SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(12), 1253–1268 (OCTOBER 1998)

Tailored Compression of Java Class Files

R. NIGEL HORSPOOL AND JASON CORLESS
Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6
(email: nigelh@csr.uvic.ca)

SUMMARY

Java class files can be transmitted more efficiently over a network if they are compressed. After an examination of the class file structure and obtaining statistics from a large collection of class files, we propose a compression scheme that is tailored to class files. Our scheme achieves significantly better compression than commonly used methods such as ZIP. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: data compression; Java; class file

INTRODUCTION

The Java programming language[1] and its implementation using a Java Virtual Machine (JVM)[2] have greatly simplified the task of developing web-based application programs. In this and in other roles, Java has been a runaway success. When a Java program is compiled, it is translated into a collection of *class* files. Each class file contains a variety of components, including instructions for the JVM as well as data constants, interface specifications and other information. When a remote user executes a Java applet, the class files are downloaded over the internet onto the user's machine and interpretively executed by a copy of the JVM on that machine. An alternative is that the class files might be translated on the user's machine into native machine code using a Just-In-Time (JIT) Java compiler.

It is clearly advantageous for the Java class files to be made as small as possible. The smaller the file, the shorter the transmission time to deliver the file to its destination. If the user is being charged for connect time or for the number of data packets delivered, then smaller files would also have an economic benefit. Transmission of a class file in a compressed format and decompressing the file on the user's machine should improve overall performance provided that the decompression process consumes reasonable amounts of computation, and does not occupy excessive amounts of main memory.

There are many general purpose data compression programs which could be used to reduce the size of a Java class file. Some examples of widely available compression programs are *gzip*, *zip*, and *compress*.[3] However, they tend not to be as effective on Java class files as on file formats. The reason is that these compression programs work by finding repetitions and by coding the repeated patterns of data in an efficient

Received 10 September 1997
CCC 0038–0644/98/121253–16$17.50 *Revised 19 March 1998*
 Accepted 17 April 1998

manner. The longer the input, the more opportunity there is for finding repetitions and the better the compression usually becomes. Unfortunately, a typical class file for a Java program is quite short, perhaps just a few hundred bytes in size, and the file is organized into several sections. Each section contains data in a different format. It is unlikely that the data in one section of a class file will repeat any patterns of bytes encountered in a previous section of the file. Given the short size of a class file and the fragmented nature of each file, a general-purpose compression algorithm has too little opportunity to adapt to a class file section before the end of that section is reached.

There has been some work on compression methods that are specifically targeted to executable files or tuned for such files. Recently, Ernst *et al.*[4] reported a compressed 'wire' representation that reduces SPARC code to 21 per cent of its original size. However, as we will argue, their techniques would be relatively ineffective on Java class files because of their small size. Yu's[5] approach achieves excellent compression on executable files and is well suited to its main task of compressing software that is being distributed on floppy disks. It too benefits from having reasonably large quantities of data to compress and would also be ineffectual if applied separately to small files like the Java class files. Yu's algorithm is based on LZSS[3,6] with a non-greedy matching heuristic and, in that regard, is similar to gzip.

If we wish to achieve good compression on Java class files, the only realistic approach is to develop a compression program which has been customized to the Java class file format. Such a program would waste very little time adapting itself to a particular file and would be able to achieve some compression on even the smallest files.

In this paper, we report on a compression/decompression program named clazz which was developed specifically for Java class files. Our program outperforms both gzip and bzip2[7] by a wide margin. The bzip2 method is close to being the best available general-purpose compression method for text files. Detailed information about the clazz program and the experimental methodology is available in the second author's MSc dissertation.[8]

The following sections of this paper provide an overview of the Java class file structure, then explain how we developed a customized compression method, and finally report on how well our method works on a collection of class files.

STRUCTURE OF JAVA PROGRAMS AND JAVA CLASS FILES

A Java program could comprise a stand-alone application program or an applet to be invoked from a web page. In either form, the program will have been constructed as a collection of Java classes. It is a requirement of some of the standard Java compilers that each class declared as 'public' must be compiled separately. In effect, a Java source code file should contain a declaration of exactly one public class plus, optionally, some declarations of private classes. Each source code file is translated by the compiler into a so-called *class file*. For example, if an application program is constructed from source code files named Main.java, One.java and Two.java then the compilation command

```
javac Main.java One.java Two.java
```

using Sun's JDK implementation on a Sun workstation will create three files with the names Main.class, One.class and Two.class.

A group of related class files may be stored together on the computer's hard drive as a *package*. It is a common practice to use zip file format[9] for this collection of files, thus making it simpler to treat a package as a single entity while also saving disk storage. The Java Archive (JAR) format also provides a mechanism for grouping class files into a single entity for shipping over a network connection. It too is based on the zip file format, although the capability of compressing members of the archive does not appear to be used by Sun when distributing their class file libraries. If the compression capabilities of the zip file or JAR file format are used, it should not be forgotten that the zip file or JAR file is constructed from independently created class files, and that individual class files can be extracted from the collection and used. The files are compressed independently in order to facilitate extraction and replacement of a single file. Consequently, no benefit to compression performance is obtained by combining several small files into a single archive.

