Car: The Class Archive Format

Denis N. Antonioli

Technical Report 2001.01
January 2001

Department of Information Technology, University of Zurich
Winterthurerstrasse 190, CH-8057 Zurich, Switzerland

<antonioli@ifi.unizh.ch>

"If I eat one of these cakes," she thought, "it's sure to make some change in my size; and, as it can't possibly make me larger, it must make me smaller, I suppose."

Lewis Carroll, *Alice's adventure in wonderland*

Abstract

A Java application is built of a large number of Java class files, which are collected and compressed in Java archive (jar) files. But the jar files typically shrink original class files by only fifty percent. Various projects have pursued ever smaller class files and they achieved very impressive results, but these results come at the cost of complicated and slow transformations. The class archive (car) format is an alternative for groups of class files. Car files are between one third and two thirds of the size of the corresponding jar files and between one seventh and one half of the size of the original class files. Although there are more compact archive formats, the car tools themselves are smaller and take far less time than the alternatives. This article also introduces a method to compare wire formats. The method does not only consider the size of the archive but also the bandwidth and the decompression speed. This method demonstrates that car is a better archive format in many situations.

Keywords

object file, object file compression, wire format, class file, reflection, custom `ClassLoader`, Java.

1 Introduction

Dynamic linking is an often overlooked main feature of Java: Every class is managed independently of the others and the Java Virtual Machine loads them as the need arises. This gives an extraordinary amount of flexibility to the developer, as he can easily replace any classes, but the consequence is that Java binaries are composed of a large number of class files, which are predominantly small or even very small.

This proliferation of files hinders the distribution of the developed application: As each file has to be transmitted separately, their large number affects the communication delays. A large number of class files also increases the storage requirement, in memory or on the file system. In Java 1.1, Sun introduced the Java archive as a mean to aggregate the components of an application, its class files and the other resources. The Java archive format can compress its content as well, but the compression ratio achieved is not very good. Typically, the Java archive compresses the class files to about half their original size.

An important aspect of Java is the distribution of the program over a network or a slow communication channel. In this case, it is interesting to develop a program representation that results in faster transmissions, even if the receiver has to decompress the program before executing it. Such a format, also called a wire format, has to reconcile three goals: (1) the format has to compact as much as possible, yet (2) the format has to expand fast and (3) its decoder has to be small, if the decoder has to be transmitted with the compacted program.

This article presents the class archive (car) format, a wire format for archives of class files that is on the average one third of the size of the original class files. The class archive is not a replacement for the Java archive, which usually contains other kinds of data, and they can be used together. Although this format does not achieve the smallest size, the decompressor is both noticeably smaller and faster than all the other existing tools. This advantage does not result from clever programming but it is inherent in the approach taken.

2 Tools and Benchmarks

Concurrently with the definition of this new format, I wrote a prototype implementation of the tools necessary to pack, unpack and load in the Java Virtual Machine the content of car archives. The tools require Sun's JDK 1.2. To create car archives, the tool has to perform simple transformations of the class files; the library `JavaClass` [5, 6] provides all the methods necessary to read, parse and write the content of the class files.

I tested the new format on the applications listed in Table 1. Along with their names and a short description, the table presents the compression ratios achieved with Sun's Java archiver and gnu's gzip, a popular, all-purpose compression tool.

	size [byte]			**compression [% of raw size]**			
benchmark	**raw**	**jar**	**jar0.gz**	**jar**	**jar0.gz**	**description**	**vers.**
200	40’381	23’651	17’208	59	43	SPEC: Test features of the JVM	1.03
201	17’821	11’451	6’270	64	35	SPEC: Modified LZW compress	1.03
202	396’536	182’969	85’403	46	22	SPEC: Java Expert Shell System	1.03
205	57’000	30’810	21’077	54	37	SPEC: ray tracing	1.03
209	10’156	6’106	5’058	60	50	SPEC: database	1.03
213	1’955’030	1’134’101	751’358	58	38	SPEC: Sun’s `javac` (jdk 1.0.2)	1.03
222	120’182	67’308	46’103	56	38	SPEC: mpeg audio decompressor	1.03
227	859	895	728	104	85	SPEC: driver for benchmark 205	1.03
228	130’889	74’002	51’924	57	40	SPEC: Java parser generator Jack	1.03
999	5’899	4’023	2’937	68	50	SPEC	1.03
cs	1’851’883	1’181’008	753’738	64	41	Crédit Suisse: Internet banking	1.4.0
hotjava	1’975’655	1’115’681	804’372	56	41	Sun: Web browser	3.0
JavaCC	783’731	371’216	311’719	47	40	Metamata: Parser generator	1.0
JavaClass	446’493	279’569	150’657	63	34	Bytecode engineering library	3.2.7
jdk-swing	157’480	84’560	49’019	54	31	Sun: Swing	1.2.2
jdk-tool	1’735’412	925’727	674’558	53	39	Sun: `javac`, `javadoc`, ...	1.2.2
jgl	874’244	355’397	179’677	41	21	ObjectStore: Java Generic Library	3.1.0
jre-i18n	5’464’618	2’974’146	2’688’584	54	49	Sun: Internalization library	1.2.2
jre-rt	10’001’194	5’927’797	3’824’231	59	38	Sun: Run time	1.2.2
jws	6’013’281	3’199’604	2’429’377	53	40	Sun: Java Workshop	2.0

Table 1: Benchmarks Used and Compression Achieved with Popular Tools

The column *raw* is the sum of the sizes of all the class files, as they are distributed. The column *jar* is the size of a Java archive[a] that contains only these classes; they are individually compressed. For the column *jar0.gz*, the classes are grouped uncompressed in a Java archive that is then compressed with gzip. The fifth and sixth columns show the compression achieved in these two cases as a percentage of the original size.

a. The jar files were produced with zip, for speed and convenience.

