Java Bytecode Compression for Low-End Embedded Systems

LARS RÆDER CLAUSEN, ULRIK PAGH SCHULTZ, CHARLES CONSEL, and GILLES MULLER
Compose Group, IRISA/INRIA

A program executing on a low-end embedded system, such as a smart-card, faces scarce memory resources and fixed execution time constraints. We demonstrate that factorization of common instruction sequences in Java bytecode allows the memory footprint to be reduced, on average, to 85% of its original size, with a minimal execution time penalty. While preserving Java compatibility, our solution requires only a few modifications which are straightforward to implement in any JVM used in a low-end embedded system.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; *Intepreters*; *Run-time environments*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Code compression, embedded systems, Java bytecode

1. INTRODUCTION

The Java language [Gosling et al. 1996], 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 and EmbeddedJava specifications [Sun Microsystems, Inc. 1997; 1998a; 1999b; 1999c; 1999d]. The primary advantage of Java in this context is portability, which is realized through the Java bytecode format [Lindholm and Yellin 1996]. The use of a standard format allows any third-party developed services to be installed on any Java-compatible embedded system.

Low-end embedded systems, such as smart-cards, have strong restrictions on the amount of available memory, severely limiting the size of 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 available physical space is limited.

This research was supported in part by Bull.
Authors' addresses: L. R. Clausen, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; U. P. Schultz and G. Muller, IRISA/INRIA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France; C. Consel, LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

Thus, it is desirable that an embedded application consumes as little memory as possible, including 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 embedded into the system.

Existing efforts at reducing the size of Java programs have concentrated on reducing the size of Java class files for transmission and subsequent execution on a standard workstation [Bradly et al. 1998; Pugh 1999]. In the Java class format, the constant pool comprises most of the space; the bytecode instructions only contribute about 18% of the total size [Antonioli and Pilz 1998]. However, the size of a class file is unimportant in the context of low-end embedded systems; only the memory footprint of the loaded program matters. In a low-end embedded system, the constant pool is either completely removed (when dynamic loading is not needed) or reduced using ad hoc techniques. We estimate[1] that the bytecode accounts for roughly 75% of the memory footprint in a system, using the token-based constant-pool approach of JavaCard 2.1 [Sun Microsystems, Inc. 1999d] (which allows dynamic loading of code).

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 [Bell et al. 1990; Ziv and Lempel 1978]. A traditional solution would involve decompressing different parts of the program as they are needed, and discarding them afterward. However, this approach is usually not applicable in the context of low-end embedded systems. First, in a low-end embedded system, there may not be sufficient memory to decompress even a single method. For example, an existing JavaCard system such as the Java Ring is limited to 32K of ROM and 6K of RAM [Dallas Semiconductor Corp. 1998]. Second, the time taken to uncompress such a segment of code might exceed time constraints defined by the application domain.

This paper presents a solution that reconciles the need to conserve space on low-end embedded systems with fixed time constraints. We propose to factorize recurring instruction sequences into new instructions. This factorization allows more concise programs to run on a Java Virtual Machine (JVM) extended to support new instructions.

By expressing the new instructions as macros over existing instructions, the JVM only needs to be extended to support these macro instructions, not to support instructions specific to any one program. Using this technique, program memory footprint is on the average reduced to 85% of its original size, at an average run-time speed penalty between 2% and 30%.

The rest of the paper is organized as follows. Section 2 fixes the setting by discussing various applicable techniques for reducing the memory footprint of Java programs. We factorize code into macros over instructions, as illustrated in Section 3 by an example. Section 4 describes the actual factorization algorithm that we employ. Section 5 describes how macro support is implemented in the JVM. The experimental results are presented and discussed in Section 6. Finally, related work is described in Section 7, and concluding remarks are presented in Section 8.

[1] Based on measurements done using standard JavaCard CAP files, described in Section 6.

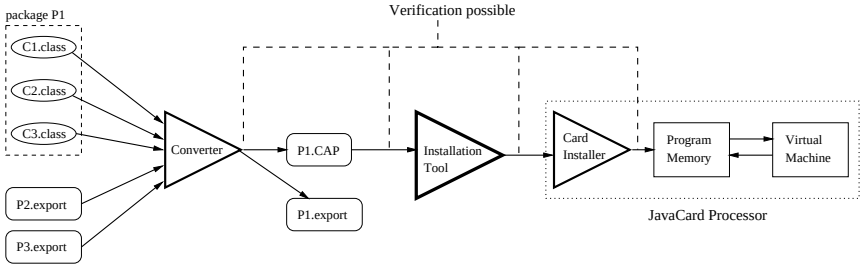

Fig. 1. Transferring Java classes into a JavaCard system.

2. APPROACHES TO REDUCING MEMORY FOOTPRINT

The standard Java class file format contains information that does not need to be present in a low-end embedded system. Thus, an internal, more compact format is used, as discussed in Section 2.1. However, the size of the bytecode can be further reduced, by using standard compression techniques in the limited fashion proposed in Section 2.2, or by factoring out recurring instruction sequences, as described in Section 2.3.

2.1 Conversion to an Internal Format

Although the Java bytecode instruction set was designed with embedded systems in mind, it is evident that standard Java class files produced by compilers such as Sun's `javac` compiler are not intended for use on such systems: debugging information and names of internal (private) identifiers are, for example, included by default. Although these are easily stripped from class files,[2] much precious space is still taken up by names that are not needed during execution. For this reason, it is natural for a low-end embedded system to use its own internal space-efficient representation. Throughout this paper, we will use the JavaCard 2.1 environment [Sun Microsystems, Inc. 1999b; 1999c; 1999d] as a reference, since it is the only documented, freely available low-end Java execution platform.

