Compression of Java Class Files

by

Jason David Corless
B Sc , University of Victoria, 1994

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr R N Horspool, Supervisor (Dept of Computer Science)

[redacted]

Dr M Levy (Dept of Computer Science)

[redacted]

Dr H Kwok (Dept of Electrical and Computer Engineering)

[redacted]

Dr P Agathoklis (Dept of Electrical and Computer Engineering)

Supervisor Dr R N Horspool

Abstract

The Java environment differs from traditional environments because application code can be dynamically loaded over a network The application code, contained in Java class files, is loaded as it is required If the application is large, there can be a significant waiting period for the user One solution to this problem is to compress the code so that it can be transmitted more quickly Existing general-purpose data compressors tend to perform poorly on Java class files We develop a specialized data compression algorithm for Java class files that, on average, outperforms general-purpose compressors by 28% Our algorithm's CPU and memory requirements are comparable to existing general-purpose compressors

Examiners

[redacted]

Dr R N Horspool, Supervisor (Dept of Computer Science)

[redacted]

Dr M Levy, Departmental Member (Dept of Computer Science)

[redacted]

Dr H Kwok, Outside Member (Dept of Electrical and Computer Engineering)

[redacted]

Dr P Agathoklis, External Member (Dept of Electrical and Computer Engineering)

Table of Contents

List of Tables

List of Figures

Acknowledgments

This research could not have been completed without the advice and guidance of my supervisor, Dr. Nigel Horspool. He was always available to answer my questions, point me to a relevant paper, and generally keep me motivated.

I would also like to thank my parents, who, in their wisdom, decided that their twelve-year old son should have a Commodore 64 for Christmas. If they hadn't realized the importance of that computer, I might not have ever started this degree.

Mike Zastre and Jochen Stier deserve thanks for their help at various stages of this research.

Finally, I would like to thank Cathy. She was a constant source of understanding and encouragement.

To my parents

1 Introduction

Imagine not having to drive to a computer store to purchase your software products. Instead, you can simply sit down at your computer, pay a small fee, and use the latest version of your favorite program. No more installation hassles, no need to purchase upgrades every year, no need to apply bug-fixes, and no need to purchase a separate copy of each software package for both your Mac and your PC. In theory, this is what Sun's Java language is going to do to the software industry.

Unfortunately, in practice, there are some problems with the scene described above. It can take upwards of 15 minutes to download, and start executing even a moderately sized Java program (or Applet). By extrapolating, one could expect that a full-blown word processing application written in Java could take upwards of an hour to download. Clearly no user is going to wait an hour each time they wish to write a letter!

Once a Java application has been downloaded, it can be executed at almost the same speed as natively compiled applications. The bottleneck is transferring the Java code from the network server to the user's local machine. With the ever increasing popularity of the internet, it seems unlikely that hardware increases in bandwidth will keep up with the demand.

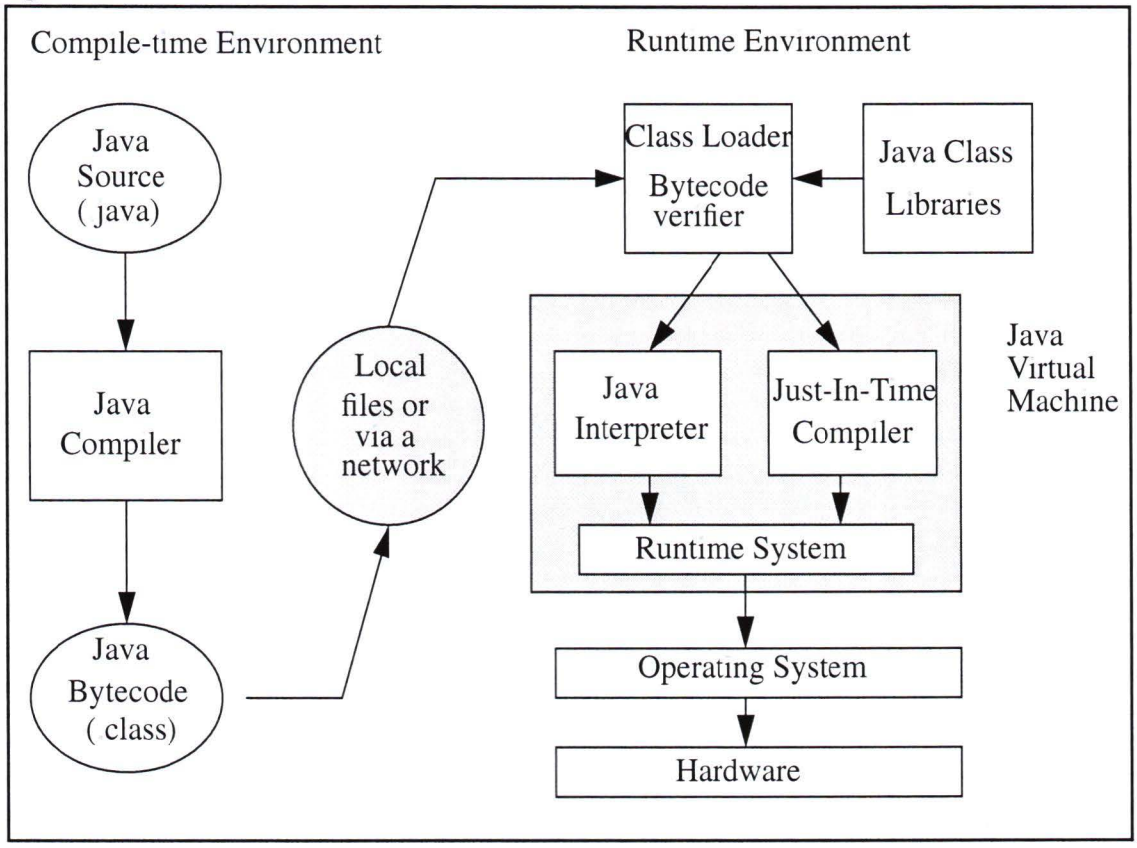
This research focuses on one of the fundamental problems. We propose a compression algorithm that makes the Java code smaller, thereby reducing the transfer time over the network. We have developed an algorithm that, on average, outperforms existing general-purpose data compression algorithms by 28%.

1.1 The Java Environment

The Java environment consists of two parts, the programming language Java, and the runtime environment required to execute compiled Java code. Figure 1.1 shows a graphical representation of the Java environment.

Java differs from traditional programming languages like C and C++ because it is not compiled to native code (the machine-code for the platform it is running on). Instead, Java is compiled into a platform independent representation, called Java bytecode, which is executed on each host platform by a Java Virtual Machine.

Figure 1.1 The Java Environment



Each class defined in a Java program is compiled into its own "object" file, called a class file. The class file contains all the information required by the Java Virtual Machine to execute the methods defined in the class.

The Java Virtual Machine was originally released as an interpreter for a simple stack-based machine. However, in recent months, several Just-In-Time (JIT) compiler based

Java Virtual Machines have been released. A JIT compiler translates the Java bytecode into native code as the code is being executed. These implementations provide anywhere from a two to twenty-fold increase in execution efficiency.

There are several good introductions to the Java Environment, the interested reader is referred to [10][13][17].

1.2 Goals of this Research

The main goal of this research is to develop a data compression algorithm to compress Java class files with the following properties.

- The algorithm need not compress anything other than Java class files, it could be a specialized algorithm created specifically for class files.
- The algorithm should consistently outperform existing general-purpose data compressors. At present, there are no other algorithms published that are optimized for class files.
- The algorithm should compress well on both large and small class files. Traditionally, general purpose compressors tend to perform poorly on small files.
- Compression is permitted to be slower than decompression, since it will typically be performed only once, while decompression will be performed often.
- Decompression speeds must be comparable to existing general purpose data compressors.

Compression performance is the most important. Without good compression performance, it is difficult to justify the overhead of using a specialized algorithm over an existing general-purpose algorithm. The algorithm developed in this research meets all of the requirements defined above.

1.3 Thesis Overview

In Chapter 2 we summarize the format of Java class files. We detail the areas of the class file that are likely to be compressible. In Chapter 3 we give an overview of several popular data compression methods, as well as background information required to

understand the data compression techniques applied in later chapters. Chapter 4 describes, in detail, the compression algorithm developed in this research. We also discuss how much each area of the class file contributes to the overall size of class files, and justify the compression decisions we made. Chapter 5 reports the results of our research, detailing the compression performance of our algorithm. Chapter 6 summarizes the effectiveness of our algorithm, and outlines possible future work.

2 Java class File Format

In this chapter we discuss the format of the class file so the reader can understand the compression techniques applied in later chapters. We discuss the areas of the class file that will be important to compression. Additionally, we give an overview of Java bytecode, the instructions executed by the Java Virtual Machine.

2.1 Java File Structure

When a Java program is compiled, each class (or interface) defined by the programmer is compiled into a file which is the class's name with the extension ".class". If the program in Figure 2.2 were to be compiled, the results would reside in the file "Small.class".

The overall structure of the class file is given in Figure 2.1. In the following discussion, the notations *u1*, *u2*, and *u4* represent unsigned integers of length one, two, and four bytes respectively. A detailed description of the entire class file can be found in [17].

Figure 2.1 Class file structure

```
ClassFile {
    u2             magic
    u2             minor_version
    u2             major_version
    u2             cp_count
    cp_info        constant_pool[cp_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             method_count
    method_info    methods[method_count]
    u2             attributes_count
    attribute_info attributes[attributes_count]
}
```

The class file is roughly divided into four parts: the constant pool, fields definitions, method definitions, and attributes. We will discuss each of these areas in the following sections. Examples will be based on the class file produced from compiling the Java program in shown Figure 2.2.

Figure 2.2 Example program Small.java

```
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) );
  }
}
```

2.2 The Constant Pool

The constant pool is a table of constants required by the Java Virtual Machine during the execution of the bytecode for the given class. It includes not only numeric and string constants defined explicitly by the programmer, but also many constants generated by the Java compiler. Figure 2.3 is the constant pool generated by compiling the example program in Figure 2.2.

Figure 2.3 Constant Pool for Small java

```
 1  String [  39]
 2  Integer [0x00 0x00 0x00 0x08 ]
 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  Field [   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"
```

It is important to note that even a small program has a constant pool of significant size In the example, only two constants (one integer constant, and one string constant) were

defined by the programmer These constants are represented by entries 2 and 39 in the constant pool The other 41 entries in the constant pool were generated by the compiler!