3 The Canonical Architecture

The distribution and the execution of Java applications build upon three pillars: The Java class file, the Java archive and the `ClassLoader`. The Java class files are the individual components of the application, the Java archive groups these components in a single file and the `ClassLoader` brings them into memory.

3.1 The Java Class File

The Java class file is the compiled representation of a Java class. According to the specification [16], the class file holds three kinds of information. The class file contains the information that is necessary to execute the class, that is the definition of the class with its bytecode. The class file also contains the information that is necessary to link the class, that is all the references to other classes and all the references accessible from other classes. These first two categories encompass the required content of the class file: Without this content, the class would not be recognized by the Java Virtual Machine. Finally, the class file may contain any supplementary data, such as debugging information, or support for language extensions [21, 23]. Typically, tools make use of this last category to cache the results of code analyses or to supplement the information available to the run-time system.

Analysis of class files [1] measured the size of files; Java binaries are mainly made of small files: 50 percent of the files are smaller than 2’000 bytes and 80 percent smaller than 6’000 bytes. The same analysis found on the average eight percent of the file taken by unnecessary information.

3.2 The Java Archive

The Java archive groups together the files needed to run an application. The Java archive is not restricted to class files, and it may also contain any kind of data: Resources for the application, such as images, or meta-information for the Java Virtual Machine, such as certificates for the security manager.

The format of the Java archive builds upon the zip format [22]. This format stores both the description and the content of the archived files, and this allows the decoder to extract and restore any individual archive member. The member files may, optionally, be compressed. In this case, they are compressed individually, in order to grant fast access to any member file. However, this independent compression holds down the compression ratio because it limits the scope of investigation for the compressor to one file and this file is, in the majority of the cases, small.

3.3 The ClassLoader

The `ClassLoader` brings class files into memory whenever the Java Virtual Machine needs the definition of a new class. The `ClassLoader` does not internalize the data, it does not consider the content of the class file, it merely locates the definition of the class file, reads the bytes into memory and handles them to the security manager of the Java Virtual Machine. This latter component will check their correctness and build an internal representation of the class for the interpreter.

The `ClassLoader` itself is a Java class and the Java Virtual Machine can be extended with new `ClassLoader`s. Although the intention of Sun was for new `ClassLoader`s to define other search strategies, such as the `SecureClassLoader`, this mechanism also allows the introduction of other formats for the class file. Indeed, the Java Virtual Machine is only concerned with the bytes delivered by the `ClassLoader`, not those in the file.

4 Related Works

The search for smaller program representations proceeds along three independent directions: The pruners, the semantic compressors and the syntactic compressors.

The pruners pare down the content of the class files. Through different analyses, they determine which parts are actually needed during the execution of the application and remove all the other parts. Whereas the first tool described, JDistill [17], restricted itself to class members, methods or fields, more recent tools, such as Jax [20], also transform the class hierarchy, collapsing or removing classes that are not directly used. These tools give reduction of 17 to 32 percent and are best used right before archiving the application.

The semantic compressors of the object file try to determine an other instruction set which encodes the same program with less bytes. The general idea is to replace recurring instructions sequences with super-operators. Proebsting et al. [18, 11] present an iterative algorithm to select an optimal set of super-operators for a given program. With this process, every application is written with its own dedicated instruction set and needs a custom interpreter. Franz [10] and Kistler [15] propose to recognize patterns of instructions and to encode them with a dynamic dictionary. Semantic compression is typically limited to the instructions sequences and does not handle the rest of the object file, such as the constants.

The syntactic compressors consider the structure of the program instead of its content and attempts to detect the redundancy there. The transformation of the class name presented in Section 5.2 is a good example: It is entirely based on the grammatical definition of the Java identifier and not on some statistical analysis. Eck et al. [8] propose to compress the syntax tree of the Java source. Corless [4, 13] established the foundation for the compression of single class files. Bradley et al. [3] extend this to groups of class files. They introduce the idea of sharing the `ConstantPool` between classes. Further along this path, Pugh [19] builds with `pack` a completely new data structure to represent a set of class files.

5 The Car Approach

I designed the car format with the twin objectives of compressing more than the standard Java archive and of having a small and fast decompressor. This section details these goals and explains the techniques I used in achieving them.

5.1 Goals

The aim of car is to become a wire format for the distribution of Java applications. Wire codes [9] are codes “that need not be interpreted directly but can be at least partly decompressed in an interpretable form or even compiled before they are used.” In this case, a custom `ClassLoader` expands the content of the car file into the format expected by the Java Virtual Machine.

As with the Java archive, car is to be used once the application leaves the realm of the developers and is packaged for the distribution. One can make at this time the following assumptions:

- as this step occurs infrequently, the speed of the compressor is much less important than the performance of the decompressor;
- the aggregation of many classes becomes an entity, and the revision of the entity subsumes the individual revisions of the classes;
- the application has been exhaustively debugged.

The Java Virtual Machine has hefty requirements, so the decompressor will certainly have sufficient processing power and memory at disposal. But the applications are often distributed over a medium with limited bandwidth, be it a congested network or a modem. The size of the transmitted data is then a decisive factor and the car archives want to be as small as possible.

Finally, the decompressor for car wants to stay as simple and lean as possible. The decompressor is built around a simple `ClassLoader` and it does not require changes to the Java Virtual Machine. The decompressor itself will be distributed with the application, so its size is an important factor. It would not make sense to use car if the decompressor took more space than was saved by the compression.

