose, providing indexes to UTF8 entries in the constant pool to provide the method's name and descriptor. Finally, the specification allows the *Code*, *Exceptions*, *Synthetic* and *Deprecated* attributes to be associated with a method (more details on Attributes are given in Section 6.5).

With tokenisation, the name and descriptor strings, and therefore the indexes that refer to them, are no longer required and are replaced with the method's token value and argument count respectively, giving the structure in Figure 6.10. The tokenised method info section has the same size as the standard method info section, but without the need for the method name and descriptor strings in the constant pool. The token value is simply the token that represents the method, while the *arg_count* value is required during method dispatch. A method call is performed by pushing the arguments onto the operand stack of the caller method, then executing an invoke* bytecode. Previously the VM could use the descriptor string to know how many arguments a method had and therefore how many of the

```
class A {
    public byte m(int b, float c, Object d) {
        return 5;
    }
    public static void main (String args[]) {
        A var = new A();
        var.m(1, 2.0, new Object());
    }
}
```

Figure 6.11: Example of code that performs an invokevirtual.

values on the operand stack were arguments for the method call. Since the descriptor string has been removed, the *arg_count* value provides the VM with the missing information.

Figure 6.11 shows an example program that declares a variable “var”, creates an object, and then performs a method call on that object. Figure 6.12 shows the state of the top frame on the call stack before, during and after the method call. The *main* method will first push the reference to the object to call the method on (the “var” variable), followed by the three parameters. The invokevirtual instruction ensures that the parameters, as well as the reference to the “var” object, are copied into the local variables for the new method frame for the *m* method. In this case the “var” becomes local variable 0, which is used to implement the “this” keyword in Java. The bytecode of the method *m* will assume that the parameters have already been assigned to local variables 1-3, and will reference them as such. When the method returns, its return value will be pushed onto the stack of the caller, as shown in the last box where the *main* method no longer has the parameters, but now has the return value of 5 on top of its operand stack.

6.4.1 access_flags

The *access_flags* value in the *method_info* structure is used to represent a series of bit flags that give the access information of the method. The allowable values are defined in Section 4.6 of the Java Virtual Machine Specification [57] and are shown in Table 6.4. This section

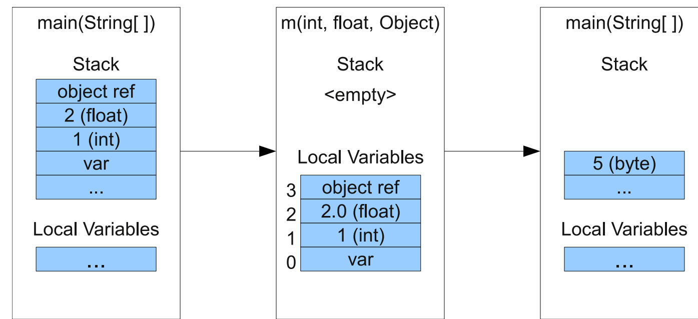


Figure 6.12: Call to method var.m(int a, float b, Object c)

describes additional flags needed for execution of the tokenised class files.

As with fields, the method name and descriptor strings provided more information than just a unique name, namely they also identified constructors and main methods. The unused bit flags in the *access_flags* field have been used to represent the information needed by the virtual machine, and are shown in Table 6.5. When starting an application, the VM is typically given only the class that should be used to start execution. The VM will search in that class for a method with the name “main”, that takes an array of String objects as a parameter. Since method names have been removed during tokenisation, the virtual machine can find a static method that takes one String array parameter, but can not know if the original method had the name “main” or not. Therefore, the ACC_MAIN flag is set at conversion time to indicate any method that is a main method. Since there can only ever be at most one such method per class, and the virtual machine is provided with the class during startup, the VM will search that class for the first method with the ACC_MAIN flag set.

Classes can optionally contain a class loader initialisation method, that will be executed by the class loader after loading the class, but before it is used. The class loader initialisation method is generated by the compiler and has the method name “<clinit>”, which is not a valid name in a Java source file, but is valid within a binary class file. Any default values assigned to static fields in the class, or any code within a *static* block in the source code, will be placed inside the <clinit> method by the compiler. The ACC_CLINIT flag allows the class loader to find any such method (there can be at most one such method per class file and it never takes any parameters) and if found, execute it.

Table 6.4: *access_flag* values from the Java Virtual Machine Specification.

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared *public*; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared *private*; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared *protected*; may be accessed within sub-classes.
ACC_STATIC	0x0008	Declared *static*.
ACC_FINAL	0x0010	Declared *final*; may not be overridden.
ACC_SYNCHRONIZED	0x0020	Declared *synchronized*; invocation is wrapped in a monitor lock.
ACC_NATIVE	0x0100	Declared *native*; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared *abstract*; no implementation is provided.
ACC_STRICT	0x0800	Declared *strictfp*; floating point mode is FP-strict.

In a Java source file, constructors must have the same name as the class, however they are compiled to a method with the name <init> in the binary class file (again because "<init>" is not a valid method name in the source code, hence identifying the method as a compiler generated constructor). There can be multiple init methods, one for each constructor that was declared in the source code, with each taking different parameters. The ACC_INITV flag indicates a default constructor, this is one which takes no parameters, while the ACC_INIT flag indicates a constructor that will take one or more parameters.

The final flag, ACC_INITS, indicates a constructor that takes a single String object as a parameter. The Java Virtual Machine Specification requires that certain operations cause the virtual machine to throw an exception (e.g. performing a method call on a null object requires a NullPointerException to be thrown). Exception classes contain a constructor which takes a single String argument with a descriptive message. The ACC_INITS flag identifies the constructor which takes this single String parameter. Without this flag, the VM can not identify the appropriate constructor and must resort to the default constructor, producing exceptions with no clarifying messages.

Table 6.5: Additional *access_flag* values added to the tokenised VM.

Flag Name	Value	Interpretation
ACC_INITV	0x0200	This is an <init> method that takes no parameters (i.e. a default constructor).
ACC_MAIN	0x1000	This is a main method.
ACC_CLINIT	0x2000	This is a <clinit> method.
ACC_INIT	0x4000	This is an <init> method that takes parameters.
ACC_INITS	0x8000	This is an <init>(Ljava/lang/String;)V method (i.e. constructor that only takes a single String parameter). Used by the VM to build Exception objects with message strings.

6.4.2 Static Methods

Standard Java class files store static and non-static methods within a single method table. Since non-static and static methods are called by different bytecode instructions, they are stored separately within a tokenised class file.

Section 3.6.2 discusses how a method token for a static method only needs to be unique within the class that contains that static method. Thus the static method tokens do not need to be referenced from the virtual method tables, leading to two types of tokens: *method tokens* for virtual methods and *static method tokens* for static methods. The method tokens must be allocated using the algorithms described earlier in this thesis, to ensure virtual method tables can be built appropriately. However, static method tokens only need to be unique within a given class. Differentiating between static and non-static methods simplifies the virtual method tables, since tokens do not need to be allocated to static methods, reducing the size of the table. Also, since static method tokens only need to be unique within the class, they can indicate the method's position within the static methods section of the tokenised class file, reducing the need for the virtual machine to search for the target method. The entries in this table have the exact same form as those in the non-static method table, which was described above.

```
attribute_info {
    u2 attribute_name;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

Figure 6.13: Standard attribute_info structure

6.5 Attributes

Attributes are used to add extra data to a class, method, field or even another attribute. Each attribute has a standard form, consisting of an identifier for the type of attribute, its length and the attribute's data. The Java Virtual Machine Specification [57] defines a set of attributes which all virtual machines must recognise. Implementers are free to create custom attributes and a virtual machine must ignore any attributes it does not understand. However, any custom attributes cannot modify the classes semantics, but can provide things like extra debugging or optimisation information. The format for attributes in a standard class file is given in Figure 6.13. The *attribute_name* value is a 2 byte index to a string in the constant pool which defines the type of the attribute and hence how to interpret any data in it. The length entry defines how many bytes of data are present in the *info* entry, allowing a class file reader to skip over any data in an unknown attribute. Finally, the *info* array contains the data of the attribute, how that data is interpreted is determined by the type of the attribute.

Since an attribute not defined in the Java Virtual Machine Specification is not allowed to affect the semantics of a class file, it is safe to remove it during tokenisation, leaving the known set of attributes defined in the Java Virtual Machine Specification. String names for attributes can take many bytes, however a simple 1-byte tag is sufficient to represent the required attributes, saving the 2 bytes for a constant pool reference and the many bytes needed for a UTF8 entry in the constant pool. Since only those attributes required by the specification are allowed, the format of all the attributes is known, also removing the need for the 4-byte length field.

Table 6.6: Class File Attributes

Tag	Attribute Type	Original Attribute Name
1	ConstantValue (1 byte)	ConstantValue
2	ConstantValue (2 bytes)	ConstantValue
3	Code	Code
4	Exceptions (1 byte)	Exceptions
5	Exceptions (2 bytes)	Exceptions
6	InnerClass	InnerClass
7	Synthetic	Synthetic
8	Deprecated	Deprecated
9	StackMap	StackMap
10	VMT	–
–	–	SourceFile
–	–	LineNumberTable
–	–	LocalVariableTable

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantValue_index;
}
```

Figure 6.14: Standard ConstantValue_attribute structure

Table 6.6 shows the names of the attributes in standard class files, and the tag value used by these in tokenised class files. In the case of the ConstantValue and Exceptions attributes, two different versions with different sized fields were added in tokenised class files. The VMT attribute is a new addition to store a class's virtual method table, while the SourceFile, LineNumberTable and LocalVariableTable attributes have all been removed. The following sections cover the use and format of each attribute in more detail.

6.5.1 ConstantValue Attribute

The ConstantValue attribute can be attached to a field_info section of a static field and denotes the value that field must have after class loading/initialising. For non-static fields, the compiler will add code to the constructor to ensure any initial values are assigned. A

```
ConstantValue1_attribute {
    u1 tag = 1;
    u1 constantvalue_index;
}
```

Figure 6.15: Tokenised ConstantValue1_attribute structure

```
ConstantValue2_attribute {
    u1 tag = 2;
    u2 constantvalue_index;
}
```

Figure 6.16: Tokenised ConstantValue2_attribute structure

standard ConstantValue attribute has the form given in Figure 6.14. The data part consists of a 2-byte constant pool reference, which can point to any of the value types in the constant pool (i.e. integer, long, float, double, string). In the case of a CONSTANT_String entry, it merely contains an index to a CONSTANT_Utf8 entry. Therefore, if a ConstantValue attribute points to a CONSTANT_String, it will be updated to point directly to the CONSTANT_Utf8 entry, removing the need for the CONSTANT_String entry.