2.2.1 ConstantPool Entries

Entries in the constant pool are variable length structures prefixed by a tag byte which designates the type of the entry Table 2 1 shows the valid constant pool entry types, and what type of information they contain Each entry contains either raw data (such as string or numeric constants) or references to other constant pool entries

Table 2.1 Constant Pool Entry Types

Entry type	Type of Information
Utf8	data
Unicode	data
Integer	data
Float	data
Long	data
Double	data
Class	reference
String	reference
Fieldref	reference
Methodref	reference
InterfaceMethodref	reference
NameAndType	reference

Consider entry 16 from Figure 2 3, a NameAndType entry This entry contains two integer parameters 18 and 19, which are references to the Utf8 entries "println" and "(I)V" Entry 16 defines the method "println", which takes a single integer argument and has a `void` return type

By way of comparison, entry 2 reflects the declaration of the integer constant 8. For more information on method signatures and the constant pool see [17].

2.2.2 Utf8 Entries

Utf8 entries are similar to Pascal strings. They contain an explicit 2-byte length indicator as well as the actual string data. All string constants required in a class are stored as Utf8 entries. Figure 2.4 shows the structure of a Utf8 entry.

Figure 2.4 Utf8 Entry

```
Utf8 {
  u1 tag
  u2 length
  u1 data[length]
}
```

There is an abundance of Utf8 entries in the constant pool because the Java compiler doesn't compile references to variables, functions, and structure members down to numeric offsets in the same way as traditional compilers [1]. Instead all references to fields and methods are symbolic. The compiler generates a Utf8 entry for each class, method, and field referenced in the source program.

In addition, field and method references generate a Utf8 entry for their "signatures". A method signature defines the type and number of arguments for the method, and a field signature defines the type of the field.

Symbolic references enable Java to avoid the "Fragile Superclass problem", where adding a field or method to a class can require complete recompilation of all classes that reference that class [13].

2.3 Fields and Methods

The structures for fields and methods are very similar. Figure 2.5 shows the structure of a method.

Figure 2.5 Structure of a method definition

```
Method_info {
  u2 access_flags
  u2 name_index
  u2 signature_index
  u2 attribute_count
  attribute_info attributes[attribute_count]
}
```

The method definition contains only very basic information about a method, including its access flags, type, and signature. Much of the information about a method is contained in its attributes. Attributes are discussed in the next section.

2.4 Attributes

Excluding the constant pool, most of the information contained in a class file resides in what Sun has called *Attributes*. The general structure of an attribute is shown in Figure 2.6. The `attribute_name` field is a reference to a Utf8 entry which contains a textual representation of the attribute's name.

At present, Sun has defined six attributes. They are *SourceFile*, *ConstantValue*, *Code*, *Exceptions*, *LineNumberTable*, and *LocalVariableTable*. In Figure 2.3 we see that entries 21, 24, 25, 27, and 29 correspond to attribute names. The only attribute not present in the example program is *LocalVariableTable*.

Figure 2.6 A Generic Attribute

```
GenericAttribute {
  u2 attribute_name
  u4 attribute_length
  u1 info[attribute_length]
}
```

Discussion of all defined attributes is beyond the scope of this paper. We discuss the LineNumberTable attribute in Section 2.4.1, the LocalVariable attribute in Section 2.4.2, and the Code attribute in Section 2.5. Complete discussions of all defined attributes can be found in [17].

2.4.1 LineNumberTable

This attribute is used by debuggers and exception handlers to determine which source code lines correspond to which bytecode instructions. This attribute is associated with the *Code* attribute discussed in Section 2.5. The *LineNumberTable* attribute is not always present in a class file. Specifically, Sun's `javac` compiler does not include it when the `-O` flag is used (optimization), but `guavac [8]` always includes the *LineNumberTable* attribute. The structure of the attribute is shown in Figure 2.7.

Figure 2.7 The LineNumberTable Attribute

```
LineNumberTable_attribute {
  u2 attribute_name
  u4 attribute_length
  u2 line_number_table_length
  {
    u2 start_pc
    u2 line_number
  } line_number_table[line_number_table_length]
}
```

The entries in the *LineNumberTable* are not necessarily one-to-one with source line numbers. That is, there may be more than one entry per physical source line [17].

It is interesting to note that there is explicit redundancy in this attribute. The fields `attribute_length` and `line_number_table_length` both provide the same information. That is:

```
attribute_length = 2 + 4*line_number_table_length
```

This type of explicit redundancy is present in many of the attributes and will be eliminated in our compressed format for the class file.

2.4.2 LocalVariableTable Attribute

The *LocalVariableTable* attribute is used by debuggers to determine the value of a given local variable during the execution of a method. This attribute is associated with the *Code* attribute, discussed in Section 2.5. Figure 2.8 shows the structure of the *LocalVariableTable* attribute.

Figure 2.8 The LocalVariableTable Attribute

```
LocalVariableTable_attribute {
  u2 attribute_name
  u4 attribute_length
  u2 local_variable_len
  {
    u2 start_pc
    u2 length
    u2 name_index
    u2 descriptor_index
    u2 index
  } local_variable_table[local_variable_table_len]
}
```

The table defines the regions of the bytecode where the local variable named by `name_index` contains a value. The value of the given local variable can be found at index `index` in the current method's local variables. There can be a *LocalVariableTable* attribute for each local variable defined in the method.

2.5 Code Attribute

The *Code* attribute contains information about a specific method, constructor, class constructor, or interface implemented by a class. The compiler determines the maximum stack size, the number of local variables, and the exception handlers for this method. This information is stored with the bytecode for the method in a *Code* attribute, whose structure is shown in Figure 2.9.

Figure 2.9 Code Attribute

```
Code_attribute {
  u2 attribute_name
  u4 attribute_length
  u2 max_stack
  u2 max_locals
  u4 code_length
  u1 code[code_length]
  u2 exception_table_length
  {
    u2 start_pc
    u2 end_pc
    u2 handler_pc
    u2 catch_type
  } exception_table[exception_table_length]
  u2 attribute_count
  attribute_info attributes[attribute_count]
}
```

There are over 200 opcodes used by the Java Virtual Machine, and a detailed description of each can be found in [17]. We will cover several "classes" of instructions here, so that the reader has enough understanding to appreciate the compression techniques applied in later chapters. Figure 2.10 shows the bytecode for one of the methods in the example program "Small.java."

The Java Virtual machine is a simple stack based machine. The simplicity of the underlying machine is reflected in the bytecode instructions.

Figure 2.10 Bytecode from Small.java

```
public int foobar (int, int )
   0  iload_1
   1  iload_2
   2  iadd
   3  istore_3
   4  iload_3
   5  bipush   0x08
   7  if_icmple      7 (  14)
  10  iconst_4
  11  iload_3
  12  imul
  13  ireturn
  14  iconst_3
  15  iload_3
  16  imul
  17  ireturn
```

2.5.1 Instructions that affect control flow

There are 27 bytecode instructions that can affect control flow. Table 2.2 shows several of the instructions and their purpose. All instructions that affect the flow of execution have program counter (pc) relative operands. If the branch is taken, the value of the operand is added to the current program counter, and execution proceeds from the instruction at the new program counter location. The operands are signed numbers, so branches can be both forwards and backwards.

Table 2.2 Control-flow Instructions

Instruction	Operands	Purpose
`ifeq`	16-bit offset	if the value on the top of the stack is zero, add *offset* to the current pc and continue execution from there
`ifge`	16 bit offset	if the value on the top of the stack is greater than or equal to zero, add *offset* to the current pc and continue execution from there
`if_icmpeq`	16-bit offset	compare the two integer values on the top of the stack. If they are equal add *offset* to the current pc and continue execution from there
`goto`	16-bit offset	always add *offset* to the current pc and continue execution from there

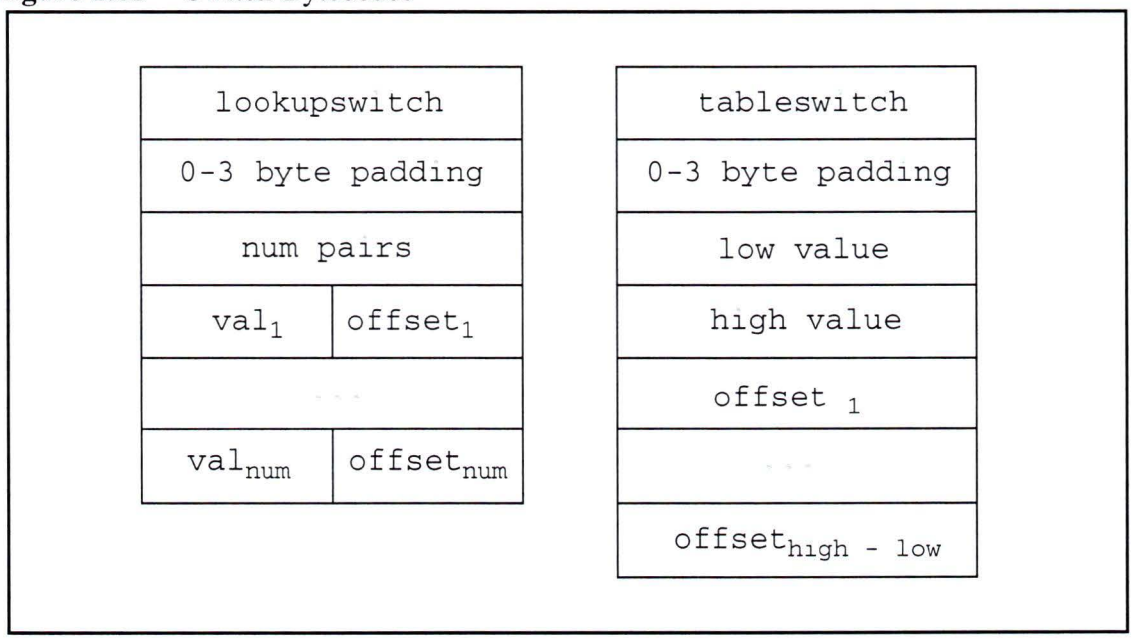
In Figure 2.10, the instruction `if_icmple` has an operand of 7 (the value in parentheses is computed by the disassembler, and shows the destination instruction). This means that if the comparison is true, 7 is added to the contents of the program counter and execution continues from that new address. In this case, execution would continue from location 14.

