Compressing Java Class Files

William Pugh
Dept. of Computer Science
Univ. of Maryland, College Park
pugh@cs.umd.edu

Abstract

Java class files are often distributed as `jar` files, which are collections of individually compressed class files (and possibility other files). Jar files are typically about 1/2 the size of the original class files due to compression. I have developed a wire-code format for collections of Java class files. This format is typically 1/2 to 1/5 of the size of the corresponding compressed jar file (1/4 to 1/10 the size of the original class files).

1 Introduction

This paper examines techniques for compressing (collections of) Java class files. Java class files are generated by Java compilers, are the standard distribution medium for Java programs and are the usual way of providing programs to a Java virtual machine. Java class files contain a substantial amount of symbolic information. In the `javac` benchmark from SPEC JVM98, only 21% of the uncompressed class file size is actually taken up by the method bytecodes. One purpose of this is to avoid the need to recompile all Java classes that use a class `X` whenever `X` is changed. So long as the functionality depended on doesn't change, previously compiled Java classes will work with the new version of `X`.

Few interesting Java applications are comprised of a single class. Many applications are composed of hundreds or even thousands of classes. Java class files can be collected in `jar` files, which are collections of compressed Java class files (and possibly other files, such as images). `Jar` files are used both on disk and for network transmission.

In many applications, Java programs are transmitted across the network. While ample bandwidth is available in some situations, there are many applications in which there are slow modem or mobile communication links in the network. The `jar` format normally uses the gzip compression mechanism to compress the files in a jar file. This typically provides a factor of 2 compression over standard Java class files. However, the compressed jar files for substantial applications can still be quite large (50-200K is not unusual), and take several minutes to transmit over a slow communication link.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
© 1999 ACM
To appear at the ACM SIGPLAN Conference on Programming Language Design and Implementation, May 2–4, 1999, pages 247-258

I use a number of approaches to creating smaller files that contain the same information as a jar file:

- Developing a more efficient and compact organization of classfile information.
- Taking steps to allow gzip to do a better job of compressing the information we use.
- Sharing information across class files, to reduce transmission of redundant information.

Although this paper focuses solely on the problem on compressing Java class files, many of the techniques described would be generally useful for developing compact object serialization protocols.

2 Methodologies and Baselines

In this paper, I explore wire-formats for collections of Java class files. I assume that bandwidth is the most precious resource. Time required to compress a Java archive is relatively unimportant, while the time required to decompress must be reasonable (not significantly longer than using gzip). The wire-format is a sequential format: all of the class files must be decompressed in sequence. As they are decompressed, they can be written to disk as a conventional jar file or separate classfiles. These would be completely conventional classfiles that could be used by a standard JVM. Alternatively, each class can be directly loaded into a JVM as it is decompressed, saving the expense of constructing the classfile. For this, a custom classloader would be required, but no other changes to the JVM would be

Benchmark	sj0r	jar	sjar	sj0r.gz	sjar/ sj0r	sjar/ jar	sj0r.gz/ sjar	Description
	Size in Kbytes							
rt	8,937	5,726	4,652	2,820	52%	81%	61%	Java 1.2 runtime
swingall	3,265	2,193	1,657	998	51%	76%	60%	Sun's new set of GUI Widgets (JFC/Swing 1.1)
tools	1,557	950	737	513	47%	78%	70%	Java 1.2 tools (javadoc, javac, jar, ...)
icebrowserbean	226	125	116	88	52%	93%	76%	HTML browser
jmark20	309	189	173	91	56%	91%	53%	Byte's java benchmark program
visaj	2,189	1,524	1,157	703	53%	76%	61%	Visual GUI builder
ImageEditor	454	359	257	162	57%	72%	63%	Image editor, distributed with VisaJ
Hanoi	86	57	46	31	54%	80%	67%	Demo applet distributed with Jax
Hanoi_big	56	37	30	20	53%	80%	67%	Hanoi, partially jax'd
Hanoi_jax	38	22	21	16	55%	96%	74%	Hanoi, fulled jax'd
javafig	357	198	170	143	48%	86%	84%	Java version of xfig
javafig_dashO	269	136	131	113	49%	96%	86%	javafig, processed by dashO
Programs from SPEC JVM98 (http://www.spec.org/osg/jvm98/)								
201_compress	15	11	10	6	64%	85%	59%	Modified Lempel-Ziv method (LZW)
202_jess	270	183	136	64	50%	74%	47%	Java Expert Shell System based on NASA's CLIPS expert shell system
205_raytrace	52	31	24	15	47%	78%	64%	Raytracing a dinosaurs (invoked by 227_mtrt)
209_db	10	6	6	5	56%	94%	84%	Performs multiple database functions on memory resident database
213_javac	516	274	226	143	44%	82%	63%	Sun's JDK 1.0.2 Java compiler
222_mpegaudio	120	68	62	45	51%	91%	73%	Decompresses MPEG Layer 3 audio
228_jack	115	74	55	36	48%	74%	65%	A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS)

sj0r non-classfiles excluded, debugging information stripped, no compression
jar non-classfiles excluded, class files as distributed (debugging information often not stripped), files compressed individually
sjar non-classes excluded, debugging information stripped, files compressed individually
sj0r.gz non-classes excluded, debugging information stripped, individual files not compressed, jar file gzip'd as a whole

Table 1: Benchmark programs studied in this paper

required. See Section 11 for a discussion of eager class loading.

While it would be possible to include debugging information in a wire-format, we would typically prefer to save space by excluding it. I do not encode the attributes LineNumberAttribute, LocalVariableTable nor SourceFile. Also, because my approach requires that we renumber entries in the constant pool, I exclude any unrecognized attributes (we would not be able to update references to the constant pool in unrecognized attributes).