Java programs are transferred to JavaCard systems in units of packages, each package implementing either a set of applets or a library. The process of transferring a Java package to a JavaCard system is illustrated in Figure 1. First, a set of Java class files that make out a package is converted into a single CAP (converted applet) file and an export file describing the package interface. Export files describing other packages that are used by the classes in the package are also given to the converter. This scheme allows all the name information to be stored in export files, with two-byte tokens as the only representation of names in the CAP file. The CAP file is transferred onto the JavaCard device, which is then free to convert it into whatever internal representation is used for execution. Verification of the class files can be done at all stages in the process, but the properties that can be verified become

[2]Numerous utilities, such as IBM's Jax (accessible from `http://www.alphaworks.ibm.com/`) reduce the size of Java class files by removing such superfluous information. Jax also performs class hierarchy specialization [Tip and Sweeney 1997], which removes unused features from Java programs, and is orthogonal to the techniques presented in this paper.

more limited as more and more information is removed.

The low-level implementations of JavaCard systems are strictly proprietary, making it difficult to give a precise description of what internal format could be used to store Java programs. As an approximation, we use the standard CAP file format as in-memory format; information not needed after installation onto the card is assumed to have been stripped from the CAP file. Concretely, a CAP file is separated into several components, and we exclude those components not needed for execution when the CAP file format is the in-memory format. The stripped CAP file format is explained in the appendix, where a more detailed overview of the CAP file format also can be found. The stripped CAP file format is not ideal in terms of space consumption, and a realistic embedded system would probably use a more optimized format, giving a smaller memory footprint. Nonetheless, due to the lack of precise information regarding concrete embedded systems, the stripped CAP file format will serve as memory footprint measure throughout this paper. According to the experiments with stripped CAP files reported in Section 6, the bytecode takes up most of the memory footprint. We thus concentrate on reducing the size of the bytecode.

2.2 Basic-Block Compression

Looking beyond simple conversion of the Java class file into a more compact format, an often-used solution for compression is word-stream compression techniques such as Huffman encoding [Huffman 1952] or Lempel-Ziv compression [Ziv and Lempel 1978]. The bytecode of 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 low-end embedded systems, it is not even possible to decompress each method as it is invoked. Rather than storing complete 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. (The disadvantage is that decompression on-the-fly makes the overhead proportional to the program running time rather than the program size.) While such generic compression algorithms may not be optimal for the kinds of patterns found in program code [Ernst et al. 1997], they are well-known and can easily be implemented. However, because stream compression techniques are not well suited for the compression of many small, individual blocks, the expected gains in compression are limited.[3] Also, the restrictions on the amount of available RAM would impose strong restrictions on the size of the dynamic dictionary; these restrictions would have detrimental effects on the degree of compression. In addition, a significant time overhead would be associated with decompressing each basic block, slowing down the overall speed of the system to an unacceptable degree.

[3]On average Java methods are small, and basic blocks are even smaller. For example, in the programs used for experiments in Section 6, the average method length is roughly 50 bytes.

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

```
public class Rectangle {
  public Point p1, p2;
}
```

Fig. 2. Java source code for the `Point` and `Box` classes.

2.3 Code Factorization

Most Java bytecode programs contain repeated occurrences of instructions. As a simple example, consider a specific object field that is manipulated throughout a class; furthermore assume that each access to this field is performed by the same sequence of operations. Common-subexpression elimination can be used to eliminate some of this redundancy. However, this optimization only applies to the rare cases where the instruction sequences actually compute the same value.

A simple way of eliminating code redundancy is to create methods that store repeated instruction sequences. Each original sequence of instructions is replaced by a call to such a method. However, there is a space overhead for defining a method and for invoking it (three bytes per invocation). Furthermore, any changes to the local state have to be copied explicitly back and forth, introducing significant time and space overheads. The code space overhead can be reduced to a few bytes per replaced instruction sequence by using Java bytecode subroutines instead of methods. However, such subroutines are intraprocedural, making the applicability of each subroutine too limited for our purposes.

As an alternative to a pure Java solution, our proposal is to extend the instruction set of the virtual machine with instructions that can replace recurring instruction sequences. In contrast to the workstation world, JVMs for embedded systems are proprietary and are as a rule written specifically for, and manually optimized to, each system. This makes it feasible to add new features to the JVM, as long as the changes are minimal and systematic, and as long as the JVM is still able to run standard Java bytecode.

Adding a fixed set of new instructions would be a nontrivial change that would significantly increase the size of the JVM. Furthermore, if the new instructions are specific to a given program, then they would have to be replaced if a different program is to be used. Our alternative is to extend the virtual machine to read new instruction definitions from the CAP 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 by a table specific to the program being interpreted. The macro instructions can be stored in the run-time system, with very little memory overhead. With this approach, the number of new instructions is limited only by the number of instructions not used in the standard instruction set.