5.2 Reduce the Meta-Information

A car file is an archive, which Howes [14] defines as “a single file containing one or (usually) more separate files plus information to allow them to be extracted (separated) by a suitable program.” The intended usage of the archive shapes the breadth of these meta-information.

Zip, upon which the Java archive format is built, is in this regard a backup tool. Zip is not only interested in restoring the content of the member files, but also all the file system attributes, such as the creation date or the owner of the file.

In a car archive, the meta-information consists of only two elements: The name and the size of the member class. The name is required to identify the correct class file member in the archive, and the size is needed to bound it. It would in theory be possible to leave out even these two items because they are already present in the class file itself. I left them in, as omitting these two elements requires the `ClassLoader` to completely parse the class file, and parsing the class files has three adverse effects:

- the `ClassLoader` becomes larger because it needs its own code to parse the class file;
- the `ClassLoader` becomes slower because it has to parse the class file;
- the work performed by the `ClassLoader` will be redone by the Java Virtual Machine, which expects an array of bytes and not some internalized structure.

The names of the member files are at the same time the names of the Java classes, which means that they abide by a very strict grammar (Code 1). In particular, the fact that the names are taken out of a restricted alphabet enables their further compactions.

```
className
      packageName javaName
      innerClassName.
innerClassName
      { '$' javaName }.
packageName
      { javaName '.' }.
javaName
      JavaLetter { JavaLetter | Digit }.
```

Code 1: Java Grammar for the Names of Classes
Java uniquely identifies the classes with their fully qualified names. See Gosling [12] § 3.8 for the definition of the terminals `JavaLetter` and `Digit`.

Car compresses the class names with a simple three-steps dictionary coder. Car first sorts lexicographically all the names, taking care to group the inner classes after their enclosing class. Car then takes advantage of this proximity and rewrites the name of the inner class with a colon and just the name of the inner class; the colon means “this is an inner class of the preceding top level class.” Finally, car numbers the packages sequentially as they come and replaces further occurrences with a dot followed by the sequence number of the first appearance of a package.

This description calls for two comments. First, the productions of Code 2 use illegal characters, the dot and the colon, to extend the productions of Code 1; it is hence always possible to mix compacted and un-compacted name. Second, the `knownPackageIndex` is coded as a Unicode character; this allows to elegantly introduce the index into the name but at the same time restricts car archive to 2^{16} packages. Although this should be sufficient for any application, it is always possible to split the archive.

```
className
      ( packageName javaName )
      | innerClassName.
innerClassName
      ':' javaName { '$' javaName }.
packageName
      [ '.' knownPackageIndex ]
      { javaName '.' }.
```

Code 2: Car Grammar for the Names of Classes
The production of this grammar extends and supplants those in Code 1 to take advantage of the packages’ structure. The terminal `knownPackageIndex` is the (short) index of another class in the same package.

To judge the effect of this transformation, I measured the size of the Utf8 encoding of all the class names in the benchmarks both before and after the conversion. The median compression ratio is 52 percent and the extrema are 36 and 79 percent.

5.3 Strip the Classes

The class files typically contain additional informations that are not required to execute correct Java applications but help during the development stage. As car is to be used for the distribution of applications, these parts of the file can be thrown away without incurring any loss of functionality. Note that this does not prevent further developments with these classes, only their debugging or their integration in some tools.

I stripped the classes by performing the following operations:

- Remove all but the `Code`, `Exceptions` and `ConstantValue` attributes.
- Rid the `ConstantPool` of all unreferenced entries.
- Sort the `ConstantPool` by the type of the entries.
- Sort lexicographically the `Utf8`s.

Pugh presented the idea for the last two operations [17]. They did not prove very effective in classes compiled by Sun’s `javac`, but I kept them as safeguard against other, more sloppy tools.

5.4 Compress the Classes Together

The third step writes and compresses the car file. The principal advantage of car over jar comes now: Car submits all its member files together to the compression engine whereas jar submits every member file individually. That the member files are grouped together instead of separately has two consequences. First, the compression engine wraps every compression unit with initialization data for the decompressor, which is repeated with every member of a Java archive file. Second, compression is performed by replacing repeating sequences of bytes by a reference to a previous occurrence of the same sequence. With a larger input, the likelihood to find repetitions is also greater and the compression is consequently more effective.

The structure of the file is given in Code 3. A very short header identifies the file type and version of the format; it is followed by a series of file members. Each member consists of its coded name, its uncompressed length and the content of its class file. The single character ‘]’, which is an illegal Java name, marks the end of the archive. The resulting byte stream is compressed with the zip library.

```
carFile
     cookie version
     { className length classFile }
     ‘]’.
cookie
     0x53736172. -- ASCII ‘Ssar’
version
     0x0100. -- major.minor
length
     u8.
```

Code 3: Structure of an Car File
Car files are reduced to the name, the size and the content of the class. The length of the size is encoded in a long, written `u8`.

6 Integration in the Java Virtual Machine

Car integrates with the standard Java Virtual Machine at two places, to create car archives and to execute the packed applications directly out of the car archive.

The tool `Car`[1] manages the car archives. It is similar in fuctionality with Sun’s `jar` tool. `Car` is a Java application that packs a set of class files into an archive, or extracts the content of an archive to conventional class files. `Car` can read and write the class files as individual files or as members of a Java archive. The tool contains 17 classes, for a total size of 34’799 bytes. Packed itself into a car archive, the tool has just six files and 20’311 bytes.