For tokenised class files, two versions of this attribute were defined, the first with a 1-byte constant pool index, the second with a 2-byte index. Since tokenisation reduces the number of entries in the constant pool, often only 1-byte indexes are required, saving space. The 1-byte version has the form shown in Figure 6.15, while the 2-byte version is shown in Figure 6.16.

During testing it was found that all except the J2SE API could be tokenised using only the 1-byte version of this attribute, meaning the 2-byte version is only needed for compatibility with large applications or libraries. The use of two different length indexes does add to the complexity of the device however, since it must deal with the two different sizes, but the trade-off seems appropriate to give support to larger applications while reducing the size of all other applications.

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.17: Standard Code_attribute structure

6.5.2 Code Attribute

The Code attribute is used to store the bytecodes and associated information needed to execute a method, which is only required by non-abstract methods. It is provided as an attribute so it will not take up space in an abstract or interface methods. The format for a standard Code attribute is given in Figure 6.17. The *max_stack* and *max_locals* provide the maximum depth of the operand stack and the maximum number of local variables used by the method, respectively. The *code* entry provides the bytecodes for the method, while the exceptions table provides the details needed to implement try/catch blocks. The *catch_type* value is a constant pool index to a CONSTANT_Class entry, denoting the type of exception this block catches. If that exception is thrown from one of the instructions between *start_pc* and *end_pc*, then execution should jump to *handler_pc*. A Code attribute can contain additional attributes, however the specification only defines two that can appear, both consisting of debugging information (the LineNumberTable and LocalVariableTable attributes). The

```
Code_attribute {
    u1 tag = 3;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

Figure 6.18: Tokenised Code_attribute structure

CLDC Specification [63], which extends the Java Virtual Machine Specification, defines the StackMap attribute, which must be present in any Code attribute. The StackMap attribute provides details on the state of the stack at certain points during the method and can optionally be used by the virtual machine to simplify verification of the class file.

The tokenised version of the Code attribute has the form given in Figure 6.18. The attribute name is replaced by the appropriate tag value (as discussed in Section 6.5), while the *catch_type* entry, which previously indexed the constant pool to define the type of exception, is replaced with the class token, removing the need to dereference the constant pool. Finally, some of the instructions have had their formats modified and some will change length as a result of tokenisation (Section 5.3 covered the changes to individual instructions in detail).

Branch instructions store the branch target as a signed offset of the number of bytes to jump forwards or backwards in the byte array. Since tokenisation can change the length of instructions, this can change the relative offset of a branch target, requiring the branch

offset to be updated. The following sections go into detail on updating the bytecodes.

6.5.2.1 General Format of Java Instructions

Each Java instruction contains an unsigned byte value as the opcode of the instruction, followed by zero or more operands (with the number or operands determined by the opcode). There are 256 possible instructions, 201 of which are used (the values from 0 to 201, with 186 not used for historical reasons), with the others being reserved for special use (i.e. debuggers and VM implementers) or reserved for future use. Most instructions have a fixed number of operands, however some, like *tableswitch* or *lookupswitch,* have a variable length. Therefore, to determine which bytes in the bytecode array are instructions and which are operands, the bytecode must be read in a serial manner, starting from byte 0, which is always, by definition, an instruction. Using the opcode of the first instruction, the number of operands can be determined and therefore the location of the second instruction, and so on, until the bytecode array has been processed.

The branching instructions, such as *goto* for unconditional branching, the various *if** instructions for conditional branching, *jsr* and even the aforementioned *tableswitch* and *lookupswitch* all provide, as an operand, an offset in bytes from the start of that instruction to the jump point. The byte at that position must be an instruction and therefore cannot be an operand to another instruction.

6.5.2.2 When Instructions Need to Change

The field and invoke instructions have new formats in tokenised class files, and a direct result of tokenisation can cause instructions to change size. There are three main causes for this:

Tokenisation Certain instructions get longer or shorter after tokenisation, with constant pool references being replaced with tokens. In particular, the field and invoke instructions (full details are in Section 5.3).

Constant pool changes With the removal of the string linking data from the constant pool, there are less entries and therefore many constant pool references will be to smaller

indexes. Some instructions that reference the constant pool have two forms, a “wide” form that uses two-byte index (allowing it to reference any constant pool entry), and a “narrow” form with a one-byte index. Since constant pool indexes are smaller, some instructions can be changed to the narrow version, reducing the code size.

Instruction changes Many branching instructions also have narrow or wide versions, with the branch target given as a number of bytes to jump forward or back in the instructions. With the other changes in the instructions above, these branch targets can get either closer or further away, in some cases requiring a change from a narrow to a wide, or wide to a narrow version of the instruction.

As mentioned above, whenever any instruction changes size, any branch instruction on one side of of the change that targets an instruction on the other side, must have its offset updated. Updating the branch targets is further complicated since updating one of the branch instructions may in turn cause its size to change, requiring further updates. Therefore updating the bytecodes was achieved with a multi-pass process, detailed in the following section.

6.5.2.3 Process to Update Bytecode

The bytecode is represented in memory as a single byte array (the same as it is stored in a standard binary class file) and consists of a mix of instructions and operands. Some operands constitute an offset to another bytecode instruction. If instructions change length, then these offsets can require adjustment. To perform the update, each instruction in the method is numbered, starting with instruction 1 at byte 0. Since instructions are not added or removed, this instruction number will not change (i.e. if a jump instruction will jump to the 10th instruction, it will always be the 10th instruction, even if its byte position has changed). Therefore, the instruction number can be used to reference instructions independent on the number of bytes between a pair of instructions. The first pass for updating the bytecode consists of overwriting the byte offset (which can be positive or negative) with the instruction number that is the target of the jump.

Next, a new byte array is created, copying instructions over one by one, converting each

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

Figure 6.19: Standard Exceptions_attribute structure

instruction as necessary. The new byte array will then have wider or narrower instructions as needed, and as such could be longer or shorter than the original. During the construction of this new code array, the offset of where each instruction is now located will also be generated, resulting in a modified bytecode array (where branch targets are given by the instruction number) and an array indicating the index of each instruction in the new array.

Using the new location for each instruction, a final pass is made to replace each of the instruction numbers with the new offset. Using the array of instruction locations and the instruction which is being jumped from and that of the one being jumped to, the new offset can be calculated.

Each of these passes is performed in a serial fashion, starting at the first instruction at byte 0, and progressing through the byte array. The first and last pass can happen in-situ, just overwriting existing values. The middle pass, when the instructions are updated requires a second array, since some instructions can become longer.

6.5.3 Exceptions Attribute

Each *method_info* structure can have, at most, one Exceptions attribute to indicate the checked exceptions that the method can throw. If the method does not throw any checked exceptions, then this attribute can be left out. In a standard class file, this attribute has the form given in Figure 6.19. Where the *exception_index_table* stores a list of constant pool references, each to a CONSTANT_Class entry. Each of these classes will be a checked exception that the method may throw during execution.

In standard class files, the use of string names for classes necessitates the use of the

```
Exceptions1_attribute {
    u1 tag = 4;
    u2 number_of_exceptions;
    u1 exception_table[number_of_exceptions];
}
```

Figure 6.20: Tokenised Exceptions1_attribute structure

```
Exceptions2_attribute {
    u1 tag = 5;
    u2 number_of_exceptions;
    u2 exception_table[number_of_exceptions];
}
```

Figure 6.21: Tokenised Exceptions2_attribute structure

constant pool. Tokenised class files however make use of fixed width class tokens, making references to the constant pool unnecessary, with the class tokens instead being stored directly in the Exceptions attribute. A Class token consists of a 16-bit unsigned integer, however, for smaller packages it is rare to get tokens larger than an 8-bit unsigned integer. Therefore, it is possible to reduce the size of this attribute by having two versions, one with a list of 16-bit values, the other with a list of 8-bit values. If all class tokens are small enough, the 8-bit version can be used to save space. Figure 6.20 shows the form when all tokens can be represented with 1 byte. If one or more of the class tokens is greater then 255, then all of them are stored as unsigned two-byte values in the structure shown in Figure 6.21. Although this will slightly increase the complexity in the device when it must handle the two lengths for class tokens, it allows for additional space saving.

6.5.4 InnerClass/Synthetic Attribute

The InnerClass and Synthetic attributes were added in Java 1.1 to support nested classes and interfaces [57]. The InnerClass attribute indicates which classes are an inner class or interface, while the Synthetic attribute marks classes, fields or methods that did not exist

in the source code and were generated by the compiler. The constant pool references in the InnerClass attribute are updated to tokens, as with other attributes, while the Synthetic attribute is reduced to a single byte for its tag value, since it is a marker and contains no data.

6.5.5 Debugging Attributes

There are three attributes that are used purely for debugging of class files, the SourceFile, LineNumberTable and the LocalVariableTable attributes. The SourceFile attribute is attached to a class and stores the name of the original source file that the class was compiled from, to allow a debugger to match up the compiled class with its source file. The LineNumberTable and LocalVariableTable are attached to a CodeAttribute. The LineNumberTable indicates the line number in the source code and which instruction that maps to, so that a debugger can break at the instruction for a given source line, or show the current source line being executed. When a class is compiled, fields will retain their name from the source code, however, local variables are compiled into indexes into the local variable table. The LocalVariableTable attribute maps these indexes back to the variable name in the source code, so a debugger can show the values of these local variables along with their names during debugging.

Since all three of these attributes are only used by the debugger, they are not going to be of use on a production device. To save space, the converter will remove them entirely during conversion.

6.5.6 Deprecated Attribute

The Deprecated attribute is used to mark a class, field or method that has been declared deprecated. Compilers or other tools that read class files use this attribute to warn the programmer that they have used a class, field or method that is deprecated. It is then up to the programmer to find appropriate documentation on what new classes, fields or methods have been added to replaced the deprecated ones.

Since the deprecated attribute is purely a marker, with no additional information, the

tokenised version consists of the 1-byte tag value and no data, making the attribute only one byte long.

6.5.7 StackMap Attribute

The StackMap attribute is not a part of the standard J2SE Specification, rather it was added by the CLDC Specification [63]. Because of the complexity and memory requirements of the class file verifier from J2SE, it is often not feasible for a CLDC device to implement the entire verifier. Therefore, Sun define a simplified class file verifier that can provide the same service with less complexity through the use of the StackMap attribute. The specification requires that classes that will be used on a CLDC device must contain StackMap attributes, while virtual machine implementers can choose to implement either the reduced verifier or the full J2SE verifier (but they must implement at least one).

The StackMap attributes provides the types stored in a method's local variables and on the method's operand stack at the start of each basic block within a method. The format of the StackMap attribute is relatively complex, however the only important part from a tokenisation perspective are references to CONSTANT_Class entries in the constant pool to define the type of objects found on the stack or in local variables. Since CONSTANT_Class entries will remain after tokenisation, the tokeniser only needs to update the references as appropriate if they have moved within the constant pool.

6.5.8 VMT Attribute

The VMT attribute stores an encoded virtual method table. These do not exist in standard Java class files and are added by the tokeniser as part of the tokenisation process. Section 4.6.3 showed the format for an encoded VMTEntry, which can be one of a *null*, *single* or *conflict* entry. A VMT attribute will be present in any non-interface class file, with exactly one VMT attribute per file.

The format for a VMT attribute will consist of:

