Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments

Dimitris Saougkos, George Manis, Konstantinos Blekas, *Member*, *IEEE*, and Apostolos V. Zarras, *Member*, *IEEE*

Abstract—Pattern-based Java bytecode compression techniques rely on the identification of identical instruction sequences that occur more than once. Each occurrence of such a sequence is substituted by a single instruction. The sequence defines a *pattern* that is used for extending the standard bytecode instruction set with the instruction that substitutes the pattern occurrences in the original bytecode. Alternatively, the pattern may be stored in a dictionary that serves for the bytecode decompression. In this case, the instruction that substitutes the pattern in the original bytecode serves as an index to the dictionary. In this paper, we investigate a bytecode compression technique that considers a more general case of patterns. Specifically, we employ the use of an advanced pattern discovery technique that allows locating patterns of an arbitrary length, which may contain a variable number of wildcards in place of certain instruction opcodes or operands. We evaluate the benefits and the limitations of this technique in various scenarios that aim at compressing the reference implementation of MIDP, a standard Java environment for the development of applications for mobile devices.

Index Terms—Java, compression (coding).

1 Introduction

The Java language has become a dominant means for the realization of embedded and mobile computing environments. The main feature of Java that led to this is its *portability*. More specifically, the compilation of Java applications results in device independent code, generated in terms of a standard format, called Java *bytecode*. The Java bytecode can then execute on top of different device-specific Java Virtual Machines (JVMs), which take charge of translating the bytecode into device-specific machine code.

The memory limitations imposed by embedded and mobile devices certainly constrain the set of applications that may possibly execute on top of them. Confronting the aforementioned issue fosters research in two orthogonal directions. The first one concerns the reduction of the physical size and cost of memory chips, while the second one involves reducing the size of the code of embedded applications. Advances in both of these research directions are equally valuable. No matter how much we increase the amount of available memory, there will always be more demanding applications. Similarly, even if we manage to diminish the size of embedded and mobile applications, we may always require concurrent execution of as many of them as possible.

Many significant research efforts have already been made toward the generation of compressed code [1]. However, most of these efforts involve the compression of either machine or assembly code. Among the few approaches that focus on the case of Java bytecode, we have those proposed in [2], [3], and [4]. In [2], the authors examine various approaches for bytecode compression, relying on Huffman codes and Markov chains. In [3], bytecode compression is based on the use of canonical Huffman codes and the generation of fast decoders. In [4], the approach followed is based on the discovery of instruction sequences that occur more than once within the Java bytecode. Each sequence of instructions defines a *pattern*. Each pattern occurrence is substituted by a single instruction that is called a *macro*.

However, a pattern, in its broadest sense may have the following characteristics:

1. It may be of an *arbitrary* length.
2. It may contain *wildcards* in place of a particular opcode or operand. Hereafter, we use the term *parameterized* to refer to patterns that contain a variable number of wildcards. Respectively, we use the term *nonparameterized* to refer to patterns that do not contain wildcards.

So far, existing approaches for Java bytecode compression do not deal with the aforementioned generic form of patterns. This fact is recognized in [4], where the authors further highlight the need for more sophisticated pattern discovery techniques. *The main contribution of this paper is to assess the use of such a technique in the context of Java bytecode compression*. Specifically:

- We customize a well-known pattern discovery technique, called agglomerative clustering [5], [6], toward the identification of parameterized and

• *The authors are with the Computer Science Department, University of Ioannina, PO Box 1186, GR 45110, Greece.*
E-mail: {dsaougkos, manis, kblekas, zarras}@cs.uoi.gr.

Manuscript received 1 Oct. 2005; revised 16 Apr. 2006; accepted 15 Mar. 2007; published online 27 Mar. 2007.
Recommended for acceptance by T. Ball.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0261-1005.
Digital Object Identifier no. toolbars 10.1109/TSE.2007.1021.

nonparameterized patterns within a given Java bytecode. The proposed technique allows discovering patterns of an arbitrary length, which may contain a variable number of wildcards.

- We assess the advantages and the limitations of the aforementioned technique in various scenarios that aim at compressing MIDP, a standard Java environment that supports the development of applications for mobile devices. The main feature of the parameterized pattern discovery technique is that it allows finding a large variety of patterns that can be combined to obtain better bytecode size reduction. This, however, is also its main limitation. Exploring the large variety of patterns toward finding a combination that gives a good bytecode size reduction is a complex task. Given this fact, in our assessment:
  1. We employ/compare two heuristic methods toward the combination of patterns.
  2. Based on the two heuristics, we investigate the impact of using patterns that contain a variable number of wildcards for the compression of Java bytecode. To this end, we compress MIDP using patterns that contain a variable number of wildcards, patterns that do not contain wildcards, and patterns that contain a fixed number of wildcards, and we perform a comparative study of the results.
  3. Moreover, we examine the impact of the increasing length of the patterns in the compression of Java bytecode.
  4. Finally, we study the behavior of the decompression overhead introduced by the examined technique.

The remainder of this paper is structured as follows: In Section 2, we discuss related work. In Section 3, we introduce a typical nonparameterized pattern discovery technique, followed by the advanced parameterized pattern discovery technique that we investigate. Then, we highlight the benefits of the parameterized pattern discovery technique, as opposed to the nonparameterized one, with respect to a number of motivating examples. In Section 4, we introduce the complementary heuristic techniques, which serve for handling the complexity of combining large numbers of parameterized and nonparameterized patterns. Moreover, we present the experimental results we obtained in the case of MIDP. Finally, in Section 5, we summarize the contribution of this paper and point out the future directions of this work.

2 Related Work

Code compression techniques can be divided into two major categories [7]. The first one aims at producing a reduced-size *wire code* that can be transmitted to the CPU as fast as possible. In this case, what matters is achieving the best possible compression. For the particular case of Java, there have been several approaches for the construction of wire code. Among the prominent ones, we have JAZZ [8], an alternative to the standard JAR format, and Slim Binaries [9], an alternative to the standard bytecode format. Moreover, we have the approach proposed in [2], where the authors discuss methods for reducing the size of the constant pool. In [10], Pugh also discusses interesting ideas toward a wire code format that aims at reducing the size of collections of class files. Finally, in [11], Tip et al. investigate techniques for the removal of redundant class file attributes and methods, along with techniques for constant pool compression and class hierarchy transformations.

The second category of techniques aims at producing a reduced-size *interpretable code* that can be stored and executed without being fully decompressed. In this case, what is important is the reduction of the overall amount of memory required for the execution of the application. The various techniques proposed for the generation of interpretable code rely either on Huffman codes and arithmetic coding or on the identification of patterns.

The use of Huffman codes [12], [13] and arithmetic coding [14] in code compression aims at shortening a sequence of instructions by mapping them into the shortest possible sequence of bits. Huffman-based techniques have been criticized for their decompression complexity. However, in [3], Latendresse and Feeley propose an approach for fast Huffman decoding. Their approach focuses on virtual instructions and is evaluated for the case of Java bytecode. In [2], Rayside et al. also investigate the use of Huffman codes for compressing Java bytecode.

Pattern-based techniques for interpretable code compression focus on the identification of multiple occurrences of instruction sequences within an application. Each such sequence is called a *pattern*. Patterns are usually stored in a dictionary and their occurrences in the original code are substituted in the compressed code by dictionary indexes. Indexes are frequently called *macros* and their size is usually the size of a single instruction. Specifically, in [15], Cooper and McIntosh propose a dictionary-based approach that can be applied in RISC intermediate representations. This approach allows searching for nonparameterized patterns of an arbitrary length. Nonparameterized patterns of an arbitrary length are also used by the IBM CodePack compressor [16], which is deployed on PowerPCs. This particular technique originates from the one proposed by Lefurgy et al. in [17].

In [18], Debray et al. present an approach that deals with parameterized patterns. The approach relies on the discovery of similar basic blocks (i.e., code fragments with a unique entry and a unique exit point). As a similarity metric, the authors use a fingerprint function. Moreover, several classical compiler optimization techniques (e.g., dead code elimination) are applied. In [19], De Sutter et al. go one step further by searching for similar procedures and code regions. Again, the fingerprint function is used as a similarity metric. Parameterized procedures are also used in Krinke's work [20]. Moreover, in BRISC [7], Ernst et al. propose a dictionary-based technique that relies on patterns consisting of two instructions. The patterns may further contain wildcards in place of instruction operands. Parameterized pattern discovery is also used in [21]. This approach focuses on the identification of similar basic blocks in ARM code. For a set of similar blocks, a

representative function is built. The representative function comprises predicated instructions, corresponding to the differences met among the original basic blocks. In [22], Evans and Fraser also propose an approach that employs a sort of parameterized pattern. The proposed system accepts as input a grammar and a training set of programs and produces an expanded grammar that allows shorter derivations of the training and other similar programs. The rules of the expanded grammar can be seen as parameterized patterns, leading to different derivations. In [20], legacy source code is transformed using procedural abstraction, so that it becomes more understandable, maintainable and small. Extracted sets of statements form procedures and the extracted code is replaced with procedure calls. Similarly, in [23], the authors identify similar segments of source code based on program dependence graphs.

Considering the particular case of Java bytecode, in [4], Clausen et al. propose a technique which considers nonparameterized patterns of an arbitrary length. Their main contribution is that they do not directly use the notion of a dictionary; instead, they specialize the JVM with more complex instructions that actually realize the execution of the patterns. Then, the patterns can be substituted in the original code by the new instructions, thus reducing the size of the original code. In this paper, we investigate a dictionary-based bytecode compression technique, which goes one step beyond [4] by considering both parameterized and nonparameterized patterns of an arbitrary length. Currently, the technique is applied in bytecode basic blocks, but it could as well be used in the case of whole Java methods. The technique that we investigate is more fine-grained with respect to the other parameterized pattern discovery techniques discussed in this section. It does not search for similarity among whole basic blocks (or procedures). Instead, it searches inside the basic blocks for similar instruction sequences of an arbitrary length. To cope with the increased complexity of this task, we employed the agglomerative clustering algorithm [5], [6].

3 Java Bytecode Compression Process

The bytecode of a compiled Java program is a sequence of binary-encoded JVM instructions. Each instruction consists of an opcode and possibly a number of operands. The size of instruction opcodes or operands is 1 byte. As an example, consider the Java program given in Fig. 1a. This simple program comprises a class of objects that represent vectors in the three-dimensional space. Each vector is characterized by three coordinates (x, y, z, attributes) and provides a method, called `distance`, which calculates the euclidean distance norm, $d = \sqrt{x^2 + y^2 + z^2}$, for the vector.

The compiled bytecode for the `distance` method of our program is given in Fig. 1b. The overall size of the sequence is 34 bytes. In the remainder of this section, we use the example of Fig. 1b to highlight the main steps of the nonparameterized and the parameterized compression approaches, discussed in this section. In both cases, the overall compression process consists of the following steps:

```
class xyz {
    public float x,y,z;
    public xyz(){
        x=0;
        y=0;
        z=0;
    }
    public xyz(float x1,float y1,float z1){
        x=x1;
        y=y1;
        z=z1;
    }
    public double distance(){
        return Math.sqrt(x*x+y*y+z*z);
    }
    public static void main(String Args[]){
        xyz z=new xyz(10,20,10);

        System.out.println
            ("Distance="+z.distance());
    }
};
```

(a)

```
Compiled from "xyz.java"
 class xyz extends java.lang.Object
 public double distance();
  Code:
    1:   aload_0
    4:   getfield     0 2
    5:   aload_0
    8:   getfield     0 2
    9:   fmul
    10:  aload_0
    13:  getfield     0 3
    14:  aload_0
    17:  getfield     0 3
    18:  fmul
    19:  fadd
    20:  aload_0
    23:  getfield     0 4
    24:  aload_0
    27:  getfield     0 4
    28:  fmul
    29:  fadd
    30:  f2d
    33:  invokestatic 0 5
    34:  dreturn
```

(b)

Fig. 1. A simple Java program. (a) Source code. (b) Compiled bytecode for the `distance` method.

1. The Java bytecode is segmented into *basic blocks*.
2. A pattern discovery technique is used for the identification of patterns in the basic blocks of the bytecode.
3. The resulted patterns are collected; possible combinations of patterns are examined and, for each one of them, the corresponding bytecode size reduction is calculated.
4. Finally, the combination of patterns that gives the highest bytecode size reduction is selected and used for the generation of the compressed code.