To execute an application packed in an car archive, the Java Virtual Machine must look for the classes in three different places: The classes can be compressed in a car archive, they can be packed in a Java archive or they can stand alone as a set of class files. Furthermore, the classes of an application can be distributed simultaneously in different formats, maybe for a part in a car archive and the rest in a Java archive. Thanks to the standard class loader, the Java Virtual Machine knows how to look in the usual Java archives and class files, but it needs a custom `ClassLoader` to access to the content of the car archives.

A small Java application, `Run`, functions as a bootstrap loader for the car format. `Run` instantiates a `ClassLoader` for the car format and asks it to search for the main class in a given set of car archives. The `ClassLoader` further delegates the search to the default class loader to look in the standard places[2]. `Run` then uses the reflection API to call the `main` method with the correct arguments and hence launches the application. Further requests for classes will automatically route through this custom `ClassLoader`, hence providing access to the rest of the car archive. The `ClassLoader` and the launcher application contain together five classes and are 8’219 bytes.

1.
```
java fully.qualified.class.name arg0
     arg1 ...
```
2.
```
java li.antonio.car.Run fqcn.car
     fully.qualified.class.name arg0
     arg1 ...
```
3.
```
java -jar fqcn.jar fqcn.car
     fully.qualified.class.name arg0
     arg1 ...
```
4.
```
java -jar fqcn.jar arg0 arg1 ...
```
5.
```
java -jar zip.jar -car zip arg0 arg1 ...
java -jar zip.jar -car unzip arg0 arg1
     ...
```

Code 4: Launching Java Application
The code demonstrates four ways to launch the application `name` in the package `fully.qualified.class` with the two arguments `arg0` and `arg1`. The application is present (1) as a set of class files, (2) in the car archive `fqcn.car`. In (3), the car archive is packed into the Java archive `fqcn.jar` with a simple bootstrap loader. In (4), the car archive is packed into the Java archive `fqcn.jar` with the enhanced bootstrap loader. (4) has the same syntax as with a traditional Java archive. Finally (5) shows how aliases allow to access different applications packed in a Java archive.

The car archive format is not a substitute for the Java archive format but a more efficient way to compress the Java class files. It is consequently still useful to pack the car archives along with the bootstrap loader and other resources in a Java archive. Among other advantages, the Java archives are directly executable on some platforms (see the fourth example in Code 4) and they should not lose this property when they contain car archives.

When asked to execute a Java archive, the Java run-time environment launches the class identified by the `Main-Class` attribute of the manifest file with the arguments specified on the command line. If the Java archive contains car archives, this `Main-Class` attribute must necessarily identify the bootstrap loader. But the bootstrap loader still needs the names of the actual main class and of the set of car archives to search. The third example in Code 4 shows the command line necessary to employ the bootstrap loader `Run`: The two arguments are inserted between the name of the Java archive and the arguments of the application. This is not satisfactory because the name of the main class, which was neatly stashed away inside the manifest, is now exposed on the command line.

1. Unless explicitly specified, all the classes belong to the package `li.antonio.car`.
2. The different versions of Java define how and where the classes are searched with mechanisms such as the `CLASSPATH` variable.

The solution to this problem is the introduction of a second manifest, a *car manifest,* stored in the file CAR-INF/MANIFEST.MF in the root directory of the Java archive. The primary function of this manifest is to identify the main class and the car archives needed to execute the application. Two further functions came during the development of this format. First, the manifest allows for the bootstrap loader to identify alternate applications. This is a practical feature when a set of small applications are build upon the same library; for example, the zip and unzip tools wrap around the zlib library. In that case, the library and all the applications can be distributed in a single Java archive. Second, the manifest recognizes aliases for the names of the class files. This feature lets invoke the application with short identifiers instead of the fully qualified class names.

The car manifest follows the syntax defined by Sun [7] and is compatible with the class `java.util.jar.Manifest`, but it has other attributes. The main attributes identify the application called per default. The other attributes either map an alias name to a fully qualified class name or they identify alternate applications stored in the Java archive. The fully qualified name of the main class and the set of car archives required identify each application.

```
Manifest
     Default { \n Alternate }
Default    main attributes
     MainClass
Alternate
     Alias | MainClass
Alias      Name1 is an alias for Name2
     ‘Name: ’ Name1 \n
     ‘Alias:’ Name2
MainClass
     ‘Name:’ Name \n
     ‘Path:’ Name { Name }
```

Code 5: CAR-INF/MANIFEST.MF

The manifest identifies the main class; it also maps names to class and car archives; *aliases* let different names be synonymous.

The bootstrap loader `JarRun` performs the same function as `Run`, but `JarRun` gets its arguments from the manifest instead of the command line. With `JarRun` installed in the Java archive, the invocation of the default application looks like the fourth example of Code 4. In the fifth example, `JarRun` gets from the option `car` the alias of an application stored in the Java archive; `JarRun` uses the manifest to map this alias to a class name. This example posits that the zip and unzip applications are both thin wrappers around a common library.

7 Security Considerations

The integration of the security in the run-time environment is one the strong features of Java. It is also important to consider the influence of this new archive format and, in particular, of the self executing archive, on the security of the system.

In Java 2, the security results from the combined action of different mechanisms. As the classes are loaded, the bytecode verifier checks their correctness, which effectively prevents malicious code to execute. The Java Virtual Machine then checks the behavior of the application during its execution. At the lowest level, it prevents direct access to important resources and controls the access to all the resources. Finally, Java provides an authentication architecture, which allows the Java Virtual Machine to relax, or tighten, its policy according to the application provider.

Corless [4] identifies next to the lossless and lossy compressions a third kind of compression: The semantically-preserving compression. A file that is packed and unpacked with a semantically-preserving compression algorithm is not bit-for-bit identical with the original, but it acts the same. Car is a typical semantically-preserving compression tool: Car removes attributes from the class file, reorders the `ConstantPool`, and eventually rewrites the bytecode, yet it does not influence the execution of the application.