A Java program is normally executed by invoking the Java interpreter, specifying to it the class file where execution is to begin. A less common alternative, but one that is growing in popularity as JIT compiler technology develops, is to translate the bytecode of each method into native code when it is invoked for the first time. For a stand-alone application, execution begins at a standard method named main; for applets, the class file has to implement other standard methods. While the program executes, it will occasionally make a reference to a class type defined in another file and which has not been previously accessed. In this case, the corresponding class file must be dynamically loaded before execution can continue. For a program invoked as an applet from a web browser, dynamic loading will typically require that the class file be fetched from a host computer elsewhere on the network.

Regardless of the architecture of the client computer where the Java program is to run, a class file always has the same format. The instructions in the class file are instructions for a Java Virtual Machine (JVM).[2] The client computer would execute the class file either by using an interpreter for the JVM or a JIT compiler. However, to maintain architecture neutrality, JIT translation is performed on the client computer. The class file is in the standard format and not in native code form when it is fetched over the network.

A slightly simplified picture of the overall layout of a class file is shown in Figure 1. What should be observed from the picture is that the class file is organized as a series of sections. One section in particular, the *Methods Section*, is itself organized as a series of method entries, where each entry is itself subdivided into sections. We explain the structure of some of the more important sections below.

The Constant Pool

Constants occurring in a Java source code file are converted by the compiler into entries in the constant pool. The compiler may also create many additional entries for constants which do not explicitly appear in the source code but which are needed at execution time. As represented in a class file, each entry in the constant pool consists of a tag byte followed by a group of bytes that contain the value of the constant. For example, a UTF8 string constant containing 10 characters would be stored as a 13-byte entry in the constant pool. The first byte is a tag with value 1,

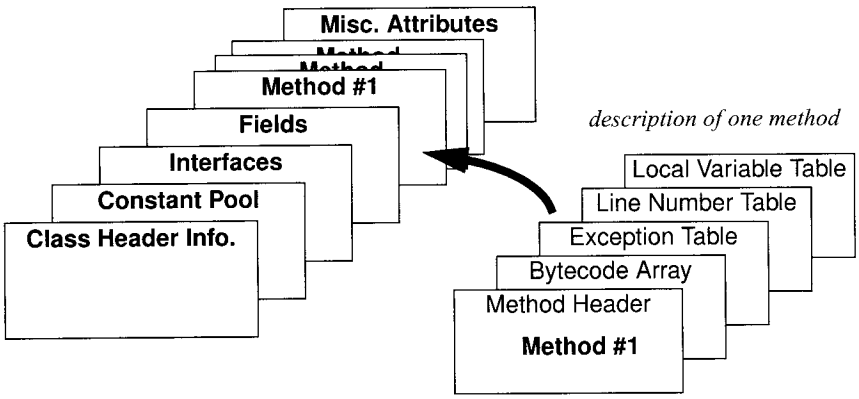

Figure 1. Class file layout

the next two bytes hold the string length, and the following 10 bytes hold the characters of the string constant.

The constant pool produced for the small sample program of Figure 2 is shown in Figure 3. Each entry is written as a tag name followed, in square brackets, by the additional information required for that tag.

The relatively large size of the constant pool compared to the original program should be noted. Many field names, class names, and type signatures are included as character strings.

```
class Small {
        static final int CUTOFF = 8;

        public int foobar( int a, int b ) {
                int val = a + b;
                if (val > CUTOFF) {
                        return 4*val;
                } else {
                        return 3*val;
                }
        }

        static void main( String args[] ) {
                Small s = new Small();
                System.out.print("Return value is:");
                System.out.println(s.foobar(4,3));
        }
}
```

Figure 2. A sample Java source file

```
 1:  String [39]
 2:  Integer [8]
 3:  Class [42]
 4:  Class [23]
 5:  Class [35]
 6:  Class [32]
 7:  Methodref [6, 13]
 8:  Methodref [5, 13]
 9:  Methodref [4, 16]
10:  Methodref [6, 17]
11:  Fieldref [3, 14]
12:  Methodref [4, 15]
13:  NameAndType [40, 43]
14:  NameAndType [33, 41]
15:  NameAndType [20, 34]
16:  NameAndType [18, 19]
17:  NameAndType [36, 22]
18:  Utf8 [7, "println"]
19:  Utf8 [4, "(I)V"]
20:  Utf8 [5, "print"]
21:  Utf8 [13, "ConstantValue"]
22:  Utf8 [5, "(II)I"]
23:  Utf8 [19, "java/io/PrintStream"]
24:  Utf8 [10, "Exceptions"]
25:  Utf8 [15, "LineNumberTable"]
26:  Utf8 [1, "I"]
27:  Utf8 [10, "SourceFile"]
28:  Utf8 [14, "LocalVariables"]
29:  Utf8 [4, "Code"]
30:  Utf8 [10, "Small.java"]
31:  Utf8 [6, "CUTOFF"]
32:  Utf8 [5, "Small"]
33:  Utf8 [3, "out"]
34:  Utf8 [21, "(Ljava/lang/String;)V"]
35:  Utf8 [16, "java/lang/Object"]
36:  Utf8 [6, "foobar"]
37:  Utf8 [4, "main"]
38:  Utf8 [22, "([Ljava/lang/String;)V"]
39:  Utf8 [16, "Return value is:"]
40:  Utf8 [6, "<init>"]
41:  Utf8 [21, "Ljava/io/PrintStream;"]
42:  Utf8 [16, "java/lang/System"]
43:  Utf8 [3, "()V"]
```