3.1 Nonparameterized Pattern Discovery

The nonparameterized pattern discovery can be reduced into a simple string search problem. Specifically, consider a finite set $\Sigma = \{c_1, \ldots, c_\Omega\}$ consisting of $\Omega = |\Sigma|$ individual characters. An arbitrary string over the set Σ is any sequence $S_j = \{s_{jk}\}_{k=1}^{L_j}$ of length L_j, where $s_{jk} \in \Sigma$ denotes the character at the kth position of the sequence S_j. Let $\mathcal{S} = \{S_1, \ldots, S_M\}$ be a set of M sequences of length $L_1, \ldots, L_M$, respectively. In our case, $\mathcal{S}$ denotes the set of

basic blocks of the Java bytecode, identified during the first step of the code compression process. Then, the pattern discovery amounts to finding common substrings that are repeated in the sequences of $\mathcal{S}$. Suppose that we search for substrings of a variable length $k = 2, \ldots, K$. Then, to locate them, we perform the following tasks:

1. For each $k = 2, \ldots, K$, obtain the collection $X_k = \{x_i\}_{i=1}^{n_k}$ of k-length substrings of the sequences that belong in $\mathcal{S}$ by sliding a window of size k in each sequence $S_j \in \mathcal{S}$. For every S_j, the resulted number of substrings is $L_j - k + 1$. Hence, the cardinality of X_k is $n_k = |X_k| = \sum_{j=1}^{M} \{L_j - k + 1\}$.
2. Then, the set of nonparameterized patterns P is calculated by searching within each collection X_k for identical substrings. The union of the identical substrings obtained during this step constitutes P.

To accomplish the first of the above tasks we have to perform $\sum_{k=2}^{K} \sum_{j=1}^{M} \{L_j - k + 1\} = \sum_{k=2}^{K} \{|X_k|\}$ sliding steps. Moreover, to accomplish the second task, we have to perform an overall total of $\sum_{k=2}^{K} \sum_{i=0}^{|X_k|-1} \{|X_k| - i - 1\} = \sum_{k=2}^{K} \{|X_k| \times (|X_k| - 1)/2\}$ substring comparisons.

All the possible combinations of the patterns retrieved are $2^{|P|}$. The subset of P that gives us the highest bytecode size reduction is stored in the dictionary. The dictionary we use is actually a table of characters. Each table element is used to hold a pattern instruction opcode or operand. Moreover, there are table elements that contain a special character used to signify the end of a pattern; hereafter, we use the term `END_OF_PATTERN` to refer to that character. The occurrences of each pattern are substituted in the original code by a character (i.e., one byte) that indexes the dictionary element that contains the first byte of the pattern. For the selection of the indexing characters we employ the typical approach of Clausen et al., which amounts to using unused bytecode instruction opcodes [4]. Different standard Java platforms for embedded systems comprise unused instruction opcodes, whose number ranges from 52 to 152. As discussed by Clausen et al., the number of unused opcodes limits the number of patterns that can be used. However, this problem can be alleviated by using a second character, along with the ones that correspond to unused instruction opcodes.

Getting to our example scenario, the bytecode for the distance method (Fig. 1b) constitutes a basic block as it does not comprise any branch operations. Alphabet Σ consists of 12 characters that encode the various opcodes (e.g., `getfield`, `aload_0`, etc.) and operands (e.g., `0`, `2`, `3`, `4`, `5`) used in the basic block. The basic block contains patterns of length 4. Since the overall size of the basic block is 34 characters (i.e., 34 bytes), the set of substrings of length 4 for this basic block contains 31 elements. The set of patterns that results from the 31 substrings includes the three patterns given in Fig. 2a.

Let us consider the first of the three patterns. When stored in the dictionary the required space is 4 bytes, plus one more byte for the `END_OF_PATTERN` character. Since this pattern appears twice in the examined Java bytecode, 2 bytes are needed in the compressed bytecode to index into the dictionary the position of the pattern. Hence, the gain from substituting the occurrences of the first pattern in the original bytecode is

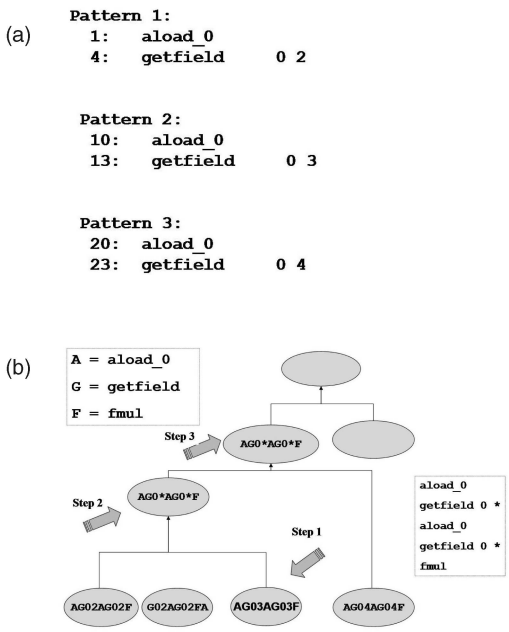


Fig. 2. Patterns discovered for the `distance` method. (a) Nonparameterized patterns. (b) Parameterized patterns.

$$2\ occurences \times 4\ bytes - (5\ dictionary\ bytes + 2\ indexing\ bytes) = 1\ byte.$$

Thus, compressing the bytecode with respect to the first pattern saves us 1 byte. If we repeat the same procedure for the other two patterns, we can save two more bytes and the final bytecode will be 31 bytes. Therefore, the bytecode size reduction obtained is 8.82 percent. As we demonstrate in the following subsection, this is much smaller than the reduction obtained by using the parameterized pattern discovery technique.

3.2 Parameterized Pattern Discovery

The parameterized pattern discovery technique is actually an extension of the nonparameterized one. It starts from the point where we have already identified the different collections $X_2, \ldots X_K$ of substrings of the sequences that belong in $\mathcal{S}$. Following, the discovery of parameterized patterns can be viewed as a *clustering problem* in the sense of searching for disjoint subsets (clusters) in each set X_k that are characterized by a high degree of similarity among the samples that they enclose. Several algorithms have been proposed for clustering discrete data [24], [5], [6]. From among them, we selected *agglomerative clustering* (AC) [5], [6], which is a hierarchical approach that relies on the bottom-up generation of a treelike structure of clusters.

AC is performed for every X_k toward obtaining a subset of candidate patterns P_k. Specifically, AC starts with a set of $|X_k|$ clusters (leaf nodes), each one containing one bytecode substring x_i from the set X_k. Then, a multinomial distribution with θ_v parameters is generated for each cluster v. In particular, θ_v can be seen as a two-dimensional matrix of size $k \times |\Omega|$. The rows of the matrix correspond to the k elements of the samples of v (i.e., x_i, in the first step of the

algorithm), while the columns correspond to the different characters of Ω that can become the values of each element. Then, the value of a matrix element $\theta_v[m,l]$ denotes the probability of observing character c_l at position m of the samples of v. Specifically, $\theta_v[m,l]$ is maximum-likelihood (ML) estimated as follows:

$$\theta_v[m,l] = n_{vml}/n_v. \quad (1)$$

n_{vml} counts the number of occurrences of the character $c_l \in \Omega$ at the mth position of the n_v samples of cluster v. Formally,

$$n_{vml} = \sum_{x_i \in v} \delta_{iml}, \text{ where } \delta_{iml} = \begin{cases} 1 & \text{if } x_{im} = c_l \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

In the first step of AC, $n_v = 1$ and each row m of the θ_v matrix contains one element equal to 1. The values of all the other elements of the mth row are 0. Taking our example program of Fig. 1, suppose that we search for patterns whose maximum length is 9 bytes. AC must be applied in the different collections $X_2, \ldots, X_9$ that contain bytecode substrings of length $2, \ldots, 9$, respectively. For the case of X_9, AC results in creating leaf clusters, each one of which includes a sequence of length 9. Some of these clusters are given in Fig. 2b (note that, in order to simplify the figure, we used `A`, `G` and `F` to denote the opcodes `aload_0`, `getfield`, and `fmul`, respectively). Let v be the first leaf cluster from the left side of the figure. Then, θ_v is a 9×12 matrix. Similar matrices are created for the rest of the leaf clusters of Fig. 2b.

At each next step of AC, the algorithm searches the current set of clusters to identify the two most similar ones v, u that can be merged into a new cluster denoted by $v \cup u$. In our example, the substrings of the first and the third cluster differ only in the operands that reside in their fourth and eighth positions. These clusters can be merged into a new cluster that contains the aforementioned substrings. Moreover, the fourth and eighth rows of the $\theta_{v \cup u}$ matrix comprise two nonzero valued elements. Consequently, the $\theta_{v \cup u}$ matrix represents a parameterized pattern of the two substrings, which contains two wildcards in place of its fourth and eighth elements. A similar merge takes place in the third step of the algorithm. Let w be the fourth leaf cluster from the left of Fig. 2b. This cluster is merged with the one created in the previous step into a new one that is still represented by the parameterized pattern created in the previous step.

The distance between two clusters is formally defined as follows [5], [6]:

$$D(v,u) = L_v(\theta_v) + L_u(\theta_u) - L_{v \cup u}(\theta_{v \cup u}). \quad (3)$$

The quantity $L_v(\theta_v)$ represents the log-likelihood value that characterizes the cluster v and is given by the following formula:

$$L_v(\theta_v) = \sum_{x_i \in v} \sum_{m=1}^{k} \sum_{l=1}^{|\Omega|} \delta_{iml} \log \theta_v[m,l]. \quad (4)$$

In general, a wildcard in the ith position of a parameterized pattern that characterizes a cluster signifies that the substrings of the cluster differ in their ith byte. The pattern may contain more than one consecutive wildcards, if the substrings of the cluster differ in more than one consecutive bytes. The AC algorithm may further construct nonparameterized patterns by merging clusters that contain identical substrings.

The algorithm terminates when no pair of nodes is allowed to be further merged. To assess the final set of patterns and obtain P_k, we perform a depth-first visit upon the nodes of the constructed tree. In particular, starting from the high-level nodes, we traverse each subtree until finding the first cluster (node) v, whose multinomial density parameters θ_v represent a required degree of similarity. This is done by setting a threshold $T(k)$ to the number of nonwildcard elements of the pattern that is represented by the node that we look for. The value for $T(k)$ should be experimentally determined.[1] Once such a node is found, it is stored in P_k. By construction, P_k may comprise patterns that overlap. More precisely, two patterns in P_k may refer to the same opcode or operand of the original bytecode. Overlapping patterns may contribute differently in the overall bytecode size reduction and, therefore, they are included in P_k. At the end of AC, the retrieved patterns are further simplified. Specifically, if P_k contains a pattern with a wildcard in place of the first (or the last) byte, this pattern is substituted with the pattern that results from removing this wildcard. Therefore, the resulting set P_k contains patterns that do not start or end with a wildcard. Nonparameterized patterns may also result from the simplification procedure when it is applied in patterns that contain wildcards only in place of their first or last bytes.

The $P_2, \ldots, P_K$ subsets of significant patterns obtained from the application of AC to the $X_2, \ldots, X_K$ collections of substrings are finally merged into a single set of significant patterns P. Again, P may comprise overlapping patterns. Moreover, P may comprise nested patterns, in the sense that a pattern p that was originally included in P_i exactly matches with a part of a pattern q that was originally included in P_j, where $j > i$. The elements of P are combined during the last two steps of the code compression process mentioned at the beginning of this section.

As in the case of the nonparameterized patterns technique, the total number of possible combinations of patterns is $2^{|P|}$. The subset of them that gives us the highest bytecode size reduction is stored in the dictionary, whose structure is, however, slightly different from the one used in the nonparameterized pattern discovery technique. Specifically, the dictionary is still a table of characters. Each table element holds a pattern instruction opcode or operand and there are elements that contain the `END_OF_PATTERN` character. The wildcards are not stored in the pattern. Instead of wasting one table element for storing a value that signifies the existence of a wildcard, we only spend 1 bit. Specifically, for patterns of maximum length K we use $\lceil (K-2)/8 \rceil$ table elements at the beginning of each pattern to encode the positions of the wildcards within it. Recall that the patterns produced by AC do not contain wildcards

1. In our particular experiments discussed in Section 4, a value of $T(k) = \lceil k/2 \rceil$ was sufficient for obtaining good percentages of bytecode size reduction.

in their first and last bytes; hence, a pattern of maximum length K may contain at most $K-2$ wildcards. Then, the value of the ith bit of a table element that encodes the positions of wildcards within the pattern is 1, if the pattern contains a wildcard in the ith position. Hence, for patterns of maximum length 10, we use one extra table element. Similarly, for patterns whose length is between 11 and 18, we use two extra table elements, and so on. In the original bytecode, the occurrences of a pattern are substituted by a character (i.e., 1 byte) that indexes the dictionary elements that encode the positions of wildcards within the patterns. The indexing byte is followed by the actual values of these wildcards. As indexing characters we use unused bytecode instruction opcodes [4].

