INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Java Bytecode Compression for Embedded Systems

Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller

N° 3578

Décembre 1998

THÈME 2

Rapport de recherche

ISSN 0249-6399

Java Bytecode Compression for Embedded Systems

Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n3578 — Décembre 1998 — 22 pages

Abstract: A program executing on an embedded system or similar environment faces limited memory resources and fixed time constrains. We demonstrate how factorization of common instruction sequences can be automatically applied to Java bytecode programs. Based on a series of experiments, we argue that program size is reduced by 30% on the average, typically with an execution time penalty of less than 30%. The one-time, minor modifications necessary to make a standard Java interpreter compatible with this factorized code are presented on the Harissa virtual machine, together with an algorithm for performing the factorization of Java bytecode.

Key-words: Java, intermediate code, factorization, compression, embedded systems

(Résumé : tsvp)

This research is supported in part by Bull.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Compression de bytecode Java pour systèmes embarqués

Résumé : La conception et l'exécution d'un programme dans un système embarqué doit répondre à des contraintes en taille de mémoire et en temps de réponse. Dans cet article, nous montrons qu'il est possible de factoriser automatiquement des séquences d'instructions communes pour du code intermédiaire Java. Sur un ensemble de programmes Java représentatifs, nous avons obtenu un taux moyen de compression de la taille du code intermédiare de 30%, ceci avec un temps d'exécution augmenté au plus de 30%. L'implémentation de la factorisation requiert des modifications mineures au sein d'une machine virtuelle Java. Nous décrivons ces modifications pour la machine virtuelle d'Harissa et présentons un algorithme permettant de factoriser le code intermédiaire.

Mots-clé : Java, code intermédiaire, factorisation, compression, systèmes embarqués

1 Introduction

The Java language [9], while enjoying widespread use in many application domains, is by design also meant to be used in embedded systems. This is witnessed by the availability of specific APIs, such as the JavaCard specification [15] and the EmbeddedJava specification [16]. The primary advantage of Java is, in this context, portability. Portability is realized through the Java bytecode format [11], allowing any third-party developed services to be installed on any Java-compatible embedded system.

Embedded systems have strong restrictions on the amount of available memory, severely limiting the size of those applications that they can run. Memory is scarce for a number of reasons: production costs must be kept low, power consumption must be minimized, and the available physical space is limited. Thus, it is desirable that embedded applications consume as little memory as possible, including not only space allocated dynamically at runtime, but also the space taken up by the program code itself; the less space is taken by the code for each feature of the application, the more features can be added to the embedded system.

Although Java bytecode is reasonably concise, programs still contain repeated patterns of code. Compression comes to mind as a viable solution for those situations where the size of the program code storage must be minimized.

Data compression has a very wide range of applications, and is a well-studied area [20, 1]. A traditional solution would involve decompressing different parts of the program as they are needed, and discarding them afterwards. However, this approach is usually not applicable in the context of embedded systems. First, in an embedded system, there is often not sufficient memory to decompress even a single method. For example, an existing JavaCard system such as the Java Ring is limited to limited to 32K of ROM and 6K of RAM [3]. Second, the time taken to uncompress such a segment of code might exceed the fixed time constraints that an embedded system must obey. As an example, consider that each JavaCard user operation should not take more than a second.

This paper presents a solution that reconciles the need to conserve space with fixed time constraints. We propose to factorize recurring instruction sequences into new instructions, yielding a more concise program that is executable on a Java virtual machine (JVM) with an extended instruction set. By expressing the new instructions as macros over existing instructions, the JVM need only be extended to support such general macro instructions, not instructions specific to any one program. Using this technique, program size is on the average reduced by 30%, at a runtime speed penalty not exceeding 30%.

The rest of the paper is organized as follows. Section 2 fixes the setting by discussing various applicable techniques for compressing Java programs. We factorize code into bytecode macros, as illustrated by an example in Section 3. Section 4 describes the actual factorization algorithm that we employ. Section 5 describes how we implement macro support in the JVM. The experimental results that we obtain are presented in Section 6. Finally, we describe related work in Section 7 and conclude in Section 8.

2 Approaches to compression of Java bytecode

An often used approach for reducing the size of Java class files is to remove all non-essential information. However, this technique is not useful in the context of embedded systems, as described in Section 2.1. It is, however, possible to use standard compression techniques to compress bytecodes, in the limited fashion proposed in Section 2.2. As an alternative to compression, standard Java bytecode can be used to express factorization of recurring instruction sequences, as described in Section 2.3. However, the only alternative that we judge worthwhile is that of generating new instructions, which we present in Section 2.4.