Figure 3. Constant pool entries

The Methods Section

Each entry in the Methods Section is a variable-length structure that describes one method in the class. In addition to some information about the class (its name, access permissions, etc.), the entry normally contains both a *Code* attribute and an *Exceptions* attribute. The Code attribute contains an array of the bytecode that is to be interpretively executed by the JVM when this method is invoked. It also includes a table providing information about exception handlers in the code and, potentially, a *LineNumberTable* and a *LocalVariableTable* which would enable a debugger to relate the bytecode and the local variables of the method to its source code. The Exceptions attribute is normally small.

A COMPRESSED 'WIRE' REPRESENTATION FOR JAVA?

A recent paper[4] describes an approach that compresses intermediate code from the lcc C compiler to as little as 21 per cent of the corresponding executable code for the SPARC architecture. This is a compression rate of 4·9 to 1. At first sight, it would appear that a similar approach should be effective for Java. The Class file format is similar in nature to the intermediate code, or IR code, generated by the front-end of a conventional compiler.

The 'wire' format for C code is explained in Ref. 4. It is based on a heuristic tree pattern matching method for compressing IR code that is described in Ref. 10. It involves splitting the sequence of tree patterns into separate streams and then applying three different compression methods to the streams—Move-To-Front encoding, Huffman coding and gzip compression.

If we were to develop a similar approach for Java, we would need to re-engineer the Java compiler so that it generates trees instead of the linearized byte code and

where constants appear as leaves in those trees instead of having been separated out into a special Constants section. In principle, it should be possible though inefficient to construct such trees from the information in the Class file.

The reason why we consider such an approach to be unappealing however is provided by the compression results reported in Ref. 4. These results are reproduced in Table I. As can be seen, wire format achieves its reported 4·9 to 1 compression ratio only on the largest file. For the smallest of the three files reported in the paper, the compression ratio is worse than that produced by gzip. This file has an uncompressed size of 60 KB, which is already much larger than the average for Java Class files. Consequently, we conclude that a Java wire format would rarely achieve better compression than gzip. The tailored approach that we have developed invariably achieves better compression than gzip.

DEVELOPING A TAILORED SOLUTION FOR JAVA CLASS FILES

Our review of the structure of the class file should have brought out the importance of the constant pool section. First, every class file will almost necessarily include many ‘standard’ character strings containing names and type signatures for methods in the Java class libraries. These strings will have a major influence on small class files, and the class files in package libraries do tend to be small. Second, the other sections of the class file all contain indexes into the constant pool. The net result is that the constant pool occupies 50–90 per cent of the entire file size when measured on a test collection of about 1000 class files. A chart that shows the contributions of the constant pool to the total file size is shown in Figure 4. The chart should be read as follows: the height of the column centred around 0·7, for example, represents the number of class files in our collection where the constant pool occupied between 65 per cent and 75 per cent of the whole file, i.e. 380 of our 1000 class files had constant pools that represented between 65 per cent and 75 per cent of the whole file.

Another important observation is that character string constants dominate the constant pool. These are strings represented in the UTF8 format. Each string is represented by a two-byte length immediately followed by the bytes that comprise the string constant. Together, the string constants account for about 40–80 per cent of the entire file size. Figure 5 graphically shows the size contribution fo UTF8 strings to the overall class file size. It should be abundantly clear that a good compression scheme for Java class files would have to perform well on UTF8 constant strings.

After the constant pool, we found the next most significant component of the class file to be the code attribute. Figure 6 shows the contribution of bytecode to the overall class file size in our experimental measurements.

Table I. SPARC code compression reported by Ernst *et al*.