Summarizing, the most important concept of AC is the *intercluster distance* $D(u,v)$, which we use as a basic criterion for merging clusters during the construction of the tree. The distance used (3) is derived from a probabilistic model; it reflects the likelihood decrease that results by merging the clusters u and v. During each of its steps, the algorithm merges the best pair of clusters, i.e., those having the lower likelihood decrease when putting together. AC has several advantages as it requires $O(|X_k|)$ memory for every $X_k, k=2,\ldots,K$, while its complexity is typically quadratic to $|X_k|$ [5]. Nevertheless, the application of other statistical methods for finding patterns in Java bytecode constitutes one of our future directions in this subject area.

Returning to our example scenario, suppose that we use the pattern identified in Fig. 2b to compress our simple Java program. The pattern appears in three bytecode sequences in the program. Originally, for these sequences we need $3 \times 9\ bytes = 27\ bytes$. In the dictionary, we need 7 bytes to store the nonwildcard pattern elements and 1 byte extra to encode the positions of the wildcards.

Finally, we need 1 byte for the `END_OF_PATTERN` character. In the bytecode, each pattern occurrence is replaced by 3 bytes, one for indexing the directory and two for the actual values of the two wildcards. Thus, in the compressed code, we need 9 bytes for specifying the positions of the pattern occurrences in the bytecode and 9 bytes for storing the pattern in the dictionary. This means that the compression results in saving 9 bytes. Hence, the use of parameterized patterns gives us a much better bytecode size reduction (26.47 percent), compared to the one (8.82 percent) obtained in the case of the nonparameterized pattern discovery technique.

To further motivate the investigation of the advanced parameterized pattern discovery technique, we applied it in a set of simple Java programs that realize standard algorithms. The sizes of these simple programs are given in Table 1a. The percentages of the bytecode size reduction we obtained in these examples are given in Table 1b. It should be noted that these results are optimal in the sense that we performed all possible combinations of the retrieved patterns toward locating the subset of these patterns that gives the highest bytecode size reduction. Once more, we can observe that the parameterized pattern discovery technique results in better percentages of bytecode size reduction, compared to the ones obtained from the nonparameterized one. It is worth noticing that in some of the examples (e.g., fibonacci numbers, selection sort, etc.), the nonparameterized technique completely failed to locate any patterns, while the parameterized technique results in a respectable percentage of bytecode size reduction.

TABLE 1
Using the Nonparamaterized and the Parameterized Techniques to Compess a Typical Set of Algorithms Implemented in Java

Java Program	Binary Search	Bubble Sort	Fibonacci Numbers	Minimum in Array	Matrix Addition	Prime Numbers	Selection Sort
class file size (bytes)	789	871	712	758	840	576	889
bytecode size (bytes)	56	60	27	30	29	27	71

(a)

Java Program	Binary Search	Bubble Sort	Fibonacci Numbers	Minimum in Array	Matrix Addition	Prime Numbers	Selection Sort
parameterized	14.28%	6.66%	3.70%	3.33%	8.82%	4.83%	7.04%
non-parameterized	10.71%	6.66%	0.00%	0.00%	1.47%	0.00%	0.00%

(b)

(a) Class file and bytecode sizes. (b) Percentage of bytecode size reduction.

3.3 Bytecode Decompression

The bytecode decompression procedures for the techniques we discussed in this section are quite straightforward. In the case of the nonparameterized technique, the decompression module is initialized with a given dictionary. The module provides an operation that accepts as input an index to the dictionary. This particular operation should be called by the main loop of the Java bytecode interpreter, upon the discovery of a byte that corresponds to an unused bytecode opcode. The decompression module uses the input index to find the first byte of the pattern. This corresponds to the first instruction opcode contained in the pattern. The decompression module reasons based on the given opcode about the number of bytes that need to be fetched from the dictionary. Then, the required number of bytes are fetched from the dictionary toward forming an instruction. The remaining instructions that constitute the particular pattern are decompressed similarly. The overall decompression procedure ends up by fetching the `END_OF_PATTERN` element from the dictionary.

For the parameterized technique, the decompression is similar. The decompression module is also initialized with a given dictionary and the maximum length used for deriving the patterns contained in this dictionary. The decompression module accepts as input an index to the dictionary. Following, it uses the input index to repeatedly fetch from the dictionary a number of elements that encode the positions of the wildcards within the indexed pattern. The number of these elements depends on the maximum length used for discovering the patterns contained in the dictionary (by default, the maximum length of patterns is 9 and therefore 1 byte is fetched). The byte that follows the

TABLE 2
MIDP Packages Basic Features

MIDP Package	Class file size (bytes)	Bytecode size (bytes)
java.io	19568	3454
java.lang	27714	3998
javax.microedition.io	4599	509
javax.microedition.media	2065	362
javax.microedition.midlet	2025	121
javax.microedition.pki	2563	184

elements that encode the positions of wildcards is the first instruction opcode contained in the pattern. As previously, the decompression module figures out about the number of bytes required for forming the first instruction that should be executed. Some of the required bytes are fetched from the dictionary, while some others are found in the compressed code. Specifically, the ith byte is fetched from the dictionary if the value of the ith bit of the bytes that encode the positions of wildcards within the pattern is 0. Otherwise, the ith byte is found in the compressed code. The remaining instructions that constitute the particular pattern are decompressed similarly. Again, the overall decompression procedure ends up by fetching the `END_OF_PATTERN` element from the dictionary.

4 Assessment

Although some benefits from using the parameterized pattern discovery technique are evident from the simple examples discussed in Section 3, we further conducted experiments involving a real-world case study. Specifically, the target of our investigation is the MIDP (Mobile Information Device Profile) v2.0 reference implementation from Sun Microsystems.[2] MIDP is part of the Java 2 Platform Micro Edition (J2ME) and relies on CLDS (Connected Limited Device Configuration). It provides a basic environment for the development of Java applications for mobile information devices (MIDs) such as mobile phones and PDAs. The MIDP reference implementation consists of 11 packages. In this section, we present the results we obtained from six of these packages, which we consider as the core of MIDP. The basic characteristics of each package (size of class file and bytecode size) are given in Table 2. The sizes of the particular Java class files are representative, considering real world applications aimed at mobile and embedded devices. Specifically, the standard JTWI (Java Technology for the Wireless Industry)[3] specification sets the limit of a standard-size-application to 64 Kbytes. JTWI actually defines a standard framework for the development of mobile applications and MIDP is part of it. Applications that are under the limit of 64 Kbytes are guaranteed to work correctly over any kind of device that complies with JTWI. Regarding the MIDP packages, java.io provides classes for input and output through data streams. The java.lang package consists of basic language classes coming from J2SE (Java 2 Standard Edition), javax.microedition.io includes networking support relying on CLDC. The javax.microedition.media package allows accessing device-dependent resources for multimedia processing. The javax.microedition.midlet package defines the basic MIDP application model. Finally, javax.microedition.pki enables managing certificates, used for securing connections.

The goals of our experiments were:

1. To investigate the *benefits* and the *cost* of the parameterized pattern discovery technique, measured in terms of the *bytecode size reduction* obtained and the *time spent* for discovering and combining patterns, respectively.
2. To study the cost of the decompression procedure that comes along with the parameterized technique in terms of the *time overhead*, introduced in the execution of Java bytecode.

4.1 Bytecode Compression

To evaluate the benefits and the cost of the parameterized pattern discovery technique, we performed two different sets of experiments. The first one aims at evaluating the impact of using patterns that contain a variable number of wildcards in the compression of MIDP. The second set of experiments aims at investigating the impact of the increasing length of the patterns in the compression of MIDP. In the first set of experiments, the maximum length of the patterns is set to 9, while in the second one we increase this length from 9 to 11. In both sets, the threshold for nonwildcard elements in the patterns was set to $T(k) = \lceil k/2 \rceil$.

The strong point of the advanced parameterized pattern discovery technique is its ability to track down a rich variety of patterns, whose combination may lead to more effective bytecode size reduction. This fact is particularly highlighted in Table 3a and Table 3b, which give the total number of patterns and the number of nonparameterized patterns discovered in the MIDP packages. Table 3a specifically refers to the first set of experiments, in which the maximum length of the patterns was 9, while Table 3b refers to the second set of experiments, in which the maximum length of the patterns was 11. In general, we can observe that increasing the maximum length of the patterns results in increasing the overall number of patterns found.

Nevertheless, the time required by the advanced parameterized pattern discovery technique for the discovery of patterns is quite high (Table 3a and Table 3b). Moreover, the discovery of a rich variety of patterns also increases the complexity of combining these patterns to obtain a good bytecode size reduction. Therefore, an issue that should be studied in our assessment is whether the discovered patterns are useful, that is, worthy of the time required for discovering and combining them.

As already discussed, the complexity for finding all possible combinations of patterns is $2^{|P|}$. In the first set of experiments, the average number of parameterized patterns found per MIDP class is 157. On the other hand, the average number of nonparameterized patterns found per MIDP class is 85.6. If we assume that each combination of patterns

2. http://java.sun.com/products/midp/.
3. http://java.sun.com/products/jtwi/.

TABLE 3
Comparing the Numbers of Parameterized and Nonparameterized Patterns Discovered in MIDP

MIDP Package	Number of patterns found (pattern length = 2 to 9)		Execution time for AC (sec)
	parameterized	non-parameterized	
java.io	4408	1870	6903
java.lang	4160	2322	25403
javax.microedition.io	66	41	632
javax.microedition.media	541	245	500
javax.microedition.midlet	531	295	12
javax.microedition.pki	91	23	55

(a)

MIDP Package	Number of patterns found (pattern length = 2 to 11)		Execution time for AC (sec)
	parameterized	non-parameterized	
java.io	4583	1981	8701
java.lang	4375	2504	32639
javax.microedition.io	99	43	818
javax.microedition.media	577	253	646
javax.microedition.midlet	533	295	15
javax.microedition.pki	98	23	74

(b)

(a) Maximum length of patterns $= 9$. *(b) Maximum length of patterns* $= 11$.

requires 1 μsec to be performed, then the assessment of all possible combinations of parameterized patterns would require $5.39864 * 10^{31}$ hours to complete. Similarly, the assessment of all combinations of nonparameterized patterns would require $1.69879 * 10^{10}$ hours. The assumption that the assessment of a particular combination of patterns requires 1 μsec is rather optimistic and refers to combinations of two patterns. As we can observe in the experiments detailed later, the required time depends on the number of combined patterns and it is usually higher than 1 μsec. Therefore, for both the parameterized and the nonparameterized sets of patterns found in MIDP, the assessment of all possible combinations is virtually impossible. To deal with this particular problem, we investigate the use of two heuristics. The purpose of the heuristics is to allow us to efficiently obtain good suboptimal bytecode size reduction.

