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.