2.1 Class file trimming

Java is has achieved widespread popularity as a means of distributing platform-neutral programs over the internet. In this context, a commonly used approach to reducing the size of Java class files is to preprocess the class file by removing debugging information and by replacing the constants that name various internal parts of the Java class by shorter but unique representations. In the context of embedded systems such preprocessing steps are redundant. Most of this information is stripped when converting the class file to the internal format used in the embedded system. This also means that the size of the class files are a minor concern. The important metric is how much space the program takes once transmitted to the card.

Having eliminated debugging information and most, if not all, of the constant pool, all that remains is the actual bytecode that defines the methods of the Java program. Another way to reduce the size of the code is to eliminate unused methods. However, the bytecode of the remaining methods still takes up space, so we consider this approach orthogonal to compression techniques.

2.2 Basic block compression

Looking beyond simple manipulation of the Java class file, an often-used solution for compression is word-stream compression techniques such as Huffmann encoding [10] or Lempel-Ziv compression [20]. The whole program can be stored in compressed form, decompressing each part of the program from ROM to RAM, as it is needed during execution. However, given the limited memory resources of embedded systems, it is not even possible to decompress each method as it is invoked. Rather than storing whole parts of the program in RAM, stream compression can be applied individually on each basic block of the code. Since the instructions in a basic block are used sequentially, they can be decompressed on the fly by a modified JVM, without having to store them to RAM. While such generic compression algorithms may not be optimal for the kinds of patterns found in program code [4], they are well known and can easily be implemented.

However, we consider that this solution, while interesting, is not sufficient for our purposes. First, stream compression techniques are not well suited for the compression of many small, individual blocks, limiting the expected gains in compression. Second, a significant time overhead would be associated with decompressing each basic block, slowing down the overall speed of the system to an unacceptable degree.

2.3 Methods and subroutines

Most Java bytecode programs contain repeated occurrences of expressions. As a simple example, consider a specific object field that is manipulated throughout a class, each time being accessed through the same operations. An optimizing compiler can eliminate this redundancy by common-subexpression elimination, but only when all such expressions compute the same value. In general, bytecode programs contain repeated occurrences of the same instruction sequences, used to compute different values.

A simple way of eliminating this code duplication in a way that requires no changes to the JVM, is to use methods to define repeated instruction sequences. A specific sequence of instructions that are used many places within a single class can be placed in a fresh method. Each original sequence of instructions is replaced by a call to the method. This requires no change to the run-time system, but there is a space overhead for defining a method (26 bytes in a Java class file), and each invocation of such an instruction sequence occupies three bytes. Furthermore, any local changes to the state have to be copied explicitly back and forth, introducing significant time and space overheads.

Using subroutines to define the instruction sequences allows the local state to be shared, so the space overhead is limited to three or four bytes per subroutine defined. However, subroutines are intra-procedural, limiting the number of places where each subroutine can be used to the point where this technique is uninteresting for our purposes.

2.4 Instruction set extension

As an alternative to a solution made purely in Java, the instruction set of the virtual machine can be extended with instructions that can replace recurring instruction sequences. The Java bytecode instruction set is generic, as it was not designed with any particular application domain in mind. There are special versions of commonly used instructions, such as instructions for accessing the first few local variables. However, it is not possible to have similar specific versions of instructions for manipulating objects, since these instructions refer (through the constant pool) to names specific to each program.

A JVM is written specifically for each embedded system, so we can add new features as long as the JVM is able to run normal Java bytecode. However, adding new instructions requires some amount of work to modify the interpreter, and would have to be redone if different code is to be used. Furthermore, some commonly used instructions (field manipulation and method invocation in particular) take arguments that depend on the constant pool of the class the instructions reside in. The constant pool here serves to perform dynamic linking between classes, arguments being symbolic indices into the constant pool rather than absolute offsets into other classes. It must be possible to resolve these arguments correctly.

Alternatively, this approach can be realized in a more general way, by extending the virtual machine to read new instruction definitions from the class file. These *macro instruction* definitions consist of bytecode instructions, and replace common instruction sequences in the code. Any instruction not in the standard instruction set is assumed to be a programmable instruction, defined through a table specific to the program being interpreted. The macro instructions can be stored in the run-time system, with very little memory overhead. Each macro instruction is specific to the program so it can be defined relative to the constant pool of a specific class file. With this approach, the number of new instructions is limited by the number of instructions not used in the standard instruction set.

This last option is the most promising: the space and time overheads are negligible, it is easily implemented, and it can provide sufficient levels of compression. The rest of the paper describes the implementation of and experiments with this approach.

3 A Simple Example

Our approach is to factorize repeated instruction sequences, and represent the result using instruction macros.

As an example, we use the Java classes of Figure 1. The class `Point` represents a geometrical point, with a method that computes the distance to the center of the coordinate system. The class `Box` simply holds a reference to a content. The corresponding bytecode program is shown in Figure 2. Default constructors for both classes have been automatically introduced by the Java compiler.