```
VMT_Attribute {
```

```
        u1 tag;
        u2 length;
        VMTEntry entries[length];
}
```

The *length* field denotes how many entries are in the virtual method table, followed by that many VMTEntry's. The use of the VMT attribute was discussed in Section 5.2.

6.6 Code Compression

During tokenisation, the strings used for symbolic references to classes, fields and methods are all removed and replaced with tokens. Also, some previously used constant pool references now store the token value, thereby further reducing the number of constant pool entries per class. These changes result in significantly smaller files, with the following sections detailing the compression results. For testing, the same packages that were used to test tokenisation (originally presented in Section 3.9) were also used for measuring compression.

Jar files can include more than just class files, they can also include properties files, pictures, sound or other data files used by the application or library. A Jar file will store these additional files with zip compression, the same as class files, however tokenisation of the class files will have no effect on these extra data files. Therefore, these extra files were removed and the Jar files recreated, containing just the class files.

6.6.1 Overall Code Compression for Global Tokenisation

The first set of results focus on global tokenisation and the size of resulting class files, presented in Table 6.7, with all sizes given in bytes. The first column is the size of the uncompressed class files stored in the appropriate folders for their package, within a normal filesystem. The sum of the length of each class file was used to arrive at this figure, which therefore ignores any filesystem overheads.

Table 6.7: Overall Size of class files (bytes)

Test Case	Original Size	Jar File Size	Tokenised Size	% of Original Size	% of Jar Size
CLDC1.0	158804	102248	89110	56.11%	87.15%
CLDC1.0+MIDP	901875	486127	489725	54.30%	100.74%
CLDC1.0+MIDP+ MIDPExamples	1166447	852625	620793	53.22%	72.81%
CLDC1.0+ Javolution3	627758	369924	311637	49.64%	84.24%
CLDC1.1	134109	85154	81393	60.69%	95.58%
CLDC1.1M	134960	85467	82069	60.81%	96.02%
CLDC1.1M+ Javolution5	676619	418644	305602	45.17%	73.00%
J2SE	29071272	15411892	17903020	61.58%	116.16%

The second value is the size of a Jar file containing just the class files and with compression enabled. The Jar file size is considered since this is the standard packaging format for Java and is defined in the Jar File Specification [60]. Therefore, this column represents the size of each package as it would normally be distributed (although with any additional data files removed from the Jar file, as noted in Section 6.6).

The tokenised size is the sum of the size of each tokenised class file, calculated as for the Original Size column, except using the tokenised files. The final two columns compare the size of the tokenised files against the original class files and the Jar file respectively.

The results show a significant size reduction compared to the original class files and in most cases the tokenised files are smaller than the compressed Jar file of the standard files. Tokenised class files are also directly executable, however, since Jar files are compressed,each file must first be uncompressed to memory before being accessed, adding a processing delay and memory overhead. By storing uncompressed standard class files (either directly or in a Jar file with no compression), the need to decompress each class is removed, however, with an increased amount of storage needed. Therefore, tokenisation can offer similar or better space savings than a Jar file, but without the need to decompress each class before accessing it.

The tokeniser produces descriptor files along with the tokenised class files. The descriptor file for a package contains the string information required to link new classes into

Table 6.8: Size of Descriptor Files (bytes)

Test Case	Descriptor Size
CLDC1.0	43944
CLDC1.0+MIDP	205135
CLDC1.0+MIDP+MIDPExamples	244093
CLDC1.0+Javolution3	158366
CLDC1.1	36315
CLDC1.1M	36368
CLDC1.1M +Javolution5	179746
J2SE	4540374

Table 6.9: Comparison to Pack format (using pack200 binary from Sun) (bytes)

Test Case	Original Size	Jar File Size	Pack Size	% of Original Size	% of Jar Size
CLDC1.0	158804	102248	54597	34.38%	53.40%
CLDC1.0+MIDP	901875	486127	322657	35.78%	66.37%
CLDC1.0+MIDP+ MIDPExamples	1166447	852625	603758	51.76%	70.81%
CLDC1.0+ Javolution3	627758	369924	184739	29.43%	49.94%
CLDC1.1	134109	85154	48425	36.11%	56.87%
CLDC1.1M	134960	85467	48735	36.11%	57.02%
CLDC1.1M+ Javolution5	676619	418644	109098	16.12%	26.06%
J2SE	29071272	15411892	4913366	16.90%	31.88%

an existing tokenised set of classes (i.e. incremental tokenisation). Table 6.8 presents the size of the descriptor file produced for each of the test cases. However, the descriptor file is not needed by the virtual machine, only by the tokeniser for incremental tokenisation, meaning the descriptor file is not needed on the device. Since descriptor files are only needed by developers when producing new libraries or applications, their size is relatively unimportant.

While the Jar file format is the standard for storing, and in many cases distributing, collections of class files, a more efficient wire format known as Pack is also defined (originally discussed in Section 2.7.1.4). The Pack format is highly optimised to reduce the size of an application for transmission across a network, at the cost of requiring the entire archive to

Table 6.10: Overall Size of class files and Descriptor file after Incremental Tokenisation (bytes)

Test Case	Global Size	Incremental Size
CLDC1.0-MIDP	489725	489646
CLDC1.0-MIDP-MIDPExamples	620793	620735
CLDC1.0-Javolution3	311637	311890
CLDC1.1M-Javolution5	305602	305770

be unpacked before execution can take place. Table 6.9 compares the same test packages, but this time compressed using the 'pack200' binary that comes with the Sun JDK [64], instead of tokenisation. While pack does focus on the constant pool (in particular the string entries), since they make up a large percentage of the size of class files, it also compresses many other parts of the class file. Tokenisation has removed many of the strings that pack focuses on, however, as can be seen when comparing the results, pack achieves better compression than tokenisation alone. Since pack is not an executable format, it is unfair to compare it directly with tokenisation. However, the results suggest that the application of the various approaches used by pack could result in a tokenised wire-format that could achieve significant reductions in size, since it would not have to deal with as much string data.

6.6.2 Overall Code Compression for Incremental Tokenisation

While global tokenisation produced good compression, it remains to be shown that incremental tokenisation does not seriously alter the results. The test cases in the previous section where more than one package was tokenised together are considered again, however, this time the packages are tokenised incrementally. Table 6.10 shows the results from both the global and incremental tokenisation of the set of packages. As can be seen, the incremental tokenisation is slightly different than global tokenisation, although the difference is negligible. The slight change in size can be attributed to the different allocation of tokens, and inclusion of some conflict entries in the virtual method tables. Therefore, there is no significant difference in compression between global or incremental tokenisation.

6.6.3 Further Compression

The current work has focused on making the implementation of the *invoke** and *get/set* field instructions simple, so that a virtual machine implementation can be realised in hardware. The compression result is a side-effect of the tokenisation. While significant space savings have been realised, the format has not been optimised nor tailored specifically for size. Considering previous work in the area of compression (presented in Section 2.7), some of these approaches could be applied to the current work. Clausen *et al.* [24] shows how bytecodes can be compressed, while Bizzotto & Grimaud [13] extend the idea further. While the current work does modify some instructions, the majority of the instruction set is unchanged. Therefore, further compression of the bytecodes such as is presented in [24] and [13] is likely possible, and would still preserve the directly executable nature of the files.

6.6.4 Constant Pool Usage

Another effect of tokenisation is the removal of many of the constant pool entries. The CONSTANT_*ref entries for field, method and interface method references have all been removed. Similarly, the entries that held class, field and method names and descriptors are also gone. The result is that the number of entries in a given constant pool is much smaller.

Table 6.11 shows for the test packages, how many of each type of constant pool entry remain after tokenisation. The Class and ArrayClass entries remain to represent symbolic references to other classes (although using tokens now instead of strings). The UTF8, Integer, Long, Float and Double entries represent constant values that are used by the program and therefore must remain.

6.7 Conclusions

As a result of tokenisation, the class file format needed updating to store the appropriate token information, and to represent a class in the absence of the strings used to describe fields and methods in standard class files. These changes have resulted in class files that are

Table 6.11: Number of Constant Pool Entries by Type

CP Entry Type	UTF8	Class	ArrayClass	Integer	Long	Float	Double
CLDC1.0	336	184	37	118	0	33	0
CLDC1.1	157	148	30	90	20	64	50
CLDC1.1M	158	148	30	90	20	64	50
MIDP	1239	936	126	705	0	36	0
MIDPExamples	879	421	29	87	0	13	0
Javolution3	717	1065	49	103	0	25	0
Javolution5	870	1115	29	114	0	44	1
J2SE	101704	21777	1409	8692	460	1327	325

significantly smaller than standard class files, and of a similar size to a Jar file containing the standard class files. Unlike a Jar file however, the tokenised files can be executed directly in their current form, without needing to be decompressed first.

These compression results have been achieved primarily because of the removal of string data that was previously used for linking. Since compression was not the primary focus of this thesis, further steps to reduce the file size have not been taken. Depending on the target device, further compression could be applied to reduce the file size further, following the ideas that presented in Section 6.6.3.

Chapter 7

Conclusions & Future Work

7.1 Tokenisation

Chapter 1 and Chapter 2 present the background for this thesis, including the Java Card system, which makes use of virtual method tables to allow for efficient dispatch of the *invokevirtual* instruction. However, in Java Card, the *invokeinterface* instruction must make use of additional lookup tables, making it slower. Previous work in object oriented languages has presented various approaches to dispatch tables that would allow *invokevirtual* and *invokeinterface* instructions to both use the same dispatch table, but at the price of very large tables.

Chapter 3 presented a new tokenisation scheme, based on that used in Java Card, but where the *invokeinterface* instruction does not require additional lookup tables, allowing it to be dispatched in the same way as the *invokevirtual* instruction. Previous research in object oriented languages required either single inheritance (Java Card), multiple dispatch tables (C++) or very large and sparse tables. The new tokenisation scheme allows for Java's limited form of multiple inheritance (in the form of interfaces) to be taken into account, while still producing compact virtual method tables. The new tables can not always be fully used in the presence of interfaces (i.e. occasionally "null" values must be left in the virtual method table), but these spaces are kept to a minimum.

Initially, the tokenisation process presented in this thesis required complete knowledge of the system, i.e. it required access to all classes that were going to be tokenised, so that the tokeniser could determine what interfaces existed and the relationships between those

classes and interfaces. The next stage of this work, presented in Chapter 4, extended the tokenisation scheme to operate incrementally. This allows a group of class files to be tokenised, then later, a new set of class files to be tokenised, without needing to alter the initial tokenisation. Incremental tokenisation permits the manufacturer to distribute a device with an already tokenised API, then third-parties can produce, tokenise and distribute applications which make use of the existing tokenised classes. Since the initial tokenisation can not know what future classes will be added to the system, some token allocations in the first tokenisation can cause ambiguities in the newly tokenised classes. The addition of conflict entries allow these ambiguities to be resolved, at the cost of a slightly more complex lookup mechanism. In practise however, the number of conflicts has been found to be very small.

During the tokenisation process, fields were also updated to use tokens instead of string references. Tokens are allocated such that each field in an object will have a unique token, which allows for efficient implementation where the token represents location of the field as an offset within the object. Static fields are similarly tokenised, where the field's token is unique within that class. The relevant get and set instructions for fields were updated to reference token values, instead of strings.

To prove that the tokeniser produced files that could still be executed, a simplified virtual machine was implemented, as described in Chapter 5. The Javolution library's benchmark suite was used as an example of an executable program, which was tokenised and executed on the simplified virtual machine. Execution was observed to be normal, indicating that the tokeniser had produced correct class files. Testing of the virtual method table based method dispatch also showed a four to forty times speed improvement over previous virtual machine implementations, specifically the Kilobyte Virtual Machine from Sun. From the point the KVM decodes an invoke instruction, it must follow references to the constant pool to locate the string name for the method, then perform string matching to find the appropriate class and method to call. To improve performance, the KVM can then replace the instruction with a faster version, making later executions of the same instruction quicker. By contrast, tokenised class files can use the virtual method table to very quickly locate the target method and do not need to update the instructions as the KVM

does. Therefore, tokenised classes provide consistent and fast execution of invoke instructions, and also, does not require the bytecode to be stored in a writable location, since the bytecode is not modified during execution.

7.2 Compression

Chapter 2 presented previous work on compressing Java class files, with some of that work focused on formats that remained executable (interpretable formats) and others on reducing the size of files for transmission (wire formats). The interpretable formats reduce the space needed to store application code, with only minimal overheads to execution. Wire formats give smaller files than interpretable formats, but require decompression before they can be executed and are aimed at reducing bandwidth for distributing applications.

During tokenisation, the strings used for linking are removed, resulting in class files that are much smaller. Chapter 6 presented a new format for tokenised class files, and included a discussion on the compression results in Section 6.6. The size of tokenised class files is 45-60% that of standard class files and on par with the size of compressed Jar files. However, unlike Jar files, the tokenised files can be executed directly. For mobile devices, where memory can be limited, having a format which offers the same size as a Jar file, without needing to be decompressed before execution, is ideal.

The gains made in compression have come mostly because of the removal of redundant string data, as well as a few optimisations in other parts of the class files (detailed in Chapter 6). However, some of the previous work on compression detailed in Section 2.7 could also be applied to the tokenised class files, allowing for further compression, in particular, work by Clausen *et al.* [24] and Bizzotto & Grimaud [13] focused on compression of the Java bytecodes, while remaining directly executable. The goal of this thesis has been to apply tokenisation to class files, therefore the application of such additional tokenisation and the implications for tokenised class files has not been considered here. It is likely that more could be done to reduce the size of tokenised class files if space savings are a priority.

7.3 Key Contributions

The key contributions of this thesis are:

- Successfully applied a Java Card like tokenisation to J2ME class files.
- Extended the tokenisation approach to include interfaces, removing the need for addition lookup tables.
- Demonstrated that the dispatch tables that are produced are smaller than previous approaches.
- Further extended the tokenisation approach to allow for incremental generation of the dispatch tables.
- Demonstrated that the resulting tables still produce correct execution in a virtual machine and that the execution was more efficient that previous approaches.
- Successfully compressed the size of the class files to 45-60% of their original size.

7.4 Future Work

While a successful tokenisation scheme has been produced, the ultimate aim is to also provide an efficient hardware based solution for executing the tokenised class files. While hardware processors capable of executing Java bytecodes exist, operations such is the *invoke** instructions remain too complex for a pure hardware solution. With the introduction of tokenisation and virtual method tables, the complexity of implementing an *invoke** instruction has been greatly reduced, making a hardware based solution feasible. Future work would involve development and prototyping such a processor, to measure what speed gains would be possible.

The work in this thesis has not aimed to produce the smallest possible class files, rather it has been aimed at producing tokenised class files. There remain elements of the class file that could be compressed further, without preventing the direct execution of the class files. While some previous work focused on the compression of string data in class files,

the need for which is reduced due to the removal of many strings, other work has focused on compression of other parts of class files, such as the bytecodes. Further work could be done on minimising the size of class files, with analysis of the performance impacts given the different nature of tokenised class files.

Section 2.3 presented previous work that has been done to optimise the Java virtual machine. While this thesis has focused on the *invoke** instructions, the current work could be merged with optimisations which target other areas of the virtual machines operation, potentially leading to greater improvements.

Finally, while the tokenisation approach here has focused on J2ME specifically, it would be interesting to examine its extension to other object oriented languages, such as C++. Going even further, it would also be possible to examine the implications of applying this tokenisation to object oriented languages in general.

Appendix A

Tokenised Class File Binary Format

The following appendix details the final format of a tokenised class file. The general structure is similar to that of a standard class file, however, the two are not compatible. The format is specified using the same notation as that used in the Java Virtual Machine Specification to define the standard class file format, which is somewhat similar to a C struct specification.

The fundamental types are: 'u1' and 'u2', which refer to an unsigned 1 or 2 byte value, respectively. All other types are defined as a structure consisting of one or more elements, each element can either be one of the fundamental types, or another structure. For elements that are repeated, an array notation is used consisting of square brackets, such as: 'u1 bytecodes[bytecode_count]'. Here 'bytecodes' is the name of this element and it consists of an array of unsigned 1-byte values, repeated 'bytecode_count' times, where bytecode_count would be another element, typically stored just before the array.

Below is the overall structure of a tokenised file. The following sections describe various parts of the file format in more detail.