3. A SIMPLE EXAMPLE

As an example of our approach, we use the Java classes of Figure 2. The class `Point` represents a geometrical point, with a method that computes the distance to the

```
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 invokespecial #5
   <Method java.lang.Object.<init>()V>
 4 return

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

Fig. 3. Java bytecode for the `Point` and `Rectangle` classes.

center of the coordinate system. The class `Rectangle` represents a geometrical rectangle defined by two opposing corner points. The corresponding bytecode program is shown in Figure 3. Default constructors for both classes have been automatically introduced by the Java compiler.

In the bytecode program of Figure 3, 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. This field access yields 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 invokespecial #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. Let us now factorize the common instruction sequences identified in the program of Figure 3. Figure 4 shows the factorized bytecode program, along with the corresponding table of macros. The repeated instruction sequences for accessing fields have been factorized into macro instructions, as has the body of the constructors.

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. We give a high-level description of the algorithm used to obtain the results of this paper; implementation

```
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 Rectangle()
   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 invokespecial #5
     <Method java.lang.Object.<init>()V>
   4 return
```

Fig. 4. Factorized Java bytecode for the `Point` and `Rectangle` classes.

details can be found in our technical report [Clausen et al. 1998]. Computing the optimal set of patterns is an NP-complete problem [Garey and Johnson 1979]. Our algorithm is designed to be simple and fast, while computing a set of patterns that is sufficient for the purpose of our experiments.

Conceptually, recurring sequences of operations are abstracted by factorizing them into single units. Each sequence of bytecode instructions is called a *pattern*. Factorizing a program with respect to a pattern yields a reduced program, where each *occurrence* of the pattern has been replaced by the corresponding new instruction. We refer to a control flow branch going from code surrounding an occurrence of a pattern into this occurrence as an *incoming branch*, whereas a branch going from a pattern to the code surrounding it is referred to as an *outgoing branch*. An outgoing branch is found in all occurrences of a pattern, whereas an incoming branch may be specific to a given occurrence of a pattern.

For a given program, factorization is done in two steps. First, repetitive instruction sequences are identified as patterns. Second, the bytecode is factorized with respect to these patterns, generating new instructions on-the-fly.

4.1 Pattern Generation

To find the set of patterns with which to factorize the program, all combinations of instruction sequences occurring in the program are generated; identical sequences are treated as a single *occurrence group*. First a group of length one is created for each set of equivalent instructions. These groups are iteratively expanded, either elongating each group or splitting it to create new groups of longer, equivalent occurrences.

Pattern generation must take into account how the constant pool is represented on the embedded system, to correctly reference constants after factorization. In the CAP file format, there is a separate constant pool for each package. The virtual machine keeps track of the current package to reference constants correctly, so two occurrences of an instruction in different packages with equal constant pool indices can be considered as being equal (permitting them to be factorized into the same pattern): the virtual machine will correctly interpret the instruction in the context of the current package. Alternative strategies to deal with constants in the virtual

machine and in the factorization algorithm are discussed in Section 5.3.

We reduce the occurrence groups to avoid unfactorizable instructions in the patterns, and to avoid branch instructions that cross the boundaries of an occurrence group. The instructions `tableswitch`, `lookupswitch`, `jsr`, and `ret` are considered unfactorizable: a switch instruction has alignment constraints that makes it difficult to place inside a macro, and a subroutine instruction causes problematic intraprocedural control flow. Neither of these instruction types are used very often, so we do not consider it worth the extra complexity to factorize them. Unfactorizable instructions are removed from a pattern by splitting the pattern into two new patterns, and splitting the occurrence group accordingly. Similarly, whenever an outgoing branch leaves a pattern, the branching instruction is removed from the pattern, creating two new patterns. Next, incoming branches are checked. Since incoming branches may be the result of a single branching statement, we only remove occurrences that have an incoming branch. The first instructions of an exception handler, and the first and last instructions of a code region where exceptions are caught, are treated in the same way as targets of incoming branches.

4.2 Pattern Application

Having computed the set of patterns, we now generate the macro instructions. Macros are generated greedily by selecting the occurrence group that gives most savings first and continuing until we either run out of unused instruction codes or occurrence groups that save space. We replace each occurrence by a macro instruction and update any other occurrences that contained the replaced occurrence to reflect this change.

The number of unused instructions in the instruction set determines the possible number of new macro instructions. The number of unused instructions depends on the Java platform used; it ranges from 52 to 152 free instructions (the various JavaCard instruction sets will be discussed in Section 6.1). Representing a macro as a single byte is simple and has very little overhead. However, to overcome the limit imposed by the number of free instructions, we may wish to define macros that have a two-byte instruction length. We refer to such macros as double-byte macros with a double-byte instruction coding (as opposed to single-byte macros with a single-byte coding). Although this coding yields less size reduction than using a single-byte coding, it is still worth doing in some cases. The loss in size reduction can be minimized by assigning the double-byte coding to macros that have few occurrences. The first byte of a double-byte macro is the instruction code, and the second byte indicates an offset into a secondary table of ordinary single-byte macros. This gives room for 255 more double-byte instructions for each free single-byte instruction code used this way.