Executable file	Original size	Gzipped size	Wire code size
lcc	315,636	75,928	64,475
gcc	1,381,304	380,451	287,260
agrep	61,036	15,936	16,013

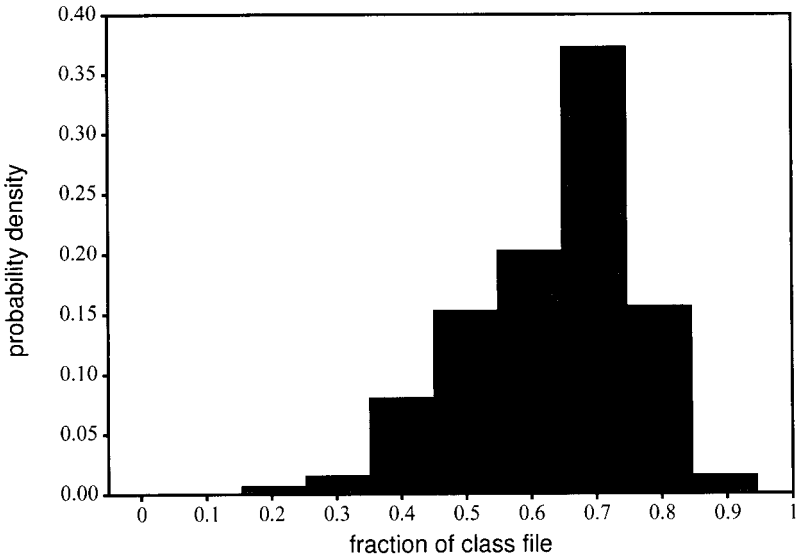

Figure 4. Distribution of constant pool contributions

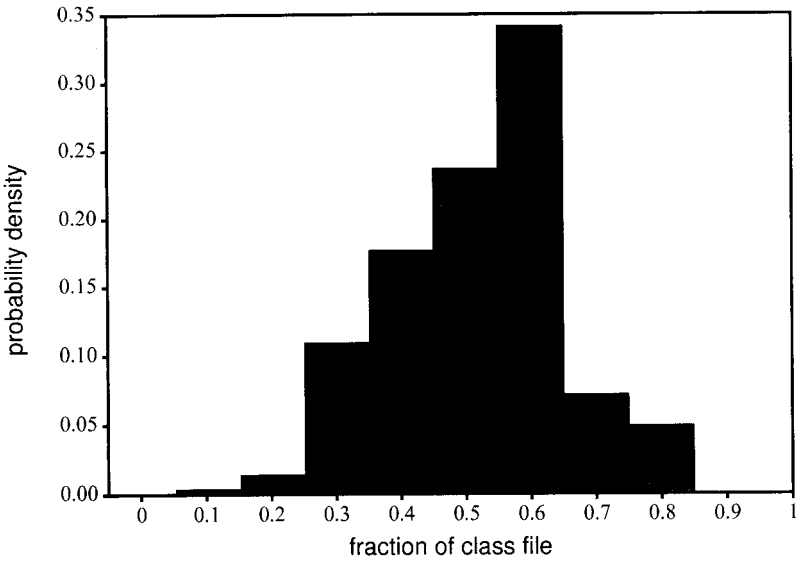

Figure 5. Distribution of UTF8 string constant contributions

The third most important component of the class file is the LineNumberTable Atribute. It is not necessarily present in a class file because it is needed only for debugging and error reporting. If the table is present, it holds one entry for every time the line number of the corresponding source code changes when making a sequential scan through the bytecode array. Its size should be roughly proportional to the bytecode array size.

The other components of the class file did not make significant contributions to

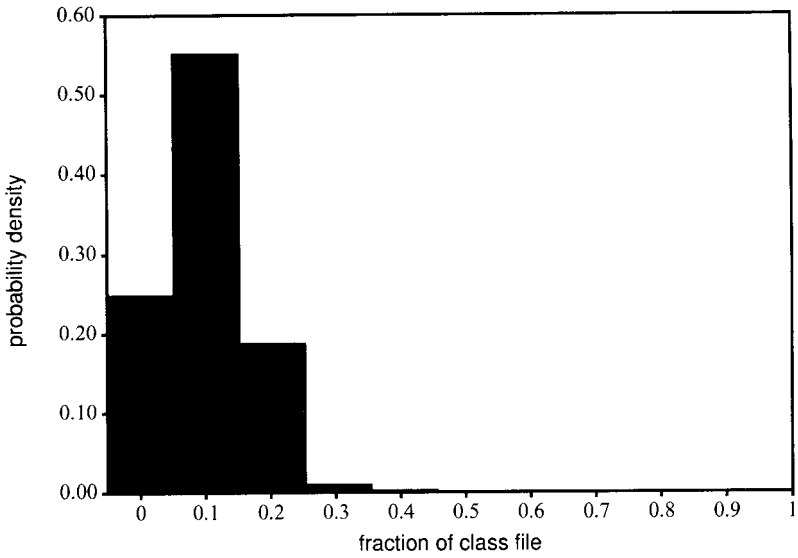

Figure 6. Contribution of Bytecode to class file size

overall file size in our experimental measurements. We therefore do not consider it necessary to provide tailored compression schemes for them.

A general compression program such as gzip is lossless in the sense that a decompressed file will be *identical* to the original file. A key observation is that a compression program for Java class files does not need to be perfectly identical to the original. It is good enough if the decompressed file *executes* in the same way as the original. That is, we only need to preserve semantic equivalence and not textual equivalence between the two files. For example, one of our biggest improvements to space efficiency comes from reordering the constant pool. As long as all indexes into the constant pool are adjusted to reflect the new order, the bytecode should execute in exactly the same way as before.

