Compact Java Binaries for Embedded Systems

Derek Rayside, Evan Mamas, Erik Hons
Electrical & Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
{drayside, evan}@swen.uwaterloo.ca, eshons@shannon.uwaterloo.ca

Abstract

Embedded systems bring special purpose computing power to consumer electronics devices such as smartcards, CD players and pagers. Java is being aggressively targeted at such systems with initiatives such as the Java 2 Platform, Micro Edition, which introduces certain efficiency optimizations to the Java Virtual Machine. Code size reduction has been identified as an important future goal for ensuring Java's success on embedded systems [20]. However, limited processing power and timing constraints often make traditional compression techniques untenable. An effective solution must meet the conflicting requirements of size reduction and execution performance. We propose modifications to the file format for Java binaries that achieve significant size reduction with little or no performance penalty. Experiments conducted on several large Java class libraries show a typical 25% size reduction for class files and a 50% size reduction for JAR files.

1 Introduction

Embedded systems represent a rapidly growing segment of software development activity. It has been estimated that the market for embedded PC and 'soft' PC devices will exceed US $1 billion by the year 2001 [12]. These include new classes of devices such as smart pagers and cellular phones, traditional consumer electronics devices such as VCRs and televisions, and industrial products such as barcode scanners and point-of-sale terminals.

Embedded systems are characterized by severe restrictions on processing power, available memory and storage space. The K Virtual Machine (KVM), a part of the Java 2 Platform, Micro Edition, is a Java run-time environment targeted at 16 or 32-bit processors with approximately 128K of memory and clock speeds as low as 16Mhz [20]. Vendors such as Motorola, 3Com and NTT DoCoMo (Japan) have produced prototype cellular phones, pagers and Personal Digital Assistants (PDAs) using the KVM [16].

An important advantage of using Java for embedded systems is that code can be developed and debugged on workstations with sophisticated tools before it is moved onto the embedded system. This is the approach taken by IBM's VisualAge family of development tools. Motorola used this strategy successfully when replacing the scripting language of their PageWriter 2000X pager with a Java KVM [17].

Reducing the size of Java binaries has been identified as an important goal to ensuring Java's success on embedded systems [20]. In this paper we propose modifications to the file format of Java binaries that show significant size reductions for both `.class` and `.jar` files with little or no run-time performance penalty.

Regular class files can be converted to the proposed smaller form simply and quickly; the reverse is also true. The proposed form meets the Release-to-Release Binary Compatibility constraints described in the Java Language Specification [9, §13.1] (namely that external references be symbolic). Executing files in the new form requires either a slightly modified virtual machine or a customized class loader.

1.1 Related Work

Past work has achieved Java code size reduction by special application of standard compression techniques or by eliminating unused code. There has also been some work on other intermediate forms that are more compact than bytecode intermediate forms.

Code Compression Ernst et al [8] draw a distinction between compressed code formats that can be directly interpreted and those that are intended primarily for efficient transmission ('wire' formats). Past work in Java bytecode compression has focused on wire formats, such as CLAZZ, JAZZ, and Pugh [18]. CLAZZ [4, 10] explores the application of standard compression techniques, including ZIP, to particular sections of the Java class file. The results reported in [4, 2] show that CLAZZ achieves compression that is up to 25% better than ZIP alone. JAZZ [2] also uses standard compression techniques (in a slightly different fashion than CLAZZ), but focuses on JAR files instead of individual class files. Pugh has explored a number of compression techniques and is able to produce better results than JAZZ [18].

The significant difference between CLAZZ/JAZZ/Pugh and this work is that we are proposing a new and smaller format for Java binaries instead of compressing the existing structure (i.e. an interpretable format instead of a wire format). This new format will impose no significant penalty on execution time, as decompression is not required. The compression techniques proposed in CLAZZ/JAZZ/Pugh could also be applied to our new format.

There is one study [3] that focuses on compressing Java opcodes for embedded systems, but it disregards the constant pool. The argument given is that the symbolic information is discarded anyway, which may be true for certain embedded systems, but is not true for the Java KVM.

Eliminating Unused Code An effective way to reduce the size of a class file is to eliminate parts that cannot be used during execution. This is particularly true when small portions of large class libraries are used. Class Hierarchy Analysis (CHA) [7] performs a static analysis of the system to determine which methods may be invoked (taking polymorphism into account). Rapid Type Analysis (RTA) [1] is another static analysis technique that intersects the results of CHA with knowledge of the instantiated ('live') classes to further reduce the set of possible method invocations. Both CHA and RTA are also used by newer optimizing compilers to try and statically resolve polymorphic method invocations. These techniques are complimentary to the work presented here.

Two tools (JAX [21] and TOAD [15]) that perform CHA and RTA for Java code size reduction are available on the IBM alphaWorks website [14]; another such tool is described by Rayside and Kontogiannis in [19].

Alternative Intermediate Representations Kistler and Franz [11] have proposed the *slim binary* intermediate representation. A slim binary is based on the program's abstract syntax tree, and is encoded with an adaptive compression scheme. These files tend to be about half the size of Java class files. However, slim binaries cannot be interpreted: they require dynamic code generation at load time. This is considered to be competitive with traditional off-line compilation because processing speed has increased at a much greater rate than disk I/O: the load time of the slim binary file plus in memory compilation does not take much longer than loading a pre-compiled file. As slim binaries are compiled directly to native code, their execution speed is on par with code generated by traditional compilers.