Since macros by definition are nonrecursive, the factorization program also computes the maximal intraprocedural macro nesting. This information can be used to simplify the modifications that need to be made to the JVM. The factorized bytecode replaces the unfactorized bytecode in the method bytecode component, and the macro table is placed in a custom CAP file component (see the appendix). Alternatively, the unfactorized bytecode can also be kept, and a choice between what code to be used can be made during code installation, so a JVM without support for execution of factorized code still can run the program.

5. IMPLEMENTING AN EXTENSIBLE JVM

This section describes the changes that are needed to make a standard JVM extensible. A few simple changes must be made once in the interpreter main loop.

5.1 Macro Representation

We make a standard JVM extensible by enabling execution of macro instructions. A macro is essentially defined by two values: the instruction code and 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 packages.

Macros are stored using a standard file format, enabling the modified interpreter to use macros produced by any factorization program [Clausen et al. 1998]. When transferring a factorized package into an embedded system, the macros must be transferred as well. This transfer can be done automatically, since the factorized code and the macros are already present in the CAP file for the package.

Verification of the factorized bytecode before transfer into the embedded system must also take macros into account. A trivial preprocessor could expand the macros before verification is performed. As a result, existing bytecode verifiers could be used. Verification of factorized bytecode on the embedded system is more difficult, but the properties that are typically verified at this stage are usually fairly simple, making it possible for the verifier to directly process factorized code.

5.2 JVM Main-Loop Modifications

Basically, the JVM modifications consist of enabling the interpreter to detect and subsequently call macros, dispatching based on the instruction number. The macro call is always local to a method; to enable returning to the calling instruction, a small stack must save return addresses. Thus, each method invocation stack frame must contain a fixed-size macro call stack of program counters. Since macros are nonrecursive, the maximum stack depth can be computed by the factorization algorithm along with the set of macros, and be verified by the preprocessor for the verifier.

When a macro instruction is invoked, the current value of the program counter is pushed on the macro stack. Afterward, the program counter is set to point to the first instruction of the body of the executed macro. Execution continues in the macro until either the macro return instruction `macro_end` is executed, an exception is thrown, or a return is made from the current method.

When the `macro_end` instruction is executed, the program counter is reset to the top value of the program counter stack (which is popped), and execution continues at 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, (1) the entire stack of program counters is popped, (2) the program counter is reset to the last program counter popped from the stack, and (3) control is transferred directly to the appropriate exception handler.

Table I. Instruction Set Sizes for Java Platforms

Java platform	key	instructions used	instructions free
Standard Java (EmbeddedJava and up)	`java`	203	52
JavaCard 2.1, with integer support	`jc21+i`	184	71
JavaCard 2.1, without integer support	`jc21-i`	135	120
JavaCard 2.0	`jc20`	103	152

5.3 Constant Pool Representation

We assume that the virtual machine uses the CAP file format as its in-memory format, and therefore does not resolve constants before execution, which implies that the factorization algorithm does not need to resolve constants either. For this mechanism to work correctly, the virtual machine must keep track of the current package. Given that the virtual machine implicitly keeps track of the current class, and that the package of a class can be trivially found from the class itself, this requirement does not impose any significant overhead.

As an alternative to referencing constants indirectly through the constant pool, an embedded system can resolve all constants when a package is installed onto the card. Indeed, this appears to be the purpose of the CAP file `Reference location` component (see the appendix for details). If constants are resolved globally, then the virtual machine no longer needs to keep track of the current package to correctly reference constants. However, the factorization algorithm must be modified to resolve all constants before factorization, to ensure that the factorized code can be resolved correctly during installation onto the card. A macro must only be shared between two different packages if for both packages constants referenced by the macro resolve to the same global address. We believe that performing global constant resolution would have a positive impact on the number of recurring patterns in the code and thus on the compression ratio, since instruction sequences performing the same action would be identical. However, all experiments reported in this document are performed without constant resolution.

6. PERFORMANCE EVALUATION

We have implemented factorization of standard Java class files and extended the JVM of the Harissa environment [Muller and Schultz 1999] with macro support. The Harissa environment integrates an optimizing off-line Java compiler with an interpreter; the interpreter allows execution of dynamically loaded programs. Using these tools, we have performed a number of experiments to evaluate our factorization technique. In this section, we first present our considerations for what JavaCard instruction sets to include in our experiments; then we describe the actual experiments and their results; last we provide an assessment of the results.