In the bytecode program of Figure 2, there are some obvious opportunities for factorization. To access the `Point.x` field, the `Point` instance is loaded onto the stack and a `getfield` instruction is used to extract the value, giving two repetitions of the following instruction sequence:

```
0 aload_0
1 getfield #4 <Field Point.x I>
```

Furthermore, both classes have been extended with a default constructor, which consists of an invocation of the constructor of `Object`:

```
0 aload_0
1 invokenonvirtual #5 <Method java.lang.Object.<init>()V>
4 return
```

Every default constructor in a program has exactly the same body, representing an ideal opportunity for factorization.

Faced with such sequences of generic instructions that are used repeatedly in specific programs, we replace each sequence by a new instruction. Figure 3 shows the bytecode program that results from factorizing the program of Figure 2, along with the corresponding table of macros. The repeated instruction sequences for accessing the fields have been factorized into macro instructions, as well as the body of the constructors.

```
public class Point {
  public int x, y;
  public int dist() {
    return Z.intSqrt( (x*x)+(y*y) );
  }
}
public class Box {
  public Object content;
}
```

Figure 1: Java source code for the `Point` and `Box` classes

```
Method int dist()
   0 aload_0
   1 getfield #4 <Field Point.x I>
   4 aload_0
   5 getfield #4 <Field Point.x I>
   8 imul
   9 aload_0
  10 getfield #7 <Field Point.y I>
  13 aload_0
  14 getfield #7 <Field Point.y I>
  17 imul
  18 iadd
  19 invokestatic #6 <Method Z.intSqrt(I)I>
  22 ireturn
Method Point()
   0 aload_0
   1 invokenonvirtual #5 <Method java.lang.Object.<init>()V>
   4 return
Method Box()
   0 aload_0
   1 invokenonvirtual #5 <Method java.lang.Object.<init>()V>
   4 return
```

Figure 2: Java bytecode for the `Point` class

```
Method int dist()
   0 Macro#204
   1 Macro#204
   2 imul
   3 Macro#205
   4 Macro#205
   5 imul
   6 iadd
   7 invokestatic #6 <Method Z.intSqrt(I)I>
  10 ireturn
Method Point()
   0 Macro#206
Method Box()
   0 Macro#206
```

Macro table:

```
Macro instruction 204:
   0 aload_0
   1 getfield #4 <Field Point.x I>
Macro instruction 205:
   0 aload_0
   1 getfield #7 <Field Point.y I>
Macro instruction 206:
   0 aload_0
   1 invokenonvirtual #5 <Method java.lang.Object.<init>()V>
```

Figure 3: Factorized Java bytecode for the `Point` and `Box` classes

4 Factorization

We now present an algorithm for transforming a Java bytecode program into an equivalent program factorized with respect to a set of patterns. Computing the optimal set of patterns is an NP-complete problem [14]; the algorithm presented in this section is polynomial-time, and computes a sub-optimal set of patterns.

Conceptually, recurring operations are abstracted by factorizing them into single units. Each such sequence of bytecode instructions is viewed as a *pattern*, each pattern having the semantic effect of the corresponding sequence of bytecode instructions. Factorizing the program with respect to a pattern yields a reduced program, where each *occurrence* of the pattern has been replaced by a new instruction representing the instructions of the pattern. The collection of all replaceable occurrences of a pattern is termed an *occurrence group*.

Factorization is done in three steps. First, patterns are generated to factorize recurring instruction sequences. Next, those patterns that cannot be expressed as macros are pruned from the generated set of patterns. Last, the bytecode is factorized with respect to these patterns, producing factorized bytecode.

4.1 Pattern generation

Instructions are first abstracted down to numbers, retaining branching information (instructions that target the instruction). These numbers are found by sorting all the instructions by an arbitrary metric which considers instructions with differing arguments to be different. Constants are resolved before comparing, so that instructions from different classes can be compared correctly.

To find the set of patterns with which to factorize the program, the maximal set of occurrence groups are created. First a group of length one is created for each set of equivalent instructions. These groups are then iteratively expanded, either elongating each group or splitting it into new groups. If all the instructions following the last instructions of a group are equivalent, the group becomes one instruction longer. Otherwise, a new group is created for each set of following equivalent instructions.

An elongated occurrence group may overlap itself, in the sense that some of its occurrences overlap other occurrences. Wherever this happens, the later occurrence is removed before we check the next occurrence, thus removing as few occurrences as possible from the occurrence group. After all occurrence groups are found, we remove groups that are totally covered by other groups.

4.2 Pattern pruning

We reduce the occurrence groups so that certain problematic ("unfactorizable") instructions do not occur in the patterns, and so that no branch instructions cross the boundaries of a group. We do this in two steps: first unfactorizable instructions and outgoing branches are handled, then incoming branches are handled.

