JAZZ: An Efficient Compressed Format for Java Archive Files

Quetzalcoatl Bradley, R. Nigel Horspool and Jan Vitek

Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, BC
Canada V8W 3P6

Centre Universitaire d'Informatique
Université de Genève
24 rue Général-Dufour
CH-1211 Geneva, Switzerland

Abstract

The Jazz file format is intended to be a replacement for the JAR file format when used for storage and distribution of Java programs. A Jazz file is compressed to a degree that far exceeds what is possible with a JAR file. The smaller size of the Jazz format permits faster transmission speeds over a network and has the additional benefit of conserving disk storage. The compression is achieved as a combination of different data compression methods, adapted to suit the characteristics of collections of Java class files.

1 Introduction

A typical Java application program consists of numerous small class files. Distribution of the program over a network requires a suitable means of shipping and storing the class files as a single unit. The official approach is to use Sun's JAR format. The JAR format combines the class files, protecting them from file systems with length limitations, and optionally compressing the individual members. The JAR format also allows other types of data to be bundled with the class files.

Unfortunately, class files are not easy to compress and JAR does not achieve competitive compression results. We have an approach that achieves great compression at the cost of making single member extraction more difficult. Our implementation, called *Jazz*, is intended to be a replacement for JAR in the domain of software distribution, archiving class files, and network distribution of class files where bandwidth is limited.

2 JAR files and Compression

JAR files are the standard means of packaging Java class files for distribution and storage. The JAR file format is based on and almost identical to the Zip format [6]. In fact, it is often possible to extract members from JAR files using Zip decoders. A JAR file contains a directory and the member files. A member file is, optionally, in a compressed format. A constituent file is compressed separately from the other members, and that allows for rapid retrieval of the file. However, it limits the possible gains that would be achieved by compressing several member files together. This is a particular problem if the files tend to be small, such as is normally the case with Java class files.

2.1 Class File Structure

Class files are usually quite small. They have a format [4] which structures the contents into many, even smaller, sections. Each section is likely to contain types of data very different from those in surrounding sections. The compression method used in Zip file and JAR files tries to take advantage of repetitive patterns in a file in order to get good results. The fractured structure of Java class files makes it extremely difficult for Zip or JAR to find the kinds of redundancy necessary for good compression. Redundancy does exist in a class file. However, Zip and other general purpose

compression algorithms based on similar principles are ill suited to take advantage of it. Jazz takes full advantage of knowledge of the structure of class files to identify redundancy, making selective use of existing compression algorithms and custom coding techniques to compress class files efficiently.

The largest component of a class file is typically the constant pool. It is not unusual for the constant pool to occupy more than 50% of the class file. The Java constant pool contains all the information needed for run-time linking of Java class files. As its name implies, it also contains constant data used by the Java bytecode instructions. Within the constant pool, the largest portion usually consists of text strings. Despite the size of the constant pool, its structure is broken up into small sections containing type information, indexes to other constant pool entries, and the constant data itself. Furthermore, the constant pool entries are in no particular order, hampering the performance of general purpose compression algorithms.

The remainder of the class file consists of methods, fields, and attributes. Except for the code attribute, most of a class file consists of constant pool indices, counts, and structure information. The code attribute contains the bytecode, the instructions executed by the Java Virtual Machine (JVM). It can occupy a significant fraction of a class file. Java instructions are variable length and have several different formats, making them difficult to compress.

2.2 Zip Compression

The compression algorithm used in Zip is based on LZSS. It works by checking its input for a repeated sequence of bytes and replacing it with a pointer to the previous occurrence. It is an algorithm that adapts to the data it is compressing. This allows it to adapt to local changes in a file. In a typical executable file, the characteristics of the file change considerably at the boundary between the code portion and the data section. Zip would adapt to such a change. However, if changes occur frequently, Zip would not have time to adapt from one to the other before it changes again, limiting the effectiveness of the compression algorithm.

3 The Jazz Approach

The goal of Jazz is to provide all of the features and functionality of the JAR format, while attaining the best compression possible in a reasonable amount of time. Like JAR, Jazz allows a number of class files to be bundled together and compressed. Later, a class file can be extracted and uncompressed. This class file can be stored as a separate file or put into a JAR file. Unlike JAR, Jazz always compresses the class files.