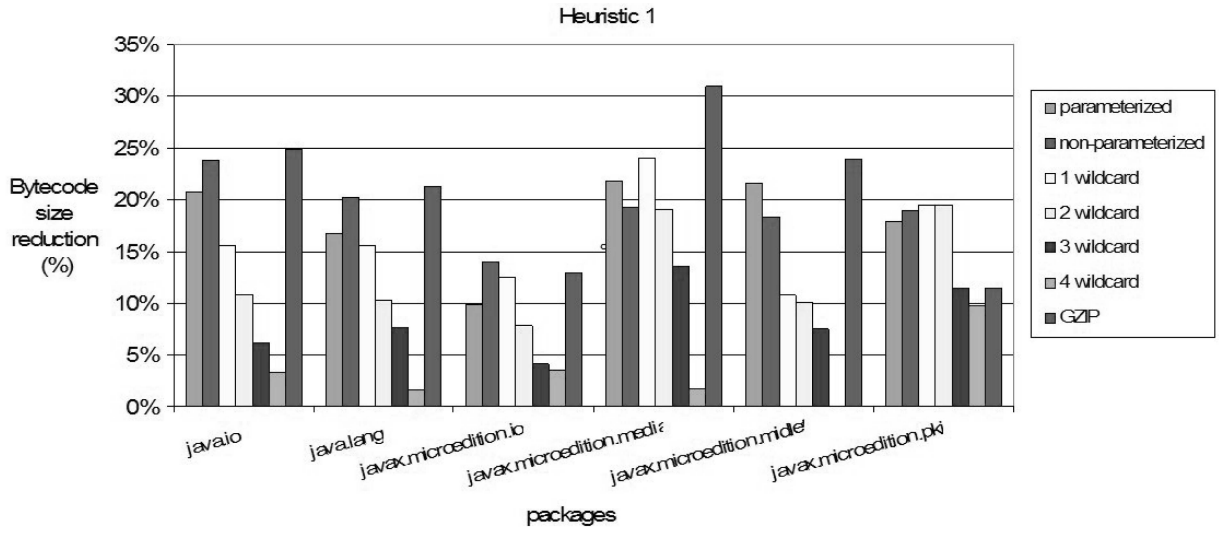
In both heuristics, we sort an overall set of patterns P based on the bytecode size reduction provided by these patterns when they are used in isolation for the compression of the bytecode within which they were found. In a sense, we quantitatively group/cluster the patterns with respect to the bytecode size reduction that they provide. Then, in the first heuristic, we start from the pattern that offers the best bytecode size reduction and we examine its combination with the second-best pattern. If the bytecode size reduction obtained using the two patterns is greater than the one obtained from the first pattern, we keep the second pattern as well. Otherwise, the second-best pattern is useless. We follow the same procedure for the rest of the sorted patterns, ending up with a set of patterns that gives us a suboptimal bytecode size reduction. The second heuristic amounts to performing the first heuristic $|P|$ times. During each iteration $i : i = 1, |P|$, we start from the ith best of the sorted patterns and we proceed by checking the usefulness of combining it with the $i+1, i+2, \ldots, |P|, 1, 2, \ldots, i-1$ patterns. At the end of this procedure, we end up with $|P|$ sets of patterns, among which we select the one that gives us the highest bytecode size reduction.

In general, the impact of the second heuristic in the overall time required for the bytecode compression is expected to be much higher than the impact of the first heuristic (since the application of the second heuristic consists of applying the first heuristic $|P|$ times). On the other hand, the combinations of patterns examined by the second heuristic are a superset of the combinations of patterns examined by the first heuristic. Hence, the percentage of bytecode size reduction obtained from the application of the second heuristic is expected to be at least as good as the percentage of bytecode size reduction obtained from the application of the first heuristic.

4.1.1 Experimental Results—First Set of Experiments

In the first set of experiments, we used as input to both heuristics the full set of patterns, P_{param}, discovered by the parameterized pattern discovery technique in each one of the MIDP packages. Moreover, we used as input to both heuristics the set of nonparameterized patterns, $P_{non\text{-}param}$, discovered in each one of the MIDP packages. To compare the contribution of patterns that contain a variable number of wildcards against the contribution of patterns that contain a fixed number of wildcards, we further constructed four input sets, P_{1w}, P_{2w}, P_{3w}, and P_{4w}, consisting of patterns that contain one, two, three, and four wildcards, respectively. P_{1w}, P_{2w}, P_{3w}, and P_{4w} were constructed from the full set of patterns P_{param} that resulted from AC. Finally, to investigate the impact of the increasing number of wildcards in the compression of the MIDP packages, we constructed three more input sets, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$, consisting of patterns that contain one-to-two, one-to-three, and one-to-four wildcards, respectively. $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$ were also constructed from the full set of patterns P_{param} that resulted from AC.

Fig. 3 compares the results we obtained per MIDP package from the application of the first heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w} (more detailed results regarding the bytecode size reduction obtained per different class of the MIDP packages are given in the Appendix). Specifically, the time required to combine the patterns (Fig. 3b) is acceptable in every case. However, the bytecode size reduction we obtained for each package largely depends on the patterns used. In particular, in some packages (e.g., javax.microedition.midlet), P_{param} gives the best bytecode size reduction. In other packages (e.g., java.io), the nonparameterized set, $P_{non\text{-}param}$, performs better. Finally, there are also packages where the sets that contain patterns with fixed numbers of wildcards give the best bytecode size reduction (e.g., P_{1w} and P_{2w} in javax.microedition.pki). Therefore, the first heuristic is quite fast but



(a)

MIDP Package	Parameterized	Non-parameterized	Parameterized patterns with 1 wildcard	Parameterized patterns with 2 wildcards	Parameterized patterns with 3 wildcards	Parameterized patterns with 4 wildcards
java.io	2.03	1.17	0.66	0.21	0.09	0.08
java.lang	3.975	7.06	1.96	0.4	0.46	0.2
javax.microedition.io	0.2	0.1	0.09	0.02	0.02	0.02
javax.microedition.media	0.27	0.18	0.05	0.01	0.01	0.01
javax.microedition.midlet	0.03	0.02	0.02	0.02	0.02	0.02
javax.microedition.pki	0.01	0.01	0.01	0.01	0.01	0.01

(b)

Fig. 3. Experimental results from the application of the first heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w}. (a) Percentage of bytecode size reduction per MIDP package. (b) Time to combine patters per MIDP package (sec).

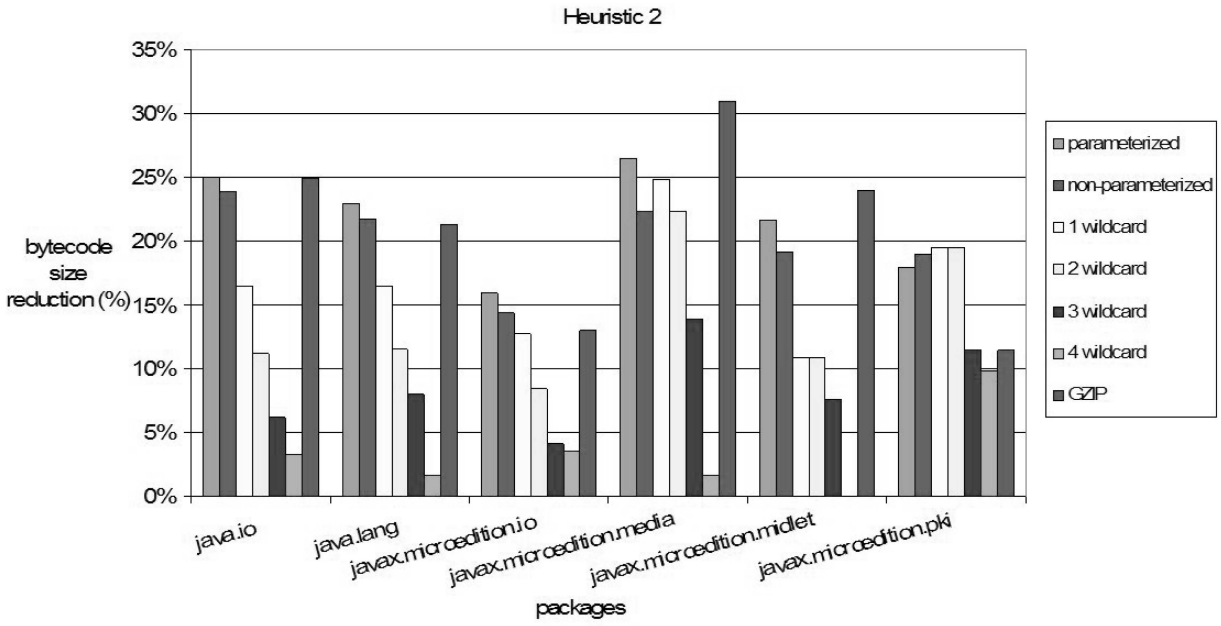
rather unpredictable regarding the set of patterns that should be used to obtain the highest bytecode size reduction. Moreover, in most packages, the first heuristic appears incapable of taking advantage of the flexibility provided by the large variety of patterns contained in P_{param}.

The results obtained from the application of the second heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w} are given in Fig. 4 (more details regarding the bytecode size reduction obtained per different MIDP class are given in the Appendix). As expected, the time required to combine patterns (Fig. 4b) is much higher than the time spent when using the first heuristic. However, it is still reasonable compared to the time required for deriving all possible combinations of patterns. The bytecode size reduction we obtained is generally better than the corresponding reduction obtained from the application of the first heuristic. The use of P_{param} gives the highest bytecode size reduction in all packages, except for the javax.microedition.pki one (Fig. 4a). Hence, the second heuristic renders the use of P_{param} more beneficial. The reduction obtained from the sets that contain patterns with fixed numbers of wildcards is in most packages poorer than the reduction resulted from P_{param}. However, among P_{1w}, P_{2w}, P_{3w}, and P_{4w}, the first two sets give better bytecode size reduction. Hence, patterns with a relatively small number of wildcards contribute more in reducing the size of the MIDP packages.

Fig. 5 and Fig. 6 compare the results we obtained from the application of the first and the second heuristic in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$ (more details can be found in the Appendix). Regarding the two heuristics, the observations derived from Fig. 3 and Fig. 4 are still valid. The bytecode size reduction obtained from the first heuristic is smaller compared to the one obtained from the application of the second heuristic. In both heuristics, the reduction obtained in the case of $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$ is greater than the reduction obtained in the case of P_{1w}, P_{2w}, P_{3w}, and P_{4w}. In most cases, $P_{1\text{-}2w}$ gives better bytecode size reduction than $P_{1\text{-}3w}$ and $P_{1\text{-}4w}$. This observation is additional evidence that patterns with a small number of wildcards result in combinations that give us better bytecode size reduction.

The inclusion of patterns with three and four wildcards in the set of patterns that contain one and two wildcards results in the worst bytecode size reduction in certain cases because of the overlapping between these patterns. For example, a pattern that contains three wildcards could appear in the bytecode in the form of two consecutive patterns that contain one and two wildcards. In the same bytecode, there may also be individual occurrences of the two constituent patterns. If used in isolation,[4] the two

4. That is, according to the way that we assess the contribution of patterns in the heuristics.



(a)

MIDP Package	Parameterized	Non-parameterized	Parameterized patterns with 1 wildcard	Parameterized patterns with 2 wildcards	Parameterized patterns with 3 wildcards	Parameterized patterns with 4 wildcards
java.io	2126.65	487.64	131.21	11.73	3.64	0.18
java.lang	12731.71	4122.76	139.11	39.34	16.84	1.19
javax.microedition.io	119.94	22.24	18.47	1.05	0.07	0.07
javax.microedition.media	0.27	0.18	0.05	0.01	0.01	0.01
javax.microedition.midlet	0.12	0.02	0.02	0.02	0.02	0.02
javax.microedition.pki	0.01	0.01	0.01	0.01	0.01	0.01

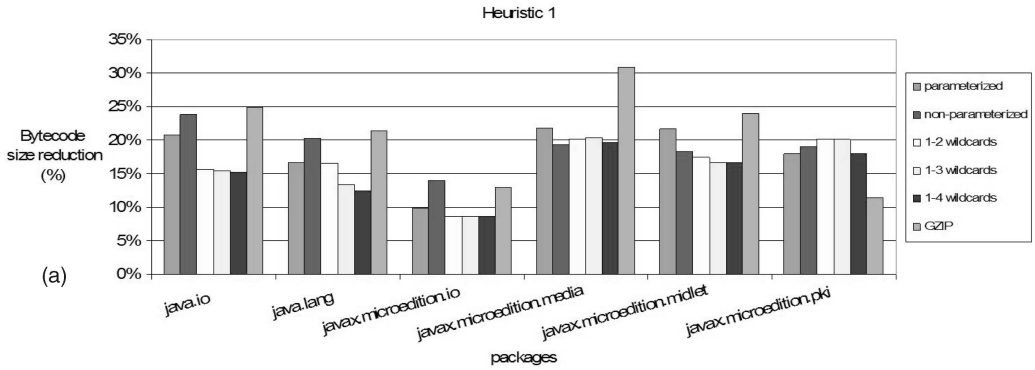
(b)

Fig. 4. Experimental results from the application of the second heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w}. (a) Percentage of bytecode size reduction per MIDP package. (b) Time to combine patterns per MIDP package (sec).

constituent patterns may result in bytecode size reductions that are smaller than the one that can be obtained from the use of the three-wildcards-pattern. On the other hand, if both the constituent patterns are used we may get a reduction that is better than the one obtained by the use of the three-wildcards-pattern. However, the selection of the three-wildcards-pattern may render the selection of the two constituent patterns impossible (e.g., because, after the substitution of the three-wildcards-pattern occurrences, the constituent patterns no longer appear in the bytecode more than once).