The bytecode verifier and the low-level access control are only concerned with the semantic of the application, so they are not affected by the compression.

The authentication architecture rests on digital signatures that rely by nature on the specific byte sequences. This part of the security system is not supported yet by car, but there are no fundamental hurdles to the integration of car archives into the authentication architecture.

8 Results

This section presents experimental results for the car format. The experiments study the compression ratio achieved and the speed of the decompression. In order to better appreciate the efficacy of car, the section also reports the results for the standard Java archive and the best available compressor, *pack* (see Section 4).

8.1 Compression size

Table 2 presents the size of car archives for the benchmarks. In the table, *car* are archives which are simply compressed. The members of the *sorted car* archives are first stripped of unnecessary content and their `ConstantPool` reordered. The best available compressor, *pack*, shows the bottom limit. The table also reports the compression ratios of these three archive formats against uncompressed class files (*raw*) and against compressed Java archives (*jar*). The reference sizes are in Table 1.

The average compression of the simple car format is 38 ± 10 percent of the original size and 65 ± 12 percent of the equivalent Java archive file. This second improvement comes solely from the compression of all the archive members as a single stream, which offers the compressor more potential redundancies. Stripping the classes and sorting their constant pools result in even better compression of 30 ± 9 percent of the original size, respectively 51 ± 11 percent of the Java archive.

Compared to pack, the car format produces larger archives, on the average 197 ± 62 percent. The more elaborate transformations performed by pack are incontestably superior to the

	size [byte]			against jar0 [%]			against jar [%]			against pack [%]	
benchmark	car	sorted car	pack	car	sorted car	pack	car	sorted car	pack	car	sorted car
200	16’618	12’559	7’176	41	31	18	70	53	30	232	175
201	5’818	5’341	2’876	33	30	16	51	47	25	202	186
202	80’389	58’699	23’063	20	15	6	44	32	13	349	255
205	20’125	14’698	7’274	35	26	13	65	48	24	277	202
209	4’875	4’798	2’890	48	47	28	80	79	47	169	166
213	719’916	531’951	191’775	37	27	10	63	47	17	375	277
222	44’579	43’182	22’161	37	36	18	66	64	33	201	195
227	583	475	377	68	55	44	65	53	42	155	126
228	50’250	34’216	16’742	38	26	13	68	46	23	300	204
999	2’754	1’844	1’178	47	31	20	68	46	29	234	157
cs	711’854	674’529	225’509	38	36	12	60	57	19	316	299
hotjava	782’986	591’297	846’819	40	30	43	70	53	76	92	70
JavaCC	308’323	209’334	105’714	39	27	13	83	56	28	292	198
JavaClass	144’743	118’766	46’073	32	27	10	52	42	16	314	258
jdk-swing	47’326	29’688	28’423	30	19	18	56	35	34	167	104
jdk-tool	662’736	505’758	245’637	38	29	14	72	55	27	270	206
jgl	168’088	162’031	61’469	19	19	7	47	46	17	273	264
jre-i18n	2’614’042	2’157’605	1’846’200	48	39	34	88	73	62	1429	117
jre-rt	3’683’179	2’713’534	1’109’347	37	27	11	62	46	19	332	245
jws	2’375’137	1’575’787	640’612	40	26	11	74	49	20	371	246

Table 2: Compression Ratios

The table presents the compression ratios of the proposed approach and compares them to the more radical *pack*. The column *car* reports the results when all the classes are only compressed with the zlib library; for the column *sorted car*, the classes are first filtered to remove the supplementary information and their constant pools are sorted to offer more and better opportunities to the compressor.

approach advocated in car, but the next section will show that the higher compression comes at a considerable runtime penalty.

8.2 Decompression Speed

This section compares the influence of the three formats on the execution time. The compression of applications is a classic asymmetric process. The compression ends the development cycle and occurs hence infrequently. On the opposite, the decompression is the first step of every execution; it occurs often and its speed matters. Consequently, this section presents only the decompression performance.

For the Table 3, I measured the time necessary to decompress each of the archive formats into memory; the table shows the averages and the standard deviations of three runs for every cases. Beside the execution time, the table also gives the decompression speed, which is defined as the size of the compressed data divided by the time needed to re-establish the class files in memory.

	time [ms]			speed [kByte/s]		
benchmark	jar	sorted car	pack	jar	sorted car	pack
200	54 ± 20	39 ± 2	407 ± 49	473 ± 144	323 ± 17	18 ± 2
201	33 ± 2	43 ± 17	731 ± 374	344 ± 16	136 ± 45	5 ± 3
202	251 ± 2	108 ± 1	741 ± 13	724 ± 4	542 ± 6	31 ± 1
205	55 ± 1	42 ± 1	405 ± 1	562 ± 6	350 ± 8	18 ± 0

Table 3: Timing the Decompression

The table presents the run time and the speed of the three approaches considered, the traditional Java archive (*jar*), the compact *pack* and the proposed *car* format. The results are the averages and the standard deviations measured for three runs of each test case.