I also exclude any non-class files (e.g., PNG image files) from archive in performing my size calculations. I report compression as the size of the compressed object, as a percentage of the size of the original object. To have a consistent and fair comparison of the size of my archive format with standard jar files, I performed the following transformations to the benchmarks I studied:

- Remove LineNumberAttribute, LocalVariableTable and SourceFile attributes
- Garbage collect the constant pool (remove unused constants)
- Sort entries in the constant pool according to type
- Sort UTF constants according to their content

These changes typically give a 20% improvement in jar file size Sorting of the constant pool entries can give an improvement of several percent when the class file is compressed, because it enables zlib to do a better job of finding repeated patterns. In this paper, when I report the size of original and compressed class files, those sizes reflect the improvements gained by these transformations. Any improvements I report for the new techniques in this paper reflect improvements beyond those gained by removing debugging information and garbage collecting the constant pool.

I will often refer to gzip and zlib compression interchangeable. However, in most situations where I apply gzip compression I do not include the 18 bytes for the GZIP header and trailer.

2.1 Gzip'd jar files of uncompressed class files

The compression done in normal jar files are on a file-by-file basis. We can achieve better compression if we compress an entire jar file, where the individual files in the jar file have not been compressed separately. In tables and text, I refer to these as `j0r.gz` files (`0` for no compression within the jar file).

3 Basic approaches

When considering techniques for compressing Java classfile archives, one of the first techniques that jumps to mind is reusing constant pool entries. Constant pool

Component	Uncompressed Size (Kbytes) swingall	javac
Total size	3,265	516
excluding jar overhead	3,010	485
Field definitions	36	7
Method definitions	97	10
Code	768	114
other	72	12
constant pool	2,037	342
Utf8 entries	1,704	295
if shared	371	56
if shared & factored	235	26

Table 2: Classfile breakdown

entries are the things that can be referenced within a classfile: examples include classes, methods, integers, doubles, and utf8 encodings of Unicode strings. Often, constant pool entries reference to other constant pool entries. For example, the constant pool entry for a method reference consists of a reference to (the constant pool entry for) the class containing the method, and a reference to the signature of the method.

As you can see in Table 2, the constant pool entries often make up most of the size of a classfile. In fact, the Utf8 entries alone often make up most of size of a classfile. Simply sharing Utf8 entries across classfiles leads to substantial reduction.

But now we have to face another issue: encoding of references to constant pool entries. In most classfiles, the number of constant pool entries is relatively small. While the standard classfile format usually allots 2 bytes to encode a reference to a constant pool entry, we can often do so in a single byte. If we compress single byte control pool references, we are likely to get good compression. But if we pool constant pool entries, it is unlikely that we will be able to encode most constant pool entries in a single byte.

Numerous data compression algorithms have been developed. Many of the lossless compression algorithms in wide use were originally designed as text compression algorithms, and work on a stream of bytes. In particular, the Lempel-Ziv family of compression algorithms have a very strong byte orientation. While it might be possible to adapt them to compress a stream of larger values (e.g., 16-bit values), it isn't clear how efficient they would be. At any rate, efficient implementations of the byte-oriented zlib library exist on most platforms and is part of the standard Java API, so utilizing the existing library makes sense.

A first solution is to use different numbering for different kinds of constant pool entries (e.g., we can have Class 17, and MethodRef 17, and IntegerConstant 17). In almost all[1] contexts, we know the type of the constant pool entry whenever we reference it, so this won't cause confusion.

This helps some and might be sufficient for small archives. However, on large archives these techniques will not be sufficient to allow us to encode most references in a single byte. In addition to encoding most references within a single byte, we would also like the encoding bytestream to have a very skewed distribution, so that it can be further compressed. Techniques for encoding references are discussed in more detail in Section 5.

Even sharing the Utf8 entries still results in a fair bit of redundancy. Each time a classname is encoded, the full package name is encoded (e.g., java.lang), and classnames appear in full text in the types of fields and methods. For example, the type of a method that takes one string as an argument and returns a string is encoded as `(Ljava.lang.String;)Ljava.lang.String;`. If we factor out this duplication, we get another substantial reduction in the space required for string constants. Note that this factoring amounts to a wholesale reorganization of the classfile; the reorganization is described in more detail in Section 4.

The savings in uncompressed size realized by eliminating redundancy often doesn't fully materialize in the size of a compressed archive. By eliminating redundancy, we have removed one of the elements the compressor was using to get better compression. While factoring and other techniques are useful, they are often not as effective as they seem at first.

4 Structuring information

In order to reduce redundancy in my archive format, I redesigned the basic structure of information in a Java classfile. You can think[2] of this restructuring as being an in-memory format for encoding classfiles, which is built and then encoded into a bytestream.

Some of the things I did in my reorganization:

- Classnames are encoded as a package name and a simple class name. All classes from the same package will share the same package name, and classes from different packages can share the same simple class name. For example, the package name `java.lang` will occur only once.
- In Java classfiles, the types of methods and fields are encoded as strings. In my restructured format, a method type is encoded as an array of

[1] The exception to this is the bytecode instructions for loading constants (LDC, LDC_W, and LDC2_W). We can handle this by introducing new pseudo-opcodes in the compressed files that describe the type of constant being loaded (e.g., LDC_Integer).

[2] In fact, my implementation creates an encoding as it traverses the classfile without completely building an in-memory restructured classfile.

classes containing the return type and the argument types. A field type is just a class. Primitive types and array types are encoded as special class references that are converted back to primitive types when decompressed.

- Generic Attributes have been eliminated. Instead, additional flags are set in the access flags that say whether specific attributes apply to this object. For example, there is a bit in the access flags for a Field definition that tells whether the field has a constant value. If so, then there is an additional reference to a constant value (e.g., an integer or a string).