Slim binaries are not appropriate for embedded systems due to the memory and processing power overhead that they require [11].

1.2 Organization

This paper is organized in the following fashion: Sections 2 and 3 provide some background information on Java; Section 4 discusses our proposed modifications to the Constant Pool; Section 5 discusses our proposed modifications to the Code attribute; Section 6 presents experimental results on several large class libraries; Section 7 concludes.

2 Classes and JARs

Java has two binary file formats: `.class` and `.jar`. A `.class` file contains a single Java class, and the fields and methods declared in that class. Fields and methods declared in parent classes are not included. Every class file contains exactly one class, even though many classes may be contained in a single source file. Class files are also used for Java interfaces, which are essentially a special type of abstract class.

A Java Archive (`.jar`) contains many `.class` files, as well as other resources such as text or graphics. JAR files also contain some security related information. JAR files use the popular ZIP file format, although the contents are often not compressed.

The remainder of this section provides an overview of the class file format, which is defined in the Java Virtual Machine Specification [13]. JavaWorld has published a number of good introductory articles to both the class file format and the Java virtual machine [24, 23, 22].

2.1 Class File Structure

Class files have six main sections: header, constant pool, fields, methods and attributes. The constant pool is similar to traditional compiler symbol tables. The bytecode notion of attribute is interesting: every class, field and method may have 'attributes' of arbitrary structure. Sun predefines a number of attributes, such as 'Code' that contain the instructions of a method body.

Physically, class files are a stream of 8-bit unsigned bytes. All values that require more than 8-bits are represented as a sequence of bytes. The Java Virtual Machine Specification [13] uses the notation `u1`, `u2`, and `u4` to represent one-byte, two-byte and four-byte sizes respectively (the `u` stands for 'unsigned'). The structure of the class file is defined as follows [13]:

```
ClassFile {

 u4 magic;
 u2 minor_version;
 u2 major_version;

 u2 constant_pool_count;
 cp_info constant_pool[constant_pool_count-1];

 u2 access_flags;
 u2 this_class;

 u2 super_class;

 u2 interfaces_count;
 u2 interfaces[interfaces_count];

 u2 fields_count;
 field_info fields[fields_count];

 u2 methods_count;
 method_info methods[methods_count];

 u2 attributes_count;
 attribute_info attributes[attributes_count];

}
```

2.2 Constant Pool

The `constant_pool` is a table of variable-length structures of type `cp_info`. `cp_info` records are used to represent class names, field names, method names, other string constants and numeric constants. The `constant_pool_count` is the number of `cp_info` entries in the `constant_pool` array.

The `cp_info` structure is the following [13]:

```
cp_info {
 u1 tag;
 u1 info[];
}
```

The `tag` is one byte long and indicates the type of the `cp_info` entry. A list of all the allowed tags is given in Table 1.

The info array varies depending on the type of the tag. The info array could be of fixed length in the case of an integer, or of variable length in the case of a UTF8 encoded string.

Constant Type	Tag Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_UTF8	1

Table 1. Constant Pool Entry Types

UTF8 Strings Strings in the constant pool are encoded in a variation of the 'standard' UTF8 format, as described in [13]. UTF8 encoding is, in a simplified sense, an extension to ASCII that allows for Unicode characters. The regular ASCII characters are encoded in one byte, and extended Unicode characters are encoded in two bytes.

The CONSTANT_String record in the constant pool contains a pointer to a CONSTANT_UTF8 record.

Constant Pool Indirection The constant pool contains some indirection that deserves discussion. The Fieldref, Methodref and InterfaceMethodref entries all have the same structure, described in Figure 1. The square boxes represent UTF8 encoded strings. The purpose of this structure is to take advantage of the fact that multiple classes may have fields and methods with identical signatures (often due to polymorphism).

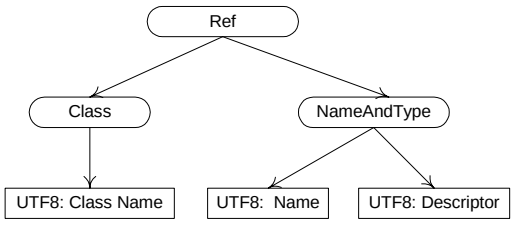

Figure 1. Ref Structures

2.3 Attributes

Attribute structures are used extensively throughout class files to describe classes, fields, methods and even other attributes. The Java Virtual Machine Specification defines a set of standard attributes, but compiler writers are free to create new attribute types.

Attribute structures are of variable length, and follow the form defined in [13]:

```
attribute_info {
 u2 attribute_name_index;
 u4 attribute_length;
 u1 info[attribute_length];
}
```

The standard attribute for classes is SourceFile; for fields it is ConstantValue. Methods have three standard attributes: Code, Exceptions and Deprecated. The Code attribute may have LineNumberTable and LocalVariableTable attributes for debugging information. The Synthetic attribute may be used by compilers for fields and methods when generating inner classes. The Code, Exceptions and ConstantValue attributes are required for proper execution.