In order to achieve good compression of Java class files, the following strategies are taken.

1. Huffman codes are used for constant pool indices.
2. A unified constant pool is used for all classes in the Jazz archive.
3. Strings, opcodes, and arbitrary data are compressed with Zip.
4. Start-step-stop codes [1] are used for instruction offsets and string lengths; they are also used to encode the tables of Huffman codes.
5. Redundant constant pool entries are eliminated.

Figure 1 shows the rearrangement of information that takes place when a Jazz file is created from two class files.

3.1 Huffman Codes

Huffman coding [1] [5] is an optimal method of assigning variable length codes to symbols, where the symbols occur independently and randomly with known probabilities. Jazz uses Huffman codes for all indexes into the constant pool.

The constant pool contains more than one type of information and one Huffman table is created for each type of constant. During encoding, a count is made of the number of references to each unique constant in all classes. This frequency information is used to construct the Huffman table which maps constants to variable length codes and vice versa. This table is stored in an efficient form in the Jazz file. Any time a reference to a constant pool entry is needed, the appropriate variable length Huffman code is used.

With one exception, the appropriate Huffman table to use is always known from context and an identification of the kind of constant does not need to be stored in the compressed file. That exception

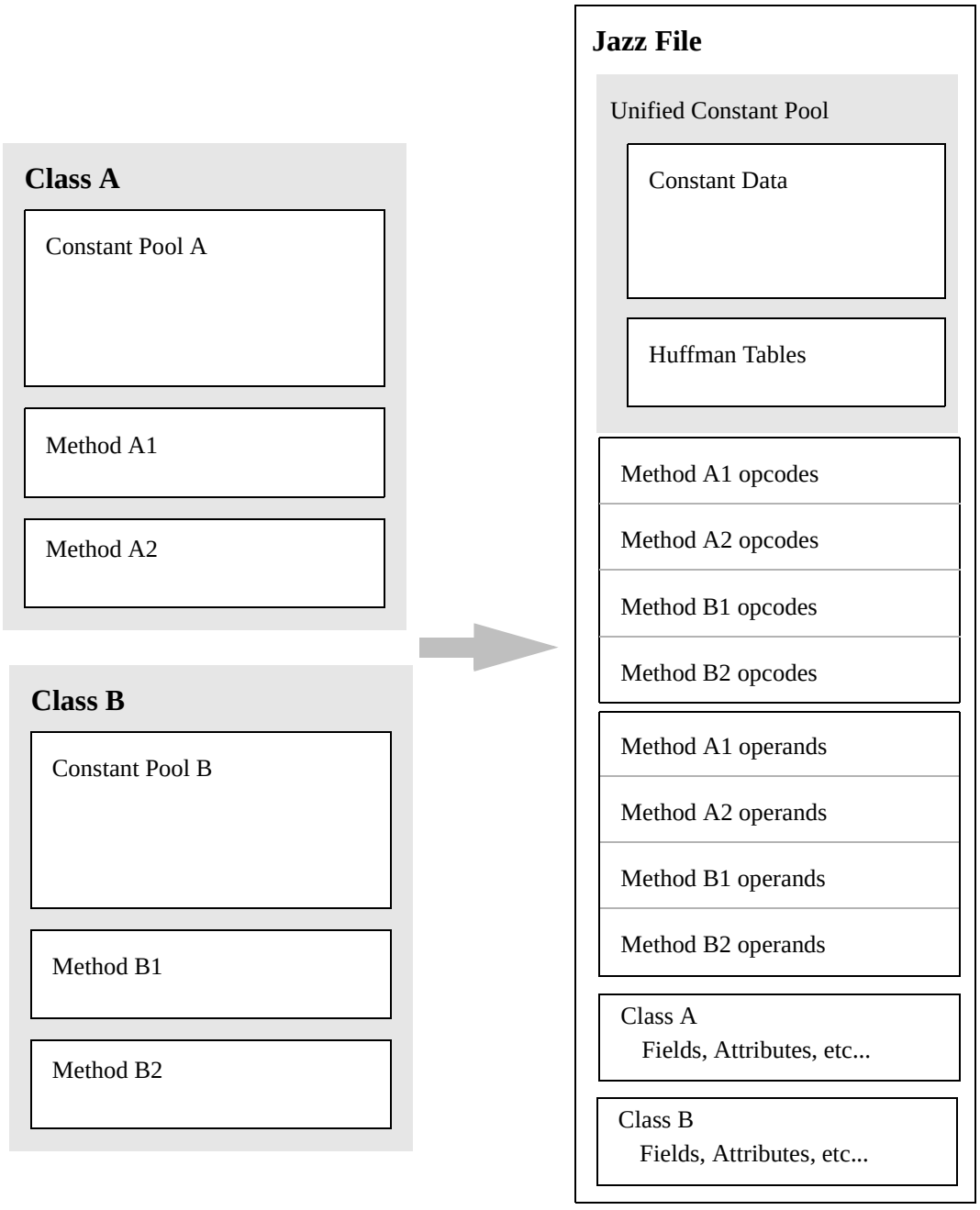

Figure 1: Jazz File Format

occurs with the LDC (load constant) instruction which has a parameter that refers to a constant pool entry that can be one of a number of different types of constant. In this case it is necessary to output 1 or 2 extra bits to indicate the appropriate table.

The advantage of using a Huffman code for constant pool indices is a constant pool pointer size that is much less, on average, than the normal 8 or 16 bits. This is a consequence of the fact that some constant pool entries are referenced frequently, while others might be referenced only rarely. The savings are offset by the need to store the decoding table in the Jazz file. In practice, the savings are significant.

3.2 Unified Constant Pool

Any Java class file must store in its constant pool the name of any class referenced by that class file. The name and signature of any method call also needs to be stored. Java has a rich set of class libraries that are used by almost every class file in one capacity or another. This causes exceptional amounts of redundancy between class files, exacerbated by the fact that every class is contained in its own individual file. Any reasonably complicated part of a program will consist of many class files but they may be working on similar problems. As a result, many related class files will contain constant pools with references to the same classes and methods. Jazz capitalizes