The instructions `tableswitch`, `lookupswitch`, `jsr`, and `ret` are considered unfactorizable: the switch instructions have alignment constraints that makes it difficult to place them inside macros, and the subroutine instructions induce problematic intra-procedural control flow. Neither of these instruction types tend to occur very often, so we consider that it is not worth the extra complexity to factorize these instructions.

Unfactorizable instructions are removed from a pattern by splitting it into two new patterns, splitting the occurrence group accordingly. Similarly, where an outgoing branch leaves a pattern, the branching instruction is removed from the pattern, creating two new patterns. To reduce the complexity of this step, we disallow backward branches in patterns. By far, the most branches are forward, caused by if- and break-statements. By checking the branches bottom-up, we ensure that splitting a pattern cannot cause already checked branches to become illegal.

Then, incoming branches are checked. In contrast with outgoing branches, which occur in all occurrences of a pattern, incoming branches may be the result of a single branching statement and not be shared by the whole group. Therefore, we only remove the occurrence that has an incoming branch. The alternative would be to split the pattern between the instruction being branched to and the preceding instruction. However, this would require extra checking to ensure that intra-pattern branches are not affected. The first instructions of an exception handler, or the first and last instructions of a code region where exceptions are caught, are treated in the same ways as targets of incoming branches. After these reductions, totally covered groups and groups of size 1 are removed.

4.3 Pattern application

Having computed the set of patterns, we start generating macro instructions. Macros are generated greedily by selecting the occurrence group that gives most savings first and continuing until we either run out of available macro instruction codes or groups that save space. We replace the first instruction of each occurrence by a macro instruction with information on how many instructions have been factorized. We remove other occurrences that are partially or fully covered, but not occurrences that cover a whole occurrence of this pattern. While occurrences that are fully covered are removed from their respective groups, the groups will contain an occurrence in the newly-made macro. Occurrences that share the same first instruction as an occurrence of the group being removed need to point to the macro, if they are shorter than the occurrence, or to the macro instruction, if they are longer than the occurrence.

The number of unused instructions in the instruction set determines the possible number of new macro instructions. The normal Java instruction set uses 203 instructions. We allocate one instruction for marking the end of a macro, so we have available at most 52 single-byte macro-instructions. In the case of the JavaCard instruction set, which uses only 103 instructions, we have at most 152 single-byte macro instructions available.

Representing a macro as a single byte is simple and has very little overhead. However, the number of macros that can be defined is limited by the number of unused instructions in the JVM instruction set. For this reason, we may wish to define macros that have a two-byte instruction length. Though this yields less reduction in size than using a one-byte instruction, it is still worth doing in some cases. The loss in size of reduction can be minimized by choosing the two-byte instructions such that they occur only rarely.

The first byte of a double-size macro is the instruction code, the second byte indicates an offset into a secondary table of ordinary single-byte macros. This gives room for 255 more instructions for each instruction code used this way.

Since macros by definition are non-recursive, the factorization also computes the maximal intra-procedural macro nesting, simplifying the modifications that need to be made to the JVM. The factorized code is stored in two separate attributes, as described in Appendix A. The original code is also kept, so a JVM without support for execution of factorized code will still be able to run the program.

5 Implementing an Extensible JVM

This section describes how to make a standard JVM extensible, enabling it to run factorized program code expressed in an extended instruction set. The extended instructions are expressed in terms of macros over existing instructions. The extensible JVM must support loading and execution of macro-enriched class files. We first describe how the macros are represented in Section 5.1, then detail in Section 5.2 the modifications necessary to the main loop of the JVM to add macro support.

5.1 Macro representation

We make a standard JVM extensible by enabling execution of macro instructions. Before execution on the JVM, a program can be factorized into modified code that uses macros, together with a set of macro instructions. A macro is essentially defined by three values: the instruction code, the *body*, and the class that defines any constants in the body. The body of the macro is a code block which is terminated by the special instruction `macro_end`. It may contain other macro instructions. The set of macros is global to all classes. All references to constants refer to the constant pool of a specific class as described individually by each macro.

Macros are stored using a standard file format, enabling the modified interpreter to use macros produced by any factorization program. For reference, the format is described in Appendix A. When transferring a factorized program into an embedded system, the macros must be transferred as well. This can be done automatically since the necessary information already is present in the class files of the program. When the extensible JVM of the embedded system is initialized, the factorized bytecode program and the macro definitions are already in place, in the internal format used by the JVM.

5.2 JVM main loop modifications

Extending a JVM to support macro-instructions is a simple, one-time change. This change enables it to execute the normal code of a method mixed with the code stored in macros. Since the macro-instructions are expressed in terms of standard instructions, the extended interpreter supports execution of any factorized program.