2.5.2 Switch Instructions

Other instructions that affect control flow are the opcodes `lookupswitch` and `tableswitch`, shown in Figure 2.11. Unlike other opcodes that affect control flow, the switch opcodes have 32-bit operands.

The switch opcodes are generated in response to the **switch** language feature, which is similar to that of C, and the Pascal **case** statement. These two opcodes are the only variable length opcodes in the Java Virtual Machine instruction set. The lengths of the instructions are determined by their operands.

Figure 2.11 Switch Bytecodes



The switch opcodes require their operands to start on a 4-byte boundary. There is, therefore, a zero to three byte padding field between the opcode and the operands to guarantee this alignment.

2.5.3 Instructions that access the Constant Pool

Several instructions obtain additional information from the constant pool. These include all the instructions for invoking methods, creating new objects and arrays, and instructions for transferring constants from the constant pool to the Java Virtual Machine stack. These instructions have 16-bit references to the constant pool as operands.

2.5.4 Other Instructions

There are many other classes of instructions that are not discussed here. In particular there are instructions for pushing constants onto the stack, transferring data between the stack and local variables, accessing arrays, manipulating the contents of the stack, arithmetic and logical instructions, conversion operations, and field manipulation instructions.

A complete understanding of the various instructions is not required to understand the remainder of this dissertation. The interested reader is referred to [17]

3 Existing Work

Java is a fairly recent language, with the first release of a development environment occurring only in 1995. There are no other published compression algorithms that work exclusively on Java class files. Java enabled Web-browsers from Netscape and Microsoft support compression of class files using existing general-purpose lossless compression techniques.

Perhaps the closest compressors in the literature to our compression algorithm are those that are designed to compress the source code for a specific programming language. These compressors parse a syntactically correct program, and perform compression using the abstract syntax tree generated from the source code. These type of compressors generally outperform general purpose compressors, but, in some cases, at the expense of losing information such as whitespace and comments [4][6].

In this chapter we will give an overview of several existing general-purpose data compressors. Additionally, we discuss variable length coding of integers, used in later chapters. There are several comprehensive references for the general subject of data compression, the interested reader is referred to [2][19][20].

3.1 Entropy

Entropy is a measure of how much information is contained in a message. The entropy of a message is related to the probability the message can occur. If the probability that a given message will occur is p_i the entropy, E_i, of that message is defined to be

$$E_i = -\log p_i \quad \text{bits}$$

A perfect representation of information would use as many bits as its entropy. The definition of entropy makes intuitive sense. In the average conversation, the message "I had a burger for lunch" has much less information content than the message "I was abducted by aliens last night" simply because the probability of the latter occurring is very small.

3.2 Arithmetic Coding

Using arithmetic coding, a message is represented by an interval of real numbers between 0 and 1. As each successive symbol of the message is transmitted, the interval becomes smaller. The less likely the symbol, the smaller the interval becomes, thus requiring more bits to distinguish it from other intervals. Arithmetic coding has been discussed at length in the literature [2], and there are several implementations available in source code form. We will not elaborate on it here because we haven't used it in our compression algorithm.

The most interesting property of arithmetic coding is that it has been shown to be optimal with respect to the entropy. That is, if the entropy of a message is n bits, arithmetic coding encodes that message using exactly n bits, where n is a real number. In practice, implementations of arithmetic coding tend to be just slightly less than optimal for efficiency reasons.

According to Bell, Cleary and Witten, the fact that arithmetic coding is optimal reduces the problem of data compression to finding an appropriate model for the input data. This model predicts, at each stage of compression, the probabilities of all possible input symbols, and arithmetic coding is used to encode the symbol that occurs[2, Section 6.0].

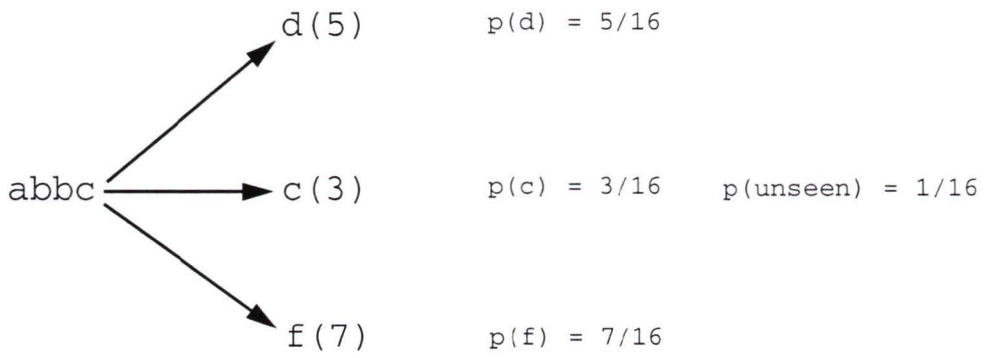
3.3 Modeling for Compression

Many models used for data compression maintain counts of how often each symbol in the input alphabet has followed the current *context*. A *context* is defined to be the symbols preceding the current symbol in the input stream. Some models have a maximum context length, but several compressors have been published recently which allow for an unbounded context length. That is, the model can predict using all previously seen symbols in the input stream [3][5]

Probabilities for the arithmetic coder are obtained from the counts maintained by the model. Figure 3.1 shows the model for the context "abbc". We can see that the model predicts the symbol "d" with probability 5/16, the symbol "c" with probability 3/16, the symbol "f" with probability 7/16, and all other symbols with probability 1/16.

For every context, the model must be able to predict all possible symbols in the input alphabet. If a symbol hasn't followed the current symbol, there will be a zero count associated with it, and hence the model would predict it with probability zero. This is called the "zero-frequency problem", and there are several methods of dealing with it. The solution we have illustrated is to reserve a count of one for all previously unseen symbols. When the algorithm encounters a symbol it cannot predict with the current model, it switches to a different model.

Figure 3.1 Modeling for Compression



This model shows all the symbols that have followed the context "abbc", and their corresponding probabilities.

Some of the best data compressors available today tend to use this style of modeling and arithmetic coding. The principal drawback to these compressors is their speed and memory requirements. Maintaining the statistics used to estimate the next input symbol tends to require a lot of memory and CPU time. Encoding values with arithmetic coding also tends to be slower than other techniques. However, with hardware speeds ever increasing, we are starting to see these types of compressors become more common.

We chose not to implement a compressor that uses arithmetic coding because we considered its decompression speed to be unacceptable for our application.

3.4 Dictionary Compressors

Dictionary compressors work by replacing sequences of characters (phrases) by an index into a dictionary. The dictionary contains phrases that are likely to occur frequently, and the indices are chosen so that, on average, the indices take less space than the phrases they represent.

Dictionary compressors tend to be outperformed by compressors using statistical modelling and arithmetic coding. Indeed, the current "best" general-purpose data

compression algorithm in the literature is PPM* [5]. However, dictionary based compressors are still commonly used in practice because they run faster and require less memory.

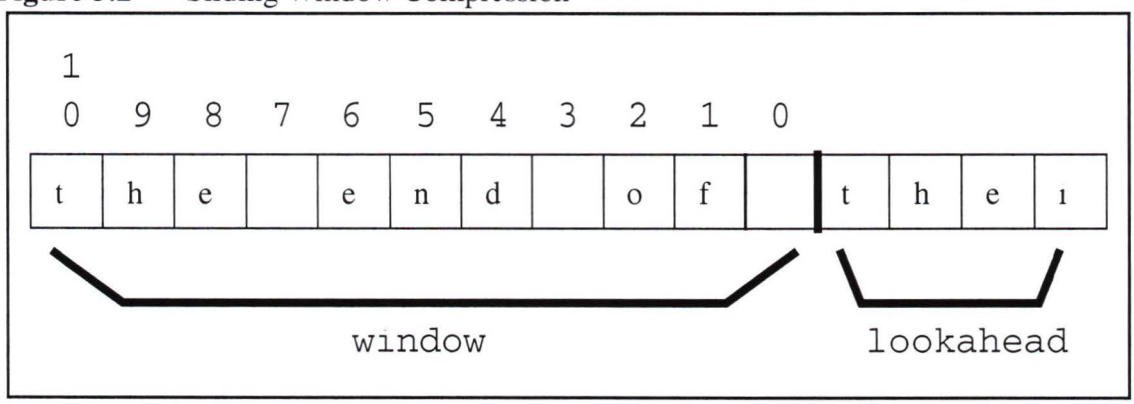
Almost all general-purpose dictionary compression techniques used today are derived from one of the two key papers by Jacob Ziv and Abraham Lempel, published in 1977 and 1978. Dictionary compressors have come to be known as either LZ77-, or LZ78-based. We discuss these styles of compressors in the following sections.

3.4.1 LZ77

LZ77 is commonly referred to as "sliding-window" data compression. The compressor has a fixed sized "window" that slides over the input file. Text that has been compressed is contained in the window, and uncompressed text is in the lookahead buffer.

The compressor searches the previously seen text for a match to the text in the lookahead buffer. If a match is found, the compressor outputs a representation of how far back in the buffer the match was found, and the length of the match found. Variations on the LZ77 algorithm differ in how they represent these matches, and how they deal with the case where no match is found. In most cases, if a match is not found, the first character in the lookahead buffer is output in its raw form. A comprehensive survey of LZ77 variants can be found in [2]. We will use the common LZ77 variant LZSS for our examples.

Figure 3.2 Sliding Window Compression



Consider compression at the stage in Figure 3.2. The current text in the lookahead buffer is "thei". The longest match in the window is "the", which occurs at offset 10. The LZSS algorithm would output this match as the length-offset pair <3,10>. If a match is not found, LZSS outputs the first character in raw format. The compressor uses a single bit to distinguish between length-offset pairs and raw characters.

Decompression is simple and extremely fast. The decompressor maintains the same buffer as the compressor. When it encounters a length-offset pair, it simply looks into the buffer and outputs the characters at that position. Raw bytes are output directly.

Several well known general purpose data compressors are LZ77-based, including `zip`, `gzip`, Stacker, and DoubleSpace.

3.4.2 LZ78

LZ78-based compressors maintain explicit dictionaries, against which phrases in the uncompressed text are matched. Instead of allowing pointers to reference any text previously seen, text seen so far is parsed into phrases and the compressor outputs dictionary indices corresponding to the longest phrase matched. Unlike LZ77, phrases must be explicitly added to the LZ78 dictionary.