on this by combining the constant pools of all of the classes that are compressed. In this way, any given constant string will only appear once in a Jazz file, regardless of the number of classes that make use of it. The same is true for method names, signatures, integer constants, and so on. The unified constant pool is normally larger than the constant pool for any class file in the Jazz archive. That would normally cause an index into the combined pool to require more bits than an index into the constant pool for the original class file. However, the reduction of duplication between the original constant pools results in a net gain, as our experimental results confirm.

Two kinds of constant pool entries are completely eliminated in the Jazz format. These are the CLASS, and STRING constant pool entries. They consist solely of a constant pool pointer to a UTF8 string and serve only to differentiate between CLASS and STRING where necessary. The distinction is not needed when extracting a class file from a Jazz archive.

3.3 Strings, opcodes, and data

Strings make up a large proportion of any class file. Class names, method names, signatures, and the class file attribute names are stored as strings in the constant pool, in addition to the constant strings used by the methods themselves. Zip uses a general purpose compression technique that is particular effective at compressing repetitive text.

To achieve good compression, all strings in all the class files are concatenated, compressed using Zip, and stored in one part of the Jazz file. The lengths of the strings also need to be stored. To make this efficient, the strings are sorted by length and delta coding used to encode the string lengths. A delta code is the difference between one value and the next. By sorting the strings on length, the values will usually be small and will require fewer bits to store. Start-step-stop codes are used to reduce the number of bits even further.

All Java opcodes are one byte. Compressing them is typically difficult. However, there are sometimes short sequences of instructions that occur frequently, such as pushing an operand before invoking a method call. Finding and compressing such sequences is precisely what Zip and other dictionary-based compression algorithms handle well. Unfortunately Zip does not perform well on unprocessed bytecode because the operands are mixed together with the opcodes, obscuring the relationship between adjacent instructions. Therefore, the opcodes and their operands are separated. Then the opcodes from all methods in all classes are combined together and compressed with Zip. The operands are processed individually because different encoding schemes are appropriate for different kinds of operands.

Some class file data is difficult to compress using anything other than a general technique. Such data includes integer constants, floating point constants, and the operands for some opcodes. These are combined together into a block and compressed with Zip.