Each method invocation stack frame must contain a fixed-size macro call stack of program counters. Since macros are non-recursive, the maximum stack depth can be computed by the factorization algorithm along with the set of macros. When a macro instruction is invoked, the current value of the program counter is pushed on the macro stack. Afterwards, the program pointer is set to point to the first instruction of the body of the executed macro. Execution proceeds from there until the macro return instruction `macro_end` is executed, an exception is thrown, or a return is made from the current method.

When the termination instruction is executed, the program counter is reset to the top value of the program counter stack (which is popped), and execution proceeds from the next instruction. If a return is performed during the execution of a macro, control is transferred back to the caller, and the current stack frame is popped from the stack, disposing any program counters stored on the macro stack. Similarly, if an exception is thrown during the execution of a macro, the entire stack of program counters is popped, the program counter is reset to the last program counter popped from the stack, and control is transferred directly to the appropriate exception handler. The choice of exception handler is determined by the layout of the exception regions relative to the current program counter; the extent of each region is reduced by the factorization algorithm when replacing code sequences by single instructions, so the correct exception handler is automatically invoked.

References to constants from within the body of a macro are resolved with respect to a specific class, as described in Section 4. If constants are resolved when transferring the program into the embedded system, this resolution is done with respect to this specific class. Otherwise, we need to keep track of the current constant pool when executing macros whose constants must be resolved with respect to the constant pool of some other class. The current constant pool can be saved on a stack when invoking a macro, as is done for the program counter.

For reference, the modifications described in this section are illustrated in Appendix B, on the main loop of an interpreter.

6 Performance Evaluation

We have implemented factorization of standard Java class files and extended the JVM of the Harissa environment [12] with macro support. The Harissa environment integrates an optimizing off-line Java compiler with an interpreter, the interpreter allowing execution of dynamically loaded programs.

To test the efficiency of the factorization algorithm and the execution speed of the resulting program, we have tested the factorization algorithm on the following program packages:

JavaCard library: The JavaCard 2.0 library classes, taken from the JavaCard Reference Implementation [17].[1] These libraries are normally placed on every JavaCard system, making them ideal candidates for factorization.

JavaCard applets: The JavaPurse applet distributed with the JavaCard Reference Implementation [17] and the Visa Open Platform Card Version 1.0 Implementation for Applet Developers [19].

JavaCard library + JavaCard applets: JavaCard applet class files together with the JavaCard 2.0 library classes.

JDK Applets: The demonstration applets included with JDK 1.0.2 [18].

CaffeineMarks: Micro-benchmark suite designed specifically for Java [13].

Javac: JavaSoft's JDK 1.0.2 Java compiler (subdirectories `asm`, `java`, `javac` and `tree` of the `sun/tools` directory).

Compactor+Cream: The factorization program itself, which is integrated within the Cream bytecode optimizer [2].

Of these, the first three tests use only the JavaCard instruction set, while the last four tests use the full Java instruction set. Table 1 shows the size of the bytecode in bytes, before and after factorization, with and without the macro definition code included. The average method size is also shown, as well as the total size obtained when using single-byte macros only. The maximal macro stack nesting did not exceed four on any of these tests, and the longest-lasting factorization was done in less than 15 minutes on a 200 MHz UltraSparc. (Factorization need only be done once, before placing the program in the embedded system.)

To compare the compression obtained using factorization with an estimate of what would be gained by compressing each method individually using a standard compression algorithm, we compare the size reduction we obtain with two different applications of the standard Unix compression tool `gzip`. To use `gzip` to uncompress methods in a JVM, each method would have to have been compressed individually using a global dictionary. To estimate the size reduction obtained using this technique, we have evaluated `gzips` compression ratio in two different ways. The local reduction is the reduction obtained by compressing the bytecode of each method separately and subtracting the 20-byte file overhead. The global reduction is obtained by compressing the bytecode of all the methods as one file. We hypothesize that a compression-based algorithm with a global dictionary would perform somewhere between those two numbers when operating on small bits of data. The compression ratios are shown in Figure 4.

To estimate the run-time speed penalty of using macros, we have performed two tests, both on Pentium (100MHz Dell Pentium) and SPARC (SPARC Station 5) architectures. Due to limitations in the Harissa interpreter, we were unable to run any JavaCard applets. The Harissa JVM was compiled using `gcc` on the Pentium and Sun's commercial `cc` compiler

[1] Excluding the part of the simulator that implements JavaCard native methods.