2.4 Methods

The methods array contains all the methods (both instance and static) declared in this class or interface. Each entry is a variable length structure of the type `method_info` that contains a complete description of the JVM code for that particular method. The `methods_count` is the number of methods declared in this class. It should be noted that a variable number of `attribute_info` structures can be included in this structure. The format of the `method_info` structure is the following [13]:

```
method_info {
 u2 access_flags;
 u2 name_index;
 u2 descriptor_index;
 u2 attributes_count;
 attribute_info attributes[attributes_count];
}
```

2.4.1 Code Attribute

The Code Attribute is only used in the attributes table in the `method_info` structure. It contains all the JVM opcodes for a single Java method. Every method may contain at most one Code attribute. (Abstract methods cannot have a Code attribute).

The structure of the `code_attribute` is the following [13]:

```
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 attributes_count;
 attribute_info attributes[attributes_count];
}
```

The four-byte variable `code_length` indicates the size of the code array. The code array contains all the Java instructions, known as opcodes, along with their operands if any.

Opcodes and Operands A Java Virtual Machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands. Each opcode is represented by one byte, and therefore the JVM can support up to 256 different opcodes, although only 204 are defined [13].

The opcodes can be classified into distinct groups according to the type of operands they accept. Opcodes for method invocation take an index to a Methodref record; opcodes for object creation take an index to a Classref record in the constant pool. Another opcode group that is related to transferring control of execution accepts a 2-byte operand that is used as an offset to the code array. There are only two opcodes with a variable number of operands: tableswitch and lookupswitch, which are used for compiling `switch` statements.

3 Types And Strings

Types in Java are organized in a hierarchical fashion, as depicted in Figure 2. The two main categories of types are *Primitive* and *Reference* types. The Primitive types include `int`, `float`, `char`, etc. The Reference types include Arrays, Classes and Interfaces. (As mentioned before, Interfaces are a special type of abstract class).

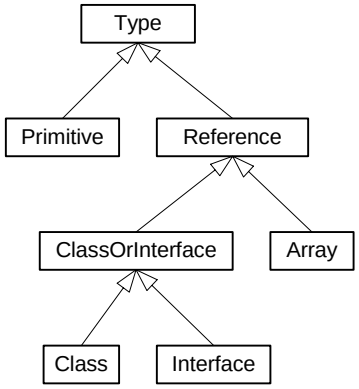

Figure 2. Types in Java

It is worth noting that the CONSTANT_Class entry in the constant pool actually represents a Reference, and not a Class type.

3.1 Mangled Names

The Java Virtual Machine Specification [13, §4.2] defines a grammar for representing the names of types in UTF8 strings that can be stored in the constant pool.

The mangled form of a class (or interface) name is the *fully qualified* name of the class (or interface) separated by slashes (`/`) instead of dots (`.`). The fully qualified name includes the complete package path of the class. For example, the fully qualified name of `Object` is `java/lang/Object`.

Primitive types (and void) are represented by a single character, such as `I` for `int` and `V` for `void`. To distinguish the class (and interface) names from the primitive names, class (and interface) names are given an `L` prefix and a `;` suffix.

The `[` character is used to indicate a one dimensional array, followed by the type of the elements in the array. An extra `[` character is used for each dimension. For example, a single dimensional array of `int`s would be represented as `[I`. A two dimensional array of `Objects` would be represented as `[[Ljava/lang/Object;`.

3.2 Method Descriptors

Method descriptors are strings formed by combining the mangled names of the parameter types and the return type of the method. All of the parameter types are surrounded by a single set of brackets, and the return type is appended at the end. The common `toString()` method, that returns the string representation of an object, has the descriptor `()Ljava/lang/String;`.

4 Symbols And Structures

The constant pool generally occupies most of the space in a class file (approximately 65% of the total size in the JDK). This section presents our proposed changes to the structure and content of the constant pool.

The proposed structure is based on a number of observations of the existing structure:

1. The fully qualified name of a class (or interface) represents a complete path, from the root

of the package tree to a node (the class or interface).

2. The constant pool does not explicitly reflect the hierarchical structure of types in Java.

3. Partitioning the constant pool according to record types could eliminate one byte per record for type identification, and could also reduce the size of constant pool indices from two bytes to one byte in some cases. This technique is also used in CLAZZ [4, 10] and Pugh [18].

4. It may be possible to eliminate some of the cross references in the constant pool by ordering the records such that these references are implicit in the ordering.

4.1 Package Tree

Java classes (and interfaces) are organized in packages, which form a tree structure. Essentially, a package is a directory in a hierarchical file system. We refer to this structure as the 'package tree': packages represent internal nodes; classes (and interfaces) represent external nodes (leaves).

Table 2 shows the average depth, average length of the tokens (strings) used to name each level, and the average total path length of some common Java class libraries. From this table we can see that the average depth of a leaf is typically about four or five levels. This makes perfect sense, as the naming convention for Java packages includes the domain name of the organization that wrote the code. For example, the XML4J packages all start with `com.ibm.xml4j`. The names for classes, interfaces, and packages tend to be between six and nine characters long. The Swing library has longer names due to the extensive use of inner classes: 59% (848 / 1444) of the classes are inner classes. The file name for an inner class is the short name of the outer class appended with `$` and the short name of the inner class.