	time [ms]			speed [kByte/s]		
benchmark	**jar**	**sorted car**	**pack**	**jar**	**sorted car**	**pack**
209	23 ± 1	31 ± 4	310 ± 1	273 ± 7	158 ± 18	9 ± 0
213	1’021 ± 62	638 ± 15	4’545 ± 108	1’107 ± 69	834 ± 20	42 ± 1
222	102 ± 2	59 ± 1	709 ± 32	655 ± 15	736 ± 14	31 ± 1
227	18 ± 1	26 ± 1	265 ± 3	56 ± 2	18 ± 1	1 ± 0
228	100 ± 1	65 ± 0	625 ± 1	737 ± 7	526 ± 0	27 ± 0
999	21 ± 1	29 ± 1	287 ± 4	199 ± 6	64 ± 2	4 ± 0
cs	1’248 ± 5	612 ± 6	6’235 ± 22	937 ± 4	1’102 ± 11	36 ± 0
hotjava	1’046 ± 133	615 ± 75	6’064 ± 573	1’078 ± 130	971 ± 121	140 ± 13
JavaCC	280 ± 7	176 ± 1	2’951 ± 644	1’324 ± 32	1’192 ± 4	37 ± 7
JavaClass	454 ± 7	200 ± 1	1’355 ± 18	618 ± 10	595 ± 3	34 ± 0
jdk-swing	92 ± 11	73 ± 16	917 ± 241	931 ± 107	421 ± 84	32 ± 8
jdk-tool	735 ± 47	486 ± 28	4’517 ± 286	1’264 ± 78	1’043 ± 60	55 ± 3
jgl	423 ± 6	233 ±2	2’743 ± 79	835 ± 11	694 ± 5	22 ± 1
jre-i18n	1’821 ± 242	1’399 ± 128	nan[a]	1’655 ± 239	1’552 ± 150	nan[a]
jre-rt	4’959 ± 327	3’259 ± 214	25’443 ± 1’638	1’197 ± 76	835 ± 53	44 ± 3
jws	2’065 ± 5	1’345 ± 96	13’630 ± 535	1’544 ± 3	1’176 ± 80	47 ± 2

Table 3: Timing the Decompression

The table presents the run time and the speed of the three approaches considered, the traditional Java archive (*jar*), the compact *pack* and the proposed *car* format. The results are the averages and the standard deviations measured for three runs of each test case.

a. The unpacker aborted with a `java.lang.IndexOutOfBoundsException` during the decompression of this application.

The comparison of the execution times shows that pack is the slowest decompressor. This is a consequence of the complicated transformations that produced the excellent compression ratios. For short bout of time, less than 50 ms, jar is the fastest decompressor, after that, the car decompressor leads. The comparison of the decompression speeds confirms that pack is one order of magnitude slower than jar and car. The comparison also tells that car is slightly slower than jar.

Figure 1 illustrates this fact and it explains the times measured: For small files, the Java archives and the car archives are about the same size and the jar decompressor finishes first. But as soon as the files get larger, car compresses more efficiently and the car decompressor has markedly less data to handle. That is why the car decompressor is then faster overall.

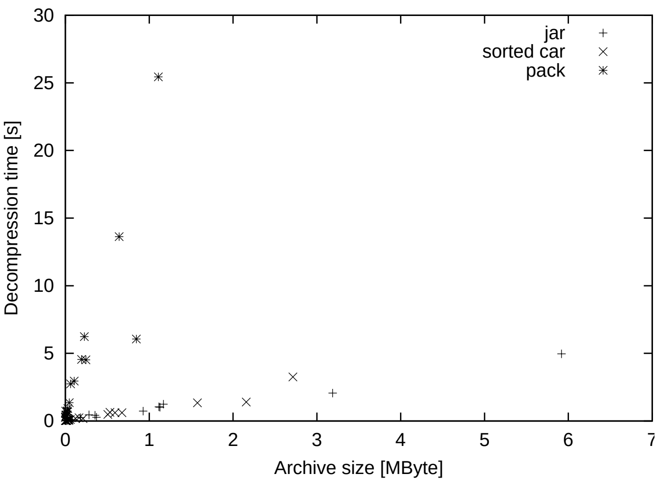

Figure 1: Comparison of the Decompression Times

The Java archive (*jar*) and car formats decompress at about the same speed, but car archives are smaller, so car is faster overall. The pack archives are more complex and their decoding is markedly slower.

9 Discussion

The Table 4 summarizes the relevant characteristics for a wire format: The compression ratio, the size of the decoder and the decompression speed. It is at once apparent that pack produces the smallest files but also that pack’s decoder is slower and larger. How these factors combine is not so apparent and will be investigated in this section.

The elapsed time between the request for an archive and its delivery is a function of the transmission time t_{trm} and the decompression time t_{dec} (Equation 1). I omit factors such as the transmission setup time or the seek time on the server as they are constant for the three archivers.

$$t = t_{trm} + t_{dec} \qquad \textbf{(EQ 1)}$$

The speed of the link and the amount of data determine together the transmission time. The choice of an archiver sets the amount of data transmitted through the size of the archive reader

	jar	pack	car
Median compression ratio [% of raw]	57	18	30
Size of decompressor [KByte]	0	36	8
Median decompression speed [KByte/s]	826	33	663

Table 4: Characteristics of the Archive Formats
The table lists the three main features of the Java archive (*jar*), pack and car formats.

sizeof(*dec*) and the size of the uncompressed archive, which is the compression factor k time the overall size of the archive members (Equation 2).

$$t_{trm} = \frac{\mathrm{sizeof}(dec) + k\,\mathrm{sizeof}(members)}{v_{trm}} \qquad \textbf{(EQ 2)}$$

The decompression time is simply the overall size of the archive members divided by the decompressor's speed v_{dec} (Equation 3).

$$t_{dec} = \frac{k\,\mathrm{sizeof}(members)}{v_{dec}} \qquad \textbf{(EQ 3)}$$

With these definitions and the values summed up in Table 4, the Equation 1 allows an objective comparison of the archivers.

This equation identifies three characteristics of a wire format, which are its compression factor, the size of its decompressor and the decompression speed. The investigation will hence first find out how important these characteristics are. The answer to this question should tell which parts of a new format are important to optimize.