3.4 Instruction Format

Instructions are variable length, ranging from one byte upward. All instructions are in the form of a one byte opcode followed by zero or more bytes of parameters. The size of each instruction is fixed except for TABLESWITCH, LOOKUPSWITCH, and WIDE. The parameters can have any one of a number of different types. A parameter could be a constant pool index, a constant value, a local variable table offset, a bytecode offset for branches, another opcode, or padding. Each case must be handled separately. Constant pool indices are encoded as variable length Huffman codes, as described above. Constant values are compressed with Zip. Local variable table offsets and bytecode offsets are encoded with Start-step-stop codes on the grounds that most such offsets are small. Padding bytes can be safely discarded.

Branch offsets are measured in bytes relative to the address of the instruction they are associated with. However, branch offsets are only valid if they branch to the beginning of an instruction. Therefore the byte offsets are first converted to instruction count offsets before being encoded with a start-step-stop code.

3.5 Preserving Java Semantics

The most important restriction on the Jazz format is that a class file which has been compressed and then decompressed must execute in exactly the same way as the original class file. It is not necessary for the decompressed class file to be byte for byte identical as long as the semantics are preserved. Although no formal proof is offered here, it is not difficult to show that Jazz produces class files which preserve all semantics except in the case of bugs in Jazz or future changes to the Java class file format.

To date there have been no changes to the Java class file format since Java 1.0, despite changes to the Java Language Specification. Java 1.1 saw the addition of Inner Classes to the Java Language Specification. This addition was implemented at the Java Class file format level using existing structures. The addition of new attributes is also not a problem for Jazz, which simply copies them byte for byte. The only likely change to the Java class file format that would break Jazz would be the addition of new instruction opcodes. In that case, Jazz would report an error rather than generating incorrect output.

Assuming that the Java class file format is stable, Jazz produces class files which are nearly the same as the original. The only possible differences are the order of items in the constant pool, the removal of redundant constant pool entries, and instructions changing between long and short formats because of the renumbered constant pool entries.

The relative position of constant pool entries is completely irrelevant. All constant pool references use absolute indexes. Constant pool entries which reference other constant pool entries also use absolute indexes. As long as these indexes resolve to the correct values, the position of each constant pool entry is not important.

The removal of duplicate constant pool entries is also not a problem. Constant pool entries are considered duplicate if they have exactly the same values, except for constant pool indexes which must resolve to constant pool entries with the same values recursively. Since the actual indexes are not relevant to the semantics of a Java class file, the fact that two references to the constant pool have different or same indexes does not matter as long as the values they resolve to are exactly the same.

Some Java instructions have two forms, one with an 8 bit index into the constant pool and one with a 16 bit index. The only difference between the short form and the long form is the size of the constant pool index which is interpreted in the same way in both cases. Changing from the short form to the long form or vice versa can have no effect on the semantics of the class file as long as branches are properly updated to reflect the new size of the instruction. There is a very small possibility that the resized instructions will cause a 16-bit branch at the extreme end of its range to not be able to reach its destination. While technically possible, it is unlikely. Java methods have a maximum size of 2^{16} bytes while the smallest relative branch is a 16-bit signed offset. Typical Java methods are very small. Therefore, the current implementation reports an error if a branch is out of range. It would be possible to implement a scheme that renumbered the constant pool entries in such a case if it was found to be necessary.

4 Experimental Results

Our prototype of Jazz is implemented in Java. Therefore one method that we could use to verify the functional equivalence of Jazz's input and output was to use Jazz to compress itself, and then uncompress the resulting Jazz file. This produced a number of class files that differed from those originally produced by the Java compiler. However, when these class files were executed, they produced identical output to the original versions. Therefore, our confidence in the implementation was increased.

4.1 Collections of class files

In the tests below, Jazz is compared with the archive files produced by the commands shown below.

- JAR file, uncompressed
  `jar cvf0 output1.jar` *classfiles*
- JAR file, compressed
  `jar cvf output2.jar` *classfiles*
- Gzip
  `cat` *classfiles* `| gzip -9 > out.gz`
- Clazz
  `clazz` *classfiles*

Uncompressed JAR files are used as a standard with which to compare the efficiency of compression of the other methods. Compressed JAR files show the efficiency of the JAR format. The Gzip results will be slightly optimistic, since concatenating the files and applying gzip to them does not permit the original files to be extracted again very easily. Clazz is the method developed in [2] and [3] but it only operates on individual class files. In this section, the results for clazz are computed from the total size of all of the class files, compressed individually.