We note that re-ordering the constant pool may cause the bytecode component of the class file to *increase* in size. This is because the LDC (load constant) instruction has an operand which is a one byte index into the constant pool. If re-ordering should cause a constant used as an operand of LDC to move out of the first 256 positions in the constant pool, then the wide form of the instruction, LDC_W, must be used instead and that occupies more memory. However, the risk is worth taking. In our experiments, the effect was observed only rarely and caused only tiny increases in the size of the decompressed file.

We now explain our transformations on the significant parts of the class file.

Constant Pool Entries

Our initial transformation is to reorder the entries of the constant pool so that all entries of the same type are grouped together and so that UTF8 strings are sorted by their lengths. The result of this reordering on the example constant pool of Figure 3 is shown in Figure 7.

There are three important benefits from the reordering. First, we no longer need

```
 1:  NameAndType [29, 20]
 2:  NameAndType [19, 42]
 3:  NameAndType [24, 41]
 4:  NameAndType [30, 21]
 5:  NameAndType [28, 25]
 6:  Methodref [16, 1]
 7:  Methodref [15, 1]
 8:  Methodref [14, 4]
 9:  Methodref [16, 5]
10:  Methodref [14, 3]
11:  Fieldref [13, 2]
12:  String [38]
13:  Class [39]
14:  Class [40]
15:  Class [37]
16:  Class [26]
17:  Integer [8]
18:  Utf8 [1, "I"]
19:  Utf8 [3, "out"]
20:  Utf8 [3, "()V"]
21:  Utf8 [4, "(I)V"]
22:  Utf8 [4, "Code"]
23:  Utf8 [4, "main"]
24:  Utf8 [5, "print"]
25:  Utf8 [5, "(II)I"]
26:  Utf8 [5, "Small"]
27:  Utf8 [6, "CUTOFF"]
28:  Utf8 [6, "foobar"]
29:  Utf8 [6, "<init>"]
30:  Utf8 [7, "println"]
31:  Utf8 [10, "Small.java"]
32:  Utf8 [10, "Exceptions"]
33:  Utf8 [10, "SourceFile"]
34:  Utf8 [13, "ConstantValue"]
35:  Utf8 [14, "LocalVariables"]
36:  Utf8 [15, "LineNumberTable"]
37:  Utf8 [16, "java/lang/Object"]
38:  Utf8 [16, "Return value is:"]
39:  Utf8 [16, "java/lang/System"]
40:  Utf8 [19, "java/io/PrintStream"]
41:  Utf8 [21, "(Ljava/lang/String;)V"]
42:  Utf8 [21, "Ljava/io/PrintStream;"]
43:  Utf8 [22, "([Ljava/lang/String;)V"]
```

Figure 7. Reordered constant pool entries

to associate a tag byte with each entry when outputting the constant pool in its compressed form. The decoding program needs to know only how many entries there are of each type in order to reconstruct a constant pool that contains the proper tag bytes. Thus, the compression program outputs a simple count of how many entries there are of each type before outputting the entries of that type. We used a start-step-stop code with parameters (1, 3, 16) to encode the count. (Start-step-stop codes are variable-length codes where small integers are encoded in a few bits while larger integers require more bits for their encoding. The scheme is explained in more detail below.) Since there are so few counts to encode in each class file, the choice of the particular encoding scheme is not critical.

The second benefit comes from encoding constant pool entries that contain references to other constant pool entries. For example, an entry of type Fieldref is normally coded as a tag byte followed by two 16-bit indexes. The first index always references a constant pool entry of type Class and the second always references an entry of type NameAndType. Since our reordered constant pool has grouped all the Class and NameAndType entries together, we can replace the first index with a number that represents the position within the group of Class entries, and similarly for the second index. These relative indexes will almost always have much smaller values than the original index numbers and we can therefore encode them using fewer bits. We encoded each index into the group of Class entries using a fixed-length binary code with $\lceil \log N_{\text{Class}} \rceil$ bits, where N_{class} is the number of entries of type Class in the constant pool.

The third benefit is that reordering the UTF8 strings into order of increasing length means that we can encode the string lengths in a more efficient manner. If we look at an arbitrary string constant, other than the first, in Figure 7, we can see that its length is almost always identical to the length of the preceding entry or is

only very slightly longer. In other words, if we encode the length of a string as the difference between its length and that of the preceding string, we will be encoding much smaller numbers. Encoding differences rather than the values themselves is known as *delta coding*. Since a typical class file contains relatively many string constants, it is worthwhile to devise a scheme for encoding the deltas (differences) as efficiently as possible. To that end, we determined the distribution of string length deltas for our collection of class files. We then determined which start-step-stop code matched that distribution best. The result is shown in Figure 8. The solid line shows the distribution of delta values; the dashed line shows what the distribution should be to perfectly match the (0, 1, 16) start-step-stop code that we picked as being the closest match.