Once we have a collection of class files in our internal format, the wire code is generated/parsed by a preorder traversal of the data-structure, starting from the roots. As each edge is traversed, an appropriate reference is encoded. As each primitive (int, long, float, or double) is encountered, it is encoded.

The internal format for Code (attached to Methods-Definitions) is more complicated. I separate bytecode into streams of opcodes, registers numbers, integer constants, virtual method references, field method references, and so on. The encoding of bytecodes is discussed more throughly in Section 7

5 Compressing References

Given a structure for the data we which to encode (Section 4), we need a way of encoding a reference to an object we may have seen before. For primitives (ints, doubles, ...), I just encode the value of the object, without bothering to check if I have seen the object previously.

Otherwise, we need an encoding that either says we have never seen the object before, or identifies the previously seen object. If we have never seen the object before, then at that point we encode all of the fields/components of the object.

I consider a number of approaches to encoding references. The basic approach that worked best was to use a move-to-front encoding. In a move-to-front encoding, we maintain an ordered list of all of the objects seen. Whenever a previously seen object is to be transmitted, we transmit the position of the object in the list (1 for the first object in the list) and move the object to the front of the list. To transmit an object not seen previously, we transmit the value 0 and insert the object at the front of the list.

I implemented move-to-front queues using a modified form of a Skiplist [Pug90] (the Skiplist structure was modified so that each link recorded the distance it travels forward in the list). By starting the search for an element at the bottom level of the Skiplist, increasing the level to the appropriate level for traversing the Skiplist, and then using a normal Skiplist traversal, I was able to achieve an expected time bound of $O(\log k)$ to do a move-to-front operation on element k of the queue, regardless of the total number of elements in the queue.

This was all that was needed in the decompressor. In the compressor, we also need a way, given an element we may have seen before, to determine if we have seen the element before and if so, where the element is now in the queue. This was implemented by a hashtable from elements to the Skiplist nodes that store them. Once we are at the Skiplist node for an element, we can walk forward to the end of the list (at each node, follow the highest link out of that node, keeping track of the distance traversed by each link). Knowing the distance to the end of the list and the total size of the list allows us to calculate the distance of the element from the front of the list. These operations can all be done in expected time $O(\log n)$, where n is the number of elements in the queue.

A move-to-front generally does an excellent job of producing lots of references with small encodings, which then can often be encoded in a single byte and compresses well with a Huffman encoding. However, a move-to-front encoding pretty much destroys any patterns in the object stream (e.g., an `aload_0` instruction is often followed by a `getfield` instruction). I tried using a move-to-front encoding for JVM opcodes, then using zlib on the result, and got much worse compression than using zlib on the original JVM opcodes. The zlib compression scheme both finds repeating patterns and uses a Huffman-like encoding to efficiently transmit a stream of bytes with a nonuniform distribution pattern. Thus, a move-to-front encoding may do an excellent job when zlib cannot find significant repeating patterns to exploit, but do poorly when they exist.

I compared using zlib on the byte stream generated by a move-to-front encoding with using a Arithmetic encoding on the indices generated by a move-to-front scheme. In the Arithmetic encoding, encoding an index that occurs with probably p requires $\log_2 1/p$ bits. Given the hypothesis that a move-to-front encoding destroys references patterns and only produces a skewed probably pattern, we would expect the Arithmetic encoding to do better. In the cases I examined, this expectation was fulfilled. For example, for references to virtual methods in rt.jar, using zlib gave results that were 2% bigger than an Arithmetic encoding.

However, these results do not include the size of the dictionaries for the arithmetic encoding, and arithmetic encoding is rather expensive to compress and decompress. The size of the dictionary would be larger than the savings unless it was fitted to a curve and just the parameters for the curve were encoded. Given the negligible or non-existent benefits and the performance cost

```
ClassDefinition [] classesDefined;

class PackageName { String name; }
class SimpleClassName { String name; }
class MethodName { String name; }
class FieldName { String name; }

class ClassRef {
    PackageName & packageName;
    SimpleClassName & simpleClassName;
}
class ClassDefinition {
    ClassRef & thisClass;
    int access_flags;
    ClassRef & superClass;
    ClassRef & [] interfaces;
    MethodDefiniton [] methods;
    FieldDefinition [] fields;
    }
class ExceptionRef {
    ClassRef & clazz;
}
```

```
class MethodRef {
    ClassRef & owner;
    MethodName & methodName;
    ClassRef & type[];
    }
class MethodDefinition {
    MethodRef & method;
    int access_flags;
    Code code;
    ExceptionRef & exceptionsThrown[];
    }
class FieldRef {
    ClassRef & owner;
    FieldName & fieldName;
    ClassRef & type;
    }
class FieldDefinition {
    FieldRef & field;
    int access_flags;
    Object & constantValue;
    }
```

& is used to indicate a reference to an object that may be shared and might have been seen before

Figure 1: Fragment of Internal format for class files

Benchmark	Simple	Basic	Freq	Cache	Move-to-front Basic	Move-to-front Transients	Move-to-front Use Context	Move-to-front Transients and Context
rt	503,522	480,535	398,303	337,201	301,704	299,159	293,451	291,052
swingall	172,372	159,869	136,241	117,254	110,370	109,211	107,247	106,223
tools	94,293	85,547	71,396	64,417	57,207	56,778	55,408	54,998
icebrowserbean	16,935	14,907	12,945	11,616	10,596	10,550	10,260	10,233
jmark20	18,041	14,497	12,583	9,897	9,879	9,954	9,622	9,658
visaj	124,297	116,316	99,216	84,854	76,585	76,080	74,800	74,400
ImageEditor	25,669	23,473	19,886	16,871	15,834	15,750	15,361	15,323
Hanoi	5,953	4,704	4,245	3,824	3,788	3,794	3,648	3,650
Hanoi_big	3,866	2,973	2,617	2,370	2,316	2,318	2,243	2,242
Hanoi_jax	3,078	2,376	2,112	1,883	1,852	1,874	1,814	1,832
javafig_dashO	22,727	19,963	17,768	16,870	15,954	15,891	15,450	15,380
javafig	27,897	23,285	20,596	19,573	18,199	18,079	17,630	17,481
201_compress	757	516	506	497	461	477	456	470
202_jess	10,032	8,256	6,831	6,347	6,224	6,176	5,969	5,876
205_raytrace	2,603	1,966	1,812	1,762	1,646	1,671	1,550	1,576
209_db	843	575	489	483	466	476	455	467
213_javac	22,338	17,815	15,109	14,325	14,193	14,041	13,622	13,504
222_mpegaudio	4,568	3,440	3,143	2,917	2,706	2,708	2,644	2,674
228_jack	6,025	4,559	4,077	3,993	3,723	3,747	3,521	3,542