Tests were performed on four collections of class files. These collections were as follows.

- The Jazz class files, compiled with debugging information using Metrowerks 3.0.1.
- The Jazz class files, compiled with Sun JDK 1.1.4 on Solaris with optimization enabled.
- The class files contained in `icebrowserbean.jar`, a collection of Java class files available on the internet.
- The class files that comprise the `java.lang` hierarchy from Sun JDK 1.1.4.

The test results are shown in Tables 1, 2, 3, and 4.

Table 1: Metrowerks 3.0.1 Class Files with Debugging Information

File Format	Size	% orig. size
JAR file uncompressed	198,191	100.0%
JAR file compressed	101,154	51.0%
Gzip	75,371	38.0%
Clazz	73,778	38.0%
Jazz	48,279	24.4%

Table 2: Sun JDK 1.1.4 Class Files

File Format	Size	% orig. size
JAR file, uncompressed	122,871	100.0%
JAR file compressed	61,312	49.9%
Clazz	44,290	36.0%
Gzip	36,020	29.3%
Jazz	31,771	25.9%

Table 3: Class Files from icebrowserbean.jar

File Format	Size	% orig. size
JAR file, uncompressed	260,178	100.0%
JAR file, compressed	132,600	51.0%
Clazz	97,341	37.4%
Gzip	97,223	37.4%
Jazz	59,321	22.8%

Table 4: Class Files from java.lang

File Format	Size	% orig. size
JAR file, uncompressed	193,788	100.0%
JAR file, compressed	93,418	48.2%
Clazz	77,641	77.6%
Gzip	69,100	35.7%
Jazz	55,408	28.6%

In summary, Jazz performed 14.6% better than the second best method in the best case, 3.4% better in the worst case, and 9.7% better on average for these four examples. In round numbers, the Zip compression used by JAR files reduces the data to one half of its original size. The Jazz approach reduces the data to one quarter of its original size.

4.2 Compression of Single Class Files

Although Jazz is optimized for compressing collections of class files, it can also be used to compress class files individually. We can compare its results to other methods. The JAR format is not really relevant to distributing individual class files, so the baseline comparison is to the actual class file itself. In all cases, gzip, clazz, and Jazz were run individually on one class file. The results were averaged over all the class files in a collection, and the results reported below. In each case, the average Jazz result was compared to the average class file size when all classes were compressed at once with Jazz. This indicates the amount of benefit gained by unifying the constant pool and combining opcodes and strings from all classes before compressing them.

The results of our two tests are summarized in Tables 5 and 6. It is clear that Jazz achieves better results than the competing methods, even when applied to individual class files. The additional benefit of applying Jazz to a collection of files is also apparent.

5 Conclusions and Future Work

Jazz already yields highly satisfactory results as currently implemented. Reducing the size of a Java class file to one quarter of its original size while maintaining full compatibility with the Java virtual machine is a significant achievement. There are, of course, a few areas that show some promise for improved performance.

Table 5: Compressing Individual Class Files from icebrowserbean.jar

File Format	Size	% orig. size
original class file	3,197	100.0%
Gzip	1,618	50.6%
clazz	1,315	41.1%
Jazz, individually	1,064	33.3%
Jazz, all together	802	25.1%

Table 6: Compressing Individual Class Files from java.lang

File Format	Size	% orig. size
original class file	2,275	100.0%
Gzip	1,094	48.1%
clazz	1,024	45.0%
Jazz, individually	879	38.6%
Jazz, all together	684	30.1%

5.1 Complete JAR Functionality

Jazz is intended to be fully compatible with JAR. However, as of this writing the implementation does not handle files other than class files. Adding support for these is a high priority and presents no technical challenges.

It is often desirable to execute Java class files directly from a JAR file without decompressing them first. This ability is used to get around filename length restrictions in some operating systems, or to avoid cluttering up the file system. Jazz should have the same functionality.

We would need to write a class loader which has access to the Jazz decompressor. To achieve reasonable performance, it would likely be necessary to decompress the entire constant pool the first time a Jazz file was accessed and to cache it. When each class file is requested, it would be possible to jump directly to that class in the Jazz file and decompress it. This requires a couple of minor changes to the Jazz file format.