Library	Depth (tokens)	Token Length (chars)	Path Length (chars)
Swing-1.1	4.54	9.02	44
JDK-1.1.7	3.56	7.43	29
JGL-3.1.0	4.88	7.76	41
XML4j-2.0.9	4.95	6.12	34

Table 2. Package Tree Averages

Table 2 shows that the average length of a fully qualified class (or interface) name is typically between thirty and forty-five characters. Therefore, the constant pool grows by thirty to forty-five bytes every time a class (or interface) is referenced (e.g. as a formal parameter in a method descriptor).

4.2 Constant Pool Modifications

We reduce the size of the constant pool by explicitly representing the package tree structure and the hierarchical organization of types in Java. This explicit representation allows us to replace string references to types with indices to the explicit representation.

Our proposed modifications to the constant pool records are described next.

4.2.1 Replaced Records

The original record format of the records that we are replacing is [13]:

```
ClassRef {
  u1 tag;
  u2 name_index;
}

(Field | Method | InterfaceMethod)ref {
  u1 tag;
  u2 class_index;
  u2 name_and_type_index;
}

NameAndType {
  u1 tag;
  u2 name_index;
  u2 descriptor_index;
}
```

4.2.2 Modified Records

We replace the Fieldref, Methodref, InterfaceMethodref and NameAndType structures with those given below. Essentially, the NameAndType structures have been collapsed into the Fieldref and Methodref structures. `ti` indicates a 'type index':

an index to the type name records described in the next section. A one-byte `ti` is used for individual class files, and a two-byte `ti` is used when combining the constant pools of many class files (i.e. for a JAR file). None of the 2500 class files in our experiments required more than 128 type indices, so a one-byte `ti` should be sufficient for any individual class file.

```
FieldRef {
  ti class_or_interface_index;
  u2 name_index;
  ti field_type_index;
}

MethodRef {
  ti class_or_interface_index;
  u2 name_index;
  u1 method_descriptor_index;
}
```

The FieldRef record contains a type index both to the type that declares the field and to the type of the field.

4.2.3 New Records

Type Names The following records explicitly represent the package tree and the hierarchical organization of types in Java. Every node on the package tree (package, class or interface) is represented by a QualifiedName record. This record points to a UTF8 string with the name of the node, and to its parent node (another QualifiedName). If a QualifiedName has no parent (i.e. is at the root), than the `parent_index` points to a reserved value (or itself if the implementation does not wish to reserve a value).

The PrimitiveAndVoidName records are not strictly required, since the complete set of primitive types is defined in the Java Language Specification [9]. These records can be eliminated by reserving eight index positions for primitives and one for void in this partition of the constant pool.

The ArrayName record has a `tag` to distinguish it from the other records. The exact value of this `tag` is only important for implementation purposes.

```
PrimitiveAndVoidName {
  u1 indicator;
}

ArrayName {
  u1 tag;
  u1 dimensions;
  ti type_index;
}

QualifiedName {
  u2 name_index;
  ti parent_index;
}
```

One byte could be used for the `name_index` field in the QualifiedName structure if the name strings were placed at the beginning of the UTF8 partition of the constant pool. Each QualifiedName requires at most one unique UTF8 string, and our experiments indicate that individual class files do not have more than 128 QualifiedName records. Our size reduction experiments were conducted using a two byte `name_index` field, as our implementation does not order the constant pool in this way.

Note that the constant pool could be made even smaller if types were not referenced by their names. The Java run-time environment uses the string representation of a class (or interface) name to search the classpath for the appropriate file. In a system where the set of types is statically fixed, which may be the case for certain embedded systems, it would be possible to replace the string representation of type names with some sort of direct reference. In that case, the QualifiedName records should be modified to include this direct reference and have the `name_index` field removed. Replacing symbolic references with direct references violates the Release-to-Release Binary Compatibility constraints specified in [9, §13.1], but this may not be a concern in tightly controlled circumstances.

Even if the symbolic linking information is removed from the class file, it is still beneficial to explicitly represent the hierarchical organization of types as proposed here.

Method Descriptor Method descriptors are represented as strings in the original constant pool, as discussed above. We represent method descriptors as a sequence of indices into the type partition of the constant pool:

```
MethodDescriptor {
  u1 parameter_count;
  ti parameter_indices[parameter_count];
  ti return_type_index;
}
```

The majority of the size reduction from our proposed modifications comes from expressing method descriptors in this fashion, instead of encoding them in UTF8 strings.

4.3 Alternative QualifiedName Structure

An alternative to the QualifiedName structure proposed above is to break a fully qualified name into only two strings: one for the class (or interface) and one for the package. This representation does not fully decompose the package tree (the QualifiedName structure is essentially a node on the package tree). Such an alternative structure is proposed in [18], and involves replacing the QualifiedName structure with the following:

```
ClassOrInterfanceName {
    u2 name_index;
    u2 package_name_index;
}
```

Table 3 shows a mathematical comparison of the cost, in bytes, of each approach. The QualifiedName approach is identified as 'Tree', and the alternative as 'Non-tree'. The 'Initial Cost' column shows the cost of adding a completely new path, and the 'Incremental Cost' column shows the cost of adding a new path that is a variant of an existing path.

n represents the number of nodes in the path, and l_i represents the length of the name of node i. The ubiquitous '+2' indicates the overhead required to record the length of a UTF8 string. The alternative approach requires a delimiter character (i.e. '/') between package names, hence the '+1'. The '+3' in the tree approach indicates the size of the QualifiedName structure: this could be reduced to 2 if a one byte index was used to the name (as discussed above).