Table 3: Size (in bytes) of compressed references

ditching the built-in zlib decoder for a arithmetic decoder, I decided that this option wasn't worth pursuing.

5.1 Variants

I considered the following variants, as

- baselines, to see the advantages given by the move-to-front encoding;
- competitors, that in the end were not as effective; and
- variants, that provide minor improvements in compression.

Except where noted, seperate pools were used for virtual, interface, static and special method references, and for static and instance field references. The resulting indicies are encoded as a byte stream and compressed as described in Section 6.

5.1.1 Baseline: Simple

Each object is assigned a fixed id. Id's are assigned sequentially, as objects are first seen. All id's are encoded as two bytes. A single pool is used for all method references, and a single pool is used for all field references.

5.1.2 Baseline: Basic

Each object is assigned a fixed id. Id's are assigned sequentially, as objects are first seen, but are encoded compactly.

5.1.3 Competitor: Freq

Like Basic, except that ids were assigned to objects so that the most frequently referenced objects had the smallest id's. Elements that only occur once are all encoded with the same special id.

5.1.4 Competitor: Cache

The Freq scheme was augmented with a LRU cache of 16 elements, implemented as a move-to-front queue. If an object is in cache, it is encoded according to its position in the cache. Separate caches were used for each context.

5.1.5 Variant: move-to-front, transients

In this scheme, objects that are seen exactly once are encoded specially and are not entered into the move-to-front queue.

5.1.6 Variant: move-to-front, use context

For method references, in addition to maintaining different MTF queues for different method kinds (virtual, interface, ...), we also maintain different MTF queues based on top two values of the computed approximate stack state (described in Section 7.1). Thus, we have one MTF queue for virtual methods invoked when there are two integers on top of the stack, and another MTF queue for virtual methods invoked when the top two values on the stack are a reference and an integer.

I do use a common pool for method names across all method types (virtual, static,...), particularly to avoid creating duplicate constant pool entries. However, under this option, we maintain different MTF queues for each method type.

One complication here is that when a method reference is seen for the first time, it must be inserted into all of the MTF queues where it might be seen later.

6 Encoding Integers

Both in encoding integers that naturally appear in a classfile (e.g., integer constants in bytecode, maximum stack size for code) and in encoding the indices arising from an encoding of references, we need to consider how to convert them into a bytestream we can hand off to the compressor.

Of course, a sequence of 16 or 32 bit integers can easily be turned into a sequence of 8 bit integers. But this sequence would contain a mixture of high bytes and low bytes, which would likely have different frequency distributions and result in poor compression.

The approach we take for encoded unsigned integers is to encode the lowest seven bits in a byte, with the high bit set if more bits are coming. This works well in cases where we don't know the maximum value or distribution but expect that the distribution is skewed towards small numbers (it works very poorly if most numbers being encoded are in the range 128-255).

In other situations both the encoder and the decoder know the range of possible values (e.g., that the integer to be encoded is in the range 0...4242). In such cases, we use a scheme that takes into account the range of values that need to be transmitted. If we know that values $0 \ldots (n-1)$ need to be transmitted ($n \leq 2^{16}$), we reserve the highest $r = \lfloor \frac{n-2}{255} \rfloor$ bit patterns in the first byte to indicate that this is a two-byte value. If $x \geq 256 - r$, x is encoded as

$$[((x - (256 - r)) \bmod r) + 256 - r, \lfloor (x - (256 - r))/r \rfloor]$$

Using variable length encodings as above for signed integers would result in a multi-byte encoding of all negative numbers since their representation is at the high end of the unsigned range. We fix this by essentially

Compression for	Benchmark programs javac	mpegaudio
Bytestream	48%	43%
Opcodes	36%	17%
using Stack State	35%	15%
using Custom opcodes	34%	13%
Register numbers	39%	34%
Branch offsets	41%	52%
Method references	35%	28%

Table 4: Compression for bytecode components

moving the sign bit of signed integers into the least significant bit position; x is encoded as $x \geq 0 \ ? \ 2x : -2x - 1$. Thus, $\{-3, -2, -1, 0, 1, 2, 3\}$ is encoded as $\{5, 3, 1, 0, 2, 4, 6\}$.

7 Compressing Bytecodes

Java bytecode sequences are a mixture of opcodes, integer constants, register numbers, constant pool references and branch offsets. As has been suggested previously [EEF+97], we might be able to achieve better compression if we separate that information into separate streams and compressed them independently.

Of course, note the "we might". It isn't guaranteed. For example, local 0 is initially (and generally throughout a method) used to store `this` (for non-static methods). There are some instruction patterns that depend on the registers and other values in the bytecode sequence. For example, an `aload` instruction is much more commonly followed by a `getfield` instruction when the `aload` instruction loads local 0. As it turns out, we would pick this up even though bytecodes are separated, because a special opcode is used for loading a reference from local 0 (`aload_0`). When we separate out the operands from the opcodes, we don't separate out the implicit operands in opcodes such as `iconst_2` and `aload_0`.