The first change would would be to make the class names easily accessible by putting them in order at the beginning of the constant pool. The second change would be to add a table of offsets which is in one-to-one correspondence with the class names in the constant pool. These offsets provide the byte offset into the Jazz file where the class-specific information resides. The extra overhead would be only 4 bytes per class, at the most.

5.2 Shared Dictionaries

Some of the largest improvements in Jazz come from the unified constant pool. It might be possible to take this a step further and have shared dictionaries. The idea would be to make a "constant pool" consisting of class names, method names, and signatures from the standard libraries. This shared dictionary could then be bundled with the encoder and decoder. The result would be that any Jazz file with a constant that also appeared in the shared dictionary could eliminate the redundant entry. The cost would be an extra bit to indicate whether a given constant was in the local dictionary or the shared dictionary. This bit would only appear in the encoded Huffman tables, so the overhead might not be that great. The idea could be extended to allow for multiple shared dictionaries. The only caveat would be that you would have to be sure that the decoder had the exact same dictionaries available as the encoder.

5.3 Implementation Improvements

The Huffman tables do not incur significant overhead for each individual table over and above the size of the indexes into the constant pool. Therefore it is desirable to partition the Huffman tables into several smaller tables if possible. The overhead is minimal and the length of each Huffman code will be shorter, on average. One area that could benefit from this is the strings in the constant pool. Some of the strings represent the names of classes, the names of methods, method signatures, and others. It is possible for one string to be used for more than one purpose. It would almost certainly be beneficial to make a separate Huffman table for class names. Another table for type signatures and yet another for method names would be less likely to be useful, but should be investigated. The overhead would increase in those cases where a string was used in more than one table, because of duplication of indexes to particular strings in the Huffman tables, but the advantage of smaller Huffman codes to the frequently referenced class names could be great.

Another potential optimization would be to eliminate the Huffman tables and use an adaptive Huffman code [1][5] instead. This would eliminate the overhead of the Huffman table while increasing the run-time cost of the algorithm. Adaptive Huffman codes yield similar compression efficiency, but the need to continuously update the Huffman table as each symbol is decoded adds an additional run-time burden. Adaptive codes based on arithmetic coding [1] are also worth considering. They would provide slightly more efficient encoding because they, in effect, generate fractional bit lengths for codes. There would be an implementation cost in the switch because the current implementation mixes start-step-stop codes and Huffman codes in the output. A consistent use of arithmetic coding throughout would probably be desirable.

Finally, reimplementing the Jazz program in C or C++ would provide a speed improvement and give a production version.

Acknowledgments

We gratefully acknowledge financial support for this work from the Natural Sciences and Engineering Research Council of Canada.

About the Authors

Quetzalcoatl Bradley is a graduate student at the University of Victoria. His research interests include distributed computing and networking, but he is interested in *everything*. His e-mail address is `qbradley@csc.uvic.ca`.

Nigel Horspool is a Professor and the Chair of the Department of Computer Science at the Uni-

versity of Victoria. His main research interests are compilers and data compression. His e-mail address is nigelh@csr.uvic.ca.

Jan Vitek is close to completing a PhD at the Université de Genève. His research interests include object-oriented programming, mobile computations and internet programming. His e-mail address is Jan.Vitek@cui.unige.ch.

References

[1] T. C. Bell, J. G. Cleary and I. H. Witten. *Text Compression*. Prentice-Hall, 1990.

[2] J. Corless. Compression of Java Class Files. M.Sc. Thesis, Dept. of Computer Science, University of Victoria, 1997.

[3] R. Nigel Horspool and Jason Corless. Tailored Compression of Java Class Files, to appear in Software – Practice and Experience (1998).

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading MA, 1997.

[5] M. Nelson and J.-L. Gailly. *The Data Compression Book*, 2nd Edition. M & T Books, New York, 1995.

[6] Info-Zip. 'General format of a Zip file', Info-Zip note 970311, URL: http://www.cdrom.com/pub/infoZip/doc/, 1997.

[7] The Zlib home page. URL: http://www.cdrom.com/pub/infoZip/zlib/, 1998.