6.1 The Various Java(Card) Instruction Sets

There are no less than four different instruction sets to consider when working with Java low-end embedded systems. Although only two of these are used in the latest specification of JavaCard (version 2.1), we consider it interesting to measure the effectiveness of our factorization algorithm on all of these instruction sets, since factorization could be used for non-JavaCard systems.

Table I shows the different Java instruction sets. The standard Java instruction

set [Lindholm and Yellin 1996] (denoted by `java`) is used in large Java embedded systems (i.e., non-JavaCard systems) [Sun Microsystems, Inc. 1998a]. The JavaCard 2.0 instruction set (denoted by `jc20`) is a strict subset of the standard Java instruction set, where instructions for operations not supported by JavaCard systems have been removed [Sun Microsystems, Inc. 1997]. There are two versions of the JavaCard 2.1 instruction set, one with 32-bit integer support (denoted by `jc21+i`), and one without 32-bit integer support (denoted by `jc21-i`) [Sun Microsystems, Inc. 1999d]. Both JavaCard 2.1 instruction sets extend the `jc20` instruction set with a number of new instructions not found in the standard Java instruction set; these new instructions are intended to permit a more compact program representation. Many of the new instructions are parameterized and are thus more general than the macros generated by our approach; we investigate issues related to the the new JavaCard 2.1 instructions in Section 6.3.

6.2 Experiments

To test the effectiveness of the factorization algorithm and the execution speed of the resulting program, we consider the following program packages:

JavaCard 2.1 Library. The JavaCard 2.1 library classes, taken from the JavaCard 2.1 Development Kit [Sun Microsystems, Inc. 2000]. These libraries are normally placed on every JavaCard 2.1 system; they represent ideal candidates for factorization. The parts of the library that require 32-bit integer support (packages `impl` and `installer` from the `com.sun.javacard` package hierarchy) are excluded in the `jc21-i` experiments.

JavaCard Applets. The demonstration applets distributed with the JavaCard 2.1 Development Kit [Sun Microsystems, Inc. 2000].

JavaRing Applets. The sample JavaRing applets available from the iButton home page [Dallas Semiconductor Corp. 1999], updated for JavaCard 2.1 compatibility. These applets require integer support and are thus excluded from `jc21-i` experiments.

Plain JavaCard. JavaCard applets together with the JavaCard 2.1 library classes.

Full JavaCard. JavaCard applets and JavaRing applets together with the JavaCard 2.1 library classes.

JES. Sun's Java Embedded Server 1.0, a full-featured Web-server for embedded systems [Sun Microsystems, Inc. 1999a].

CaffeineMarks. Microbenchmark suite designed specifically for Java [Pendragon Software 1997] (the embedded version).

Javac. JavaSoft's JDK 1.0.2 Java compiler [Sun Microsystems, Inc. 1998c], packages `acm`, `java`, `javac`, and `tree` from the `sun.tools` package hierarchy.

Of these program packages, the JavaCard library, JavaCard applets, and JavaRing applets were only tested with the various JavaCard instruction sets, and tests including the JavaRing applets always use instruction sets with integer support. The last three tests (JES, CaffeineMarks, and Javac) were all done with the full Java instruction set.

Table II shows the size of the bytecode in bytes and the total memory footprint, before and after factorization. The size of the bytecode is shown with and without

Table II. Size in Bytes of Bytecode and Memory Footprint before and after Factorization (†: Estimated memory footprint, ‡: excluding certain classes; see text.)

		Unfactorized				Factorized					**Compression**	
	Instruction set	Memory footprint	Bytecode size	Average method size	Bytecode part of memory footprint	Macros defined	Bytecode size	Macro table size	Bytecode + macro table size	Memory footprint	**Bytecode** (after/before)	**Memory footprint** (after/before)
JavaCard Library	`jc20`	16071	11241	31	69.9%	186	8221	680	9276	14106	**82.5%**	**87.8%**
	`jc21-i`‡	4941	4390	16	88.8%	119	2944	400	3582	4133	**81.6%**	**83.6%**
	`jc21+i`	14283	9453	26	66.2%	96	7560	354	8109	12939	**85.8%**	**90.6%**
JavaCard Applets	`jc20`	5624	4067	75	72.3%	150	2325	586	3211	4768	**79.0%**	**84.8%**
	`jc21-i`	4886	3329	61	68.1%	119	1977	446	2661	4218	**79.9%**	**86.3%**
	`jc21+i`	4886	3329	61	68.1%	67	2260	329	2723	4280	**81.8%**	**87.6%**
JavaRing Applets	`jc20`	7314	6305	106	86.2%	150	3402	554	4256	5265	**67.5%**	**72.0%**
	`jc21+i`	6494	5485	92	84.5%	77	3335	363	3855	4864	**70.3%**	**74.9%**
Plain JavaCard	`jc20`	21695	15308	37	70.6%	202	11216	858	12481	18868	**81.5%**	**87.0%**
	`jc21-i`‡	9827	7719	24	78.5%	149	5253	607	6161	8269	**79.8%**	**84.1%**
	`jc21+i`	19169	12782	31	66.7%	83	10418	383	10970	17357	**85.8%**	**90.5%**
Full JavaCard	`jc20`	29009	21613	46	74.5%	225	15400	1027	16880	24276	**78.1%**	**83.7%**
	`jc21+i`	25663	18267	39	71.2%	102	14327	592	15126	22522	**82.8%**	**87.8%**
JES†	`java`	206143	153133	53	74.3%	102	115374	1667	117248	170258	**76.6%**	**82.6%**
Caf.Mark†	`java`	4067	3021	37	74.3%	68	1882	439	2460	3506	**81.4%**	**86.2%**
Javac†	`java`	121189	90025	79	74.3%	127	72022	606	72885	104049	**81.0%**	**85.9%**
Averages				50.9	74.3%						79.7%	84.7%

the macro definition code included. The average method size is shown for reference, as is the percentage of memory footprint taken by the bytecode. The maximal macro stack nesting did not exceed four on any of these tests, and the factorization for all the tests reported in Table II was performed in less than 10 minutes on a 233MHz Pentium II machine. The memory footprint is given for stripped CAP files (as explained in the appendix). We cannot give a precise figure for the memory footprint of the three non-JavaCard programs, since they cannot be converted to CAP files (they all use non-JavaCard functionality). We observe that, on average, across all instruction sets, the bytecode accounts for 74.3% of the memory footprint. This figure is used as an estimate when reporting the memory footprint for these three test programs. As mentioned earlier, the factorization of programs in `jc21-i` instruction set is sometimes done on a more limited set of classes (JavaCard Library), or not included at all (JavaRing Applets and Full JavaCard).