After extracting and encoding the length prefixes, the group of string constants becomes a block of text that contains a substantial amount of repetition. (Observe, for example, the repetition of the substring 'java' in Figure 7.) This block of text is well-suited for standard text compression algorithms. For convenience, we used the ZLIB library functions[11,12] to compress the text. ZLIB implements the same compression algorithm as used in gzip, a method that is very similar to the *deflate* compression method supported in the zip file format.[9] The maximum compression option for ZLIB/deflate was used here, as in all our uses of this compression method.

Other kinds of entries in the constant pool, such as integer or floating-point constants, occurred so infrequently in our sample files that there was very little benefit from devising special coding schemes for them. We therefore left their representations unchanged.

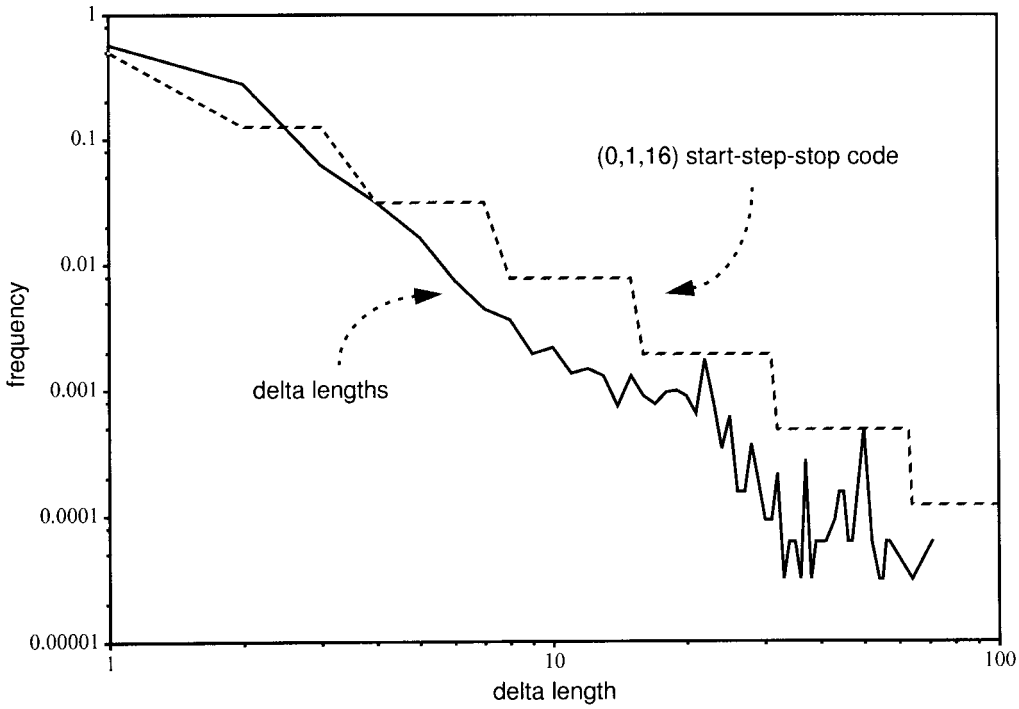

Figure 8. Distribution of delta string lengths

Code Attribute

The bytecode part of the class file contains the patterns of JVM instructions generated by the Java compiler for the constructs in the source program. Unless the compiler is sophisticated and optimizes these patterns extensively, there will necessarily be repetitions of the coding patterns. However, we were unable to find a fast and effective way to exploit these patterns. It would be easier and preferable for the compiler to generate such patterns in a compact form directly, rather than having to rediscover the patterns by analyzing the bytecode. Such is the approach of the *Slim Binaries* format of Kistler and Franz.[13]

In keeping with our desire to preserve the existing bytecode format, we chose to perform only two simple transformations on the bytecode and then apply the ZLIB compression algorithm[11,12] to it. The first transformation was to separate the opcodes and the operands into two separate arrays. By separating out the opcodes, any repeated patterns of opcodes will become apparent and amenable to compression by a general-purpose method.

The second transformation concerned operands of branching instructions. The operands of branching instructions are the addresses of other instructions in the bytecode array for the method being executed. They are normally implemented as 2-byte offsets. For example, if index positions 103–105 of the array hold the ifnonnull branching instruction, and its target is an instruction at index position 124, then bytes 103 and 104 will hold the value 21 (computed as 124–103). Such a representation is redundant because not every position in the bytecode array represents the start of an instruction—many JVM instructions occupy two or more bytes. We eliminated the redundancy by replacing byte-offsets with instruction-offsets in our compressed file format.

Following the re-encoding of all instruction-relative offsets, we separately compress the two arrays created from the bytecode using the ZLIB routines.

LineNumberTable Attribute

Each entry in the LineNumberTable contains a code array index and a corresponding source statement number. The indexes can only refer to the starts of JVM instructions. Therefore, space can be saved by converting these indexes into instruction numbers. Further compression is achieved by using delta coding. Both the instruction numbers and the statement numbers form slowly increasing sequences in our collection of sample class files. Presumably a sophisticated Java compiler could re-order code and thus break the property that statement numbers only increase through the code array; however, statements would be likely to be moved in groups and delta coding would still achieve good results. We used (2, 2, 16) start-step-stop codes for both the instruction number differences and statement number differences.