Approach	Initial Cost	Incremental Cost
Tree	$\sum_{i=1}^{n} l_i + n \times (3+2)$	$l_{n+1} + 3 + 2$
Non-tree	$\sum_{i=1}^{n} (l_i + 1) - 1 + 2$	$\sum_{i=1}^{n+1} (l_i + 1) - 1 + 2$
Difference	$4n - 1$	$-\sum_{i=1}^{n} (l_i + 1) + 3$

Table 3. Space Comparison

As can be seen in Table 3, the initial cost of the tree-based approach is more expensive by $O(n)$; however, the incremental cost of the tree-based approach is less expensive by $O(l)$ (where l represents $\sum l_i$). The initial cost is incurred less frequently than the incremental cost, and n is necessarily shorter than (or equal to) l. Table 2 shows that n is typically less than l by a factor of 6–9. Therefore, the tree-based approach will produce smaller files in most cases.

There may be individual class files where the initial cost of the tree-based approach will overshadow the incremental savings, but these will be rare. For example, the initial cost to represent `java.lang` is 11 bytes for the non-tree approach and 18 bytes for the tree-based approach. The incremental cost of representing `java.lang.reflect` is then 19 bytes for the non-tree approach and 12 bytes for the tree-based approach. With only these two packages the cost of both approaches is equal: further JDK packages will incur the incremental cost, and so the tree-based approach will produce the smaller file.

4.4 Partitioning

The constant pool can be partitioned according to record type. Using the new record types we propose, the structure of the constant pool becomes:

```
constant_pool {

  // size of each partition
  u2 field_ref_count;
  u2 method_ref_count;
  u2 method_descriptor_count;
  u2 type_count;
  u2 literal_constant_count;
  u2 utf8_count;

  // partitions
  field_refs[field_ref_count];
  method_refs[method_ref_count];
  method_descriptors[method_descriptor_count];
  types[type_count];
  literal_constants[literal_constant_count];
  utf8[utf8_count];

}
```

Our experiments show that the average constant pool has approximately 150 records, and so eliminating the record identification tag saves about 150 bytes per class file.

4.5 Implicit Indices

Some of the records in the constant pool, such as Fieldref, contain indices to other constant pool records. It is possible to eliminate some of the indices by ordering the records in the constant pool such that the index is implicit in the ordering. For

example, all of the fields declared by a particular class (or interface) can be grouped together and the `class_or_interface_index` can be removed from the records.

It is also possible to eliminate the `name_index` from some of the records by sorting the UTF8 partition in the same order as the `name_index` fields appear above. However, there may be small complications as multiple records may refer to the same UTF8 record. One possible solution to this is to duplicate the UTF8 record for each record that refers to it. Another approach would be to insert special records in the UTF8 partition (instead of the duplicates) that point to the proper UTF8 record.

Our experiments show that overall space savings can be achieved, even using UTF8 duplication. However, using implicit indices makes the file format more complicated and may introduce difficulties for verification.

4.6 Execution Characteristics

The proposed modifications to the constant pool should have little or no impact on the execution time of the class files, for the simple reason that these modifications only affect the linking phase of the virtual machine. Linking is the process of combining a class file into the run-time state of the virtual machine, and involves three steps: verification, preparation and resolution [13, §2.16.3].

The verification process ensures that the class file conforms to the constraints given in [13, §4.9]. The majority of the cost of this verification is in performing data-flow analysis on each method, which is not affected by modifications to the constant pool. Partitioning the constant pool according to record type and explicitly representing the hierarchical organization of types in Java should make the verification a bit easier. However, many embedded systems will probably not perform verification: they will assume that the code they have been given is well-formed.

Preparation is the process of configuring the interpreter's internal data structures (such as method tables) for the given class: it is not affected by the constant pool reorganization.

Resolution is the process of resolving symbolic (string) references [13, §5] in the constant pool. This process is affected by the constant pool reorganization proposed here. Typically, the symbolic references are only resolved once, and then replaced with a direct reference [13, §2.16.3], [9, §12.3.3]. Our equivalent of the ClassRef structure (QualifiedName) takes a bit longer to resolve, as the path back to the root will need to be reconstructed. This operation is linear in the number of nodes, and Table 2 shows us that there are typically three to five nodes. However, our MethodDescriptor structure should be faster: reading a string-encoded method descriptor requires simple parsing, and then a lookup for the direct reference using the string as a key. Our MethodDescriptor structure encodes indices, which can be mapped to the direct references with an array; thereby avoiding the simple parsing and string-based lookup. This array will consume a small amount of extra memory (linear in the number of types referenced), and can only be used once the symbolic reference has been resolved to a direct reference.

In conclusion, the proposed modifications to the constant pool should have little or no impact on the run-time of an application.