```
TokenisedClassFile {
    u1 minor_version;
    u1 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count];
    u2 access_flags;
```

```
u2 this_class_token;
u2 super_class_token;
u2 interfaces_count;
u2 interfaces[interfaces_count];
u2 fields_count;
field_info fields[fields_count];
u2 static_methods_count;
method_info static_methods[static_methods_count];
u2 methods_count;
method_info methods[methods_count];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

- *minor_version* - The minor version number of the file format. This document describes the minor version 0.
- *major_version* - The major version of this class file. These two numbers are combined as *major_version.minor_version*. Standard class files use major versions 45 to 50. Tokenised files use major version 100.
- *constant_pool_count* - This is the number of entries in the constant pool plus one. This is since the constant pool is indexed starting at 1 instead of 0, to allow 0 to remain a special value.
- *constant_pool* - An array of constant pool entries. These are indexed from 1 to *constant_pool_count* - 1 inclusive.
- *access_flags* - A set of bit-flags to denote access permissions and properties of the class. This has not been changed from the original specification.
- *this_class_token* - The token that represents this class.

- *super_class_token* - The token that represents the super-class of this class. In the case of java/lang/Object, which has no super-class, this value must be 65535. This also imposes the limit on the total number of classes in a system to 65535 (0-65534), since the 65535 value is reserved.
- *interface_count* - The number of interfaces this class directly implements.
- *interfaces* - This is an array, giving the class token for each of the interfaces this class implements.
- *fields_count* - The number of fields that are defined in this class.
- *fields* - An array giving the details of each field in this class.
- *static_methods_count* - The number of static methods defined in this class.
- *static_methods* - An array of entries, one for each static method in this class.
- *methods_count* - The number of non-static methods defined in this class.
- *methods* - An array of entries, one for each non-static method in this class. The format of entries in this and the *static_methods* array are identical, with the type of the method (static or non-static) determined by which list the entry is in.
- *attributes_count* - The number of attributes associate with this class.
- *attributes* - Any additional attributes associated with the class.

A.1 Constant Pool

The constant pool is used to store not just constant values needed by the program, but also all the linking symbols and information to allow a class file to link with other files. There are several different types of entries that can occur in the constant pool, with the type or each entry denoted by a 1 byte “tag” value at the start of each entry. The types of entries in a tokenised class file are given in Table A.1.