Table 4 shows sample compression factors for bytecodes, and for various components of bytecodes. As you can see, we get substantially better compression factors for a sequence of opcodes than for a sequence of bytecodes. In some unusual cases, such as mpegaudio, we get absolutely incredible compression ratios. The other sequences don't always compress as well, but the overall effect is a substantial win.

7.1 Approximate Stack State

I also performed a calculation of the current stack state (a computation of the number and types of values on the stack before executing each instruction). This stack information was used to collapse opcodes. For example, if we know the type of the element on the top of the stack, we can collapse all four addition opcodes into the `iadd` instruction, and regenerate the correct opcode upon decompression. No backwards branches were considered, and I only remembered the stack state over one forward branch at any one time (because the decompressor has to duplicate this computation, it would be impossible to consider backward branches). Because of these limitations, the calculation was an approximation: sometimes, the system would not know what state the stack was in. The improvements realized by this optimization are modest, as seen in Table 4, but not expensive to computing while compressing or decompressing. Computing the stack information is also useful in compression references (§5). I have incorporated this optimization into my baseline results.

7.2 Using Custom Opcodes

I tried a custom opcode approach to compressing JVM opcodes [EEF+97, FP95]. The program looked for pairs of adjacent opcodes, that, if replaced by new opcode, would most reduce the estimated length of the program, where an opcode that occurred with frequency p was expected to require $\log_2(1/p)$ bits. It also considered skip-pairs, that allowed for a slot between the two opcodes being combined. After each new opcode was introduced, the frequencies were recalculated.

Although this approach substantially decreased number of opcodes, gzipping the resulting sequence of opcodes gave a result that was only about slightly better than gzipping the original sequence of opcodes (see "Custom opcodes" in Table 4). As implemented, computing the custom opcodes was relatively expensive, but was very inexpensive to decompress. However, given the meager improvements, I decided not to incorporate this technology in the results reported here. Using custom opcodes may be an attractive in situations where gzip compression is not being used (because it is not available on the client or it is too expensive to run on the client).

8 Compressing Sets of Strings

The zlib compression algorithm works very well on text, and so we correctly expect that it would work well on a list of strings. However, because strings make up a substantial portion of the information in Java class file (even once we have factored out information like class names and package names), it is important to do as well as possible.

Our approach to handling strings is similar to that for objects in general. The first time a string is encounted, we encode a special index to indicate a value not seen before, and we write the Unicode string using the UTF encoding. Different categories of strings (e.g.,

Option	% of size of jar file of gzip'd classfiles javac	mpegaudio
Standard	22%	37%
Packed Separately	52%	56%
Not gzip'd	49%	99%
Packed Separately and not gzip'd	87%	118%

Table 5: Effects of separate packing and not gzipping

string constants or method names) are put into seperate streams. Strings lengths are written to a separate stream than the Unicode characters (mixing the two degrades compression). When a string is encounted again, we encode a reference to it using the scheme used for objects in general, as discussed in Section 5 (e.g., the index into a move-to-front queue or a fixed-id).

9 Other issues

One reason that my packed format is more compact is that multiple class files are combined into a single packed format that shares information. If each class file were packed separately, the total amount of data that needs to be communicated increases. Another question is how much of the compression in my packed format is due to gzip, and how much is because of the more compact encoding. On normal classfiles, gzip provides a compression factor of about 2. These effects of combining classfiles and using gzip are broken out in Table 5. Not using gzip may be appropriate on very lightweight clients where running zip is impossible or too expensive.

There is one issue we must be careful about when decompressing an archive. Normally, when we need to create a reference to a constant pool entry in a reconstructed classfile, we can just assign the element referenced to any free slot in the constant pool. However, the bytecode LDC instruction can only encode an index in the range 1-255. These instructions can only reference integer, float and string constants.

The first fix is to assign integer, float and string constant pool entries the smallest available index. Other constant pool entries are assigned in the largest available index; we transmit the total number of constant pool entries required as part of are encoding.

This almost fixes the problem. However, if there are more than 255 integer, float and string constants referenced in a classfile, which ones are assigned small indices? We would like to ensure that the same set of constants is assigned small indices as in the original classfile; otherwise, we would have to change some LDC instructions to LDC_W instructions, which are of different sizes. This would then require patching all jump offsets that traversed the changed instruction.

Instead, if a integer, float or string constant is referenced with a LDC_W instruction, then it is assigned a high constant pool index; if it is referenced with a LDC instruction, it is assigned a low constant pool index. This assumes that a classfile doesn't reference the same constant pool entry with both a LDC and a LDC_W instruction. It would be inefficient to do so, and can be fixed (and made more efficient) when the classfile is encoded if necessary.

This almost fixes the problem, except that a integer, float or string constant can also be referenced as a constant value for a field. We use an additional bit in the access flags for a field to encode whether a constant value int/float/string should be assigned a high index.

10 Evaluation

I report my compression results in Table 6. I used the move-to-front with transients and context scheme for references and used calculated stack state to collapse JVM opcodes.

I report the size of jar files, j0r.gz files (jar files without individual classfile compression but with overall zlib compression), Jazz files [BHV98] (Section 13.1) and the archives produced by the techniques described in this paper.

I also report, in the archives I produce, how much space is occupied by strings (string constants, class and method names, ...), opcodes, integers, references and other (including floating point constants, branch offsets and registers). As you can see, no one element dominates, so obtaining substantial additional reduction in archive size would likely require substantial reductions in all elements.

10.1 Execution time

We timed the execution speed of both the compression routine and the decompression routine. In the decompressor, we just computed the time required to build each classfile internally; we did not include the time required to store the class files into a jar file. Thus, these times would be appropriate for an application using eager class loading (Section 11).