Start-Step-Stop Codes

We make extensive use of start-step-stop codes[6] to encode various kinds of integers in our compressed class files. Such codes have the general property that small integers receive shorter codes than large integers. The codes are generated in a systematic manner that permits rapid conversions between an integer and its encoded representation.

The codes have three parameters which control the range of integers that can be represented and the rate at which the bit string encoding grows in length. The encoding would be optimal if the range exactly matches the range of numbers that we need to represent and if the number of bits used to encode an integer k is logarithmically related to the frequency with which k needs to be encoded. That is, if len(k) is the number of bits used to encode k, and if Freq_k is the frequency of occurrences of k, then we would desire that

$$\text{len}(k) = -\log(\text{Freq}_k)$$

should hold. In practice, we can only choose a start-step-stop code that approximately matches the frequency distribution. Huffman coding[6] will usually produce better compression, but the compression and decompression algorithms are more complicated and require that a coding table be provided.

The underlying number representation used by a start-step-stop code is the usual binary. However, the encoder and decoder must agree on how many bits comprise the binary number. Rather than using a fixed, predetermined, number of bits, the start-step-stop code prefixes the binary number with a code that specifies the number of bits in the binary part. This prefix code is implemented as a unary number; unary being a scheme that can be decoded without knowing the number of bits in advance. For example, the unary code for the integer 5 is 111110, constructed as five 1-bits and terminated by a 0-bit. If the unary number is m, then the number of bits in the immediately following binary part of the code is $a + b \times m$ where a is the start parameter and b is the stop parameter. The stop parameter c is the maximum value that the unary prefix is allowed to encode. Knowledge of this value is used to optimize the way in which the unary code is written (its final 0 bit can be safely dropped). As an example, the table of start-step-stop codes for (1, 2, 5) coding is shown in Table II. To make the codes easier to interpret, the prefix part of each code is underlined.

EXPERIMENTAL RESULTS

A C implementation of our tailored compression approach was programmed. We named this program *clazz*. Compression results for some representative class files

Table II. (1,2,5) Start-Step-Stop codes

Integer	Code	Integer	Code
0	0 0	8	10 110
1	0 1	9	10 111
2	10 000	10	11 00000
3	10 001	11	11 00001
4	10 010	12	11 00010
5	10 011	…	…
6	10 100	40	11 11110
7	10 101	41	11 11111

Table III. Compression results for representative class files

File	Original size	ZLIB deflated size	bzip2 size	clazz size
AudioClip.class	233	184	225	95
Component.class	24,622	11,154	11,269	10,050
Enumeration.class	261	203	254	107
HashTableEntry.class	630	404	458	229
Integer.class	3,733	1,919	2,113	1,610
Object.class	1,452	787	923	576

are shown in Table III. In this table, we compare the compression of our clazz program against two general-purpose text compression programs—the *deflate* method of the ZLIB library (with maximum compression selected as an option) and bzip2. The ZLIB program is relatively fast and could therefore be considered as a good candidate for compressing Java class files. In this table, we show the results for some of the largest files as well as for some of the smallest files. We can observe that ZLIB and bzip2 perform better on large class files but quite poorly on small files, where they have little opportunity to adapt to the file characteristics. Our clazz program, on the other hand, achieves significant compression for all file sizes and always outperforms both competitors. Its better performance with small file sizes is marked.

Compression results for two collections of class files are shown in Table IV. Both class file collections were taken from the Metrowerks Codewarrior distribution. The first 50 classfile members of the Swing/Rose library and the first 128 members of the standard Java class library were used. The files were compressed separately. Again, clazz outperformed the ZLIB deflate method and bzip2 by a significant margin. Expressed as compression ratios, clazz is achieving a reduction to 35–38 per cent of the original size, versus 46–51 per cent for ZLIB and 51–56 per cent for bzip2.

Since the clazz program applies a variety of compression methods to different components of the class file, it is interesting to observe how well each component is compressed. We observed the following:

- Tag bytes attached to entries in the constant pool accounted for 2–6 per cent of the size of our sample class files. Our reordering of the entries and replacing the tags with counts, using start-step-stop codes, reduced the contribution of tags to insignificance.