Table A.1: Tokenised constant pool entry types

Tag	Description
1	CONSTANT_Utf8
3	CONSTANT_Integer
4	CONSTANT_Float
5	CONSTANT_Long
6	CONSTANT_Double
7	CONSTANT_Class
13	CONSTANT_ArrayClass

Of the entries that can appear in tokenised class files, the CONSTANT_Utf8, CONSTANT_Integer, CONSTANT_Float, CONSTANT_Long and CONSTANT_Double entries all store constant values used by the program and remain unchanged from standard class files. The CONSTANT_Class entry has been modified and a new CONSTANT_ArrayClass entry type has been created.

In Java a class reference can be to a compiled class file or to an array class, which the virtual machine must create on the fly. The standard CONSTANT_Class entry points to a string in the constant pool which defines the class name, with array classes beginning with the '[' character (since this is not a valid character in a class name). During tokenisation, other classes will be referred to by their class token, however it is not practical to allocate tokens for specific array classes. Therefore two types of entries were created for tokenised class files, the CONSTANT_Class and CONSTANT_ArrayClass. The CONSTANT_Class_info entry is modified from standard class files to contain the classes token instead of an index to a string:

```
CONSTANT_Class_info {
    u1 tag = 7;
    u2 token_value;
}
```

While the CONSTANT_Array_Class_info entry has been created to represent an array class:

Table A.2: Possible values for the *type* entry in a CONSTANT_Array_Class_info entry.

Value	Description
1	Primitive *boolean* type.
2	Primitive *char* type.
3	Primitive *byte* type.
4	Primitive *short* type.
5	Primitive *int* type.
6	Primitive *long* type.
7	Primitive *float* type.
8	Primitive *double* type.
9	Object reference type.

```
CONSTANT_Array_Class_info {
    u1 tag = 13;
    u1 type;
    u2 dimensions;
    u2 class_token;
}
```

The *tag* value of 13 is used, as this is not in use in standard class files. Next the *type* entry describes what type of values are stored in this array. Table A.2 shows the possible values. The *dimensions* value indicates the number of dimensions in the array, i.e. 1 is a standard array, while 2 would be an array with 2 dimensions and so on. This value must be >0. Finally the *class_token* value gives the types of objects that can be stored in this array. This is only true if the *type* entry is 9 (i.e. the array stores objects of some type), otherwise the array will be storing primitive values, and the *class_token* value must be 65,535.

A.1.1 field_info Section

A field_info entry describes a single field within a class. A tokenised field_info entry is very similar to the standard entry, but with the name and descriptor index entries replaced with a single token entry, giving the following structure:

```
field_info {
```

Table A.3: Meaning of Bits in the *access_flags* component of *field_info* entries

(a) Bit Map For *access_flags* Entry

Bit	Usage
0	Public
1	Private
2	Protected
3	Static
4	Final
5	Reserved
6	Volatile
7	Transient
8-11	Type
12-15	Reserved

(b) Meaning of Type Value For Fields

Value	Type
0	Boolean
1	Char
2	Byte
3	Short
4	Int
5	Long
6	Float
7	Double
8	Object

```
    u2 access_flags;
    u2 token;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

In a standard class file the *access_flags* entry contains a set of flags to define certain properties of the field (i.e. public, private, protected), and some bits are reserved for future use. The bits already defined for standard files retain their meaning in tokenised files, however some of the reserved bits have been made use of. Figure A.3a shows the meaning of each bit in tokenised files, with bits 8-11 now used to define the type of value stored in the field. The type value is stored as an unsigned 4-bit integer, with Figure A.3b showing the meaning of each value.

A.2 method_info Section

The method_info structure details a single method defined within the class. The same structure is used for the static_methods and the methods lists. The list the entry appears

Table A.4: Additional *access_flag* values added to the tokenised VM.

Flag Name	Value	Interpretation
ACC_INITV	0x0200	This is an <init> method that takes no parameters (i.e. a default constructor).
ACC_MAIN	0x1000	This is a main method.
ACC_CLINIT	0x2000	This is a <clinit> method.
ACC_INIT	0x4000	This is an <init> method that takes parameters.
ACC_INITS	0x8000	This is an <init>(Ljava/lang/String;)V method (i.e. constructor that only takes a single String parameter). Used by the VM to build Exception objects with message strings.

in defines if the method is a static or non-static method respectively. Each method_info structure has the form:

```
method_info {
    u2 access_flags;
    u2 token;
    u2 arg_count;
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

The access_flags entry provides a set of flags which describe certain properties of the method. The flags already declared for standard class files all remain in tokenised files, however some additional flags have been added, as shown in Figure A.4.

The token entry provides the method's token, while arg_count indicates the number of arguments the method accepts (this was originally available from the descriptor string, but that was removed during tokenisation).

A.3 Attributes

Attributes are used to store additional information, with the Java Virtual Machine Specification allowing the addition of custom attributes. However, custom attributes are not allowed to modify the semantics of a class file and must be ignored by virtual machines that do not understand them. Tokenised class files limit attributes to the set of attributes defined in the Java Virtual Machine Specification that are required for the virtual machine to operate correctly (plus an addition VirtualMethodTable attribute defined in this thesis). Standard class files identify the types of attributes by a string name, allowing more flexibility for extension. Since tokenised files do not allow custom attributes, a single byte tag value is sufficient to identify all possible attributes, resulting in a general form for tokenised attributes of:

```
attribute_info {
    u1 tag;
    u1 info[...];
}
```

The *tag* determines the type of the entry and since the virtual machine will know the format for each of these types, no size value needs to be included. The possible types are:

Tag	Attribute Type
1	ConstantValue (1 byte)
2	ConstantValue (2 bytes)
3	Code
4	Exceptions (1 byte)
5	Exceptions (2 bytes)
6	InnerClass
7	Synthetic
8	Deprecated
9	StackMap
10	VMT

Each type is explained in the following sections.

A.3.1 ConstantValue Attribute

This attribute can be attached to a field_info section of a static field and denotes the value the field must have when it is initialised. The attribute can point to any of the value types in the constant pool, specifically, one of: CONSTANT_Integer, CONSTANT_Long, CONSTANT_Float, CONSTANT_Double or CONSTANT_Utf8.

There are two versions of this attribute defined for tokenised class files, one with an unsigned 1-byte index to the constant pool, the other with an unsigned 2-byte index, which allows for a small space saving when the wider index is not required. The 1-byte version has the form:

```
ConstantValue1_attribute {
    u1 tag = 1;
    u1 constantvalue_index;
}
```

While the 2-byte version has the form:

```
ConstantValue2_attribute {
    u1 tag = 2;
    u2 constantvalue_index;
}
```

A.3.2 Code

The Code attribute is used to store the bytecodes and associated information for a method. It is provided as an attribute so it will not take up space in a method with no implementation (i.e. abstract methods).

The format of the Code attribute is mostly the same as found in The Java Virtual Machine Specification[57] and consists of:

```
Code_attribute {
    u1 tag = 3;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attribute_count;
    attribute_info attributes[attribute_count];
}
```

The *code* entry has several changes to the *ldc, ldc_w, getstatic, putstatic, getfield, putfield, invokevirtual, invokespecial, invokestatic* and *invokeinterface* instructions. Details of these changes are in Section 5.3.

As well, the *catch_type* entry in the *exception_table* used to be an index into the constant pool to a CONSTANT_Class entry, to provide the type of the exception that handler would catch. This has been replaced with the two-byte class token for the class, removing the reference to the constant pool.

A.3.3 Exceptions Attribute

This attribute is found in a *method_info* structure, where it indicates which checked exceptions a method may throw. In a standard class file, this attribute consists of a list of constant pool references, each one to a CONSTANT_Class entry, to denote the exception types.

For tokenised class files, the attribute simply stores the class tokens instead of needing references to the constant pool. To save space, there are two versions of this attribute, one with single-byte class tokens, and one with two-byte class tokens. If every class token is less than 256, then they are stored as unsigned single byte values in the following structure:

```
Exceptions1_attribute {
    u1 tag = 4;
    u2 number_of_exceptions;
    u1 exception_table[number_of_exceptions];
}
```

If one or more of the class tokens is greater than 255, then all of them are stored as unsigned two-byte values in the following structure:

```
Exceptions2_attribute {
    u1 tag = 5;
    u2 number_of_exceptions;
    u2 exception_table[number_of_exceptions];
}
```

A.3.4 InnerClass Attribute

The InnerClass attribute indicates a class that is an inner class of another class. The standard class file version of this attribute contains indexes to the constant pool to define the inner and outer classes. These entries retain the same format (except for the attribute name being replaced with a tag value).

A.3.5 Synthetic/Deprecated

These attributes are used to mark class/methods/fields that were generated by the compiler (synthetic) or that the programmer believes should not be used anymore (deprecated). Since

both these attributes are simply markers, they require no data, resulting in an attribute entry that consists only of the one byte tag value.

A.3.6 StackMap

The StackMap attribute is not a part of the standard J2SE Specification, rather it was added by the CLDC Specification [63] and is part of a simplified class file verification process for CLDC devices. The format of StackMap entries remains unchanged from their original specification.

A.3.7 VirtualMethodTable Attribute

The VMT_Attribute is used to store the virtual method table for a class and as such, there must be exactly one of these attributes associated with every class (even abstract classes, since they can contain concrete methods, however not in interfaces). A VirtualMethodTable attribute consists of:

```
VMT_attribute {
    u1 tag = 10;
    u2 entries;
    VMTEntry vmtEntries[entries];
}
```

Each VMTEntry has the general format:

```
VMTEntry {
    u1 tag;
}
```

The tag value here denotes the type of VMT entry, between three possibilities: Null, Single or Multi. The Null entry is for an entry that was left blank by the tokeniser (because of a

conflict elsewhere, this class could not use that token value), and therefore consists of just the tag value:

```
VMTEntry_Null {
    u1 tag = 0;
}
```

A Single entry is the most common type of VMT entry and indicates a single method that should be called. To define the specific method, the entry contains a class token and method token, as follows:

```
VMTEntry_Single {
    u1 tag = 1;
    u2 classToken;
    u2 methodToken;
}
```

The final type of entry is the Multi, used when a VMT entry could result in one of several different methods being executed, because of a conflict that arose during incremental tokenisation. A Multi entry has the format:

```
VMTEntry_Multi {
    u1 tag = 2;
    u2 classToken;
    u2 methodToken;
    u2 count;
    {
        u2 interfaceToken;
        u2 classToken;
        u2 methodToken;
```

```
        } conflict_entries [count];

    }
```

The classToken and methodToken values provide the default method to call, while the conflict entries lists the alternative methods that must be called if an invokeinterface instruction is being used for the given interface type.