The decompressor can decompress about 75-120 Kbytes of wire-format classfiles per second (which would expand into a substantially larger collection of classfiles). This is on a Sun Ultra 5 workstation with a 333Mhz processor using the Sun Solaris production JVM, version 1.2fcs, which achieves a JVM98 Specmark of 16.6.

The compressor is about 15 times slower than the decompressor, but at the moment it still contains a fair bit of code for generating statistics and is a very general purpose compressor (i.e., can implement many differ-

Benchmark	Size in KBytes jar	j0r.gz	Jazz	Packed	Size as % of jar format j0r.gz	Jazz	Packed	Size as % of packed format Strings	Opcodes	Ints	Refs	Misc
209_db	6	5	4	3	84%	66%	49%	34%	28%	9%	17%	13%
201_compress	10	6	4	3	59%	41%	29%	29%	32%	14%	17%	8%
Hanoi_jax	21	16	12	7	74%	58%	32%	21%	30%	13%	27%	9%
205_raytrace	24	15	12	7	64%	50%	30%	20%	33%	9%	22%	16%
Hanoi_big	30	20	15	9	67%	52%	29%	25%	27%	14%	26%	8%
Hanoi	46	31	23	13	67%	49%	29%	22%	29%	12%	29%	8%
228_jack	55	36	30	17	65%	55%	30%	32%	21%	14%	21%	11%
222_mpegaudio	62	45	34	23	73%	54%	37%	9%	24%	37%	12%	18%
icebrowserbean	116	88	80	39	76%	69%	34%	21%	31%	11%	26%	12%
javafig_dashO	131	113	102	53	86%	78%	41%	23%	28%	8%	29%	12%
202_jess	136	64	42	23	47%	31%	17%	23%	28%	12%	26%	11%
javafig	170	143	122	64	84%	71%	38%	28%	26%	8%	27%	11%
jmark20	173	91	86	35	53%	50%	20%	22%	25%	13%	28%	12%
213_javac	226	143	90	50	63%	40%	22%	18%	29%	15%	27%	11%
ImageEditor	257	162	123	64	63%	48%	25%	22%	28%	16%	24%	10%
tools	737	513	477	204	70%	65%	28%	26%	27%	10%	27%	11%
visaj	1,157	703	691	238	61%	60%	21%	23%	26%	12%	31%	8%
swingall	1,657	998	887	338	60%	54%	20%	19%	28%	13%	31%	9%
rt	4,652	2,820	8,435	1,069	61%	181%	23%	22%	28%	13%	27%	10%

jar Size of jar file with individual class files stripped of debugging information and compressed
j0r.gz Size of gzip of jar file with class files stripped of debugging information and but not compressed
Jazz Size of Jazz archive [BHV98] (See Section 13.1)
Packed Size of archive produced by techniques in this paper

Table 6: Compression ratios

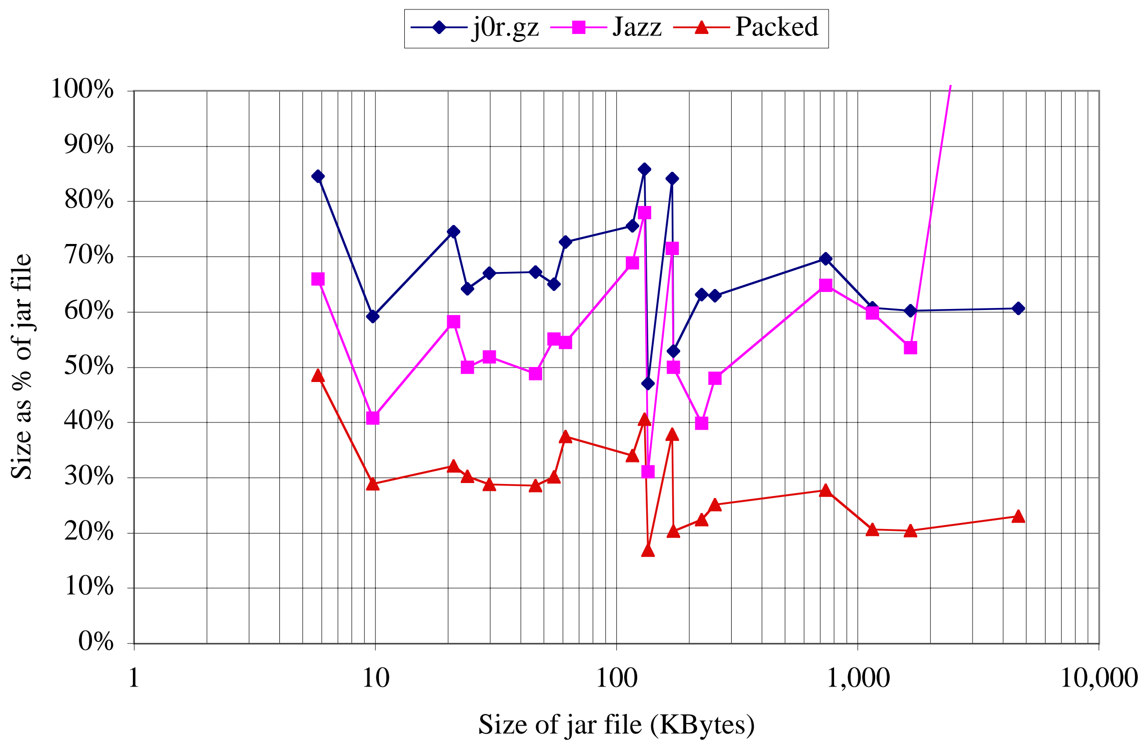

Figure 2: Graph of compression ratios

File	Compression time (secs)	Decompression time (secs)	Kbytes/sec
rt	170.62	8.58	125
swingall	54.97	2.92	116
tools	27.61	1.51	135
icebrowserbean	4.72	0.31	127
jmark20	5.59	0.42	83
visaj	36.5	2.07	115
ImageEditor	7.47	0.53	122
Hanoi	1.74	0.13	101
Hanoi_big	1.07	0.10	86
Hanoi_jax	0.80	0.09	75
javafig	8.72	0.41	157
javafig_dashO	7.16	0.49	109
201_compress	0.25	0.09	31
202_jess	3.18	0.24	95
205_raytrace	1.02	0.13	56
209_db	0.27	0.09	31
213_javac	6.56	0.39	129
222_mpegaudio	2.42	0.19	121
228_jack	2.50	0.18	92

Table 7: Execution times

ent compression schemes). A rewrite of the compressor should provide substantial speed improvements.