Under the assumption that the communication channels are always getting faster, the Equation 4 shows that the time difference between two archivers, indicated with the superscripts one and two, is determined by the ratio of the compression factor over the decompression speed: The smaller the ratio, the faster the system is overall. With the results reported in Table 4, car has a ratio of 5, Java archive 7 and pack 54.

$$\lim_{v_{trm} \to \infty} {}^1t - {}^2t = \left(\frac{{}^1k}{{}^1v_{dec}} - \frac{{}^2k}{{}^2v_{dec}}\right)\mathrm{sizeof}(members) \qquad \textbf{(EQ 4)}$$

Of course, the speed of the communication channels is hardly ever infinite and the behavior of the three formats under various speeds is important for the selection of a format. Figure 2 and Figure 3 show the overall decompression times for the three formats studied for two possible application domains. In both figures, the Java archive benefits from being a standard part of Java; as such, it is always included in the run-time system and does not need to be transmitted. For the other two formats, all transmissions have to include the decoder and this offsets the two lines upward.

The first application domain for a wire format are systems with a very slow communication channel. As an example for such a system, I offer in Figure 2 GSM, the European standard for cellular phone; GSM transmits digital data at a speed of 9.6 kbit/s. At this speed, the three formats have an utility. The Java archives are best for applications smaller than 29 KByte, then the car format is the fastest for applications up to 389 KByte, after which the pack format is faster.

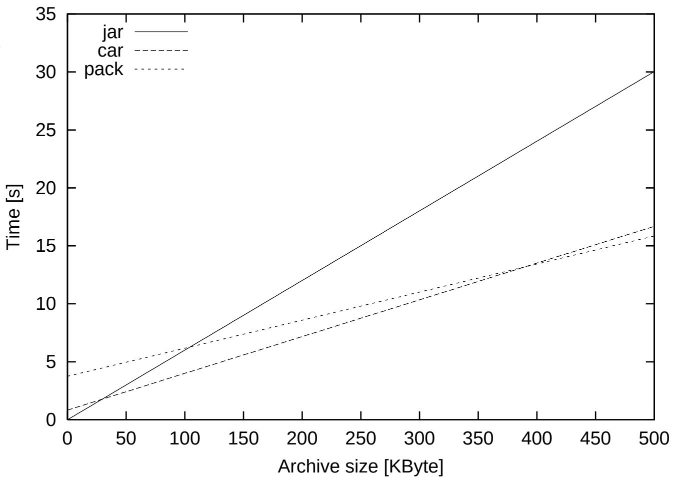

Figure 2: Decompression Time Over GSM
The figure plots the estimated time for the transmission and the decompression of archived application over a GSM communication channel. GSM transmits digital data at 9.6 kbit/s.

The second application domain considered are all the systems that are connected through a plain analog modem. At 56 kbit/s, which is the highest standardized speed for a modem, the Figure 3 shows an important difference: The car format achieves the fastest transmission for all the archives larger than 28 KByte. The decompression speed is already the dominant factor for this connection speed.

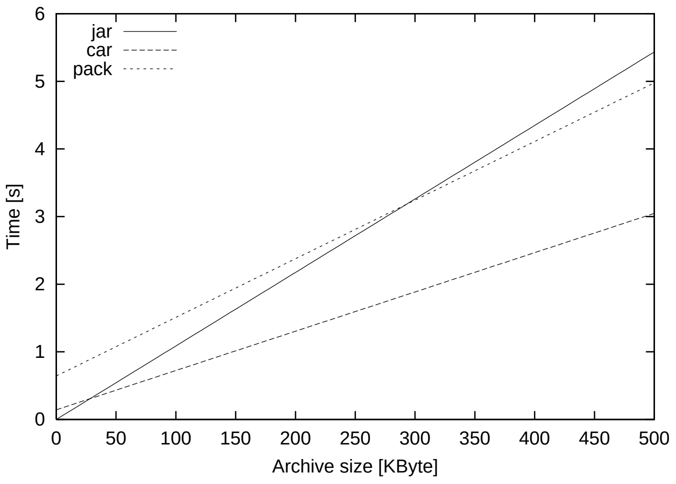

Figure 3: Decompression Time Over a Modem
The figure plots the estimated time for the transmission and the decompression of archived application over a modem. The V.90 modem standard transmits at up to 56 kbit/s.

Where are then the optimal application domains for the three formats? The Equation 1 can answer this question too. As the connection slows down, the transmission rate becomes more and more important. The Equation 5 calculates the boundaries of the three domains. At the limit, for an hypothetical transmission speed of zero bit/s, Java archives are best for applications smaller than 29 KByte and pack archives are best above 233 KByte; in between, car is the better format. A simple computation also finds pack slower than the car format for all connections faster than 24 kbit/s; pack is also slower than the Java archive above 82 kbit/s.

$$\lim_{v_{trm} \to 0} {}^1t - {}^2t = \frac{\mathrm{sizeof}({}^1dec) - \mathrm{sizeof}({}^2dec)}{{}^2k - {}^1k} \qquad \textbf{(EQ 5)}$$

10 Conclusion

I presented in this article a new wire format for archives of Java class files. Whereas others have elected to achieve the smallest possible file and completely overhauled the existing format in the process, I choose to balance the size of the archive with the simplicity and the velocity of the decoder.

The proposed format has these properties:

- car archives are on the average 30 percent of the size of the original class files and 51 percent the size of the corresponding compressed Java archives.
- car archives are twice the size of the best available archive, pack.
- the car decompressor itself is markedly smaller than pack's decompressor (five KByte against 36 KByte).
- the car decompressor is an order of magnitude faster than pack's decompressor.