Appendix B

Descriptor File Binary Format

A descriptor file describes an already tokenised set of classes, along with the string name and token for each class and for all the fields/methods within each class. The tokeniser will output a descriptor file at the end of every tokenisation operation, and in the case of incremental tokenisation, it must be provided with the descriptor file for the previously tokenised classes. Since the new classes will reference existing classes via their string names (i.e. 'java/lang/Object'), the descriptor file can be used by the tokeniser to map from the string names to the appropriate token.

A descriptor file consists of three main sections. The first is a string repository, similar to a class files constant pool, which stores the string data used by the rest of the file. These strings are referenced via their index within the string repository. Next is the method group repository, which lists all the currently existing method groups and the methods contained in that group. These are stored centrally, since when loading a descriptor file, all methods in a method group must reference the same method group object. Therefore, the method group repository allows the tokeniser to create all method group objects first, then link them to the method objects as they are created. The final section is the data for all the classes, defining the class's name and token, as well as the name, descriptor and token for all of the classes methods and fields. The general format for a descriptor file is:

```
DescriptorFile {

    u4 stringCount;

    UTF8 strings[stringCount];
```

```
    u2 methodGroupCount;
    MGEntry methodGroups[methodGroupCount];
    u2 classCount;
    Class classEntries[classCount];
}
```

- *stringCount* - The number of strings stored in this file.
- *strings* - This stores all the strings used in the file, with other structures providing an index into the strings array. This also allows the same index to be used when the same string is used in multiple places, hence saving space. Each string is stored in UTF format, which corresponds to the java.io.DataInputStream readUTF() method and java.io.DataOutputStream writeUTF(String) method.
- *methodGroupCount* - The number of method group entries.
- *methodGroups* - The method groups that exist in this file. Each entry can be either a MethodGroupSingle or a MethodGroupMulti, as described later.
- *classCount* - The number of classes in this file.
- *classEntries* - Each entry describes all the details of a single class in the system.

B.1 Method Group Entries

Each entry in the *methodGroups* array has the general form:

```
MGEntry {
    u1 type;
    ....
}
```

Where the *type* value will be non-zero to indicate a MethodGroupMulti entry and zero to indicate a MethodGroupSingle entry. Following this will be the data for the entry, the length of which will depend on the type.

B.1.1 MethodGroupSingle

A MethodGroupSingle entry is the basic type of method group that is allocated a single token. The entry will have a token value, and a list of the methods that it contains. The form will be:

```
MethodGroupSingle {
    u2 token;
    u2 count;
    u4 classNames[count];
    u4 methodNames[count];
    u4 methodDescriptors[count];
}
```

- *token* - The token value that all methods in this group require.
- *count* - The number of methods in this group. This value denotes the length of the following three arrays, where an entry at index X will give the class name, method name and method descriptor for a single entry in this method group.
- *classNames* - The class names for each entry in this method group. Each entry is an unsigned 2-byte value, which gives an index into the string repository. The string at that index will be the class name for the entry.
- *methodNames* - The method names for each entry in this method group. Each entry is an unsigned 2-byte value, which gives an index into the string repository. The string at that index will be the method name for the entry.

- *methodDescriptors* - The method descriptor for each entry in this method group. Each entry is an unsigned 2-byte value, which gives an index into the string repository. The string at that index will be the method descriptor for the entry.

At runtime, and object representing a MethodGroupSingle will contain a list of object references to the objects that represent the methods contained in that group. Also, each method object will contain a reference to the method group that it is in. Since both types of objects require a reference to the other, one of them must be saved/loaded first and therefore can not give a simple index within the file to the later object (in this case, when the method group is written to the file, the methods have not be saved yet, and vice versa during loading). Therefore, the method group stores the class name, method name and descriptor for the method and after the tokeniser finishes loading the descriptor file, it will use these to resole the appropriate runtime object that represents that method.

B.1.2 MethodGroupMulti

This represents a MethodGroupMulti, which are needed to implement incremental tokenisation. Each MethodGroupMulti consists of a list of MethodGroupSingles. This takes the form of:

```
MethodGroupMulti {
    u2 count;
    u2 MGSIndex[count];
}
```

- *count* - The number of MethodGroupSingle entries this group contains.
- *MGSIndex* - Each entry is an index into the *methodGroups* array in the current descriptor file, where the entry at that index will be a MethodGroupSingle.

The current tokeniser does not ensure that all of the referenced MethodGroupSingle entries have been saved before saving a MethodGroupMulti. Therefore, at load time, the

MethodGroupMulti entry may contain indexes to MethodGroupSingle entries that have not yet been loaded. A two-stage loading process is required, where the first stage is to create each MethodGroupMulti entry with just the index values while loading the method groups. Then after the *methodGroups* array has been read completely, each MethodGroupMulti can resolve the indexes.

B.2 Class Entries

Each Class entry will provide all the information about a single class that was either already present, or added to the system, during the conversion that produced this descriptor file. Each entry contains the name of this class, its super-class, implemented interfaces, fields and methods. With this information from every class, the tokeniser can produce an inheritance tree of every class in the system, as well as the more general interface graph. The inheritance tree and interface graph are both traversed when allocating tokens, to check for conflicts. The structure for each class consists of:

```
Class {
    u4 className;
    u4 superClassName;
    u4 thisClassToken;
    u1 flags;
    u2 interfaceCount;
    u4 interfaces[interfaceCount];
    u2 fieldCount;
    Field fields[fieldCount];
    u2 staticFieldCount;
    Field staticFields[staticFieldCount];
    u2 staticMethodCount;
    Method staticMethods[staticMethodCount];
```

```
    u2 methodCount;
    Method methods[methodCount];
}
```

- *className* - An index into the string repository. The entry at that index will be the name of this class.
- *superClassName* - If the name of this class is "java/lang/Object", then this class must have no super-class, and this entry will have the max value for an unsigned 4 byte value. Otherwise this will be an index into the string repository and the entry at that index will be the name of the super-class for this class.
- *thisClassToken* - The token value that has been allocated to this class.
- *flags* - This field consists of 8, 1-bit, flags. At present only two of these are used, they are the values 0x01 and 0x02. If 0x01 is set, then this class is an interface. If 0x02 is set, then this class is abstract. The remaining bits are reserved for future use.
- *interfaceCount* - The number of interfaces that this class or interface implements.
- *interfaces* - Each value in this array will be an index into the string repository. The entry at that index will be the name of an interface this class or interface implements.
- *fieldCount* - The number of non-static fields in this class.
- *fields* - Each entry will describe the details for a single, non-static, field in this class.
- *staticFieldCount* - The number of static fields in this class.
- *staticFields* - Each entry will describe the details for a single, static, field in this class.
- *staticMethodCount* - The number of static methods in this class.
- *staticMethods* - Each entry will describe a single static method that is in this class.
- *methodCount* - The number of methods in this class.
- *methods* - Each entry will describe a single method that is in this class.

B.2.1 Field entries

Each field entry represents either a static or non-static field (depending which list the entry is in) that is present inside a class. Each field entry will have the following format:

```
Field {
    u4 name;
    u4 descriptor;
    u2 token;
}
```

- *name* - An index into the string repository. The entry at that index will be the name of this field.
- *descriptor* - An index into the string repository. The entry at that index will be the descriptor for this method.
- *token* - The token value that has been assigned to this field.

B.2.2 Method entries

Each Method entry denotes a single method (either static or non-static, depending on which list the entry is in) that exists in a class. The entry contains the name and descriptor for the method, as well as the method group that contains it, thus providing the token. The format for a Method entry is:

```
Method {
    u4 name;
    u4 descriptor;
    u2 methodGroup;
}
```

- *name* - An index into the string repository. The entry at that index will be the name of this method.
- *descriptor* - An index into the string repository. The entry at that index will be the descriptor for this method.
- *methodGroup* - An index into the method group repository. The entry at that index will be the method group that contains this method. One of the entries within the method group must consists of this class's name, as well as the method's name and descriptor.

Appendix C

Types of Native Methods in CLDC 1.1 API

The CLDC API contains many native methods that are not implemented in Java for one reason or another. In some cases this is to improve performance, by making use of lower level specialised hardware support, or other times because the Java environment simply cannot supply the behaviour.

There are three main categories of native methods: performance, JVM interaction and IO. The following sections are broken down by the reason for the method to be native, and then by the class each method belongs to.

C.1 Performance

These are methods that could be implemented in Java, but have not been. They are implemented in native code so as to allow them to execute faster, either by using specialised hardware, or making more efficient use of the underlying hardware or JVM internals.

C.1.1 java.lang.Double

- public static long doubleToLongBits(double value) - Converts a double primitive type to a long primitive type with the IEEE 754 floating point “double-precision” bit layout.

- public static double longBitsToDouble(long bits) - Converts a long primitive type

with the IEEE 754 floating point "double-precision" bit layout to a double primitive type.

C.1.2 java.lang.Float

- public static int floatToIntBits(float value) - Converts a float primitive type to an int primitive type with the IEEE 754 floating-point "single-precision" bit layout.
- public static float intBitsToFloat(int bits) - Converts an int primitive type with the IEEE 754 floating-point "single-precision" bit layout to a float primitive type.

C.1.3 java.lang.Math

There are several methods in this class used to implement mathematical functions. These are native to make use of platform libraries, specialised instructions or hardware to implement them.

- public static double sin(double a)
- public static double cos(double a)
- public static double tan(double a)
- public static double sqrt(double a)
- public static double ceil(double a)
- public static double floor(double a)

C.1.4 java.lang.String

The native string methods appear to have been provided solely for performance reasons, since they underpin a large number of operations in the virtual machine. Most methods have a commented out Java version in the source code.

- public char charAt(int index) - Returns the character at the given index in the string.