5 Code Attribute Reduction

The most important field inside the Code Attribute is the code array. This array contains all of the Java instructions and their operands for a single method. The statistics collected on the JDK base libraries revealed that the opcodes and the operands contribute equally to the size of the code array. In our approach, we separate the opcodes from the operands and apply different techniques to each group. Reconstructing the original code from the two compressed groups will be trivial since each opcode is followed by a specified number of operands.

An exception to this rule is the `tableswitch` and `lookupswitch` opcodes, which can have a variable number of operands. For every occurrence of these opcodes we need to include the number of operands in order to be able to restore the original code array. It turns out that these two opcodes do not occur very frequently and thus our overall size reduction is not greatly affected.

The techniques applied to the opcodes and operands are described in detail in the next two sections.

5.1 Opcodes

Every opcode is represented by a single byte inside the class file, allowing up to 256 different opcodes. Only 204 of these opcodes are used by the Java Virtual Machine Specification [13]. Moreover, the frequencies with which the opcodes appear inside class files have a very uneven distribution as seen in Figure 3. These facts present two opportunities for representing the code attribute in a more compact form.

The first approach is to use the Huffman algorithm to reduce the number of bits required to represent the most frequent opcodes and increase the number of bits for the infrequent opcodes. The Huffman algorithm is used to create a tree of symbols based on their frequency distribution. A table is also constructed to associate every opcode with a unique prefix code. When an opcode is compressed a table lookup is required to determine the corresponding Huffman code. All the compressed codes are concatenated to yield a long sequence of bits. When an opcode has to be decompressed the Huffman tree is used to retrieve the original symbol in $\log n$ steps where n is the depth of the tree. The depth of the tree for 204 opcodes typically ranges between 20 and 22. The advantage of prefix codes is that when they are concatenated in a sequence we can decompress them one at a time starting from the beginning of the sequence without having to examine the whole sequence. Using the frequency distribution from Figure 3, we calculate that on average a single opcode can be represented using only 5.4 bits instead of 8 bits. This yields a size reduction of 32.5% on average.

The second approach is based on the observation that some opcodes are very likely to occur in pairs. Using the Huffman algorithm in a manner similar to the first approach we can associate each pair of opcodes with a unique prefix code. Since we examine opcode pairs only, we have to deal with a scenario where the number of opcodes is odd. In such a case we also have to associate a symbol to every single opcode. The number of symbols that have to be added to deal with the previous scenario is insignificant compared to the total number of symbols. Therefore, the overall size reduction will not be affected. On the other hand, there is a performance penalty for this approach since the number of prefix codes is much larger ($\simeq 204^2$) than the previous approach. Based on the frequency distribution of opcode pairs, we calculate as before that on average an opcode pair can be represented using only 9.2 bits as opposed to 16 bits. This yields a size reduction of 43% on average.

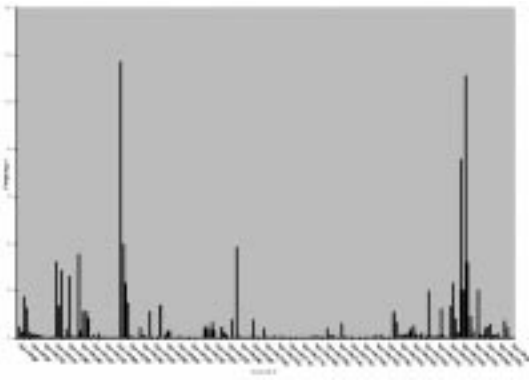

Figure 3. Opcode frequency distribution

The third approach is an attempt to exploit different patterns that exist in the code generated by Java compilers, such as `javac`. Each opcode can be represented as a unique state in a Markov state model [5]. A sequence of opcodes can then be represented by the first state and a sequence of state transitions. The benefit of such a representation is that we can identify distinct groups of opcodes that are very likely to be followed only by opcodes from some other groups. Again, the Huffman algorithm will be used to determine the prefix codes that represent the possible transitions between all states. In our attempt, we identified four distinct opcode groups using the frequency distributions in Figure 3. The opcodes were sorted by frequency and the groups were formed in such a way that group 1 had the most frequent opcodes and group 4 had the least frequent opcodes. The average number of bits required to represent the opcodes is therefore reduced from 8 bits to 5.9 bits yielding a size reduction of 26%. It should be noted that this approach was expected to achieve higher size reduction and the current results are attributed to identifying only sub-optimal groups.

In a different approach, we created 'custom opcodes' to represent commonly occuring sequences of real opcodes. These custom opcodes were numbered 204 through 256: the 52 unused instruction numbers in the Java VM specification [13]. Statis-

tics were collected on all opcodes sequences of size 2 and 3. By using custom opcodes to represent the most frequent 52 opcode sequences of size 2, we reduce the size of the total opcodes by 23%. Alternatively, a combination of the most frequent sequences of size 2 and 3 could be used to achieve similar results. Compared to the previous approaches, the custom opcode approach has lower memory and processing overhead, but does not provide the same degree of size reduction. Similar approaches have been attempted in [18] and [3].

5.2 Operands

Every opcode is followed by a predefined number of operands (except the two switch opcodes, as discussed previously). The operands that follow an opcode represent a specific type of information. We can identify opcode groups based on the type of operands they have. The majority of the operands are one of the following types: Constant Pool Index and Method Offset. For each of these types we apply the following techniques.