We expect that the higher the number of unused instructions in an instruction set, the better the compression ratio. It can be seen that consistently across all of the JavaCard experiments, the `jc20` and `jc21-i` instruction sets give rise to better compression ratios than the `jc21+i` instruction set, due to the higher number of unused instructions. There is, however, no consistent difference between the compression obtained for the `jc20` and `jc21-i` instruction sets, which we attribute to the small difference in unused instructions. For the `jc20` and `jc21-i` instruction sets, the footprint is reduced on average to roughly 84% of its original size, whereas the reduction is closer to 86% of the original size for the `jc21+i` instruction set. The compression ratios for the Java programs are relatively high compared to the JavaCard programs, when taking into account the lower number of unused instructions.

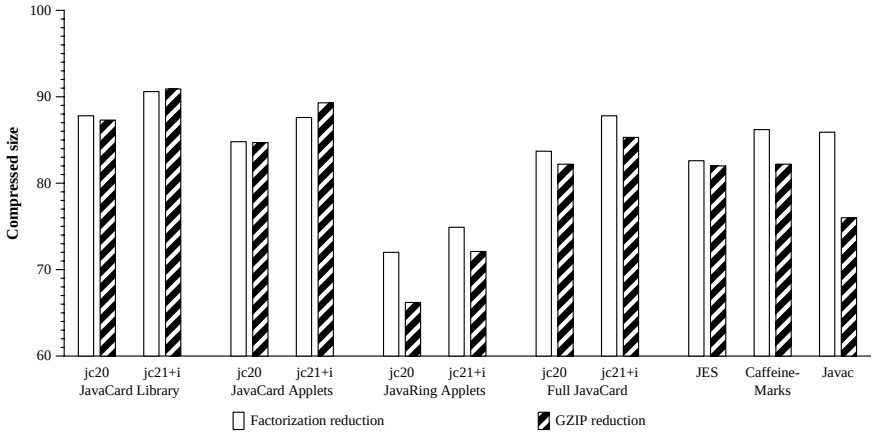

Fig. 5. Decrease in memory footprint by factorization and compression.

Table III. Factorization Options (factorization of some JavaCard 2.0 programs, memory footprint estimated, bytecode size includes macro table)

	bytecode size		memory footprint	
	bytes	effect	bytes	effect
unfactorized size	10754	—	15071	—
all options on (no CSE)	7733	0.0%	12050	0.0%
2-byte macros, nesting (no branches, no CSE))	7881	+1.9%	12198	+1.2%
2-byte macros, branches (no nesting, no CSE)	7928	+2.4%	12245	+1.6%
2-byte macros (no nesting, no branches, no CSE)	8006	+3.4%	12323	+2.2%
nesting, branches (no 2-byte macros, no CSE)	7826	+1.2%	12143	+0.8%
unfactorized size, CSE (vs. "unfactorized size")	10721	−0.3%	15038	−0.2%
all options on, CSE (vs. "all options on")	7748	+0.2%	12065	+0.1%

These programs were written with larger systems in mind, so it seems likely that the source code may contain more redundancy giving rise to more opportunities for factorization.

Factorization vs. `gzip` *Comparison.* 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 obtained with the standard Unix compression tool `gzip`. To use a `gzip`-like algorithm to uncompress methods in a JVM, each basic block would have to have been compressed individually using a global static dictionary, as was described in Section 2.2. To loosely estimate the size reduction obtained using this technique, we compress the bytecode of each method separately and subtract the overhead from the 20-byte header and file name. The compression ratios are shown in Figure 5. Factorization compression is roughly comparable to that of `gzip`, although `gzip` in general performs slightly better than factorization.

Assessment of Factorization Algorithm Features. The factorization algorithm factorizes out code containing branches, allows macros to be defined in terms of other macros, and permits the use of two bytes for defining a macro instruction code. However, it is not obvious how much additional compression is provided by

Table IV. Benchmarks Comparing Normal Execution with Macro Execution

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

these features. Also, some standard optimizations such as common-subexpression elimination (CSE) tend to reduce code size, which might be cumulative with factorization. We test the advantage of each of the factorization algorithm features and the result of applying CSE in terms of additional compression, as illustrated in Table III. All experiments are performed in the `jc20` instruction set, and we perform CSE using the Cream bytecode optimizer [Clausen 1997]. Due to limitations of the Cream implementation, we cannot use the "Full JavaCard 2.1" benchmark. As an alternative, we use the JavaCard 2.0 library (taken from the JavaCard 2.0 Reference Implementation [Sun Microsystems, Inc. 1998b]) together with the JavaCard applets from our previous tests and the Visa Open Platform Card Version 1.0 implementation for Applet Developers [Visa International Service Association 1998] (a JavaCard 2.0-only library). Macros containing other macros offer the greatest advantage, followed by macros containing branch instructions, and finally double-byte macros. As for CSE, it reduces the unfactorized program size, but apparently conflicts with the factorization algorithm, and causes an increase in the size of the factorized program.

Run-time Overhead of Factorized Code. To estimate the run-time speed overhead of using macros, we have performed two tests with the modified Harissa interpreter, both on Pentium (100MHz Dell Pentium) and SPARC (SPARC Station 5) architectures. Due to limitations in the Harissa interpreter, we were unable to run the Java Embedded Server. 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 IV. There is virtually no slowdown, as 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 microbenchmarks (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 slowdown of 19% on the Pentium and 27% on the SPARC. Code locality is strongly affected by factorization, which might have had a negative effect on our benchmarks. However, code locality is only an issue on systems with a cache, and most low-end embedded systems have a flat memory model.

6.3 Assessment

We have chosen the approach of generating bytecode macros over that of adding fixed instructions to the JVM. With the JavaCard 2.1 specification, the choice was made to add new, fixed instructions to the JVM, offering a significant advantage in terms of reduced code size beyond what can be achieved using factorization, as

was illustrated by our experiments. Naturally, factorization can still be applied to the extended instruction set, further reducing code size.

The additional compression offered by the JavaCard 2.1 instruction sets indicates that it would be worthwhile to allow parameterized macros. Parameterization could be in the form of a fixed number of instructions (possibly themselves macros), or in the form of arguments to bytecode instructions. Parameterization in the form of bytecode instructions should allow even more sharing of macro definitions, further improving compression. Parameterization in the form of bytecode instruction arguments would permit most JavaCard 2.1 instructions to be expressed in terms of a macro over JavaCard 2.0 instructions, allowing factorization to give compression comparable to that offered by the combination of factorization and the JavaCard 2.1 instruction set. Macros have a significant advantage over additional fixed instructions: a macro-enabled JVM is simpler to implement and takes up less ROM space than a JVM with a much larger instruction set (such as the JavaCard 2.1 instruction set).