Although one would expect that the bytecode size reduction resulted from $P_{1\text{-}3w}$ and $P_{1\text{-}4w}$ would be really close or equal to the one obtained from P_{param}, this is not the case. $P_{1\text{-}4w}$ comprises all the patterns that contain wildcards. However, P_{param} further comprises the nonparameterized patterns, discovered by AC (Section 3.2). The contribution of the nonparameterized patterns is quite significant, along with the contribution of the patterns that contain one and two wildcards. This becomes clear with further elaboration on the results obtained from the application of the second heuristic in P_{param}. Specifically, Fig. 7 gives the numbers of useful patterns (i.e., the patterns that are finally used for compressing the MIDP packages) that resulted from the second heuristic in relation with their lengths and the number of wildcards that they contained.

Figs. 3a, 4a, 5a, and 6a further compare the results we discussed so far with the ones obtained from the use of `GZIP`. We used `GZIP` to compress the bytecode of the different MIDP packages that constitute our case study. The resulted compressed bytecode is not interpretable, in that it cannot be executed without being fully decompressed. However, the bytecode size reduction obtained with `GZIP` is a useful measure toward our assessment. In all figures, we can observe that the percentage of the bytecode size reduction obtained with `GZIP` is comparable with the one resulted from P_{param}. It is interesting to note that in the case of the first heuristic (Fig. 3a and Fig. 5a), in most MIDP packages, `GZIP` performs slightly better than P_{param} and $P_{non\text{-}param}$. On the other hand, in the case of the second heuristic, we see that P_{param} gives, in some MIDP packages (java.io, java.lang packages), better bytecode size reduction than `GZIP`, while $P_{non\text{-}param}$ gives only, at most, as good bytecode size reduction as `GZIP`.



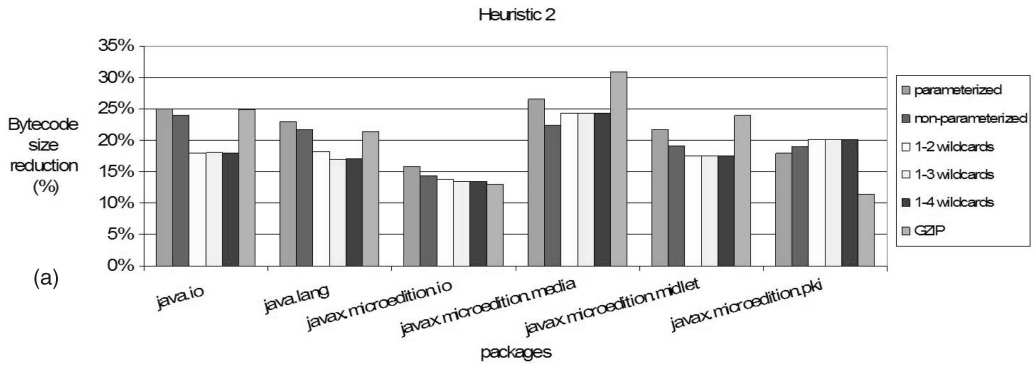
(a)

(b)

MIDP Package	Parameterized	Non-parameterized	Parameterized patterns with 1-2 wildcards	Parameterized patterns with 1-3 wildcards	Parameterized patterns with 1-4 wildcards
java.io	2.03	1.17	0.84	0.91	1.02
java.lang	3.975	7.06	1.25	1.34	1.53
javax.microedition.io	0.2	0.1	0.11	0.12	0.14
javax.microedition.media	0.27	0.18	0.1	0.1	0.11
javax.microedition.midlet	0.03	0.02	0.02	0.02	0.02
javax.microedition.pki	0.01	0.01	0.01	0.02	0.03

Fig. 5. Experimental results from the application of the first heuristic in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, $P_{1\text{-}4w}$, and P_{4w}. (a) Percentage of bytecode size reduction per MIDP package. (b) Time to combine patterns per MIDP package (sec).

Summarizing the results from the first set of experiments, we can derive the following conclusions: The examined pattern discovery technique allows the discovery of a rich variety of parameterized and nonparameterized patterns. The discovery of the patterns is quite expensive, with respect to the time spent by AC. Moreover, the complexity of selecting and combining patterns is also high. Our experimental results showed that in the case of MIDP, the most *useful* patterns out of P_{param} are the nonparameterized ones and the ones that contain one and two wildcards. However, this may not be the case in other Java applications, where patterns with more than two wildcards may also prove useful. Based on these remarks, a good strategy for balancing the trade-off between the time spent for discovering/combining patterns and the resulting bytecode size reduction is to apply the technique in an incremental manner. For a given bytecode, instead of using AC for constructing a large set of patterns, containing many wildcards, AC can be customized to construct a smaller set of patterns, containing few wildcards. Starting from this smaller set of patterns, the technique may be used at a later time, if necessary, to discover patterns that



(a)

(b)

MIDP Package	Parameterized	Non-parameterized	Parameterized patterns with 1-2 wildcards	Parameterized patterns with 1-3 wildcards	Parameterized patterns with 1-4 wildcards
java.io	2126.65	487.64	119.46	231.75	295.83
java.lang	12731.71	4122.76	227.48	1417.55	1527.37
javax.microedition.io	119.94	22.24	28.71	28.87	35.32
javax.microedition.media	0.27	0.18	22.27	22.5	36.66
javax.microedition.midlet	0.12	0.02	0.02	0.02	0.02
javax.microedition.pki	0.01	0.01	0.33	0.42	0.45

Fig. 6. Experimental results from the application of the second heuristic in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, $P_{1\text{-}4w}$, and P_{4w}. (a) Percentage of bytecode size reduction per MIDP package. (b) Time to combine patterns per MIDP package (sec).

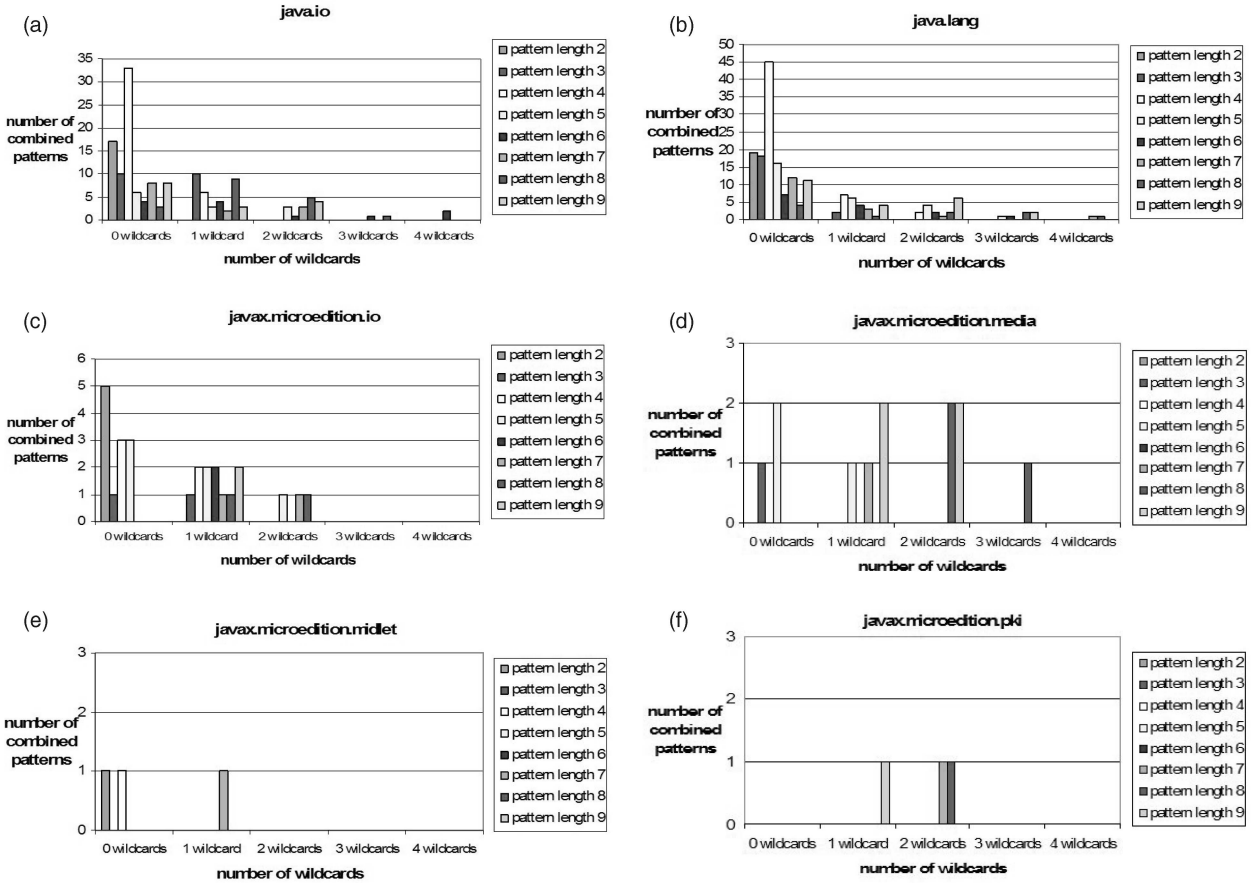


Fig. 7. Patterns contributed by the second heuristic in the case of P_{param}. (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microedition.pki.

contain more wildcards, toward further improving the bytecode size reduction.

4.1.2 Experimental Results—Second Set of Experiments

In the second set of experiments, we used as input to both heuristics the set of patterns of maximum length 11, P_{param}, discovered in each one of the MIDP packages. As discussed at the beginning of this subsection, increasing the maximum length of the patterns results in the discovery of more patterns. Moreover, increasing the maximum length implies increasing the time required for discovering the patterns and the time required for combining them. The expected benefit against the time increment is the increment of the bytecode size reduction. However, the above may not hold; increasing the maximum length of the patterns may result only in a small increment of the bytecode size reduction. In the worst case, it may result in a decrement of the bytecode size reduction. As detailed in Section 3.2, to encode the positions of wildcards in patterns of maximum length K, we have to use $\lceil (K-2)/8 \rceil$ bytes. Therefore, for patterns of maximum length 9, 1 byte is needed. On the other hand, for patterns of maximum length 11, 2 bytes have to be used. The extra bytes used for encoding the positions of wildcards when the maximum length of the patterns is long may reduce the benefits obtained from using these patterns.

More specifically, Fig. 8a and Fig. 8b give the results we obtained in the case of the MIDP packages. Fig. 8a refers to the first heuristic, while Fig. 8b refers to the second heuristic. In both figures, the left axis corresponds to the percentage of the bytecode size reduction increment or decrement resulted from increasing the maximum length of the patterns from 9 to 11 (columns that are under 0 percent correspond to bytecode size reduction decrement, while columns that are above 0 percent correspond to bytecode size reduction increment; the smaller the columns that are under 0 percent, the larger the decrement of the bytecode size reduction). The right axis shows the increment of the time required for discovering and combining the patterns when their maximum length is increased from 9 to 11. In the case of the first heuristic, time increases from 26 percent to 34 percent for the different MIDP packages. Similarly, in the second heuristic, time increases up to 34 percent. With the exception of one package (javax.microedition.pki), the bytecode size reduction decreases. In the first heuristic, the decrement ranges from -12 percent (Fig. 8a, third column—javax.microedition.media) to -42 percent (Fig. 8a, second column—java.lang), while in the second heuristic the decrement ranges from -7 percent (Fig. 8b, third column—javax.microedition.media) to -33 (Fig. 8b, first column—java.io). In the javax.microedition.pki package, the bytecode reduction increases up to 3 percent for both the heuristics (sixth column, Fig. 8a and Fig. 8b).