Table IV. Compression results for collections of class files

Library	Average Sizes (in bytes)			
	Original file	ZLIB/deflate	bzip2	clazz
50 members of Rose class library	4047·2	1881·8	2063·6	1431·5
128 members of MW class library	2405·5	1221·9	1354·7	920·6

- The length fields of UTF8 strings were reduced from 2 bytes to an average of 2·7 bits, i.e. to 17 per cent of their original size.
- The entire constant pool was reduced, on average, to 31 per cent of its original size, even though we made no attempt to compress entries for integer constants or floating-point constants.
- A simpler method to compress the constant pool would be to reorder the entries and remove the superfluous tag bytes, as explained above, and then apply the ZLIB compression routine. This achieves somewhat worse compression than that produced by our more complicated approach. For example, the file Integer.class which is compressed to 1610 bytes with our method would be compressed to 1761 bytes instead. We consider this difference to be worth the price of the more complicated method.
- Our attempts to compress the bytecode arrays were successful only for larger class files. In many cases, methods contained fewer than 20 bytes of bytecode. On average, each method had its bytecode reduced to 59 per cent of its original size. The best compression, observed for those methods with the most bytecode, reduced the bytecode to 26 per cent of its original size.
- The LineNumberTable attribute, when present in the class file, was compressed, on average, to 33 per cent of its original size. (Production code would not normally contain this attribute.)
- Reordering the constant pool and making corresponding changes throughout other sections of the class file indeed has no effect when executed by the JVM. Spot checks with several files yielded no discernible difference in behaviour at execution time.

Execution times for compressing and decompressing representative class files are shown in Table V. All times are measured in seconds and were obtained with a 120 MHz Intel Pentium CPU. Compression times are quite competitive with the

Table V. Execution times for compression and decompression

File	Size (bytes)	Execution times (in seconds)		
		ZLIB/deflate	bzip2	clazz
AudioClip.class	233	0·049 0·019	0·577 0·385	0·049 0·025
Component.class	24,622	0·368 0·088	1·340 0·577	0·338 0·370
Enumeration.class	261	0·052 0·024	0·538 0·373	0·052 0·025
HashTableEntry.class	630	0·053 0·026	0·563 0·376	0·058 0·030
Integer.class	3,733	0·105 0·032	0·747 0·417	0·103 0·070
Object.class	1,452	0·057 0·026	0·598 0·381	0·067 0·037

ZLIB library, while decompression times are only a little worse. Our timings could undoubtedly be further improved with a more careful implementation. We observe too that the decompression time could be greatly reduced by integrating decompression with the Java class loader. One reason is that our compressed format has eliminated the need for one step of the bytecode verification process that is performed before the bytecode is executed. The verifier must check that every branch address, every entry point and every exception handler begins at the start of a bytecode instruction. Our compressed file format guarantees that this property must hold. A second reason is that the decompression program re-constructs the class file as an organized collection of data structures in memory as an intermediate step. This is work that the class loader would also perform.

CONCLUSIONS

The class file compression strategy, implemented as the clazz program, achieves much better compression than general-purpose compression programs while retaining full compatibility with the JVM architecture. A key insight is that the reconstructed file does not need to be identical to the original—it need be only semantically equivalent. Our implementation is not as fast as the competing compression programs, but that issue could be alleviated or eliminated if we were to re-implement the program more carefully and if we could combine the decompression code with the Java class loader.

A longer term and more drastic way of achieving greater compression would involve a complete re-design of the class file structure of the JVM instruction set. The slim binaries proposal,[13] for example, provides a very compact alternative format for bytecode along with the constants used in that code. Yet another possibility would be to design a new JAR file format where members of the archive share a common string constants table. Class files belonging to the same package typically duplicate many string constants, representing member names and method signatures.

ACKNOWLEDGEMENTS

Financial support from Natural Sciences and Engineering Research Council of Canada, in the form of a scholarship for the second author and a research grant for the first author, is gratefully acknowledged. Comments provided by the reviewers were invaluable in improving the experimental results.

REFERENCES

1. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1997.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
3. M. Nelson and J.-L. Gailly, *The Data Compression Book, 2nd Edition*. M & T Books, 1995.
4. J. Ernst, W. Evans, C. W. Fraser, S. Lucco and T. A. Proebsting, 'Code compression', *Proceedings of PLDI'97, ACM Conference on Programming Languages, Design and Implementation*, 1997, pp. 358–365.
5. T. L. Yu, 'Data compression for PC software distribution', *Software—Practice and Experience*, **26**(11), 1181–1195 (1996).
6. T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*, Prentice-Hall, 1990.
7. J. Seward, 'The Bzip2 home page', URL: http://www.muraroa.demon.co.uk and mirrored at http://www.digistar.com/bzip2 in North America (1998).
8. J. Corless, 'Compression of Java Class Files', *MSc Thesis*, Department of Computer Science, University of Victoria, 1997.
9. Info-ZIP 'General format of a ZIP file', Info-ZIP note 970311, URL: http://www.cdrom.com/pub/infozip/doc/ (1997).

10. C. W. Fraser and T. A. Proebsting, 'Custom instruction sets for code compression', URL: http://www.cs.arizona.edu/people/todd/papers/pldi2.ps (1995).
11. L. P. Deutsch and J.-L. Gailly, 'ZLIB compressed data format specification, version 3.3', URL: http://quest.jpl.nasa.gov/zlib/rfc-zlib.html (1996).
12. The Zlib home page, URL: http://www.cdrom.com/pub/infozip/zlib/, 1998.
13. T. Kistler and M. Franz, 'A tree-based alternative to Java byte-codes', *Technical Report 96-58*, Department of Information and Computing Science, University of California at Irvine, 1996.