A Constant Pool index is two bytes long and is used to refer to a Constant Pool entry. Over 80% of the class files we examined had a constant pool with size less that 256 entries. Therefore, we can use only one byte to index the Constant pool and we save the remaining one byte. Implementing this is very efficient since we only have to check whether the constant pool entries are less than 256. An improvement on the previous approach would be to use only the required number of bits to reference constant pool entries. A Method Offset is also two bytes long and it is used by branch opcodes to refer to another opcode were the execution will continue. The Method offset is a signed number that represents the actual number of bytes from the current opcode to the target opcode. The Method offset includes both the opcode bytes and the operand bytes. This count includes not only the opcodes but the operands as well. By using only the number of opcodes we may be able to eliminate one of the two bytes. Another check we can perform is to count if the number of opcodes is less than 256. In turns out that 70% of the time the latter is true. Implementing the method offset reduction is very efficient since we only have to count the number of opcodes in our method and check if it is less than 256. As previously stated, only the required number of bits can be used.

6 Experiments

6.1 Test Data

The following four common Java class libraries were used as test input for our experiments:

Library	Files	Size (KB)
Sun Swing 1.1	1444	3542
Sun JDK 1.1.7 (`java.*`)	713	1698
ObjectSpace JGL 3.1.0	262	874
IBM XML4j 2.0.9	232	617

Table 4. Libraries

6.2 Class File Analysis

The Constant Pool comprises 65% of the total class file (on average), while the Code Attributes inside the Methods structure comprises another 15% (on average). The remaining size of the file can be attributed to all other class sections such as Interfaces, Fields and Attributes. Most of the space in the constant pool is occupied by UTF8 entries. NameAndType and Method and Field entries also have a noticeable contribution to the Constant Pool size. The contribution of the Methods structure can be attributed equally to the opcodes and the operands inside the Code Attribute.

Therefore, the primary goal of the new format is to restructure the constant pool to reduce the total size of the UTF8 strings. Recall that method descriptors are stored in UTF8 strings in the standard constant pool.

6.3 Class File Size Reduction

Experimental results for applying the techniques described above to individual class files are given in Table 5. The numbers reported for using implicit indices in the constant pool were computed on constant pools in the original format.

As expected, restructuring the constant pool gives the best results, with an average 24% size reduction. The opcode size reduction and implicit references produce about a 6% size reduction each, with low standard deviation.

Library	Restructuring	Opcodes	Implicit
Swing-1.1	23%	5.6%	6.8%
JDK-1.1.7	14%	5.8%	5.1%
JGL-3.1.0	35%	5.2%	5.5%
XML4j-2.0.9	23%	8.9%	6.3%

Table 5. Class File Size Reduction

Constant pool restructuring gives the best results consistently, but they vary between 14% and 35%. We expected the JDK to show the least improvement, as it has, based on the analysis of the package trees in Table 2. The reason for this is that the average path length, in characters, is less than for the other libraries. The JDK is a special case because it does not have to follow the package naming conventions used by other libraries.

The Java Generic Library (JGL) shows the greatest improvement from restructuring the constant pool: 35%. By examining where these savings come from in Table 6 we see that the JGL gets a proportionally bigger gain from the UTF8 strings. Eliminating the record tags, old NameAndType records, and old ClassRef records produces roughly the same results in each case.

Library	UTF8	Tag	NameAndType	ClassRef
Swing-1.1	67%	13%	16%	4%
JDK-1.1.7	60%	18%	16%	6%
JGL-3.1.0	83%	7%	8%	2%
XML4j-2.0.9	67%	13%	15%	5%

Table 6. Class File Restructuring Breakdown

As expected, the majority of savings come from replacing method descriptors encoded in UTF8 strings with the new MethodDescriptor records. This expectation is corroborated by Table 7, which shows that the JGL has a greater number of parameters per method descriptor. Each parameter of reference type typically consumes the number of characters specified by the average package tree path length.

If we assume that all of the extra parameters in the JGL are of reference type, than the extra space saved is:

$$(1.70 - avg(parm/desc)) \times 4503 \times 41 = 110,774$$

Library	Parameters	Descriptors	Ratio
Swing-1.1	20 117	17 652	1.14
JDK-1.1.7	6931	6485	1.07
JGL-3.1.0	7644	4503	1.70
XML4j-2.0.9	3495	3168	1.10

Table 7. Parameters per Method Descriptor

($avg(parm/desc) = (1.14 + 1.07 + 1.10)/3$; 4503 is the number of descriptors in the JGL (Table 7); 41 is the average path length (Table 2). Notice that 110,774 is 12.7% of 874K (the size of the JGL): this 12.7% is almost exactly the difference between the savings for the JGL and Swing/XML4J (35% – 23%).

6.4 Jar File Size Reduction

A `.jar` file contains a set of `.class` files, a manifest (text encoded), and possibly resource files that may be either text or binary. We examined the code size reduction possible by using a single constant pool for a set of class files, as would be found in a jar file. We did not examine the other files that may be found in a jar file.

The results given in Table 8 show that the overall code size can be reduced by about 45%–50% by using a common constant pool in the new form. This is approximately twice the reduction from treating class files individually.