- public boolean equals(Object anObject) - Over-rides the equals() method in Object to compare the contents of strings.
- public int indexOf(int ch) - Search for the character in the String.
- public int indexOf(int ch, int fromIndex) - Search for the character in the string.
- public String intern() - Will internalise a string. Once interned, that string object will remain for the life of the virtual machine. All future attempts at interning a String object which is equal (contains the same characters in the same order) to an already interned string, will result in the already interned object being returned. Thus if two strings, which may not be the same object, represent the same sequence of characters, using intern will guarantee they are the same object afterwards.

C.1.5 java.lang.StringBuffer

The StringBuffer class is used to hold a mutable string (the String class is non-mutable). In particular, additional sections can be appended to the end of the contents of a StringBuffer. Any place where string concatenation occurs in the source code, will actually be compiled to use a StringBuffer, and append each section to build up the string. As such, this class is used in a large majority of the string operations at runtime.

- public native synchronized StringBuffer append(String str) - Adds the contents of the given string onto the end of the contents of this string buffer. There are overloaded versions of this method that will accept any of the primitive types. All these (except for the int one below) use String.valueOf(x) to return the primitive value as a string, then calls this method.
- public native StringBuffer append(int i) - Adds the given integer (turned into a string) onto the end of this string.
- public native String toString() - Will turn a StringBuffer's contents back into a non-mutable String.

C.1.6 java.lang.System

- public static void arraycopy(Object src, int srcOffset, Object dst, int dstOffset, int length) - Copies the contents from a given range in one array into another. Provided so the VM can do a type check, then a direct memory copy. If implemented in Java it would require a type check for every individual value stored into the destination array.

C.2 JVM Interaction

These methods need to interact with the virtual machine in some way and are therefore performed as a native method call, to allow the JVM to intercept them and perform the necessary operations. These operations could not be performed in Java code.

C.2.1 java.lang.Class

Represents a class in the virtual machine, providing reflection abilities to query classes. Most of the methods in this class are native as it requires querying the virtual machine's data structures. Instances of this class will represent a given class or interface type, that must have been loaded into the virtual machine.

- public static Class forName(String className) - Will attempt to load the class with the given name.

- public Object newInstance() - Will make a new instance of this class. Allows objects to be made for classes that were unknown at compile time.

- public boolean isInstance(Object obj) - Test if the object is an instance of this class type. This will give the same result as the 'instanceof' operator, but allows it to be used on Classes that were not known at compile time.

- public boolean isAssignableFrom(Class cls) - Used to test if a cast will be successful.

- public boolean isInterface() - Test if this represents a class or an interface.

- public boolean isArray() - Test if this is an array class that is created on-the-fly by the virtual machine.

- public String getName() - Get the name of this class.

C.2.2 java.lang.Object

This is the base type for all objects in Java.

- public final native Class getClass() - Returns the java.lang.Class object that represents the class this is an object of.

- public native int hashCode() - Returns a unique hashcode for this object.

- public final native void notify() - Used for thread synchronisation.

- public final native void notifyAll()- Used for thread synchronisation.

- public final native void wait(long timeout) throws InterruptedException- Used for thread synchronisation.

C.2.3 java.lang.Runtime

This class is used to represent the current runtime state of the virtual machine as it appears to the program to allow it to query/manipulate it.

- private void exitInternal(int status) - Causes the virtual machine to exit, used to implement System.exit(int).

- public long freeMemory() - Query the VM on how much memory there is free.

- public long totalMemory() - Query the VM on how much memory there is in total.

- public void gc() - Request the VM perform garbage collection.

C.2.4 java.lang.System

- public static long currentTimeMillis() - Returns the current time as the number of milliseconds since midnight at the start of the 1st January, 1970. This method is native since a call to the underlying operating system is required to get the current time.

- private static String getProperty0(String key) - Internal method to query system properties. These are set by the virtual machine before the application starts executing and can include system dependent information, such as operating system, working directory, virtual machine version, current user, etc..

- public static int identityHashCode(Object x) - Provides a way to get the same value the java.lang.Object.hashcode() method would return, even if it has been over-ridden.

C.2.5 java.lang.Thread

An object of this class represents a thread of execution in the virtual machine, that may or may not be currently executing.

- public static Thread currentThread() - Returns the Thread object for the thread that called this method.

- public static void yield() - Causes the thread scheduler to be run again. The current thread will still be runnable, however it gives a chance for other threads to execute. This is different from sleep in that it is possible for this thread to be selected for execution again immediately.

- public static void sleep(long millis) - Will cause this thread to sleep for a given time before becoming runnable again (although it must then wait for the scheduler to actually schedule it, before it will start executing again).

- public void start() - Starts a new thread.

- public boolean isAlive() - Test if the thread is still alive. That is, it has been started and has not terminated (either via natural termination or due to an exception/error).
- public static int activeCount() - The current number of active threads in the virtual machine.
- private void setPriority0(int newPriority) - Used as an internal helper method to cause a change in the threads priority.
- private void interrupt0() - Used as an internal helper method to interrupt a thread.

C.2.6 java.lang.Throwable

- private void printStackTrace0(Object s) - Internal method to cause the VM to print the stack trace to the given output stream. This is needed since only the VM knows what state the stack is in.

C.2.7 java.lang.ref.WeakReference

This is used to implement weak references, allowing an application to hold references to objects that will not stop the object from being garbage collected if needed. As such, this requires very special and tight integration with the garbage collector to operate correctly.

- private native void initializeWeakReference() - Used to initialise a weak reference after the object has been created. This method is called from the constructor.

C.3 IO

The final category is for performing IO operations. Java does not have the necessary low level features to perform IO, as typically this will require calling system/kernel functions or interacting directly with hardware, depending on the context. While the Java libraries provide pure Java code for dealing with and using various types of IO, all those classes are underpinned by the native methods found here.

C.3.1 com.sun.cldc.io.ConsoleOutputStream

This particular class is used specifically to implement System.out (i.e. the standard output stream).

- public native synchronized void write(int c) throws IOException - Any calls to System.out that write output, will ultimately become a sequence of calls to this method, to write one character at a time to output.

C.3.2 com.sun.cldc.io.ResourceInputStream

This class is used for reading arbitrary resources from an applications JAR file. The class file will expect some form of object to be returned when it opens the stream, and that object is used for every read operation. It is up to the native code to determine what that object actually is. All these methods are private methods used internally by the class.

- private static native Object open(String name) throws IOException - Open the named resource file to be read. This returns a handle object that the Java code does not interact with, but is used to denote the stream in the native code (i.e. identifier or structure used by the native code to perform IO).

- private static native void close(Object handle) throws IOException - Will close the given stream.

- private static native int size(Object handle) throws IOException - Returns the size of the open file.

- private static native int read(Object handle) throws IOException - Reads a single byte from the file.

- private static native int readBytes(Object handle, byte[] b, int offset, int pos, int len) throws IOException - Used to read 1 or more bytes into the given byte array in a single operation.

C.3.3 com.sun.cldc.io.Waiter

This class is used to wait for an IO event.

- public native static void waitForIO()

Bibliography

[1] G. Acher, C. Trinitis, and R. Buchty. Cpu-independent assembler in an fpga. In *International Conference on Field Programmable Logic and Applications*, pages 519–522, Aug. 2005.

[2] R. Achutharaman, R. Govindarajan, G. Hariprakash, and A.R. Omondi. Exploiting java-ilp on a simultaneous multi-trace instruction issue (smti) processor. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 8 pp.–, 22-26 April 2003.

[3] Ali-Reza Adl-Tabatabai, MichałCierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *SIGPLAN Notices*, 33(5):280–290, 1998.

[4] aJile Systems, Inc. aj-100, real-time low-power direct execution microprocessor for the java platform. Online at: http://www.jempower.com/ajile/downloads/aj100.pdf (Last Accessed: 21st Jan. 2007), 2006.

[5] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 2001. ACM.

[6] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. *SIGPLAN Notices*, 27(10):110–126, 1992.

[7] Denis N. Antonioli. Car: The class archive format. Technical report, Department of Information Technology, University of Zurich, 2001.

[8] ARM Limited. ARM thumb architecture extension. Online at: http://www.arm.com/products/CPUs/archi-thumb.html (Last Accessed: 29th Jan. 2010).

[9] ARM Limited. Jazelle dbx white paper. Online at: http://www.arm.com/pdfs/JazelleDBX_WhitePaper_2007v1p1.pdf (Last Accessed: 21st Jan. 2010), 2005.

[10] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[11] Mladen Berekovic, Helge Kloos, and Peter Pirsch. Hardware realization of a java virtual machine for high performance multimedia applications. *The Journal of VLSI Signal Processing*, 22(1):31–43, August 1999.

[12] Hal Berghel. Who won the mosaic war? *Communications of the ACM*, 41(10):13–16, October 1998.

[13] Gabriel Bizzotto and Gilles Grimaud. Practical JavaCard bytecode compression. In *Symposium en Architectures Nouvelles de Machines (RENPAR14/ASF/SympA)*, Hamamet, Tunisia, April 2002.

[14] Q. Bradley, R. Horspool, and J. Vitek. Jazz: An efficient compressed format for java archive files, 1998.

[15] K. Burgaard and J. Erichsen. Virtual machines for limited devices, 2000.

[16] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalape no dynamic optimizing compiler for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM.

[17] Jon Byous. Java technology: The early years. Online at: http://java.sun.com/features/1998/05/birthday.html (Last accessed: 25th Feb. 2008), 2003.

[18] Gregory J. Chaitin, Marc A. Auslander, Ashor K. Chandra, John Cocke, Martain E. Hopkins, and Peter W. Markstein. Register allocation via colouring. *Computer Languages*, 6:47 – 57, 1981.

[19] L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung. Stack operations folding in java processors. *Computers and Digital Techniques, IEE Proceedings*, 145(5):333–340, September 1998.

[20] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. In *Proceedings of the OOPSLA '03 conference*, volume 38, pages 282–301, November 2003.

[21] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. Energy savings through compression in embedded java environments. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 163–168, New York, NY, USA, 2002. ACM.

[22] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded java environment. In *Proceedings, Eighth International Symposium on High-Performance Computer Architecture, 2002*, pages 92–103, 2-6 Feb. 2002.