	Size of class file	Size of bytecode	Average method size	Number of macros defined	Bytecode size after factorization	Size of macro table	Normal total size	Single-byte macro total size
JavaCard Library	55884	6266	26	155	3682	662	4344	5172
JavaCard Applets	13803	2940	94	128	1645	422	2067	2067
All JavaCard	74535	9206	33	167	5722	764	6486	6522
JDK Applets	116736	25643	86	147	19793	849	20642	22115
CaffeineMarks	12629	3021	34	69	1846	407	2253	2329
Javac	497074	90025	78	128	72000	607	72607	76521
Compactor+Cream	737129	112063	43	131	82518	888	83406	89081

Table 1: Size of bytecode (sizes are in bytes) before and after factorization

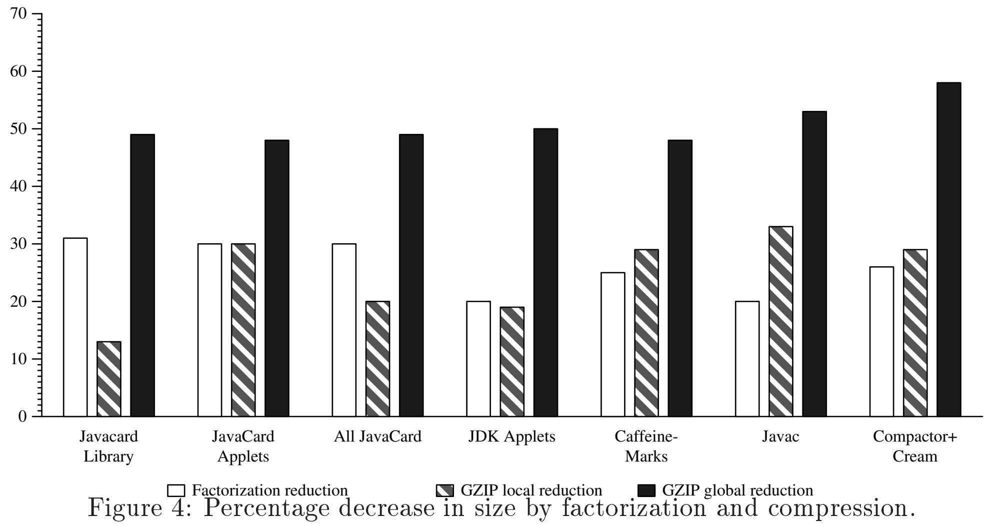

Figure 4: Percentage decrease in size by factorization and compression.

	Javac		CaffeineMarks	
	Pentium	SPARC	Pentium	SPARC
Without macros (n_1)	46.3s	69.6s	26	15
With macros (n_2)	46.6s	71.4s	21	11
Slowdown ($1 - n_1/n_2$)	2%	3%	19%	27%

Table 2: Benchmarks comparing normal execution with macro execution

on the SPARC (compiling with `cc` on the SPARC gave the best unfactorized execution time performance). The first test measures the performance of the factorized Javac compiler. Although this is not a program that is likely to be placed in an embedded system, it is a large and complex application that performs a wide range of different data manipulation tasks. The result is shown in Table 2. There is virtually no slowdown, compilation takes 2-3% longer when using the factorized code. The second test is the CaffeineMark benchmark. Given that the tests in this suite are micro-benchmarks (i.e. tight loops testing very specific instructions), we assume that they represent a worst-case scenario with respect to the speed of factorized code. Here, we observe a visible slowdown, 19% on the Pentium and 27% on the SPARC. These numbers are within the bounds that we consider to be acceptable.

The factorization algorithm clearly works at it best when it has at its disposition for macros the 100 additional free instructions of the JavaCard instruction set. Nonetheless, similar performance is almost delivered for the full Java bytecode instruction set. As can be seen from the rightmost column of Table 1, two-byte macros provide an advantage here, on the average 5%.

Examining the generated macros reveals that the average macro length is 3 bytes (median length 2), and that each macro is applied 9 times on the average. Instructions for accessing the first eight local variables were factorized into single-byte macros in most of the tests. Similarly, the instructions for invoking the constructor of `java.lang.Object` was factorized in most cases. Finally, access to fields specific to each set of class files was factorized in all cases.

These observations could be taken as indications that the existing JVM instruction set can be optimized to obtain size reductions similar to those obtained using macro instructions. However, the macro instructions are specific to the constant pool of a program, something a fixed instruction set could not be. In addition, adding more fixed instructions would make macro-instruction approaches much less efficient. Nonetheless, it is quite possible that a slightly different instruction set could be a noticably more compact representation for Java programs. We consider this aspect as future work.

7 Related work

The idea of compressing code is by no means new. Fraser, Myers and Vendt describe an approach similar to ours, using suffix trees to compress assembly code [8]. They get an average compression factor of 7%. They factorize local branches, use parameterized patterns, and also implement a cross-jumping technique to exploit merging code sequences. However, they require their code fragments to have no net effect on the stack, which limits the effectiveness of their approach.