Library	Restructuring	Improvement
Swing-1.1	49%	× 2.13
JDK-1.1.7	32%	× 2.29
JGL-3.1.0	53%	× 1.51
XML4j-2.0.9	45%	× 1.96

Table 8. Jar File Size Reduction

Only the results for constant-pool restructuring are shown in Table 8 because the opcode improvement does not change, and using implicit indices in the constant pool shows a negligible improvement with a combined constant pool. We show the results of combining the restructured constant pools: an improvement would also be expected if combining constant pools in the original format (one of the approaches used in [2, 18]).

As before, JGL shows the greatest improvement and JDK the least. Swing shows a slightly better result than XML4J (49% vs. 43%), whereas previously they were the same. This can be explained by the fact that Swing has a longer average path length (Table 2) and slightly more parameters per descriptor (Table 7).

7 Conclusions

We have shown a number of techniques for reducing the size of Java class files, including a new format for the constant pool, reorganizing the opcodes and the idea of implicit indices. Of these, restructuring the constant pool shows the most significant size reduction. Experiments conducted on four common Java class libraries indicate that a 25% size reduction can be expected in typical cases.

Experiments also show that the size reduction is about double (50%) when using a single, restructured constant pool for all of the `.class` files in a `.jar`.

The degree of size reduction varies with the code analyzed, and has been shown to correlate with the average path length (in characters) of the package tree, and the number of parameters per method descriptor. Due to the standard Java package naming convention, it can be expected that all code will show significant size reduction.

The new constant pool format is fairly simple, and should incur no run time overhead in virtual machines adapted for it. Class files can be converted to the new format quickly and easily (our prototype implementation took only a few days to build). At the suggestion of the referees, we may investigate creating a Java Specification Request through the Java Community Process for Sun to modify the class file format.

Embedded systems promise to be an important area of software development in the future, and Java will play a growing role in this area. Reduced binary code size will lower the hardware requirements, and therefore the cost, of embedded systems using Java technology.

Acknowledgments

We would like to thank Dr. Kostas Kontogiannis and Dr. Amir Khandani of the University of Waterloo; Bill O'Farrell and Stephen Perlgut at IBM Centre for Advanced Studies; the IBM Centre for Advanced Studies and the High Performance Java S/390 Group at the IBM Toronto Laboratory for their support.

We would like to express our gratitude to all the reviewers for their valueable suggestions and constructive comments.

We would also like to acknowledge the bytecode parser produced by Markus Dahm at the Free University in Berlin [6], which he graciously makes available under the GNU Public License.

About the Authors

The authors are currently graduate students in the Electrical & Computer Engineering Department at the University of Waterloo. Derek Rayside and Evan Mamas are pursuing their work in association with the IBM Centre for Advanced Studies at the IBM Toronto Laboratory.

References

[1] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of ACM/SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA) '96*, pages pp. 324 – 341, 1996.

[2] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of CASCON '98*, Toronto, December 1998.

[3] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for embedded systems. Technical Report 1213, Institut de Recherche en Informatique et Systèmes Aléatoires, December 1988.

[4] J. D. Corless. Compression of Java class files. Master's thesis, University of Victoria, 1994.

[5] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[6] M. Dahm. JavaClass API 3.1.2. http://www.inf.fu-berlin.de/ dahm/JavaClass/.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceeding of AITO European Conference on Object-Oriented Programming (ECOOP) '95*, August 1995.

[8] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997.

[9] J. Gosling, B. Joy, and G. Steele Jr. *The Java Language Specification*. Addison Wesley, 1996.

[10] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software – Practice and Experience*, 28(12):1253 – 1268, October 1998.

[11] T. Kistler and M. Franz. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21 – 34, February 1999.

[12] B. Lee. Internet embedded systems: Poised for takeoff. *IEEE Internet Computing*, 2(3):pp. 24–29, May 1998.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[14] IBM alphaWorks Website. . http://alphaworks.ibm.com.

[15] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with Java. In *Proceedings of CASCON '98*, Toronto, December 1998.

[16] Press Release. What is the Java 2 Platform, Micro Edition?, June 1999. http://java.sun.com/features/1999/06/j2me.html.

[17] Press Release. Taking it to the streets: Motorola and the K Virtual Machine, June 1999. http://java.sun.com/features/1999/06/moto.html.

[18] W. Pugh. Compressing Java class files. In *Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation (PLDI) '99*, May 1999.

[19] D. Rayside and K. Kontogiannis. Extracting Java library subsets for deployment on embedded systems. In *Proceedings Conference on Software Maintenance and Re-engineering (CSMR)*, Amsterdam, 1999.

[20] Sun Microsystems. The K Virtual Machine (KVM) White Paper. Technical report, Sun Microsystems, 1999.

[21] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Proceedings of ACM/SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA) '99*, November 1999.

[22] B. Venners. Under the Hood: Bytecode basics. *Java World*, September 1996. http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html.

[23] B. Venners. Under the hood: The Java class file lifestyle. *Java World*, July 1996. http://www.javaworld.com/ javaworld/jw-07-1996/jw-07-classfile.html.

[24] B. Venners. Under the hood: The lean, mean, virtual machine. *Java World*, June 1996. http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html.