The differences between LZ78 variants include how new phrases are added to the dictionary, and how dictionary indices are represented. LZ78-based compressors are generally more difficult to implement than LZ77 compressors, while compression performance tends to be similar. One of the first LZ78-based compressors, an LZW based implementation, was the UNIX program "compress".

LZW initializes the dictionary to contain every possible singleton in the source alphabet, ensuring that a match can always be made. After each match is made, a new phrase is added to the dictionary that is the concatenation of the last phrase matched and the first character of the current phrase matched. This method of adding new phrases to the dictionary is somewhat arbitrary, although it tends to perform well in practice. Any method of updating the dictionary can be used, as long as the decompressor can mimic the additions made by the compressor. Figure 3.3 shows an example of LZW compression.

Figure 3.3 LZW compression

Input String = "wed web wee"

Step	Match	Phrase Added
1	"w"	-
2	"e"	"we"
3	"d"	"ed"
4	"_"	"d_"
5	"we"	"_w"
6	"b"	"web"
7	"_w"	"b_"
8	"e"	"_we"
9	"e"	"ee"

LZ78-based compression algorithms are in common use in modems, graphics formats (GIF), and many other applications.

3.5 Coding Schemes for Integers

The most common coding scheme for integers is "fixed width" coding. This style of coding is most often dictated by the underlying hardware. The hardware has a word size w, which is typically 32 or 64 bits. Integers ranging from 0 to 2^w-1 are represented using w bits. This discussion applies equally to floating point numbers, which are commonly represented as a pair of signed integers.

Fixed width representations are used primarily because building the hardware to operate on this style of numbers is simple and cheap.

In many cases, a fixed width coding wastes space, since it implicitly assigns equal probability to each of the integers being represented. In practice, certain values are much

more likely to occur than others. Assigning those values shorter codes would reduce the space requirements. For general-purpose operations, other coding schemes have questionable worth, but for data compression, fixed width coding is often wasteful.

This section discusses two alternatives to fixed-width coding: Huffman coding and start-step-stop codes.

3.5.1 Huffman Coding

Given a set of symbols and the probability that each symbol will occur in a given message, we can construct Huffman codes for those symbols which will give a minimal coding over the message. Huffman codes are generated by constructing a Huffman tree. The tree is constructed by the following algorithm:

- All symbols are inserted into a priority queue, ranked by probabilities.
- We repeatedly extract the two symbols with the smallest probabilities from the queue. These two symbols become the children of a new node whose probability is the sum of its children's probabilities. We insert the new node into the priority queue.
- Huffman codes are constructed by walking the tree from the root to the leaf of the symbol. When traversing a left branch output a 0, when traversing a right branch output a 1.

Huffman coding can be performed by making two passes over the input, the first pass counts the number of occurrences of each symbol and then forms the Huffman codes from those probabilities. The second pass then encodes the input, using the codes constructed in the first pass.

The term "minimal-coding" is somewhat misleading, since Huffman codes are always an integral number of bits. Unless the probabilities of all the input symbols are a power of 2, one could obtain better coding using arithmetic coding, which, in effect, can use fractional bits.

Huffman codes are well discussed in the literature. There are numerous discussions of how to construct the codes and we will not reproduce them here. A discussion can be

found in [2], while source code and examples can be found in [19].

3.5.2 Start-Step-Stop Codes

A start-step-stop code is a sequence of fixed width codes for the integers, phased in according to the parameters (*start*, *step*, *stop*). The fixed width codes are prefixed by a unary code, which enables the decoder to determine which fixed width code is to follow.

The unary coding for the integer n is n ones followed by a zero (or, equivalently, n zeros followed by a one.) For example, the unary code for 7 is `11111110`.

In a start-step-stop code, the first fixed width code is *start* bits wide, and each successive code is *step* bits larger than the previous one. The last code is *stop* bits wide. The unary code prefixing the last fixed width code in the family can omit the final zero, since there are no larger codes.

By varying the parameters of the start-step-stop code, we can tune the implicit probability distribution of the code to match the data we are representing. Note that if the *stop* parameter is not infinite, there is a maximum integer than can be represented.

Table 3.4 shows the (1,2,5) code. For presentation purposes only, the fixed width codes are separated from the unary code by a colon. The first fixed width code is one bit wide, the second is three bits wide, and the final code is five bits wide.

Source code for generating start-step-stop codes can be found in [9]. The algorithm is both simple and efficient. We make extensive use of start-step-stop codes in our compression algorithm.

Table 3.4 Start-Step-Stop code (1,2,5)

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

4 Approach

A key insight that allows our algorithm to produce better compression is that the decompressed file does not have to be a byte-for-byte copy of the original class file. We only require that the same result is obtained when executing the code contained in the decompressed class file. This is a luxury that general-purpose data compressors do not have. We have to preserve the semantics of the class file, not the content.

In several respects we are able to make changes to the structure of the class file that helps to increase the compressibility of the file. Rather than treat the class file as a stream of bytes, our compressor parses the class file, rearranges the parsed data and then encodes it. The compressor works only on *correct* class files. That is, the behavior of the compressor is undefined if the contents of the class file do not match the specification found in [17][1].

The statistics shown in this chapter were gathered from approximately 1000 class files taken from various sources, including Sun's class libraries, the Java Generic Library (JGL) class library [15], and several graphical applications written by the author.

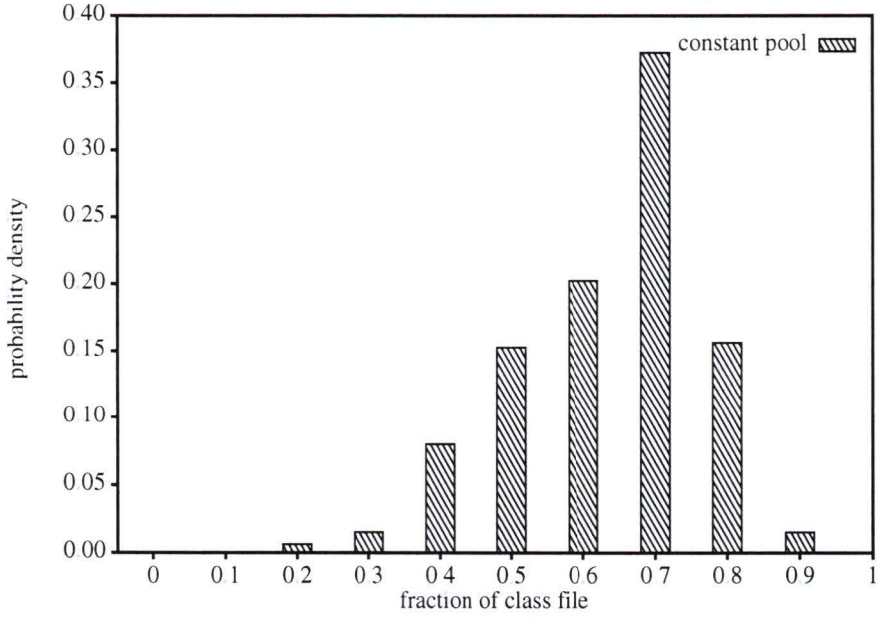
4.1 The Constant Pool

The constant pool is the largest contributor to the class file's size. We have therefore spent the greater part of our compression efforts on the constant pool.

---

1 Surprisingly there was one file in the Sun distribution which contained an invalid *LineNumberAttribute*, causing our compressor to fail.

Figure 4.1 Constant Pool contribution to overall file size



As Figure 4.1 shows, in the vast majority of class files, the constant pool makes up over 50% of the file. In some cases, it accounts for 90% of the class file.

4.1.1 Reordering

The crucial idea that allows many of our compression techniques to function is that the entries in the constant pool can be reordered. Entries are inserted into the constant pool as the Java compiler encounters various entities in the Java source code. We are free to reorder the constant pool entries, provided that we update the entire class file to reflect the new ordering.

We chose to reorder the constant pool entries so that entries of the same type are grouped together. Figure 4.2 shows the before and after ordering of an example constant pool.

Reordering the constant pool entries yields several immediate benefits, as discussed in the following sections. These benefits do not come without cost. Updating the constant pool references is conceptually simple, but the implementation can be difficult to get right. Also, reordering requires two passes over the constant pool.

Figure 4.2 Reordering the Constant Pool

Before

```
 1  Integer [0x08]
 2  "Small"
 3  Class [   2]
 4  "<init>"
 5  "()V"
 6  NameAndType [   4,   5]
 7  Method [   3,   6]
 8  "java/lang/System"
 9  Class [   8]
10  "out"
11  "Ljava/io/PrintStream;"
12  NameAndType [  10,  11]
13  Field [   9,  12]
14  "Return value is "
15  String [  14]
16  "java/io/PrintStream"
17  Class [  16]
18  "print"
19  "(Ljava/lang/String;)V"
20  NameAndType [  18,  19]
21  Method [  17,  20]
22  "foobar"
23  "(II)I"
24  NameAndType [  22,  23]
25  Method [   3,  24]
26  "println"
27  "(I)V"
28  NameAndType [  26,  27]
29  Method [  17,  28]
30  "java/lang/Object"
31  Class [  30]
32  Method [  31,   6]
33  "CUTOFF"
34  "I"
35  "ConstantValue"
36  "Code"
37  "LineNumberTable"
38  "main"
39  "([Ljava/lang/String;)V"
40  "SourceFile"
41  "Small.java"
```

After

```
 1  NameAndType [  27,  19]
 2  NameAndType [  20,  39]
 3  NameAndType [  25,  40]
 4  NameAndType [  28,  26]
 5  NameAndType [  30,  21]
 6  Method [  13,   1]
 7  Method [  15,   3]
 8  Method [  13,   4]
 9  Method [  15,   5]
10  Method [  16,   1]
11  Field [  14,   2]
12  String [  36]
13  Class [  24]
14  Class [  35]
15  Class [  38]
16  Class [  37]
17  Integer [0x08]
18  "I"
19  "()V"
20  "out"
21  "(I)V"
22  "Code"
23  "main"
24  "Small"
25  "print"
26  "(II)I"
27  "<init>"
28  "foobar"
29  "CUTOFF"
30  "println"
31  "SourceFile"
32  "Small.java"
33  "ConstantValue"
34  "LineNumberTable"
35  "java/lang/System"
36  "Return value is "
37  "java/lang/Object"
38  "java/io/PrintStream"
39  "Ljava/io/PrintStream;"
40  "(Ljava/lang/String;)V"
41  "([Ljava/lang/String;)V"
```

4.1.2 Elimination of Tag Bytes

As discussed in Section 2.2.1, each entry in the constant pool is prefixed by a single byte which denotes its type. Because we have reordered the entries so that like entries are grouped together, we can replace the tag bytes for a class of entries by a simple count.