Ernst et al. describe compression of code, both for transmission and for execution [4]. They obtain the same compression ratio as when using `gzip` for the executable code. They introduce a specific bytecode language for this purpose, using a bottom-up joining technique to form patterns. While their compression technique yields better results than ours, it requires greater amounts of RAM than is available on some embedded systems.

Proebsting describes a C interpreter using "superoperators" [14]. This kind of operators can be automatically inferred from the tree-like intermediate representation produced by `lcc` [6, 7]. The operators are then used to produce an interpreter with specialized instructions. This transformation is aimed at improving speed; it gives a modest reduction in program size, at the cost of increased size of the generated interpreter. The approach of using a specialized interpreter could also be viable for embedded systems, e.g., by specializing the interpreter with respect to the general run-time environment.

Franz and Kistler present SLIM binaries as an alternative to Java bytecode [5]. SLIM binaries provide a highly compact platform-independent structured program representation, designed to be translated into binary code by an optimizing just-in-time compiler. Due to the structured representation which can include information needed for optimizations, the compilation overhead is negligible and the generated binary code is as efficient as that generated by an ordinary (optimizing) compiler. However, SLIM binaries are not easily interpretable in their compressed form, needing to be compiled into binary code before execution. This makes them unsuitable for embedded systems.

8 Conclusion and Future Work

We have implemented factorization for Java bytecode. It handles non-trivial programs, and reduces the total size of the bytecode by about 30%, in some cases outperforming `gzip`. The execution time overhead of introducing macros is less than 30%.

We are currently investigating whether it is worthwhile to allow macros to take a fixed number of instructions (possibly themselves macros) as arguments. Rather than having patterns that represent continuous sequences of instructions, the patterns are parameterized by a number of instructions that can be used in different instantiations of the macro. Some design choices need to be made, such as whether multiple arguments are allowed and whether an argument can be more than one instruction. The more elaborate designs lead to intricate implementations that support a complex format for parameterized macros, at the cost of performance. The simpler designs, while more efficient, may not be advanced enough to support truly useful parameterized macros.

References

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

[2] Lars R. Clausen. A Java byte-code optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 1997.

[3] Dallas Semiconductor Corp. *Java-Powered Decoder Ring*, 1998. URL: `http://www.ibutton.com/jring_facts.html`.

[4] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 358–365, New York, June15–18 1997. ACM Press.

[5] Michael Franz and Thomas Kistler. Slim binaries. *CACM*, 40(12):87–94, December 1997.

[6] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software-Practice and Experience*, 21(9):963–988, September 1991.

[7] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[8] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analysing and compressing assembly code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 117–121. ACM, 1984.

[9] James Gosling, Bill Joy, and Guy L. Steele Jr. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[10] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, September 1952.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[12] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 1997. Usenix Association.

[13] Pendragon Software. CaffeineMark 3.0, 1997. URL: `http://www.webfayre.com/pendragon/cm3/index.html`.

[14] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, January 22–25, 1995. ACM Press.

[15] Sun Microsystems, Inc. *JavaCard 2.0 Language Subset and Virtual Machine Specification*, 1.0 edition, October 1997. URL: `http://www.javasoft.com/products/javacard/index.html`.

[16] Sun Microsystems, Inc. *Embedded Java Specification*, 1.0 edition, March 1998. URL: `http://java.sun.com/products/embeddedjava/note.html`.

[17] Sun Microsystems, Inc. *Java Card API 2.0 Reference Implementation*, developer release 2 edition, February 1998. URL: `http://www.javasoft.com/products/javacard/index.html`.

[18] Sun Microsystems, Inc. *The Java Developers Kit 1.0.2*, 1998. URL: `http://java.sun.com/products/jdk/1.0.2/`.

[19] Visa International Service Association. *Visa Open Platform Card Version 1.0 Implmentation for Applet Developers*, 1998. URL: `http://www.visa.com/cgi-bin/vee/nt/suppliers/open/main.html`.

[20] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable rate encoding. *IEEE Transactions on Information Theory*, 24:530–536, September 1978.

A Macro File Format

This appendix describes the exact, standard layout proposed for storing macros in Java class files. Each factorized class file includes two new attributes in addition to the original unfactorized code: the factorized code and the set of macros for which the code has been factorized. When transferring the bytecode to an embedded system, the transference mechanism can select which attributes to make use of, depending on the capabilities of the embedded JVM.