The decompressor is 36 Kbytes in jar format, so if the decompressor had to be downloaded along with a packed archived, it would only be advantageous for larger archives. The decompressor is 23 Kbytes in j0r.gz format, so it could be loaded by first downloading (a tiny) a classloader than understood j0r.gz archives, and then downloading the decompressor. If the decompressor were installed as an standard extension, then this wouldn't be an issue and would be fine for even very small archives.

11 Eager class loading

Normally, Java classfiles are loaded on demand. Particular when each classfile is loaded via a separate file or net connection, this can be a huge win. However, when classfiles are loaded out of an archive (a jar file or other Java classfile archive) that is downloaded over the net, it is a more dubious idea. To allow on-demand loading, the archive must be cached on disk or in memory. If it is cached on disk, that bytes forming the archive may need to be copied in memory several times (with a good and large file system cache, they probably won't need to be retrieved from disk).

In Sun's (Sparc Solaris 1.2) implementation of classloading from downloaded jar files, no entries can be examined until the jar directory is downloaded, which is at the very end of the jar file. The jar file is cached on disk and kept open until the virtual machine shuts down. It might be possible to fix some of these implementation issues, so that entries could be accessed once they have arrived, and the jar file would be deleted once the classloader than opened it was unreachable. But the archive would still have to be kept cached while classes were still being loaded, and classes would likely be copied in memory several times before being loaded.

An alternative approach would be *eager class loading* – to load classes into the JVM as soon as they arrive (i.e., invoke `java.lang.Classloader.defineClass(...)`), without buffering them or waiting for the entire archive to arrive. This allows us quicker access to some of the class files in the archive, and eliminates the need to cache or buffer a copy of the jar file.

If this resulted in loading many classes that were not needed, it might result in increased resource costs or performance problems. But this is already an issue for Java archive. If you are going to download a large archive over a network for direct execution, you already want to make sure that most of the classes will be actually used. Otherwise, you will pay the transmission costs for classfiles you won't use. There are several approaches to increasing the percentage of classfiles that are actually used. A tool such as JAX may be used to eliminate from third-party libraries classfiles that cannot be loaded by the application being distributed. Profiling [KCLZ98] could be used to determine a desirable order for classes. You could also break up packages into separate archives, and have rarely used classfiles loaded separately.

The eager class loading approach works with a standard jar archive, as well with the packed format. Note that before a class `X` can be loaded, the superclass of `X` and all interfaces implemented by `X` must be loaded. If the request to load `X` is done in a thread separate than the one which is handling the download and spawning of threads to load classes, the system won't deadlock, but it also won't be efficient. Instead, we should make sure that the superclass of `X` and the interfaces implemented by `X` appear in the archive before `X`.

12 Jar functionality

Java jar files provide functionality beyond just being an archive of class files. In particular, jar files can contain non-class files (gif images, property files) and the jar manifest, which contains information such as digital signatures.

The basic solution to this is to combine a packed java archive with a standard jar file that contains all of the non-class files from the jar archive being emulated. One issue that needs to be dealt with is that compressing and decompressing a Java classfile using

Paper	% of Gzip'd classfiles
Slim Binaries [KF97, KF, Fra97]	59
JShrink, DashO, and Jax	65 – 83
jar.gz format (§2.1)	55 – 85
Clazz format [HC98]	52 – 90
Jazz format [BHV98]	40 – 70
This paper (on programs > 10K bytes)	17 – 41

Table 8: Results on wire-code program compression in related work

the format proposed in this paper will likely modify the classfile by renumbering the constant pool. Thus, any signatures for the original classfiles will be invalid for the decomposed classfiles. However, the decompression is deterministic: decompression of a packed archive will always result the identical set of classfile. Therefore, if you wish to sign your classfiles, you must use the following approach: compress the classfiles, and then decompress the classfiles. Sign the decompressed classfiles, and ship the signed manifest from the decompressed classfiles along with the packed archive.

13 Related Work

Many papers have argued and discussed different aspects of machine independent intermediate forms, such as their suitability for run-time optimization. Given the wide-spread use of Java, my goal was to develop a compact wire-code format for Java, without regard to the merits or problems of the Java virtual machine. A comparison of quoted compression results from related work is provided in Table 8.

Ernst et al. [EEF+97] discussed two kinds of code compression: wire-code and directly executable. The technique I have proposed is a wire code, so I will limit myself to comparisons with the wire-code described by Ernst et al. [EEF+97]. Ernst et al. consider only code segments of full executables, and thus don't deal with significant amounts of symbolic linkage information nor data such as strings and floating point constants. They use the `lcc` intermediate form [FH95], which is a tree based format. It has been suggested that a tree based intermediate form is more suited to compression [KF97]. Despite these differences, our basic approach is very similar: break out dissimilar objects into different streams which are compressed separately.

Michael Franz has proposed Slim Binaries [KF97, KF, Fra97] as a mechanism for distribution of compact, machine independent programs. It is based on an encoding of the abstract syntax tree and symbol table of a program. The papers on Slim Binaries do not give details of the encoding, but do give limited experimental results (Table 8).