The maximum number of entries in the constant pool is 65535. Obviously there must be fewer than 65535 entries of any given type. It would be sufficient to use a 16-bit count for each entry type. This gives us compression improvement when there are more than two entries of each type. However, in many cases there are no entries of a given type, which would cause an expansion of 2 bytes.

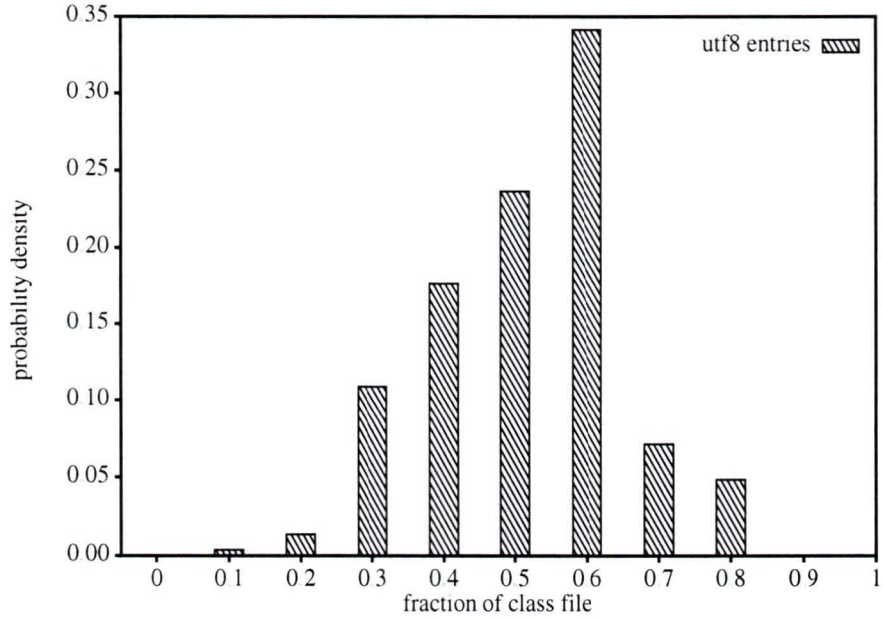
We chose to use a (1,3,16) code (see Section 3.5.2), which could at worst cause an expansion of two bits per class of entry. That is, it requires two bits to represent the fact that there are no entries of a given type. The possible compression improvement over a fixed width coding is negligible for large files, but can give an improvement for smaller files.

4.1.3 Utf8 Entries

By far the most common entries in the constant pool are Utf8 entries. In fact, as Figure 4.3 shows, the Utf8 entries contribute a significant fraction of the overall size of the class file.

We chose to split compression of the Utf8 entries into two distinct stages. We encode the lengths of all the Utf8 entries first, and then follow these by the actual Utf8 data. By separating the two byte length indicators from the actual string data, we simplified our implementation.

Figure 4.3 Utf8 contribution to overall file size



Utf8 Lengths

Figure 4 4 shows the lengths of Utf8 entries in our test data Short strings are more likely to occur than long ones We can exploit this property by choosing a coding for string lengths that reflects this property

Rather than explicitly encode the lengths, the Utf8 entries are sorted by length, as shown in Figure 4 2 We therefore can use delta coding for the string lengths That is, for each length, we encode the difference between the previous length and the current length

Figure 4 5 shows the frequencies of delta string lengths for our test data Ideally we would like to find a coding scheme optimized for a probability distribution that matches that of Figure 4 5 For simplicity and execution efficiency, we chose the start-step-stop code that most closely approximates that curve

Figure 4.4 String Lengths

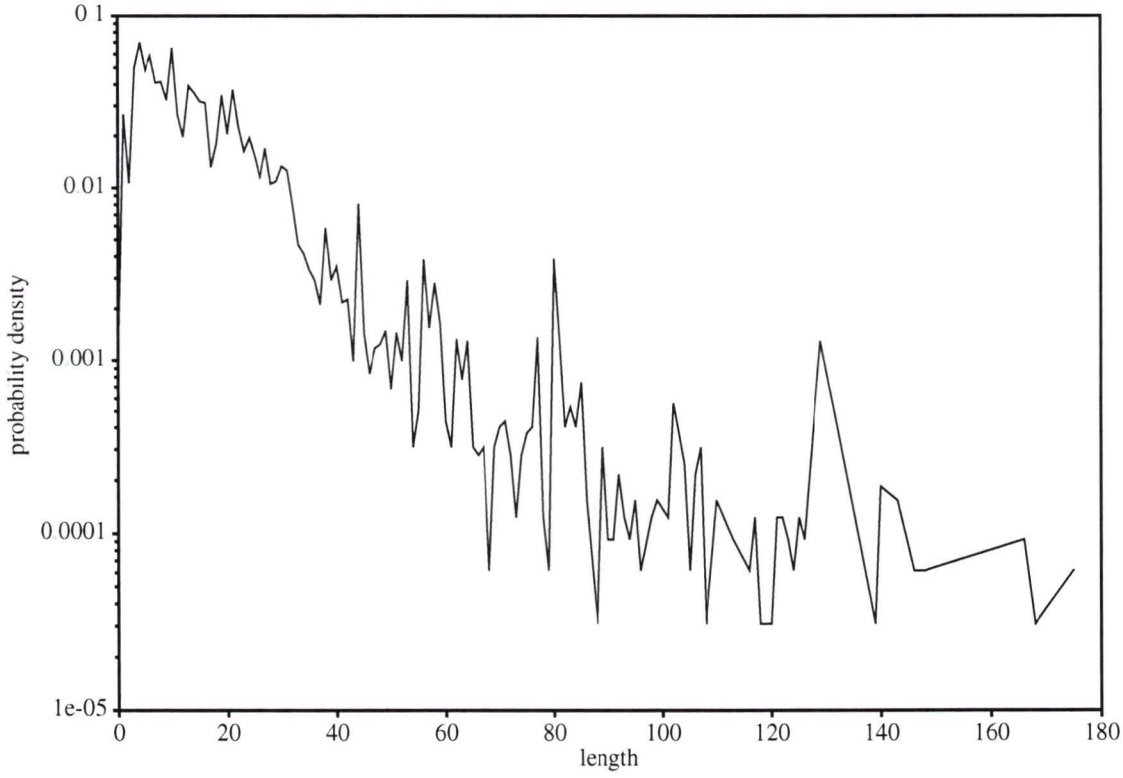
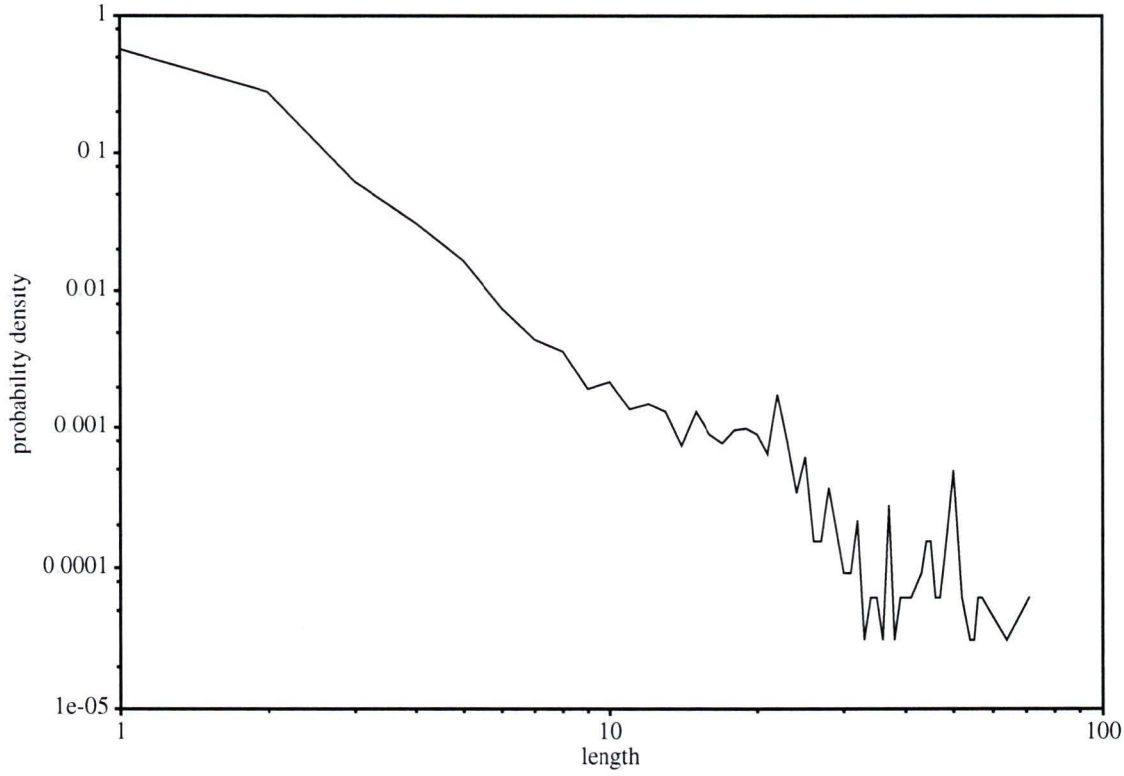
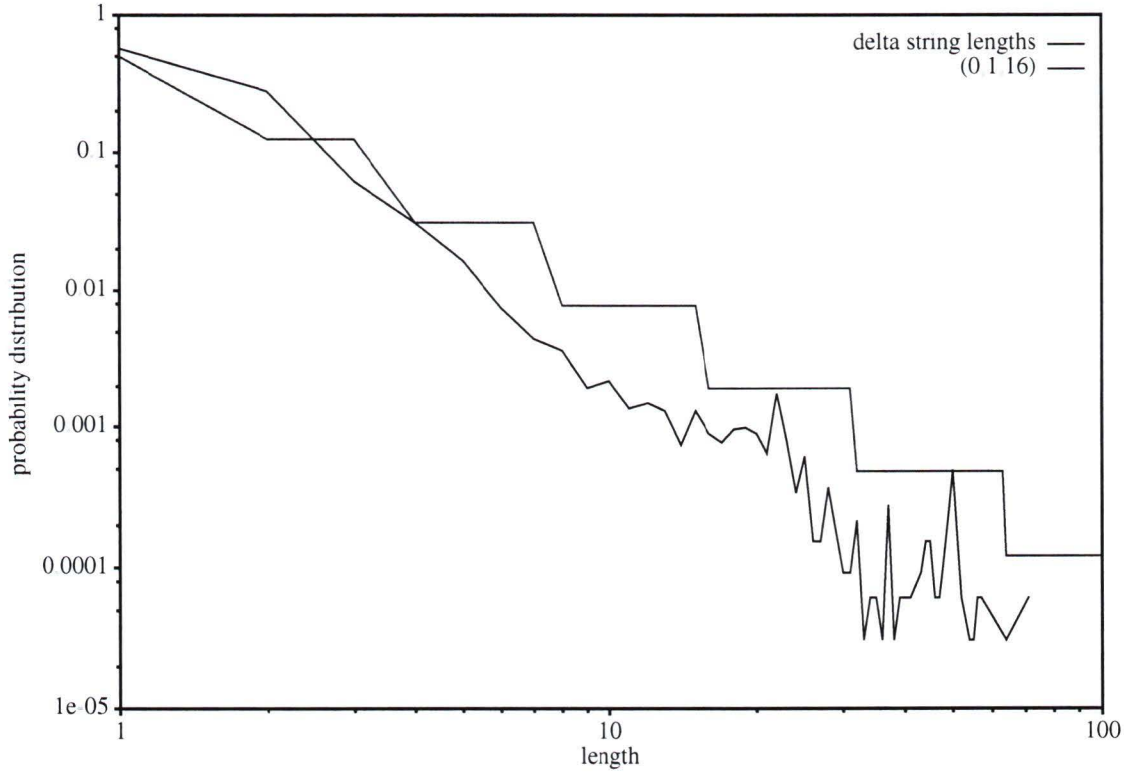