Summarizing the results from our second set of experiments, we can conclude that very long patterns may not prove beneficial for the bytecode compression process. The

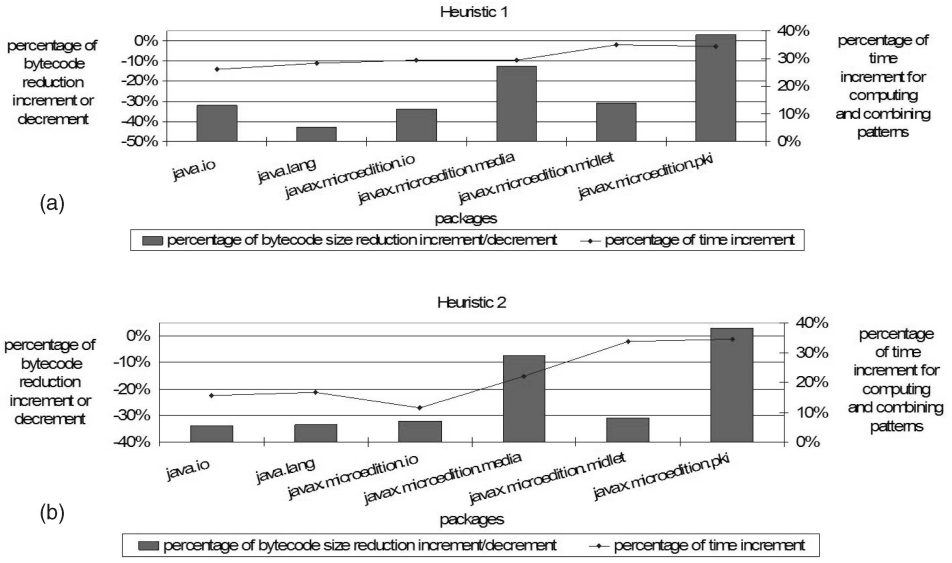


Fig. 8. The impact of increasing the maximum length of the patterns in the bytecode syze reduction obtained and the time required for discovering cominging the patterns. (a) Results obtained from the first heuristic. (b) Results obtained from the second heuristic.

best strategy is to start from discovering patterns whose maximum length is less than or equal to 10 since the bytes required for encoding them in the dictionary is small. The discovery of longer patterns could be tried toward optimizing the compression, while keeping in mind that it may lead to worse results.

4.2 Bytecode Decompression

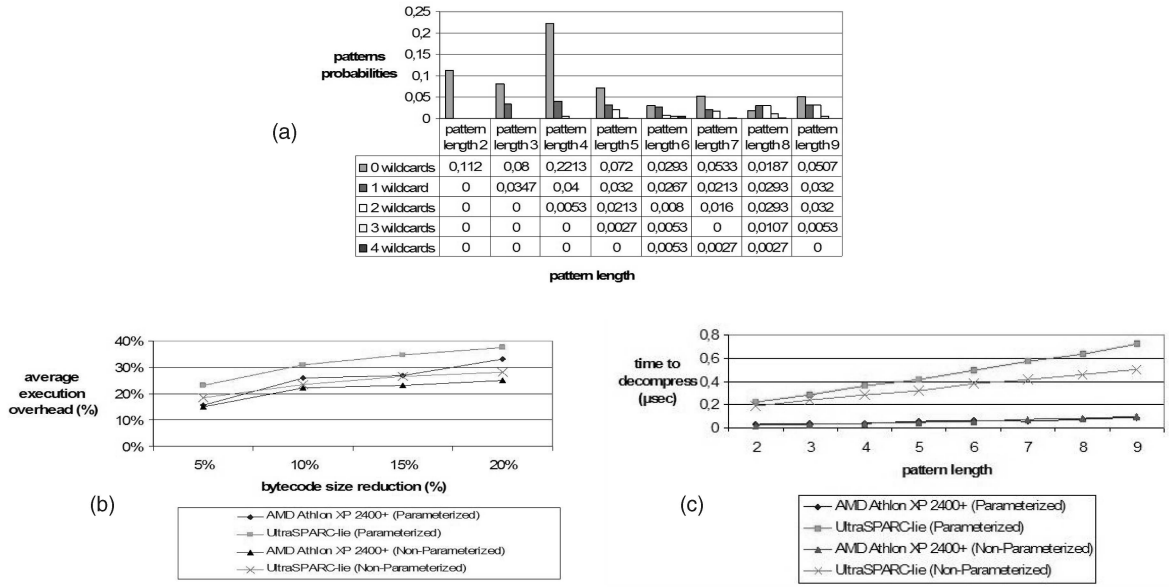
The decompression overhead introduced in the execution of applications that were compressed into interpretable bytecode is always an issue for the assessment of the compression technique that was used. However, this overhead depends on several factors concerning the application itself and the environment within which the application executes. Specifically, in the case of the technique examined in this paper, the overhead depends on the number of pattern instances encountered during the bytecode execution, the length of the patterns and the number of wildcards of these patterns. Moreover, the overhead depends on the characteristics of the particular device and the JVM used for the execution of the bytecode. It should be noted that the JVM may be even implemented as part of the processor used. The previous particularly holds in the case of Java processors such as picoJava,[5] Komodo,[6] aJile GEMCore,[7] JOP,[8] and several others. Given the previous remarks, in this paper, we focus on examining the general behavior of the decompression overhead. To this end, we used randomly generated synthetic bytecode and a simple JVM main loop [25] that interprets the synthetic bytecode. The simple JVM main loop was realized for the purpose of our experiments along with a random bytecode generator and the two decompression modules discussed in Section 3.3.

The random bytecode generator accepts as input a required percentage of bytecode size reduction and produces a synthetic bytecode sequence that can be compressed according to this percentage of bytecode size reduction. The generator further accepts as input the main features of a set of patterns that is generated with respect to the given features. This set of patterns is then used by the generator towards the construction of the target bytecode sequence, which consists of instances of the generated patterns, complemented with bytecode instructions that will not be compressed. Specifically, the main features that characterize a generated set of patterns are:

- The maximum length, K, of the patterns. Based on this feature, the generator creates patterns, whose length is uniformly distributed in the range $[2, K]$.
- The maximum number of wildcards, M, contained in the patterns. Given this feature, the generator builds patterns, whose number of wildcards is uniformly distributed in the range $[0, M]$.
- The maximum the number of pattern occurrences, L. According to this feature, the generator constructs a number of pattern occurrences in the target bytecode, which is uniformly distributed in the range $[2, L]$.
- A two-dimensional $K \times M$ matrix FR. The value of each matrix element $FR[k, m]$ (such that $k : 2, \ldots, K$ and $m : 0, \ldots, M$) corresponds to the probability of generating a pattern of length k with m wildcards.

The JVM loop does not provide any advanced JIT compilation capabilities. Moreover, several typical JVM activities (e.g., verification, resolution, and access control [25]) that precede the bytecode interpretation are omitted. Consequently, the overhead introduced by the decompression modules is expected to be high. In any case, the results are based on randomly generated bytecode and may not be representative of real applications that do not comply with the features used for the generated bytecode in our experiments. To perform our experiments, we used two different environments. The first one relies on a 2 GHz AMD Athlon XP 2400+ with 256 Kbytes L2 cache, while the

5. http://www.sun.com/microelectronics/picoJava/.
6. http://ipr.ira.uka.de/komodo/komodoEng.html.
7. www.ajile.com/products/jemcore/htm.
8. http://www.jopdesign.com/.



	pattern length 2	pattern length 3	pattern length 4	pattern length 5	pattern length 6	pattern length 7	pattern length 8	pattern length 9
0 wildcards	0,112	0,08	0,2213	0,072	0,0293	0,0533	0,0187	0,0507
1 wildcard	0	0,0347	0,04	0,032	0,0267	0,0213	0,0293	0,032
2 wildcards	0	0	0,0053	0,0213	0,008	0,016	0,0293	0,032
3 wildcards	0	0	0	0,0027	0,0053	0	0,0107	0,0053
4 wildcards	0	0	0	0	0,0053	0,0027	0,0027	0

Fig. 9. Evaluating the decompression overhead of the parameterized pattern discovery technique. (a) Probabilities of generating patterns of length $k : 2, \ldots, 9$ with $m : 0, \ldots, 4$ wildcards. (b) Decompression overhead for the synthetic bytecode sequences. (c) Absolute times for pattern decompression (μsec).

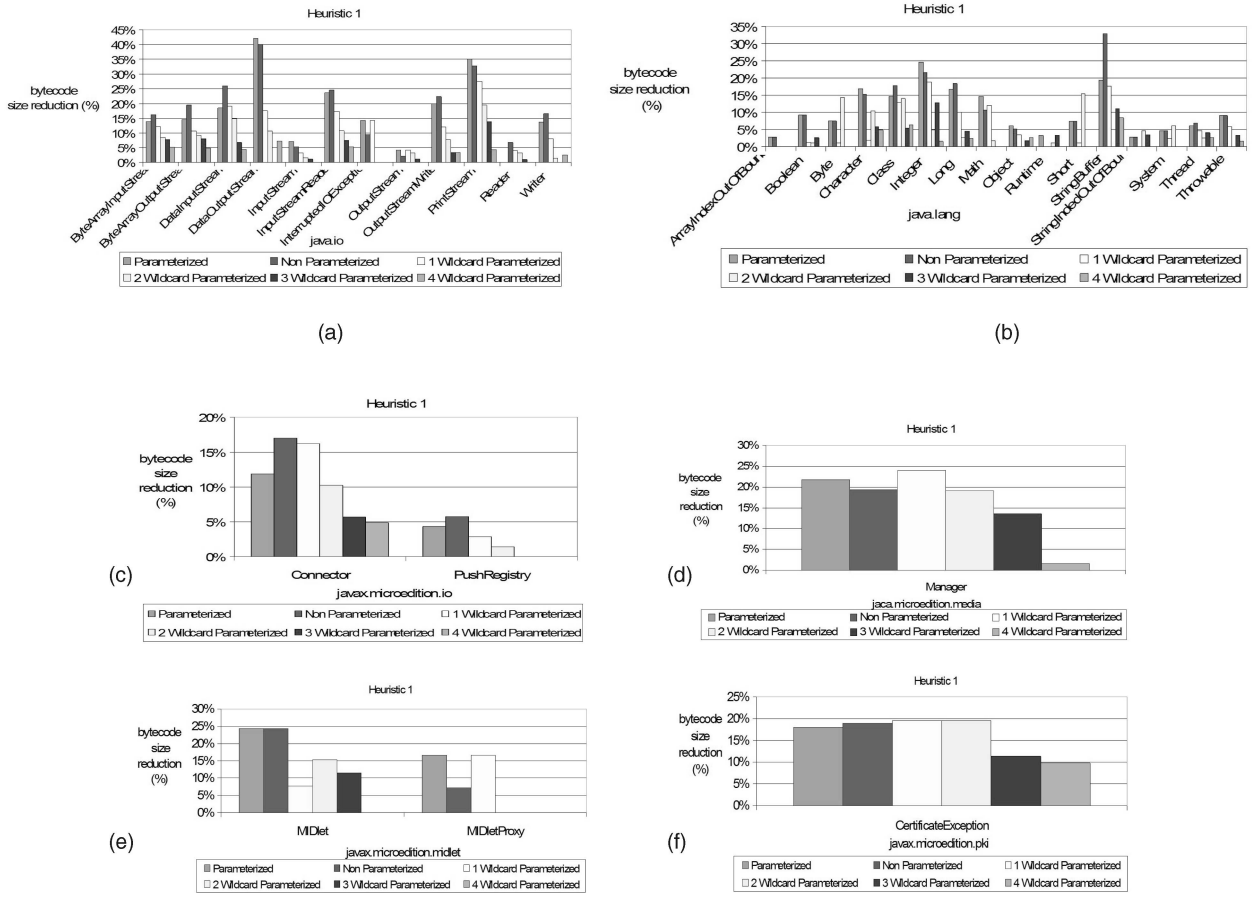


Fig. 10. Experimental results per MIDP class from the application of the first heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w}. (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

second one is based on a 500 MHz UltraSPARC-IIe with 256 Kbytes L2 cache.