[23] MichałCierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. *SIGPLAN Not.*, 35(5):13–26, 2000.

[24] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[25] Jean-Marie Dautelle. Javolution. Online at: http://javolution.com/ (Last accessed: 18th Feb. 2010).

[26] David Detlefs and Ole Agesen. Inlining of virtual methods. *Lecture Notes in Computer Science*, 1628/1999:668, 1999.

[27] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 273–282, New York, NY, USA, 1992. ACM.

[28] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. *SIGPLAN Not.*, 24(10):211–214, 1989.

[29] Karel Driesen. Method lookup strategies in dynamically typed object-oriented programming languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[30] Karel Driesen. Selector table indexing & sparse arrays. In *Conference on Object-Oriented*, pages 259–270, 1993.

[31] Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, University of California, Santa Barbara, June 1999.

[32] K. Ebcioglu, E. Altman, and E. Hokenek. A java ILP machine based on fast dynamic compilation. In *MASCOTS'97 — International Workshop on Security and Efficiency Aspects of Java*, 1997.

[33] M. Watheq El-Kharashi, Fayez Elguibaly, and Kin F. Li. Adapting tomasulo's algorithm for bytecode folding based java processors. *SIGARCH Comput. Archit. News*, 29(5):1–8, December 2001.

[34] M.W. El-Kharashi, F. ElGuibaly, and K.F. Li. A quantitative study for java microprocessor architectural requirements. part i: Instruction set design. *Microprocessors and Microsystems*, 24(5):225–236, September 2000.

[35] M.W. El-Kharashi, F. ElGuibaly, and K.F. Li. A quantitative study for java microprocessor architectural requirements. part ii: high-level language support. *Microprocessors and Microsystems*, 24(5):237–250, September 2000.

[36] M.W. El-Kharashi, F. Elguibaly, and K.F. Li. A robust stack folding approach for java processors: an operand extraction-based algorithm. *Journal of Systems Architecture*, 47(8):697–726, December 2001.

[37] M.W. El-Kharashi, F. Gebali, K.F. Li, and Fang Zhang. The jafardd processor: a java architecture based on a folding algorithm, with reservation stations, dynamic translation, and dual processing. *IEEE Transactions on Consumer Electronics*, 48(4):1004–1015, November 2002.

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

[39] Jens Ernst, William S. Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 358–365, 1997.

[40] freestandards.org. Itanium C++ ABI (revision: 1.83).

[41] S. Fuhrmann, M. Pfeffer, J. Kreuzinger, Th. Ungerer, and U. Brinkschulte. Real-time garbage collection for a multithreaded java microcontroller. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, volume 00, page 69, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[42] C. J. Glossner and S. Vassiliadis. The Delft-Java engine. *Lecture Notes in Computer Science*, 1300:766–770, November 1997.

[43] GNU. Gnu gprof info pages. http://sourceware.org/binutils/docs/gprof/Sampling-Error.html (Last accessed: 18th Feb. 2010).

[44] G. Hariprakash, R. Achutharaman, and Amos Omondi. Hardware compilation for high performance java processors. In *International Conference on Architecture of Computing Systems*, pages 125 – 134, Karlsruhe, Germany, 2002.

[45] R. Nigel Horspool and Jason Corless. Tailored compression of Java class files. *Software - Practice and Experience*, 28(12):1253–1268, 1998.

[46] Imsys Technologies. Imsys im1000 - a multimedia platform for java applications product brief. http://www.imsystech.com/documentation /product_briefs/IM1000_product_brief_506c-A4.pdf (Last accessed: 18th Feb. 2010).

[47] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, chapter 2, pages 2–12 – 2–15. Intel Corp., April 2008.

[48] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: Basic Architecture*, chapter 18, pages 18–39 – 18–39. Intel Corp., April 2008.

[49] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, New York, NY, USA, 2000. ACM.

[50] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, New York, NY, USA, 1999. ACM.

[51] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio

Nakatani. Effectiveness of cross-platform optimizations for a java just-in-time compiler. *SIGPLAN Not.*, 38(11):187–204, 2003.

[52] Alan C. Kay. The early history of smalltalk. *History of programming languages II*, pages 511–598, 1996.

[53] Kenneth Blair Kent. *The Co-Design of Virtual Machines Using Reconfigurable Hardware*. PhD thesis, University of Victoria, 2003.

[54] A. Krall. Efficient javavm just-in-time compilation. *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205–212, Oct 1998.

[55] Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM implementation: Compilers versus hardware. In John Morris, editor, *Computer Architecture (ACAC '98)*, volume 20, pages 101–110, Perth, 1998. Springer.

[56] Andreas Krall and Reinhard Grafl. Cacao - a 64-bit javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[57] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Prentice Hall PTR, second edition edition, 1999.

[58] Steven Lucco. Split-stream dictionary program compression. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 27–34, New York, NY, USA, 2000. ACM.

[59] David Matula, George Marble, and Joel Isaacson. *Graph Theory and Computing*, chapter Graph Colouring Algorithms, pages 109 – 122. Academic Press, 1972.

[60] Sun Microsystems. Jar file specification, 1999.

[61] Sun Microsystems. The java language specification, second edition, 1999.

[62] Sun Microsystems. J2me building blocks for mobile devices: White paper on kvm and the connected, limited device configuration (cldc), 2000.

[63] Sun Microsystems. Connected limited device configuration specification, version 1.1, 2003.

[64] Sun Microsystems. *Java 2 Platform, Standard Edition*. Online at: http://java.sun.com/j2se/index.jsp, (Last accessed: 13th Nov. 2004).

[65] Nazomi Communications. JA108 - multimedia application processor. Online at: http://www.nazomi.com/images/ja108_pb.pdf (Last accessed: 10th Nov. 2006), 2003.

[66] J.M. O'Connor and M. Tremblay. picojava-i: the java virtual machine in hardware. *Micro, IEEE*, 17(2):45–53, Mar/Apr 1997.

[67] Amos R. Omondi. Design and implementiation of java processors. *Lecture Notes in Computer Science*, 2823/2003:86 – 96, October 2003.

[68] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120–131, New York, NY, USA, 1997. ACM.

[69] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A Framework for Optimizing Java Using Attributes. In *Compiler Construction: 10th International Conference, CC 2001, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001: Proceedings*, volume 2027 of *Lecture Notes in Computer Science*, pages 334–354, Genova, Italy, April 2001. Springer Berlin / Heidelberg.

[70] Chris Porthouse. High performance Java on embedded devices. White paper, ARM Ltd., October 2005.

[71] William Pugh. Compressing java class files. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[72] R. Radhakrishnan, D. Talla, and L.K. John. Allowing for ilp in an embedded java processor. In *Proceedings of the 27th International Symposium on Computer Architecture, 2000*, pages 294–305, 2000.

[73] Ramesh Radhakrishnan. *Microarchitectural techniques to enable efficient Java execution*. PhD thesis, The University of Texas at Austin, August 2000.

[74] Ramesh Radhakrishnan, Ravi Bhargava, and Lizy K. John. Improving java performance using hardware translation. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 427–439, New York, NY, USA, 2001. ACM Press.

[75] D. Rayside, E. Mamas, and E. Hons. Compact java binaries for embedded systems, 1999.

[76] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[77] Martin Schoeberl, Christian Thalinger, Stephan Korsholm, and Anders P. Ravn. Hardware objects for java. *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, 0:445–452, 2008.

[78] K. Scott and K. Skadron. BLP: Applying ILP Techniques to Bytecode Execution. In *Proceedings of the 2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, September 2000.

[79] Nik Shaylor, Douglas N. Simon, and William R. Bush. A java virtual machine architecture for very small devices. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 38(7):34–41, June 2003.

[80] Isidoros Sideris, George Economakos, and Kiamal Pekmestzi. A cache based stack folding technique for high performance java processors. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 48–57, New York, NY, USA, 2006. ACM.

[81] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java™ on the bare metal. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, New York, NY, USA, 2005. ACM.

[82] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM.

[83] R. Smith, C. Cifuentes, and D. Simon. Enabling Java TM for small wireless devices with Squawk and Spotworld. In *OOPSLA Workshop Bringing Software to Pervasive Computing, Oct*, volume 16, 2005.

[84] Adam Spitz, Alex Ausch, and David Ungar. Klien metacircular virtual machine kit 0.1 release notes. Online at: http://research.sun.com/self/Klein/release.html (Last accessed: 18th Feb. 2010).

[85] Squeak. Online at: http://www.squeak.org/ (Last accessed: 18th Feb. 2010).

[86] James M. Stichnoth, Guei-Yuan Lueh, and MichałCierniak. Support for garbage collection at every instruction in a java compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 118–127, New York, NY, USA, 1999. ACM.

[87] Christian H. Stork, Vivek Haldar, and Michael Franz. *Generic Adaptive Syntax-directed Compression for Mobile Code*. Dept. of Information and Computer Science, University of California, Irvine, March 2001.

[88] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, 2000.

[89] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, New York, NY, USA, 2001. ACM.

[90] Sun Microsystems. picoJava-II: Java processor core. Sun Microsystems Data Sheet, April.

[91] Sun Microsystems. Revisions to the class file format. Online at: http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf (Last accessed: 12th Nov. 2008).

[92] Sun Microsystems. Jsr 200: Network transfer format for java archives. Online at: http://jcp.org/en/jsr/summary?id=200 (Last accessed: 28th Jul. 2009), 2004.

[93] The Standard Performance Evaluation Corporation. Specjvm98. Online at: http://www.specbench.org/osg/jvm98/, 1998.

[94] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[95] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[96] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a java processor. *ECOOP'98 - Object-Oriented Programming*, 1445/1998:330, 1998.

[97] Wikipedia. List of programming languages by category. Online at: http://en.wikipedia.org/wiki/List_of_programming_languages_by_category (Last accessed: 1st Sep. 2008), Sep 2008.

[98] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: A java

vm just-in-time compiler with fast and efficient register allocation. In *Eighth International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, volume 00, page 128, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[99] Tan Yiyu, Anthony S. Fong, and Yang Xiaojian. An instruction folding solution to a java processor. *Lecture Notes in Computer Science: Network and Parallel Computing*, 4672/2007:415–424, 2007.