Figure 4.5 Delta String Length



As shown in Figure 4 6, the (0,1,16) code provides a good match to the actual distribution of delta string lengths The exact choice of code has little effect on overall compression efficiency, provided the code favors short integers over long ones Experiments with a (0,2,16) code decreased compression efficiency by less than 1%

Figure 4.6 Delta String Lengths with a (0,1,16) Code



Utf8 String Data

The Utf8 data, which is composed mainly of ASCII strings, can be compressed using any of the existing general purpose data compressors

Some effort was put into investigating the possibility of constructing an optimal dictionary for dictionary compression, since we would be willing to expend more CPU time at the compression stage provided decompression speed was not adversely affected However, it has been shown that construction of an optimal dictionary is an NP-complete problem [20]

We investigated constructing a "nearly-optimal" dictionary using suffix trees [18] In the end this was also dismissed as being too difficult to implement for the estimated compression improvement

We decided to compress the Utf8 entries using an LZ77 variant There are several advantages of LZ77 over LZ78 for this application First, LZ77 is able to exploit all substrings previously seen in the input, whereas LZ78 is restricted to strings explicitly

added to the dictionary. Also LZ77 decompression is extremely fast, and is easier to implement than LZ78.

We are able to preload the LZ77 window with strings that are likely to occur with no penalty to compression performance if they do not occur. Specifically, we have preloaded the dictionary with all the *Attribute* names. This provides only a small compression improvement, but comes at no cost.

If we preload an LZ78 dictionary and the strings do not occur, we have wasted dictionary indices.

A straight forward implementation of LZSS provided adequate compression performance. However, since there are several other LZ77 variants which consistently outperform LZSS [2, Section 9.3], we spent some time investigating them. The highly efficient LZ77 derivative used in `zip` and `gzip` is available as a library called ZLIB [7]. Using ZLIB rather than LZSS improved overall compression by up to 10%.

4.1.4 Other Entries

Other entry types are less common than Utf8 entries. They tend to contribute less to the overall size of the class file. Still, we would like to compress them whereever possible. As discussed in Section 4.1.2, we have eliminated the tag byte for all constant pool entries.

Some constant pool entries contain references to other constant pool entries (see Section 2.2.1). Since we always know the type of entry being referenced [17], we can replace the absolute reference with a relative one. Using this representation, an entry that previously required 16-bits, now requires:

$$\lceil \log(entrycount) \rceil \text{ bits}$$

The principal advantage to using this fixed-width representation is speed. Indeed, fixed width coding is rarely optimal in terms of compression efficiency. A more sophisticated coding scheme, such as Huffman coding, would likely provide better compression at the expense of speed and implementation difficulty.

The least common constant pool entries are those that specify integer and floating point constants. In the over 1000 class files analyzed, there were only 15 floating point constants, 757 integer constants, 220 long integer constants, and 19 double floating point constants. With such a small sample to work with, and with the presence of "bit-field" constants, special coding techniques for the constants as in [14] are unlikely to provide any significant compression. We have emitted these constants as is.

4.2 Coding of Attributes

Attributes can contribute a significant portion to the overall size of the class file. However, in most cases they are dominated by the constant pool. Nevertheless, we have provided specialized coding for all the attributes presently defined by Sun.

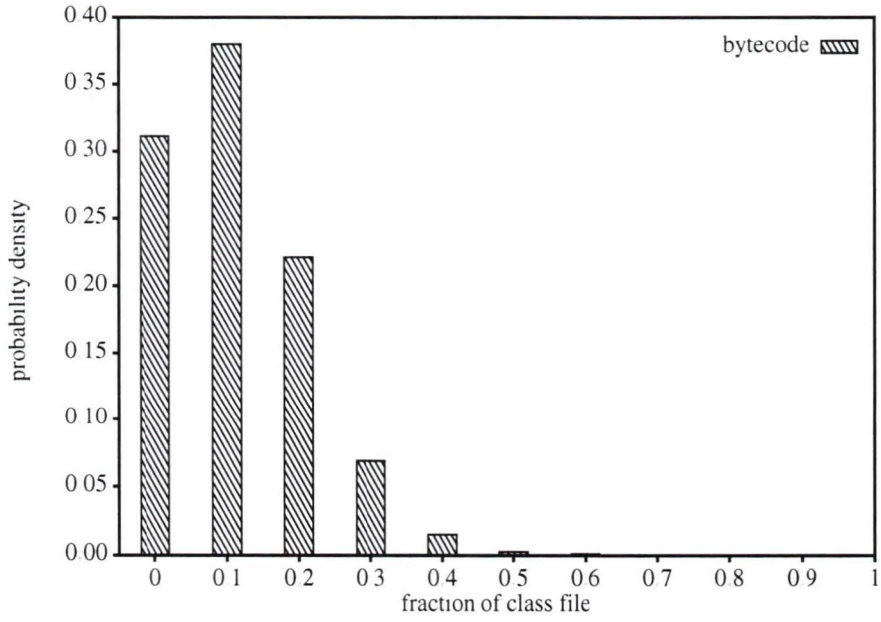
The Java Virtual Machine specification requires that compliant virtual machines ignore attributes that they do not recognize. Similarly, our compressor must preserve those attributes that it doesn't recognize. Attributes we don't specifically recognize are encoded using the LZ77 based compression provided by ZLIB. This is unlikely to provide optimal compression, but at worst will cause a minimal expansion [7].

All attributes have had explicitly redundant information removed, as discussed in Section 2.4.1. Additionally, constant pool references have been coding using the fixed width representation discussed in Section 4.1.4.

4.2.1 Code Attribute

The *Code* attribute contains information about the methods that a class implements. The major portion of this attribute is the bytecode for the method. Figure 4.7 shows the relative contribution of bytecode to the overall class file size. In the majority of files, the bytecode comprises only a small portion of the file.

Figure 4.7 Bytecode contribution to overall file size



As mentioned in Section 2.5.1, all jump offsets are program counter relative. There is a certain amount of redundancy in this coding, since we also know the size of all the instructions. It is therefore possible to change the program counter relative jump offsets to instruction relative jump offsets. See Figure 4.8 for an example.

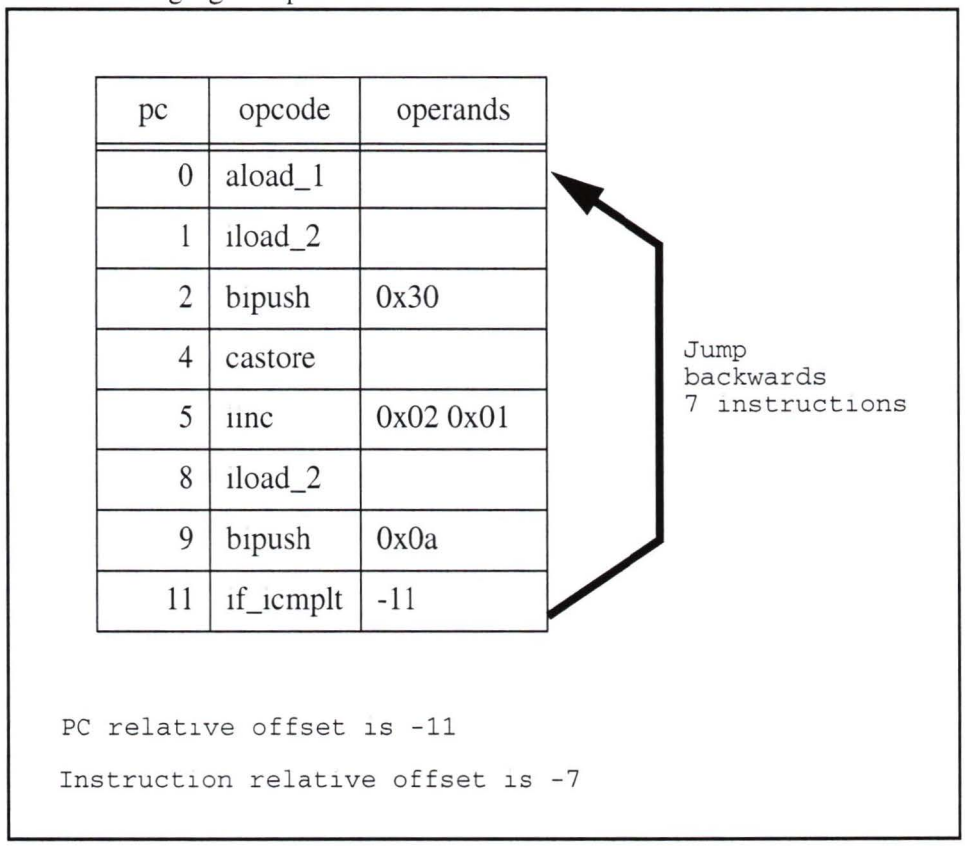
We are also able to change any reference to a program counter to a reference to an instruction number. For example, the exception table portion of the *Code* attribute lists ranges of bytecode addresses where exceptions are handled. These can be changed to ranges of instructions where exceptions are handled.