The input parameters of the generator relied on the results we obtained from the MIDP case study. In particular, we constructed four different sets of bytecode sequences, whose size was reduced 5 percent, 10 percent, 15 percent, and 20 percent, respectively. The compression of these bytecode sequences relied on patterns, which contained up to four wildcards. Similarly, we generated four more sets of bytecode sequences, whose size was reduced

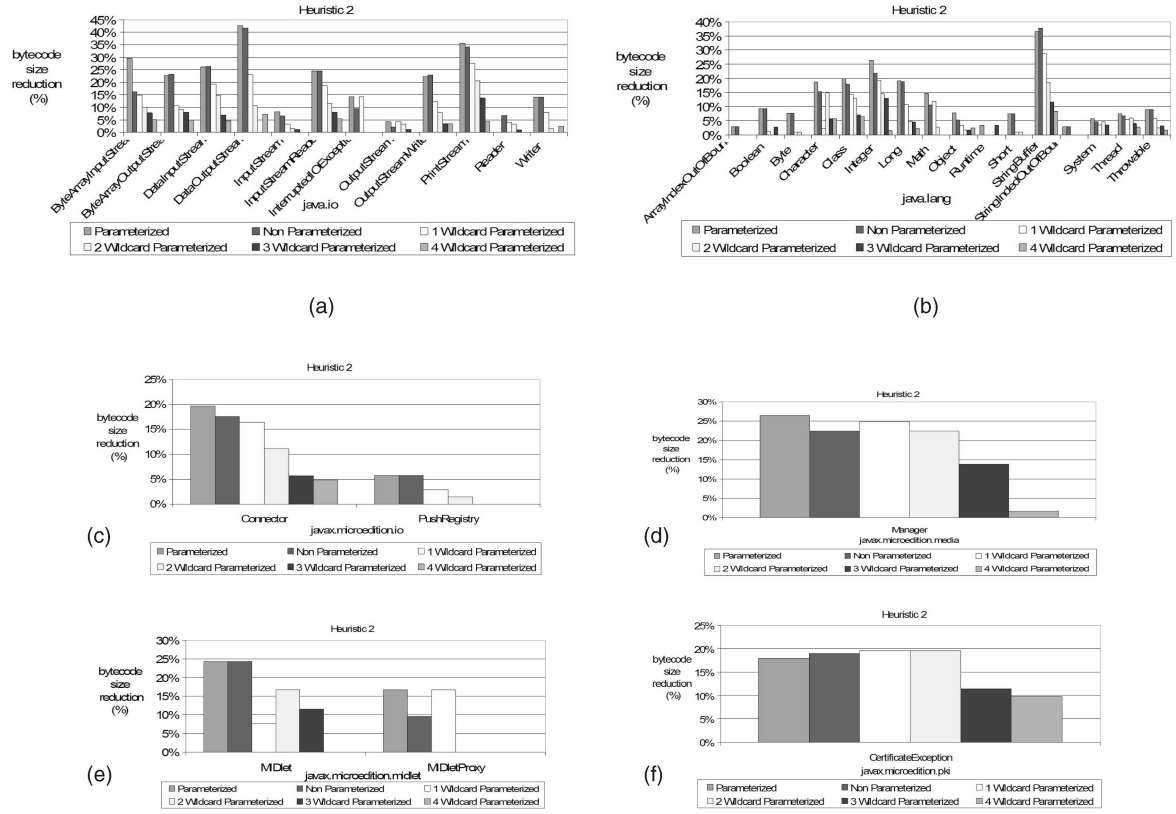


Fig. 11. Experimental results per MIDP class from the application of the second heuristic in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w}. (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

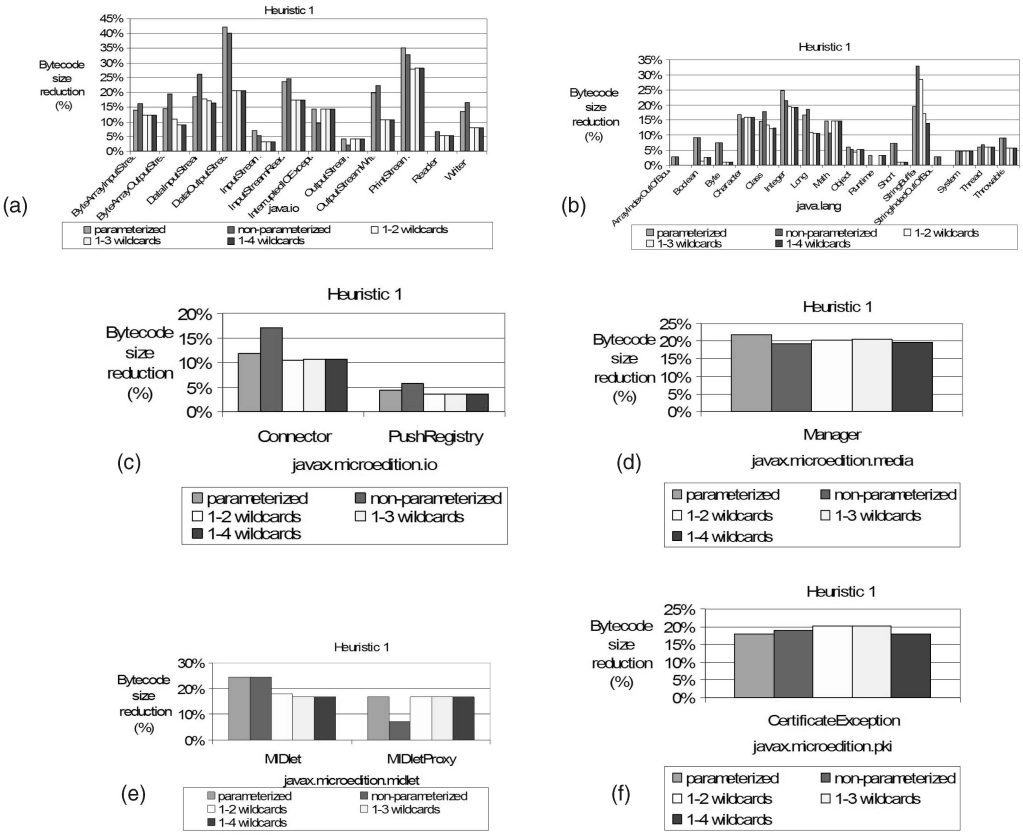


Fig. 12. Experimental results per MIDP class from the application of the first heuristic in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$. (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

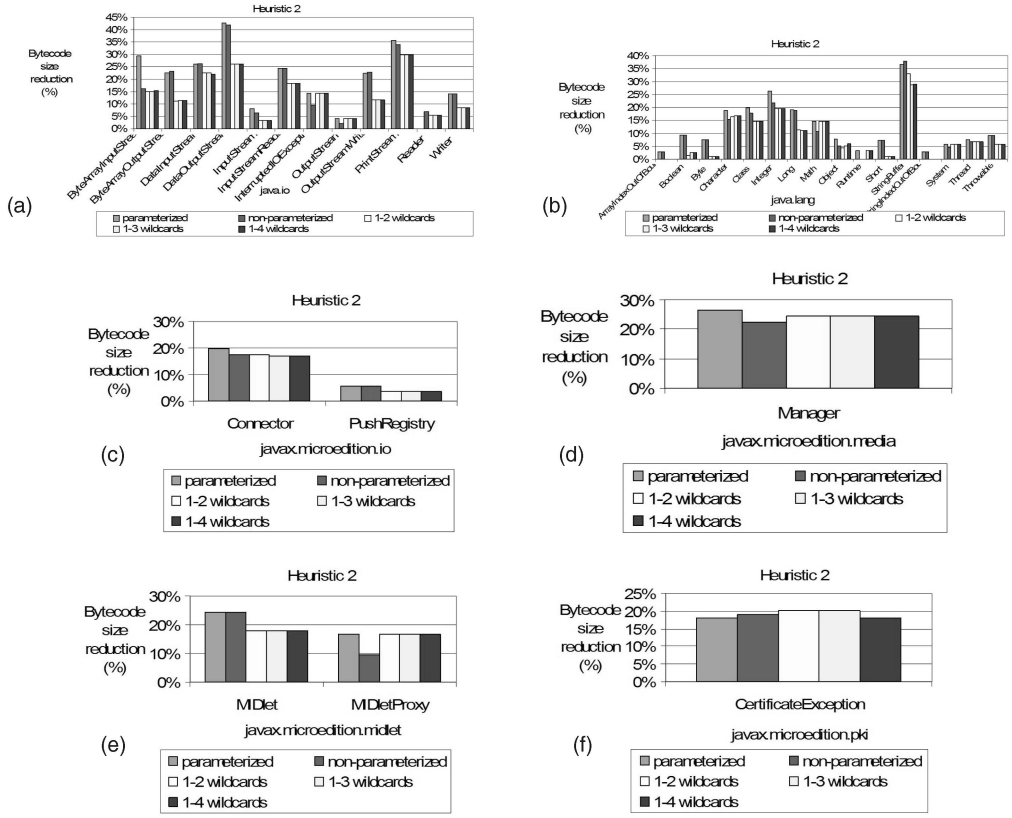


Fig. 13. Experimental results per MIDP class from the application of the second heuristic in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$, and $P_{1\text{-}4w}$. (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

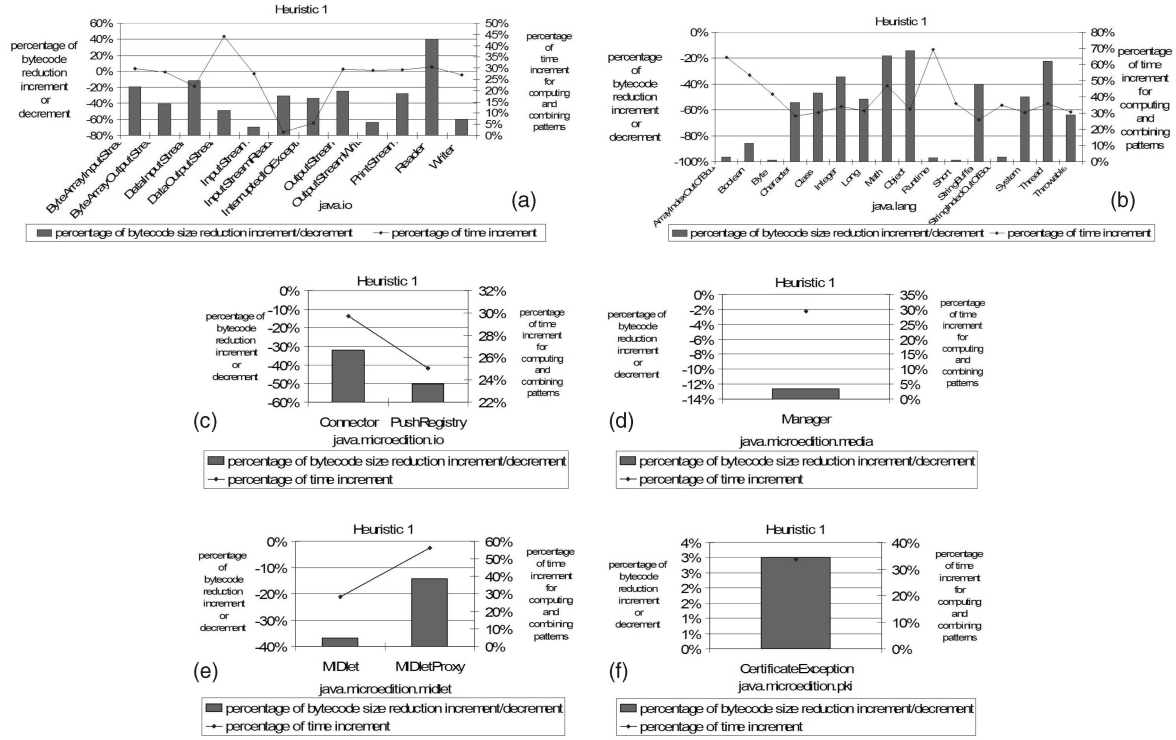


Fig. 14. The impact per MIDP class of increasing the maximum length of the patterns in the bytecode size reduction obtained and the time required for discovering and comining the patterns (first heuristic). (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

5 percent, 10 percent, 15 percent, and 20 percent, with respect to sets of patterns, which did not contain wildcards. In all cases, the maximum length of the patterns was set to 9. The range of the pattern instances was $[2, 12]$. The matrix FR that was given as input to the generator was calculated with respect to the overall set of patterns that