In the class file, the factorized bytecode is stored in an attribute similar to the standard attribute, named `IRISA.Macrocode`. Likewise, the code defining the macro is stored in an attribute named `IRISA.Macrodef`, shown in Figure 5. The attribute contains the maximum nesting depth of the macros, the number of macros it defines, and an array of macro definitions. Each macro is defined by a one-byte type tag (defined in Figure 6), an instruction code, a defining class for resolving constants, and a byte array of code. Instruction 255 is reserved for marking the end of a code array (`macro_end`), if so required by the implementation.

The macro definitions can be stored by the JVM in a very compact fashion: By placing the code for all macro definitions in a single array, the overhead for defining a new macro can be reduced to one byte plus the size of the indexing array. Depending on the architecture of the runtime system, the size of this array may vary. By using two-byte offsets to point to the start of each instruction, the array can be kept small. Rather than having a separate field indicating the length of the macro definition code, the implementation can add the `macro_end` instruction at the end of each macro definition code array. The second byte

```
IRISA_Macrodef_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 max_nesting;
  u2 macro_table_length;
  {  u1 type_tag;
     union {
       { u1 macro_code; } simple;
       { u2 macro_code; } double;
     }
     u2 defining_class;
     u2 code_length;
     u1 code[code_length];
  } macro_table[macro_table_length];
}
```

Figure 5: Layout of macro definition attribute `IRISA.Macrodef`

`MACRO_TYPE_SIMPLE = 0`: A macro with a one-byte code

`MACRO_TYPE_DOUBLE = 1`: A macro with a two-byte code

Figure 6: Type tag for each kind of macro

of the two-byte instructions will always be used from the lowest first. Thus, all double-size macros with the same prefix instruction code can be placed in a single, dense array. New instruction codes will not be allocated for double-size macros before all those already available are used.

B Interpreter Main Loop Modifications

Figure 7 shows the core of a normal bytecode interpreter. The interpreter is written as a recursive function that is invoked once for each method invocation. The local variables of the function thus correspond to the stack frame of each method in the JVM. In Figure 8, the interpreter has been modified to allow execution of single-byte macro instructions. Two small stacks keep track of program counters and constant-pool defining classes. Constant lookup is done from the defining class. Throwing an exception clears the stack. When executing a macro instruction, the old program counter and defining class are pushed on their stacks, and the new defining class is taken from the macro executed. Ending a macro involves popping the program counter and defining class from the stack.

```
StackItem *interpretMethod( Method *m, StackItem *vars ) {
  unsigned char c, d, type, *pc;
  StackItem *sp, *vars;
  /* ... other local variables ... */
  pc = m->code;
  /* ... other initializations ... */
  while( 1 ) {
    c = *pc++;
    switch( c ) {
    case ACONST_NULL: sp--; sp->r = NULL; break;
    case ICONST_0: sp--; sp->i = 0; break;
    case LDC1:
      d = *pc++;
      type = m->constant_pool[d].tag;
      switch( type ) {
      case CONSTANT_Integer:
        sp--; sp->i = m->constant_pool[d].value.c_int; break;
        /* ... other types ... */
      }
      break;
    case ATHROW:
      throwException( sp[0].r );
      break;
      /* ... other instructions ... */
    default:
      VM_error( "unkown opcode %d", c );
      break;
    }
  }
  /* ... release resources and return ... */
}
```

Figure 7: The core of a bytecode interpreter

```
StackItem *interpretMethod( Method *m, StackItem *vars ) {
  unsigned char c, d, type, *pc;
  StackItem *sp, *vars;
  /* ... other local variables ... */
  unsigned char *pc_stack[macroTable.max_nesting];
  Class *cp, *cp_stack[macroTable.max_nesting];
  int stack_index = 0;
  pc = m->code;
  /* ... other initializations ... */
  cp = m->class;
  while( 1 ) {
    c = *pc++;
    switch( c ) {
    case ACONST_NULL: sp--; sp->r = NULL; break;
    case ICONST_0: sp--; sp->i = 0; break;
    case LDC1:
      d = *pc++;
      type = cp->constant_pool[d].tag;
      switch( type ) {
      case CONSTANT_Integer:
        sp--; sp->i = cp->constant_pool[d].value.c_int; break;
        /* ... other types ... */
      }
      break;
    case ATHROW:
      stack_index = 0;
      throwException( sp[0].r );
      break;
      /* ... other instructions ... */
    case MACRO_END:
      stack_index--;
      pc = pc_stack[stack_index];
      cp = cp_stack[stack_index];
      break;
    default:
      if( macroTable.table[c] == NULL ) VM_error( "unkown opcode %d", c );
      pc_stack[stack_index] = pc;
      cp_stack[stack_index] = cp;
      stack_index++;
      pc = macroTable.table[c]->code;
      cp = macroTable.table[c]->defining_class;
      break;
    }
  }
  /* ... release resources and return ... */
}
```

Figure 8: The core of a bytecode interpreter with support for macros (added or modified lines are emphasized)

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399