Because the majority of instructions have operands that are more than 1 byte in length, the change from program counter to instruction relative addressing reduces the magnitudes of the operands. This reduction in magnitude throughout the bytecode can lead to better compression.

There are two instructions for accessing items in the constant pool: `ldc1` and `ldc2`. The `ldc1` instruction has a one byte operand, and is used to access constant pool entries less than 255. The `ldc2` instruction has a two byte operand, and can therefore be used to access items up to 65535. As a result of reordering the constant pool, we may have to

change `ldc1` instructions into `ldc2` instructions. Conversely, we may be able to change `ldc2` instructions into `ldc1` instructions. Our current implementation uses `ldc2` instructions exclusively. This can cause the decompressed file to be slightly larger than the original class file. If this is deemed to be undesirable, the implementation could be modified to use `ldc1` instructions where appropriate.

Figure 4.8 Changing Jump Offsets



A further, unexpected, benefit has been obtained by converting all operands to instruction relative addressing. Before executing the bytecode, the bytecode verifier portion of the Java Virtual Machine must satisfy a number of security constraints [17]. Among other things, it must verify that:

- The targets of all control-flow instructions are the starts of instructions.
- For each exception handler, the starting and ending points are the beginnings of instructions.

As a by-product of our coding technique, these requirements are guaranteed to be satisfied by any class file decompressed from our representation

It was originally hoped that a special-purpose coding scheme could be discovered that could take advantage of the structure of the bytecode Compressing the bytecode and their operands separately did not provide any compression improvement After evaluating several options, we decided to compress the bytecode using ZLIB

In addition to changing the jump offsets to being instruction relative, all the bytecode in a class is concatenated together and compressed as one block This increases the chance of the sliding window algorithm finding good (long) matches

Changing offsets from instruction relative is a time consuming process for the compressor, and the reverse transformation is equally time consuming for the decompressor The increase in compression is typically less than 10% for the bytecode being compressed Unless the decompressor is integrated into a class loader it is likely to not be worth the effort

4.2.2 LineNumberTable Attribute

The size of the *LineNumberTable* attribute is obviously proportional to the amount of bytecode in a method For large methods, this table can be quite large The program counter relative portions of the *LineNumberTable* have been converted to instruction relative See Section 2.4.1 for a description of this attribute

Our coding of the line number table exploits two things

- the `start_pc` field is always increasing, and
- line numbers tend to be localized

We have employed delta coding for both fields in the `line_number_table` The delta values are coded using a (2,2,16) code

4.2.3 LocalVariableTable Attribute

This attribute didn't occur in any of the class files we used for testing. These files were compiled with a variety of compilers, but none seemed to produce this attribute. We have employed a scheme similar to the *LineNumberTable* attribute, but have been unable to test its effectiveness.

5 Implementation and Results

In this chapter we discuss our implementation, compression performance, and possible improvements to our algorithm. The implementation discussed in this chapter embodies the ideas presented in Chapter 4.

There are several ways to state the compression statistics. In the following sections, we define compression ratio as `compressed size / original size`.

Using this definition, a 1000 byte file compressed to 200 bytes, would yield a compression ratio of 0.20. Better compression is indicated by a lower compression ratio.

5.1 Implementation

A compressor and a decompressor have been written in C++. The compressor consists of approximately 4000 lines of code. The compressor is large because it needs to parse the class file, and the class file has a non-trivial structure. The decompressor consists of approximately 1000 lines of code. Compression is more complicated than decompression, and this is reflected in the complexity of the code.

Our implementation is a prototype for research purposes. Little effort has been expended to optimize the program for either speed or memory utilization efficiency. Indeed, we have tried to keep the implementation as simple as possible. When there was a choice between ease of implementation and efficiency, we almost always chose ease of implementation. The programs have been written to demonstrate that the ideas presented in Chapter 4 provide good compression, not to be used in any practical application.

There are several areas where the implementation could be improved. We are currently sorting the Utf8 entries by length using simple insertion sort. For small class files, this is acceptable, but an $n\log(n)$ implementation would clearly be better. There are areas where two passes are made over the input whereas a more sophisticated implementation should be able to compute the same results in a single pass.

5.2 Compression Results

The compression performance of the algorithm is excellent compared to general-purpose data compressors. There would be little point in implementing our ideas if the compression performance were not significantly better than existing general-purpose compressors.

Table 5.1 shows the performance on selected class files drawn from a variety of applications. For comparison purposes, we have included `gzip`, one of the best Lempel-Ziv based compressors commonly in use, and `bred`. Bred has been shown to be equivalent to the best statistical compressor in the literature, PPM* [5][3][22]. We have denoted our implementation as `clazz`.

Table 5.1 Compression of Selected Class Files (results in bytes)

Filename	Original Size	gzip	bred	clazz
Agent	25714	13477	13703	9165
Applying	798	512	463	331
Assembler	9663	4894	4907	3409
AudioClip	233	217	182	123
ByteIterator	1879	909	805	672
CacheEntry	461	332	295	195
DrawingCanvas	1756	1031	961	678
King	646	509	466	329
State	3056	1721	1635	1169
TTY	28055	14143	14337	9631
Total	72261	37745	37754	25702

General-purpose compressors tend not to perform as well on small files. They typically need a few kilobytes of data to work with before they can discover and then exploit the redundancy in the data. In most applications, this is acceptable, since the absolute compression gain for smaller files is small. However, most Java class files are small because they contain the information for a single programmer defined class. In a traditional environment a large program consists of one main "executable image", but in the Java environment an application is made up of many small class files. In contrast to general-purpose compressors, our algorithm performs quite well on both small and large class files.

Table 5.2 shows the performance of our algorithm over two large Java libraries. These results are obtained by compressing each individual file in the library, and then summing the size of the compressed files. We can see that `bred` is outperforming `gzip`, but both are significantly outperformed by our algorithm.

Table 5.2 Compression of Various Libraries

Library	Total Size (bytes)	gzip	bred	clazz	Improvement
JGL V1.1	233408	0.51	0.48	0.34	28%
Sun's Class Library	1358044	0.53	0.51	0.36	29%

5.3 Analysis of Compression Performance

In the following sections, we analyze each of the key areas of our compression algorithm to determine where compression performance is coming from. Almost all the techniques depend on the constant pool being reordered, so we don't discuss the overall affect of that portion of the compressor.

5.3.1 Elimination of Tag Bytes

Tag bytes in constant pool entries contribute between 2% and 6% to the total file size in our test data. As such, the tag bytes do not constitute a major portion of the class file. By replacing these tag bytes with a counter, we are able to obtain excellent compression.

We use a (1,3,16) code to represent the number of entries of each type in the constant pool. Table 5.3 shows selected values of the (1,3,16) code. Each entry in the table reflects the number of bytes in the original file we are able to replace with the number of bits indicated in the third column.

Table 5.3 Select Values for a (1,3,16) Code

Integer	Code	Bits
0	1 0	2
2	10 0000	6
18	110 0000000	10
274	1110 0000000000	14
2322	11110 0000000000000	18

From the table, it is clear that when there are more than a few entries of a given type we are getting excellent compression! We chose not to report these results using a compression ratio because the ratios work out to be much less than 1%. The discussion above should give the reader a better understanding of what compression is achieved.

5.3.2 Utf8 Entries

As discussed in Section 4.1.3, the Utf8 entries are the most common of all constant pool entries. We split the encoding of Utf8 entries into two distinct phases: encoding the lengths, then encoding the data. Table 5.7 shows statistics gathered about compressing the Utf8 entries for our test data.

Table 5.4 Compression of Utf8 Entries

Compression	Utf8 Length	Utf8 Data
maximum	0.07	0.21
minimum	0.28	0.63
average	0.17	0.41
median	0.16	0.41

As Table 5.7 shows, we are achieving excellent compression of the length fields for the Utf8 entries. On average, the length fields are compressed to 17% of their original size.

Compression of the Utf8 data is not nearly as good. However, it is consistent with what we would expect from an LZ77 variant compressing ASCII text. Better compression of the Utf8 data is clearly an area where this algorithm could be improved. We are faced with the trade-off between compression performance and speed. We chose speed and ease of implementation over compression in this particular case.

5.3.3 Other Constant Pool Entries

As mentioned in Section 4.1.4, we have spent little effort compressing the remaining constant pool entries. Indeed, we have performed no compression on the integer and floating point constants other than the removal of tag bytes.

Table 5.5 Compression of the Constant Pool

Compression	Constant Pool Entries
maximum	0.15
minimum	0.79
average	0.24
median	0.24

With an average compression ratio of 0.24, the current method of compressing the remaining constant pool entries is acceptable. We could achieve better compression if we were willing to expend the CPU time to do so. One obvious improvement would be to replace the fixed-width coding of constant pool references with Huffman coding.

Given that the overall performance of the algorithm is already quite good, we decided to accept this compression.

5.3.4 Code Attribute

Overall, compression of the bytecode is not nearly as good as other areas of the class file. This is to be expected, since the bytecode is similar to traditional object code, and object code tends to be less compressible than textual data [2, Appendix B].

In fact, as Table 5.7 shows, we can expand the bytecode by trying to compress it! Further analysis showed that while the relative expansion can be quite large (up to 3.4 times), the size of the bytecode in these cases was well under 20 bytes. Part of the expansion is due to the eight byte header inserted by ZLIB. So, while the percentage expansion is large, the effect on the size of the compressed file is small.

Table 5.6 Compression of Bytecode

Compression	Bytecode
maximum	0.26
minimum	3.40
average	1.07
median	0.81

In a production compressor, one could check to see if compression is performed, and if none occurs, store the data in raw form. This requires the addition of only a single bit to distinguish between the compressed and uncompressed formats. Table 5.7 shows compression performance we could have achieved if we had performed this optimization.

Table 5.7 Improved Compression of Bytecode

Compression	Bytecode
maximum	0.26
minimum	1.03
average	0.79
median	0.80

5.3.5 LineNumberTable Attribute

Table 5 7 shows the compression attained for *LineNumberTable* attributes over our test data. Compression of this attribute is of questionable worth, since it is used for debugging purposes and will likely not be present in release quality code.