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 [Fraser et al. 1984]. They get an average compression factor of 7%. They factorize local branches, use parameterized patterns, and implement a cross-jumping technique to exploit merging code sequences. Lefurgy, Bird, Chen, and Mudge replace common sequences of instructions with a single instruction macro [Lefurgy et al. 1997]. Compression is done on the instructions sets of the PowerPC, ARM, and i386 processors. However, minor hardware modifications are required for the compressed code to execute, contrary to the case for Java bytecode where only the virtual machine needs to be modified. The average compression rates obtained are 39%, 34%, and 26%, respectively.

Ernst et al. describe compression of code, both for transmission and for execution [Ernst et al. 1997]. They obtain the same compression ratio as `gzip` for 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 most low-end embedded systems.

Proebsting describes a C interpreter using "superoperators" [Proebsting 1995]. These kinds of operators can be automatically inferred from the tree-like intermediate representation produced by `lcc` [Fraser and Hanson 1991a; 1991b]. 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.

As an alternative to the standard ZIP-based JAR format, Bradley, Horspool, and Vitek present the JAZZ format, which compresses collections of classes, improving the compression ratio by reorganizing data, so that similar data are compressed together using standard compression techniques [Bradly et al. 1998]. Compression is about 75%, as opposed to only 50% for the standard JAR format. Pugh goes be-

yond this result by employing more aggressive compression techniques, completely reorganizing the class file layout, and employing dedicated compression techniques to each kind of data [Pugh 1999]. The resulting programs are on the average half the size of JAZZ-compressed programs. Although achieving superior compression ratios, neither of these techniques are appropriate for low-end embedded systems, both requiring more space and computation during uncompression than is available. Rayside, Mamas, and Hons present an alternative class file format that targets embedded systems that use the standard Java class file format as in-memory format [Rayside et al. 1999]. They primarily focus on reducing the size of the constant pool, while keeping it in a directly accessible format; class files are on the average reduced to 75% of their original size. While the class pool dominates the size of a Java program in the standard class file format, this is not true for low-end embedded systems. In effect, their technique is only relevant for embedded systems larger than those targeted by our factorization approach.

Franz and Kistler present SLIM binaries as an alternative to Java bytecode [Franz and Kistler 1997]. 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 low-end embedded systems.

Liao et al. optimize the selection of instructions on embedded DSP processors that have complex instruction sets defining compound operations [Liao et al. 1995]. Unlike the factorization process proposed in this paper where we generate new instructions for a given program, Liao et al. compile high-level programs to a given instruction set.

8. CONCLUSION AND FUTURE WORK

We have implemented factorization for Java bytecode. It handles nontrivial programs, and reduces the overall memory footprint of bytecode programs for low-end embedded systems to about 85% of their original size. Our factorization algorithm seems to compare satisfactorily with traditional compression, as embodied by `gzip`. The execution time overhead of introducing macros is between 2% and 30%. In particular, factorization can trivially provide better compression ratios than the pure-Java solutions proposed earlier (methods and subroutines) with a smaller run-time overhead, making it the embedded system engineer's preferred choice.

Inspired by the strong separation between packages in the JavaCard 2.1 specification, we are currently investigating the option of having package-local macro tables. This has several advantages, chiefly that dynamically loaded packages can be prefactorized using a private set of macros ranging over all free instructions. Also, we are investigating how to permit macros to take arguments, as described in Section 6.3. An initial assessment of the effect of having parameterized local variable numbers for load and store instructions looks promising, but much work needs to be done to develop an efficient factorization algorithm. Finally, the factorization

algorithm and the extensible JVM are by design independent of one another, allowing transparent replacement of our factorization algorithm with a new one that provides better compression. We consider the development of such factorization algorithms future work.

APPENDIX

To the authors' knowledge, all existing JVMs for low-end embedded systems are proprietary, making it difficult to assess the impact of bytecode compression on the memory footprint of a Java program. The JavaCard 2.1 CAP format is publicly specified [Sun Microsystems, Inc. 1999d], but it is not necessarily an appropriate format for the internal JVM data structures. It would be inaccurate to use a metric based on an existing JVM for a nonembedded environment (such JVMs usually are optimized for speed rather than space). For the lack of better concrete information we choose to use the CAP format as the basis for our memory footprint.

A CAP file consists of a number of *components* that in combination describe a complete JavaCard package [Sun Microsystems, Inc. 1999d]:

`Header.` General information about this CAP file and the package it defines.

`Directory.` Lists the size of each of the components defined in this CAP file, including any custom components.

`Applet.` Contains an entry for each of the applets defined in this package.

`Import.` Lists the set of packages imported by classes in this package.

`Constant pool.` Contains an entry for each of the classes, methods, and fields referenced by elements in the `Method` component of this CAP file.