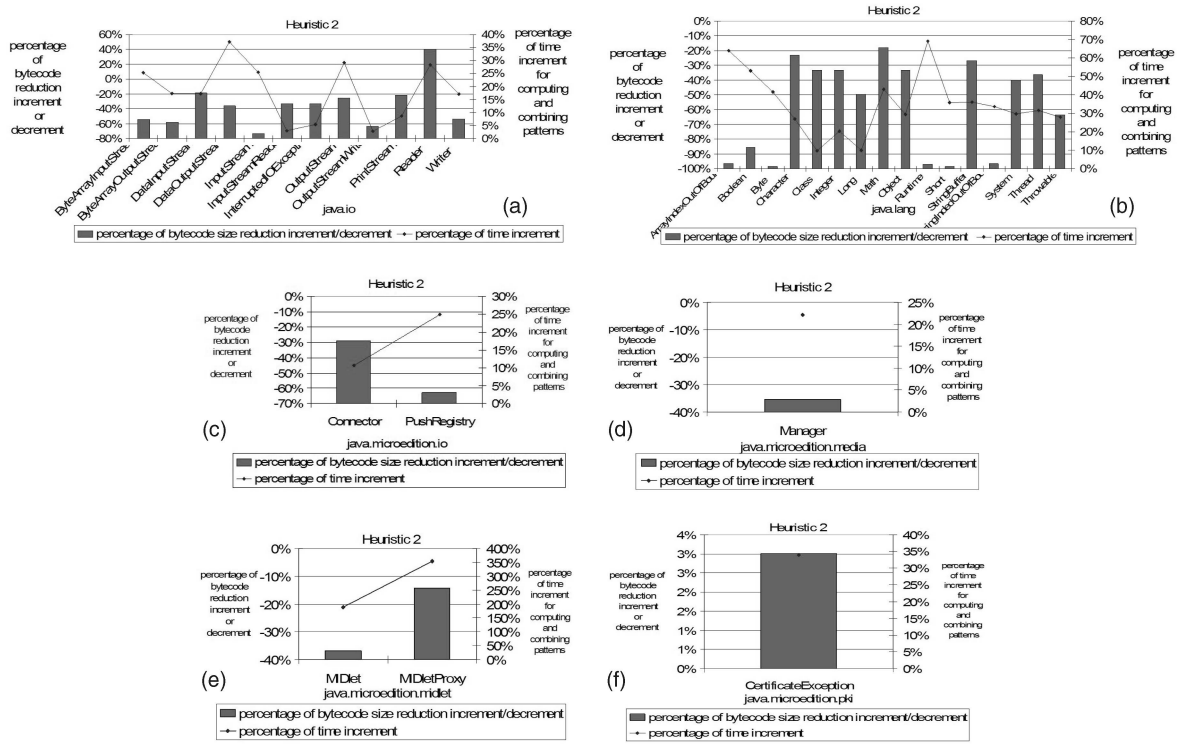


Fig. 15. The impact per MIDP class of increasing the maximum length of the patterns in the bytecode size reduction obtained and the time required for discovering and comining the patterns (second heuristic). (a) java.io. (b) java.lang. (c) javax.microedition.io. (d) javax.microedition.media. (e) javax.microedition.midlet. (f) javax.microediton.pki.

were combined using the second heuristic in the case of MIDP (Fig. 7). The specific values of FR that we used are given in Fig. 9a.

Fig. 9b gives the average decompression overhead we obtained for the aforementioned sets of bytecode sequences. As expected, the overhead is quite high (however, it is close to the overhead reported in [4] for the CaffeineMark synthetic Java programs). The overhead linearly increases with the percentage of the bytecode size reduction. In general, the overhead in the case of the bytecode sequences that were compressed using patterns that did not contain wildcards was smaller, compared to the overhead in the case of the bytecode sequences that were compressed using patterns that contained wildcards. To further elaborate on the decompression overhead, we measured the absolute time required for decompressing parameterized and nonparameterized patterns, whose length ranged from 2 to 9 bytes. The parameterized patterns comprised up to three wildcards. Specifically, the patterns of lengths 2 and 3 comprised one wildcard. The patterns of lengths 4, 5, and 6 comprised two wildcards. Finally, the patterns of lengths 7, 8 and 9 comprised three wildcards. The results we obtained are given in Fig. 9c. The time to decompress linearly increases with the length of the patterns.

5 Conclusion

In this paper, we introduced a first approach that aimed at assessing the use of statistical pattern discovery for dictionary-based Java bytecode compression. In particular, we focused on the use of agglomerative clustering, a well-known hierarchical pattern discovery technique. The main outcome revealed from our assessment is that the examined technique promotes the identification of a rich collection of parameterized and nonparameterized patterns of variable lengths, which give the opportunity for obtaining good bytecode size reduction. However, the discovery of such a rich set of patterns for a given bytecode is certainly time consuming. Moreover, the complexity of finding useful combinations of patterns out of this set that result in a good bytecode size reduction is also high. To deal with the complexity of combining patterns we investigated two heuristics. Our experimental results showed that the length of the patterns should be appropriately customized so that it does not negatively affect the compression by requiring a large number of bytes for encoding the patterns. Moreover, our experimental results showed that nonparameterized patterns and patterns that contain a relatively small number of wildcards are the most useful in our case study. However, this observation may not hold for any possible Java bytecode. Based on these remarks, a good strategy for balancing the trade-off between the time spent for discovering/combining patterns and the resulted bytecode size reduction is to apply the patterns discovery technique in an incremental manner. In a first step, the algorithm can be customized toward the discovery of small sets of patterns that contain few wildcards. Then, the small sets of patterns may serve as input to the algorithm toward the discovery of patterns that contain more wildcards, which may further improve the bytecode size reduction.

The incremental use of the agglomerative clustering algorithm is an interesting issue for further research, along with techniques that would allow pruning patterns

that are not useful, early in the patterns-discovery process. Currently, our work is also targeted toward further improving the efficiency of the pattern combination procedure. To this end, we aim at formulating the problem of pattern combination as a global optimization problem. This would allow us to investigate the use of classical global optimization techniques, such as simulated annealing, in conjunction with the proposed parameterized pattern discovery technique. Our future research further aims at the exploration of other, possibly more efficient, statistical methods for discovering patterns in Java bytecode.

APPENDIX

FURTHER RESULTS FROM THE MIDP CASE STUDY

In this section, we provide further details regarding the experiments performed for the assessment of the parameterized pattern-discovery technique discussed in this paper. Specifically, Fig. 10 and Fig. 11 provide details regarding the bytecode size reduction obtained per different MIDP class from the application of the first and the second heuristics in P_{param}, $P_{non\text{-}param}$, P_{1w}, P_{2w}, P_{3w}, and P_{4w}. Similarly, Fig. 12 and Fig. 13 give the bytecode size reduction obtained per different MIDP class from the application of the first and the second heuristics in P_{param}, $P_{non\text{-}param}$, $P_{1\text{-}2w}$, $P_{1\text{-}3w}$ and $P_{1\text{-}4w}$. Finally, Fig. 14 and Fig. 15 detail the impact of increasing the maximum length of the patterns found in MIDP from 9 to 11, in the bytecode size reduction obtained and the time required for compressing each MIDP class.

ACKNOWLEDGMENTS

This work was partially funded by the Mobius GSRT Grant for Cooperation in S&T Areas with European Countries.

REFERENCES

[1] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of Code-Size Reduction Methods," *ACM Computing Surveys,* vol. 35, no. 3, pp. 223-267, 2003.

[2] D. Rayside, E. Mamas, and E. Hons, "Compact Java Binaries for Embedded Systems," *Proc. 1999 ACM Conf. Centre for Advanced Studies on Collaborative Research (CASCON '99),* pp. 1-14, 1999.

[3] M. Latendresse and M. Feeley, "Generation of Fast Interpreters for Huffman Compressed Bytecode," *Science of Computer Programming (Advances in Interpreters, Virtual Machines and Emulators),* vol. 57, no. 3, pp. 295-317, 2005.

[4] L.R. Clausen, U.P. Schultz, C. Consel, and G. Muller, "Java Bytecode Compression for Low-End Embedded Systems," *ACM Trans. Programming Languages and Systems,* vol. 22, no. 3, pp. 471-489, 2000.

[5] M. Meilă and D. Hecherman, "An Experimental Comparison of Model-Based Clustering Methods," *Machine Learning,* vol. 42, pp. 9-29, 2001.

[6] K. Blekas and A. Likas, "Incremental Mixture Learning for Clustering Discrete Data," *Lecture Notes in Artificial Intelligence,* vol. 3025, Springer-Verlag, pp. 210-219, 2004.

[7] J. Ernst, W. Evans, C.W. Fraser, S. Lucco, and T.A. Proebsting, "Code Compression," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '97),* pp. 358-365, 1997.

[8] Q. Brandly, R.N. Horspool, and J. Viter, "JAZZ: An Efficient Compressed Format for Java Archive Files," *Lecture Notes in Artificial Intelligence,* vol. 3025, Springer-Verlag, pp. 210-219, 2004.

[9] M. Franz and T. Kistler, "Slim Binaries," *Comm. ACM,* vol. 40, no. 12, pp. 87-94, 1997.

[10] W. Pugh, "Compressing Java Class Files," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '99),* pp. 247-258, 1999.

[11] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical Extraction Techniques for Java," *ACM Trans. Programming Languages and Systems,* vol. 24, no. 6, pp. 625-666, 2002.

[12] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. 25th IEEE Int'l Symp. Microarchitecture (Micro-25),* pp. 81-92, 1992.

[13] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," *Proc. Int'l IEEE Conf. Computer Design: VLSI in Computers and Processors,* pp. 270-277, 1994.

[14] H. Lekatsas and A. Wolfe, "SAMC: A Code Compression Algorithm for Embedded Processors," *IEEE Trans. Computer Aided Design,* vol. 18, no. 12, pp. 1689-1701, 1999.

[15] K.D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded Risc Processors," *ACM SIGPLAN Notices,* vol. 5, pp. 139-149, 1999.

[16] "CodePack: PowerPC Code Compression Utility User's Manual v3.0," technical report, IBM Corp., 1998.

[17] C.R. Lefurgy, P.L. Bird, I.-C. Chen, and T.N. Mudge, "Improving Code Density Using Compression Techniques," *Proc. 30th ACM/IEEE Int'l Symp. Microarchitecture (Micro-30),* pp. 194-203, 1997.

[18] S.K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler Techniques for Code Compaction," *ACM Trans. Programming Languages and Systems,* vol. 22, no. 2, pp. 378-415, 2000.

[19] B. De Sutter, B. De Bus, and K. De Bosschere, "Sifting out the Mud: Low Level C++ Code Reuse," *Proc. 17th ACM SIGPLAN Conf. Object-Oriented Programming Systems Languages and Applications (OOPSLA '02),* pp. 275-291, 2002.

[20] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Eighth IEEE Working Conf. Reverse Eng.,* pp. 301-309, 2001.

[21] W. Cheung, W. Evans, and J. Moses, "Predicated Instructions for Code Compaction," *Proc. Seventh Int'l Workshop Software and Compilers for Embedded Systems (SCOPES '03),* pp. 17-31, 2003.

[22] W. Evans and C. Fraser, "Bytecode Compression via Profiled Grammar Rewriting," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '01),* pp. 148-155, 2001.

[23] R. Komondoor and S. Horwitz, "Effective, Automatic Procedure Extraction," *Proc. 11th IEEE Int'l Workshop Program Comprehension (IWPC '03),* pp. 33-43, 2003.

[24] Y. Bengio and S. Bengio, "Modeling High-Dimensional Discrete Data with Multi-Layer Neural Networks," *Advances in Neural Processing Systems 12,* S. Solla, T. Leenand, K.-R. Móller, eds., pp. 400-406, MIT Press, 2000.

[25] T. Lindholm and F. Yellin, "The Structure of the Java Virutal Machine," *The Java Virtual Machine Specification,* SUN Microsystems, 1999.

Dimitris Saougkos received the BSc degree in 2004 from the Department of Computer Science at the University of Ioannina, Greece. He received the MSc degree in 2006 from the same department. His main research interests include code compression, compilers, programming languages, middleware, and mobile and embedded computing systems.

George Manis received the Diploma degree in electrical and computer engineering in 1992 from the National Technical University of Athens (NTUA). In 1993, he received the MSc degree from the Queen Mary and Westfield College (QMW), University of London. In 1997, he received the PhD degree in computer engineering from NTUA. He is a lecturer in the Department of Computer Science at the University of Ioannina, Greece. His main research interests include computing systems and architectures, compilers, and biomedical engineering.

Konstantinos Blekas received the Diploma degree in electrical engineering in 1993 and the PhD degree in electrical and computer engineering in 1997, both from the National Technical University of Athens (NTUA). Since 2002, he has been with the Department of Computer Science, University of Ioannina, Greece, where he is currently a lecturer. His research interests include artificial intelligence, statistical pattern recognition, machine learning, and bioinformatics. He is a member of the IEEE.

Apostolos V. Zarras received the BSc degree in computer science in 1994 from the Department of Computer Science, University of Crete. In 1996, he received the MSc degree in distributed systems and computer architecture from the same department. He received the PhD degree in distributed systems and software architectures in 2000 from the University of Rennes I, France. He holds a lecturer position in the Department of Computer Science, University of Ioannina, Greece. From 1999 to 2001, he worked as a researcher at the ARLES Group of INRIA-Rocquencourt. His research interests include middleware, model-driven architecture development, and pervasive computing. He is a member of the IEEE.