Table 5.8 Compression of LineNumberTable

Compression	LineNumberTable
maximum	0 20
minimum	0 49
average	0 29
median	0 28

5.4 CPU Requirements

In this section we detail the CPU requirements for our algorithm. All measurements were performed on an otherwise unloaded 486-DX/4 100 MHz machine running Linux 2 0 8. Times reported are the sum of system and user times.

It is worth noting that both `gzip` and `bred` are heavily optimized for efficiency, while our algorithm has had no speed optimizations performed. As expected, both `bred` and `gzip` consistently compress and decompress faster than our algorithm.

Table 5 9 shows that compression with `gzip` is approximately three times faster than our algorithm. Compression with `bred` is approximately twice as fast. As noted earlier, there are several areas where compression could be sped up, most notably the insertion sort used to sort Utf8 entries by length.

Table 5.9 Compression Times (in seconds)

Filename	Size	gzip	bred	clazz
Agent	25714	0.17	0.41	0.57
Applying	798	0.04	0.04	0.15
Assembler	9663	0.07	0.16	0.26
AudioClip	233	0.04	0.03	0.14
ByteIterator	1879	0.05	0.05	0.15
CacheEntry	461	0.03	0.03	0.14
DrawingCanvas	1756	0.04	0.04	0.15
King	646	0.04	0.04	0.14
State	3056	0.04	0.07	0.16
TTY	28055	0.18	0.48	0.61

Table 5.9 shows decompression times. Again our algorithm is consistently outperformed by `gzip`. However, on larger files, our algorithm sometimes outperforms `bred`.

Table 5.10 Decompression Times (in seconds)

Filename	Size	gzip	bred	clazz
Agent	25714	0.06	0.37	0.34
Applying	798	0.03	0.04	0.12
Assembler	9663	0.04	0.13	0.20
AudioClip	233	0.03	0.03	0.12
ByteIterator	1879	0.02	0.03	0.15
CacheEntry	461	0.03	0.03	0.12
DrawingCanvas	1756	0.04	0.04	0.15
King	646	0.03	0.03	0.12
State	3056	0.06	0.06	0.16
TTY	28055	0.18	0.36	0.37

As we discussed in the introduction, one can imagine that compression will be performed once at the server side, and decompression will be performed many times by clients accessing the server. For this reason, we consider decompression times significantly more important than compression times.

While it is possible to imagine speeding up our compression algorithm, it is unlikely that it will ever be faster than `gzip`. Our algorithm requires that two passes be made over both the constant pool and the bytecode, whereas `gzip` requires only a single pass over the entire file.

It is, however, likely that the decompressor could be sped up significantly by a careful implementation and by integration into the Java Virtual Machine's class loader. The decompressor would no longer be duplicating work that must be performed by the bytecode verifier when it converts instruction relative addressing to program counter relative addressing (see Section 4.2.1.) It is wasteful for the decompressor to convert our compressed format into the class file format. Ideally, the compressed class file would be loaded directly into the Java Virtual Machine.

6 Conclusions

The main contribution of this thesis is a compression program that outperforms existing general purpose data compressors on Java class files. On average, our program is 28% better than general-purpose compressors. In a research area where a few percent increase in compression performance is considered good [5][6], a 28% increase should be considered exceptional.

We obtained our results by careful application of existing data compression methods to specific areas of the class file, and by transforming regions of the class file to make them more compressible.

Since Java class files tend to be small, performance on small files is important. Our program performs well on small files, whereas this is usually a weakness of general-purpose compressors. Indeed, we have yet to find a single class file where our algorithm didn't outperform both `bred` and `gzip` by at least 15%.

Many of the methods we used to obtain our compression results are considered "ad hoc" by the authors of [2]. Indeed, they go so far as to say:

> techniques tailored to take advantage of idiosyncratic repetitions and regularities are - or should be - a thing of the past. Adaptive models discover for themselves the regularities that are present, and do so more reliably than people.

Perhaps in theory this is true, but since our algorithm outperforms these "adaptive models" by a significant amount, it appears the "ad hoc" methods are still useful in practice.

6.1 Future Work

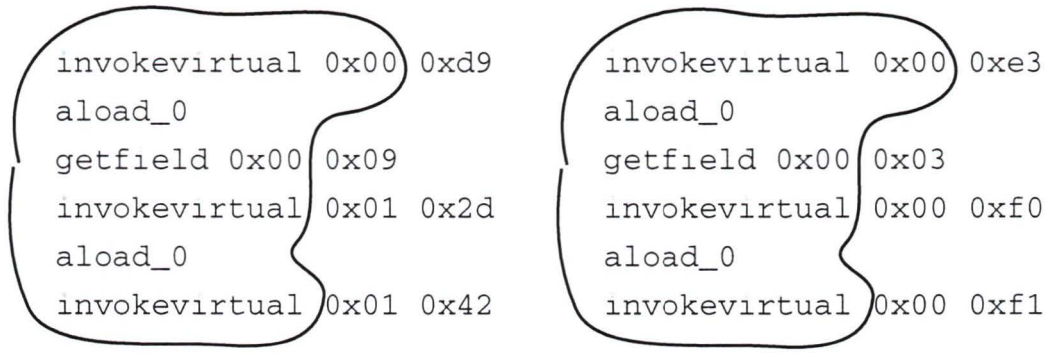
Perhaps the most practical extension to this research would be to incorporate our program into an existing Java environment. The compression algorithm could be incorporated directly into a Java compiler. Given the appropriate flag, the compiler would produce class files in the compressed format, rather than the native class file format. Depending on how the compiler is implemented, this could be quite easily accomplished. If this isn't possible, a command line compressor would still be useful.

As mentioned earlier, the decompression algorithm should be incorporated into a class loader. This would eliminate the redundant work be performed by the decompressor and the existing class loader, and would allow the Java environment to support compressed files natively.

Further experiments are needed to determine if it is indeed faster to transmit the compressed files and decompress them versus transmitting uncompressed files and loading those files.

We are left with the feeling that better compression of bytecode could be obtained. LZ77 compression is not performing exceptionally, and there surely must be an algorithm more suited to compressing the bytecode. One possibility might be to use a parameterized encoding scheme similar to that described by Franz in [12]. The algorithm is a variation on LZ78 which allows "templates", or inexact matches. The algorithm isn't directly applicable to compressing bytecode, but a similar algorithm should exist for bytecode. Consider the two fragments of bytecode shown in Figure 6.1.

Figure 6.1 Parameterized Matching of bytecode



These two sequences of bytecode differ only by a few operands, the circled areas could be matched using templates

Using a conventional LZ77 or LZ78 based coding scheme, the two fragments wouldn't provide a match of more than the opcodes involved, since matches must be consecutive in the input However, if we allowed for "templates", or "parameterized" matches, we could match the sequences of instructions as shown in the figure, and then specify the operands separately Perhaps this type of parameterized matching could be applied in a more general sense It is definitely a problem that warrants further research

The algorithm described in [11] performs this style of parameterized matching (although it is described as tree-matching) The algorithm described can require significant amounts of CPU time, while compression ratios approach only 0 50 Results are reported for only two programs, both of which are quite large It is uncertain how the algorithm would perform on the small amount of bytecode typically present in a class file For our purposes, a simpler, faster implementation is needed

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986

[2] Timothy C Bell, John G Cleary, and Ian H Witten, *Text Compression*, Prentice-Hall, 1990

[3] M Burrows and D J Wheeler, "A Block-sorting Lossless Data Compression Algorithm," *SRC Research Report 124*, Digital Systems Research Center, 1994

[4] Robert Cameron, "Source encoding using syntactic information source models," *IEEE Transactions on Information Theory*, Vol 34, No 4, 843-850, 1988

[5] John G Cleary, W J Teahan, and Ian H Witten, "Unbounded Length Contexts for PPM," *Proceedings DCC 1995 Data Compression Conference*, IEEE Computer Society Press, 52-61, 1996

[6] Rod M Davies and Ian H Witten, "Compressing Computer Programs," *Technical Report 93/7*, University of Waikato, New Zealand, 1993

[7] L P Deutsch, "ZLIB Compressed Data Format Specification," available in `http //quest jpl nasa gov/zlib/rfc-zlib html`

[8] David Engberg, "Guavac Java Compiler," available in `http // http cs berkeley edu/~engberg/guavac/`

[9] Edward R Fiala and Daniel H Green, "Data Compression with Finite Windows," *Communications of the ACM*, Vol 32, No 4, 490-505, 1989

[10] David Flanagan, *Java in a Nutshell*, O'Reilly & Associates, 1996

[11] Christopher W. F'55ser and Todd A. Proebsting, "Custom Instruction Sets for Code Compression", Unpublished Technical Report, available in `http://www.cs.arizona.edu/people/todd/papers/pldi2.ps`

[12] Michael Steffen Oliver Franz, "Code-Generation On-the-Fly: A Key to Portable Software", Ph. D. Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1994.

[13] James Gosling and Henry McGilton, "The Java Language Environment," *Sun Microsystems White Paper*, Sun Microsystems, 1995.

[14] Eric C. Hehner, "Matching Program and Data Representations to a Computing Environment," Technical Report CSRG-44, University of Toronto, 1974.

[15] The Java Generic Library (JGL), `http://www.objectspace.com/jgl/`

[16] Douglas Kramer, "The Java Platform," *Sun Microsystems White Paper*, Sun Microsystems, 1995.

[17] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[18] Edward M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, Vol. 23, No. 2, 262-272, April 1976.

[19] Mark Nelson, *The Data Compression Book*, M&T Books, 1991.

[20] James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, 1988.

[21] James A. Storer and Thomas G. Szymanski, "Data Compression via Textual Substitution," *Journal of the ACM*, Vol. 29, No. 4, 928-951, 1982.

[22] D. Wheeler, "An implementation of block coding," Cambridge University, July 1995, `ftp://ftp.cl.cam.ac.uk/users/djw3/bred.ps`

Vita

Surname: Corless Given Names: Jason David

Place of Birth: Prince George, British Columbia, Canada

Education Institutions Attended:

University of Victoria 1990 to 1996

Degrees Awarded:

B.Sc. (Co-op) University of Victoria 1994

Honours and Awards:

NSERC PGS A Scholarship 1994 to 1996

Publications:

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Compression of Java Class Files

Author: [redacted]

Jason David Corless

Date: Jan 23/97