Lars Raeder Clausen et al. [CSCM98] describe a method of factoring Java classfiles by finding custom macros or opcodes, similar to the techniques described in [EEF+97]. They focus on embedded systems with small amounts of memory, and focus on reducing the size of bytescodes for loaded classfiles. Their techniques reduce the size of the bytecodes to about 70% – 80% of their original uncompressed size and require modifications to the JVM.

Normally, an entire class file must be transmitted before a class loader will start to process it, and an entire jar file is transfered before it is used by a class loader. Krintz et al. [KCLZ98] describe methods that determine (based on profile information) which classes and methods are likely to be needed first, transmit the data needed for these classes and methods first, and allow execution to start while the remaining information about classes and methods is still being transferred. Invoking a method which hasn't arrived yet blocks until the method arrives. I have incorporated parts of this idea in my provision for eager class loading (§11). Intermingling different classes could change the effectiveness of compression (since there would likely not be as much locality of reference).

Tools such as Jax [LTS], Dash-O and JShrink perform shrinking and obfuscation by renaming classes, methods and fields to have short, meaningless names and stripping out debugging information. The Jax tool (and perhaps the others) performs transformations such as removing methods that are never called and merging a class into its superclass when it can prove that such a transformation doesn't effect the semantics of the program. These tools typically give reductions of 17% - 32% of the classfile size. Surprisingly, the techniques of Section 2, applied after applying DashO, gave an additional reduction of 2% in the size of the resulting compressed jar file. The apparent reason is the DashO doesn't sort the constant pool, leading to poorer compression. Transformations applied by these tools could be usefully combined with the techniques in this paper to provide greater compression than either technique alone. Tools such as Jax are particularlly using when an application uses a small portion of a library that is not installed on most clients. By extracting just the used portion of the library, the potential savings are unbounded.

13.1 Jazz compression

The Jazz format [BHV98] is also a custom compressed format for collections of Java class files. In that regard, it is very similar to the work described in this paper. However, the Jazz format does not achieve as good compression rations as the work described here. The Jazz format is a less radical format. It retains the exist-

ing kinds of Constant Pool entries, although it uses a global constant pool, sharing them across classfiles. But it doesn't do the factoring my work does, which eliminates the repetition of package names in classnames and of classnames in signatures. Also, it uses a fixed Huffman encoding indices for each kind of constant pool entry, that doesn't take locality of reference into account.

The Clazz format described by Horspool and Corless [HC98] was a predecessor of the Jazz format. While there are a number of similarities, the Clazz format is applied to individual classfiles in isolation, and therefore does not achieve a high compression as Jazz or the compression techniques described in this paper.

14 Conclusion

The Java classfile format is rather fluffy and it should come as no great surprise that a different format could lead to smaller files, particularly when information duplicated across multiple class files is combined. On the other hand, a good compression algorithm can work wonders, and a more efficient format with less redundant information will often not compress as well. So the amount of additional compression available over gzip'd classfiles was not obvious. As it turns out, we can obtain compression factors of 2-5 over individually gzip'd classfiles, which will make an important difference in mobile and other low bandwidth applications.

We have been making the assumption that for each kind of data, one particular encoding scheme is optimal. Of course, this isn't the case: different schemes will work better with different benchmarks. To achieve even better compression, the compression stage could try several encoding methods of each kind of data, and select the one that happens to work best. The encoded data would include a description of the encoding mechanism used for each data sequence, and would not be substantially harder to decode than if a fixed policy was used for each kind of data.

There are a number of other approaches that might give minor performance improvements. The only change I can think of that would likely give non-trivial improvements would be assume a standard set of preloaded references to frequently used package names, classes, method references and so on. It actually isn't guaranteed that this would improve compression (preloaded references that were never used would degrade compression), but I expect it would help on small archives. This would also likely increase the size of the decompressor, so in the situations where the decompressor is not pre-installed, there would not be any net benefit.

As a research tool, the goal is to get as much compression as possible. However, as a tool that might be widely distributed and reimplemented, it might be better to have a specification of the packed format that is simple and clear. It may be appropriate to simplify the format by, for example, dropping approximate stack state (§7.1).

I expect that an implementation will be available for download from http://www.cs.umd.edu/∼pugh by the date of the conference.

15 Acknowledgments

Thanks to the referees and others who provided me with feedback about the paper. Special thanks to Quetzalcoatl Bradley, R. Nigel Horspool and Jan Vitek, who provided me with an implementation of Jazz [BHV98] so that I could do a proper comparison of my work with Jazz.

References

[BHV98] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. Jazz: An efficient compressed format for java archive files. In *Proceedings of CASCON'98*, November 1998.

[CSCM98] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for embedded systems. Technical Report 1213, Irisa, December 1998.

[EEF+97] Jens Ernst, William Evans, Christopher Fraser, Steven Lucco, and Todd Proebsting. Code compression. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley Longman, 1995.

[FP95] Christopher Fraser and Todd Proebsting. Custom instruction sets for code compression. www.research.microsoft.com/~toddpro/papers/pldi2.ps, October 1995.

[Fra97] Michael Franz. *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 263–276. Springer Lecture Notes in Computer Science, 1997.

[HC98] R. Nigel Horspool and Jason Corless. Tailored compression of java class files. *Software – Practice and Experience*, 28(12):1253–1268, October 1998.

[KCLZ98] Chandra Krintz, Brad Calder, Han Bok Lee, and Benjamin Zorn. Overlapping execution with transfer using non-strict execution for mobile programs. In *Eighth SIAM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[KF] Thomas Kistler and Michael Franz. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming*. To appear.

[KF97] Thomas Kistler and Michael Franz. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[LTS] Chris Laffra, Frank Tip, and Pete Sweeny. Jax – the Java Application eXtractor. www.alphaWorks.ibm.com/formula/JAX.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commincations of the ACM*, 33(6):668–676, June 1990.