I also presented in this article a theoretical foundation for the comparison of wire formats. The characterization of the format does not only consider the compression factor, but also the size of the necessary decompressor and the decompression speed. It is possible with these criteria to derive the important features of a wire format.

- for slow transmission channels, the compression ratio and the size of the decompressor are the most important factors;
- for fast transmission channels, the compression ratio and the decompression speed are the most important factors;

With this theory, I contrast my proposed format, car, with Sun's Java archives and Pugh's pack. The comparison finds that:

- the Java archive is an optimal format whenever the application is smaller than 29 KByte;
- car is without competition for applications larger than 29 KByte and over connections faster than 24 kbit/s;
- car is also the best format over connections slower than 24 kbit/s for applications larger than 29 KByte and smaller than a moving threshold; the threshold moves between 233 KByte, at zero bit/s, and up to five MByte, at 23 kbit/s;
- in the remaining cases, for applications larger than this threshold and over connections slower than 24 kbit/s, pack is the best format.

These results assume that the Java archive decoder is already present on the receiver system but that the decoders for the other two formats have to be transmitted with the application. The results would be slightly different if the receiver had the decoders pre-installed. In this case, the more compact pack would be ideal for communications slower than 24 kbit/s and my car format for all faster transmission channel.

11 Acknowledgments

I express here my gratitude to Markus Pilz and Takashi Suezawa for their numerous but constructive criticisms.

12 References

[1] Denis N. Antonioli and Markus Pilz, *Analysis of the Java Class File Format*, Technical report ifi-98.04, Department of Information Technology, University of Zurich, Switzerland, 1998 <ftp://ftp.ifi.unizh.ch/pub/techreports/TR-98/ifi-98.04.ps.gz>

[2] Denis N. Antonioli, *Compressing Java Binaries: The Ristretto Project*, PhD Thesis, Department of Information Technology, University of Zurich, Switzerland, 2000

[3] Quetzalcoatl Bradley, R. Nigel Horspool and Jan Vitek, *JAZZ: An efficient compressed format for Java archive files*, in *Proceedings of CASCON '98*, November 1998

[4] Jason D. Corless, *Compression of Java class files*, MSc Thesis, Department of Computer Science, University of Victoria, Canada, 1996

[5] Markus Dahm, *Byte code engineering*, in *Java-Informations-Tage 1999 (JIT '99)*, ed. Clemens H. Cap, Springer-Verlag, pp 267-277, 1999

[6] Markus Dahm, *JavaClass*, <http://www.inf.fu-berlin.de/~dahm/JavaClass>

[7] Thomas Dell, David Hopwood, Dave Brown, Benjamin Renaud, David Connelly, *Manifest format*, in Sun's JDK from version 1.1 onward, 1996

[8] Peter Eck, Xia Changsong, and Rolf Matzner, *A new compression scheme for syntactically structured messages (programs) and its application to Java and the Internet*, in *International conference on data compression*, Snowbird, Utah, U.S.A., March 1998

[9] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco and Todd A. Proebsting, *Code compression*, in *Proceedings of the ACM SIG-Plan conference on programming language design and implementation (PLDI'97)*, ACM SIG-Plan Notices, v. 32 n. 5, pp 358-365, June 1997

[10] Michael Franz, *Code–generation on the fly: A key to portable software*, PhD Thesis 10497, Swiss Federal Institute of Technology, Zurich, Switzerland, 1994

[11] Christopher W. Fraser and Todd A. Proebsting, *Custom instruction sets for code compression*, unpublished, <http://reasearch.microsoft.com/~toddpro/papers/pldi2.ps>, 1995

[12] James Gosling, Bill Joy and Guy Steele, *The Java language specification*, Addison-Wesley, 1996

[13] R. Nigel Horspool and Jason Corless, *Tailored compression of Java class files*, Software – Practice & Experience, v. 28 n. 12, pp 1253 – 1268, October 1998

[14] Denis Howe (ed.), *The free on-line dictionary of computing*, <http://foldoc.doc.ic.ac.uk/>, 1993

[15] Thomas Kistler and Michael Franz, *A tree-based alternative to Java byte-code*, Technical report 96-58, Department of Computer Science, University of California, Irvine, U.S.A., 1996

[16] Tim Lindholm and Frank Yellin, *The Java Virtual Machine specification*, Addison-Wesley, 1996

[17] Éamonn McManus, *JDistill, a program to shrink Java packages*, <http://www.gr.osf.org/~emcmanus/jdistill.html>, April 1998

[18] Todd A. Proebsting, *Optimizing an ANSI C interpreter with superoperators*, in *Conference record of POPL'95: The 22th ACM SIG-Plan – SIG-Act symposium on principles of programming languages*, pp 322–332, January 1995

[19] William Pugh, *Compressing Java class files*, in *Proceedings of the ACM SIG-Plan conference on programming language design and implementation (PLDI'99*), ACM SIG-Plan Notices, v. 34 n. 5, pp 247-258, May 1999

[20] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter, *Practical experience with an application extractor for Java*, in *Proceedings of the fourteenth annual conference on object-oriented programming systems, languages, and applications (OOPSLA'99)*, ACM SIG-Plan Notices, v. 34 n. 10, pp 292-305, 1999

[21] Philip Wadler, *GJ: A generic Java*, Dr. Dobb's Journal, v. 25 n. 2, pp 23-28, Februray 2000

[22] Info-ZIP, *zip*, <http://www.cdrom.com/pub/infozip/>

[23] Sun, *Connected, limited device configuration: Specification version 1.0, Java 2 platform micro edition*, Sun Microsystems, 1.0, May 2000