`Class.` Describes each of the classes and interfaces defined in this package.

`Method.` The method component describes each of the methods declared in this package, including bytecode and exception handlers, but excluding `<clinit>` methods and interface method declarations.

`Static field.` Contains all the information required to create and initialize an image of all of the static fields defined in this package.

`Reference location.` Represents lists of offsets into the bytecode of the `Method` component to operands that contain indices into the constant pool array of the `Constant pool` component.

`Export.` Lists all static elements in this package that may be imported by classes in other packages.

`Descriptor.` Provides sufficient information to parse and verify all elements of the CAP file (optional component).

In addition, we use the CAP file custom component mechanism to store the macro table generated by the factorization algorithm in its own component. CAP files are actually generated in the Java JAR format, but will of course be decompressed during transfer to the JavaCard system. Thus, the memory footprint can be computed as the sum of the sizes of the decompressed CAP file components.

Not all of these components need to be included in the stripped CAP file format which we use to compute the memory footprint of a JavaCard system. The `Descriptor` component is only needed for verification, and is explicitly listed in

the JavaCard 2.1 Virtual Machine Specification as being optional. The `Reference location` component has no obvious use except to perform global resolution of constants into absolute addresses. Since we lack more precise information, we assume that the CAP file format is the in-memory format used for execution. Hence, constants are not resolved into addresses, and the `Reference location` component is not needed. Thus, we exclude the `Descriptor` and `Reference location` components from our stripped CAP file format.

ACKNOWLEDGMENTS

We wish to thank Peter Chang for his proofreading assistance, and the anonymous TOPLAS referees for their helpful comments.

REFERENCES

Antonioli, D. N. and Pilz, M. 1998. Analysis of the java class file format. Tech. Rep. ifi-98.04, Department of Computer Science, University of Zurich. Apr. 28.

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

Bradly, Q., Horspool, R. N., and Vitek, J. 1998. Jazz: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*. Toronto, Ontario, 294–302.

Clausen, L. R. 1997. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice & Experience 9,* 11 (Nov.), 1031–1045.

Clausen, L. R., Schultz, U., Consel, C., and Muller, G. 1998. Java bytecode compression for embedded systems. Research Report 3578, INRIA, Rennes, France. Dec.

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

Dallas Semiconductor Corp. 1999. Java-Powered Ring Download Site. URL: `http://www.ibutton.com/jiblet`.

Ernst, J., Evans, W., Fraser, C. W., Lucco, S., and Proebsting, T. A. 1997. Code compression. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices, vol. 32, 5. ACM Press, New York, 358–365.

Franz, M. and Kistler, T. 1997. Slim binaries. *CACM 40,* 12 (Dec.), 87–94.

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

Fraser, C. W. and Hanson, D. R. 1991b. A retargetable compiler for ANSI C. *SIGPLAN Notices 26,* 10 (Oct.), 29–43.

Fraser, C. W., Myers, E. W., and Wendt, A. L. 1984. Analysing and compressing assembly code. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*. ACM, 117–121.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Gosling, J., Joy, B., and Steele Jr., G. L. 1996. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA.

Huffman, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute for Radio Engineering 40,* 9 (Sept.), 1098–1101.

Lefurgy, C., Bird, P., Chen, I., and Mudge, T. 1997. Improving code density using compression techniques. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. IEEE Computer Society TC-MICRO and ACM SIGMICRO, Research Triangle Park, North Carolina, 194–203.

Liao, S., Devadas, S., Keutzer, K., and Tjiang, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the International Conference on Computer Aided Design*. IEEE Computer Society Press, Los Alamitos, Ca., USA, 393–401.

Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA.

Muller, G. and Schultz, U. 1999. Harissa: A hybrid approach to Java execution. *IEEE Software*, 44–51.

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

Proebsting, T. A. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*. ACM Press, San Francisco, California, 322–332.

Pugh, W. 1999. Compressing Java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*. Atlanta, Georgia, USA, 247–258.

Rayside, D., Mamas, E., and Hons, E. 1999. Compact java binaries for embedded systems. In *Proceedings of CASCON'99*. Toronto, Ontario, 1–14.

Sun Microsystems, Inc. 1997. *JavaCard 2.0 Language Subset and Virtual Machine Specification*, 1.0 ed. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1998a. *Embedded Java Specification*, 1.0 ed. Sun Microsystems, Inc. URL: `http://java.sun.com/products/embeddedjava/note.html`.

Sun Microsystems, Inc. 1998b. *Java Card API 2.0 Reference Implementation*, Developer Release 2 ed. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

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

Sun Microsystems, Inc. 1999a. *Java Embedded Server*. Sun Microsystems, Inc. URL: `http://www.sun.com/software/embeddedserver`.

Sun Microsystems, Inc. 1999b. *JavaCard 2.1 Application Programming Interface Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1999c. *JavaCard 2.1 Runtime Environment (JCRE) Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 1999d. *JavaCard 2.1 Virtual Machine Specification*. Sun Microsystems, Inc. URL: `http://www.javasoft.com/products/javacard`.

Sun Microsystems, Inc. 2000. JavaCard 2.1 development kit. URL: `http://www.javasoft.com/products/javacard`.

Tip, F. and Sweeney, P. F. 1997. Class hierarchy specialization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*. ACM SIGPLAN Notices, vol. 32, 10. ACM Press, New York, 271–285.

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

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable rate encoding. *IEEE Transactions on Information Theory 24*, 530–536.

Received February 1999; revised September 2000; accepted March 2000