Java Bytecode Compression for Embedded Systems

Karim Ammous, Nasser Benameur, Smail Niar, Mourad Abed.

Université de Valenciennes et du Hainaut-Cambrésis,
LAMIH/ROI, ISTV2,
Mont Houy 59313,
Valenciennes Cedex 9 - France
{karim.ammous, nasser.benameur, smail.niar, mourad.abed}@univ-valenciennes.fr

1. Introduction

Recent advances in technology have permitted more and more hardware resources to be injected into new embedded systems. Nevertheless, the applications for which these systems are designed (such as MPEG7, UMTS, voice/image recognition programs, etc.) require more and more resources, forcing developers of applications for embedded systems to use the fewest hardware resources possible. Consequently, faced with this reduction in resources like memory capacity and clock frequency, embedded systems can only contain a small set of applications. These applications must be both light and non-greedy in terms of execution time.

In this paper, we focus particularly on Java bytecode compression, and we propose a software solution to address the problematic described in the previous paragraph. Our two-part approach is an interesting alternative to existing techniques. First, by using a bytecode compressing technique, our approach makes it possible for an application to take up less memory in the embedded system. In addition, it also allows more applications to be loaded and permits applications whose required resources are higher than those of the target embedded system to be executed. Second, our techniques can, in some circumstances, speed up the execution of embedded applications.

2. The bytecode factorization method

Java bytecode factorization consists of replacing the redundant code sequences in the bytecode with new instructions, called macro-instructions. Such a technique is made possible by the existence of unused instructions in the specification of the Java virtual machine. The macro-instructions are defined in the macro table and are appended to the ClassFile[1] in the optional component at the end of the compression process. Thus, the Java virtual machine instruction set is enhanced by these macro-instructions, which allows the new instructions to be interpreted. In this section, we provide details concerning the factorization process. There are three phases: pattern detection, optimal pattern sequence selection and code transformation.

In this section, we provide details concerning the factorization process. There are three phases: pattern detection, optimal pattern sequence selection and code transformation.

2.1. Pattern detection

Pattern detection occurs during a single passage of the code. The bytecode is examined, instruction-by-instruction, and all the possible patterns and their respective occurrences are recorded. A benefit estimation for each of these patterns designates those that have a positive effect on the compression ratio, and thus should be kept. The remaining patterns will be ignored. This estimation is relevant due to the overlapping of identical or different-sized patterns. Two patterns overlap when they have common instructions. The overlapping of two patterns involves a benefit reduction of the second pattern. As a direct consequence of this overlapping, the pattern benefit can be reduced to zero, and must be excluded from the eligible pattern set. A solution that addresses this problem is presented in the following section.

[1] A file format that contains standard and personalized components.

2.2. Selection of the optimal pattern sequence

This phase has a direct effect on the compression ratio and consequently on the efficiency of our factorization algorithm. Once the set of eligible patterns is defined, the patterns that contribute to the compression stage must be determined and their order of application noted. To reach this end, all the pattern sequences applied throughout the program to be compressed are represented in a tree form. Each tree node represents the original program state after factorization according to a given pattern. The original program is at the root of the tree. The path from the root to a node indicates the pattern sequencing order. The problem consists in finding the optimal pattern set represented by the leaf node that provides the most significant benefit.

This kind of problem is classified among NP-Complete problems. Consider a tree with a branching factor N (the number of eligible patterns). At depth d, there will be N*d nodes. This causes an exponential explosion of the tree size. To reduce the complexity and consequently determine the optimal solution, we developed two different techniques. The first technique precedes tree construction. It is called the dependency analysis technique. The second technique, named benefit revaluation, is used progressively during tree construction.

2.2.1. Dependency analysis

Since the pattern sequencing order comes into play only when patterns overlap, dependency preprocessing is necessary. This preprocessing discerns two types of patterns.

- Free patterns: a pattern is considered "free", if it has no overlapping relationship with any other pattern. Free pattern order in the final sequence is unimportant since it does not alter the remaining pattern benefit.

- Semi-free patterns: semi-free patterns form subsets according to a dependency relationship. Overlapping patterns, with no relation to other patterns outside the set, form a semi-free pattern set. Set construction is done recursively.

Overlapping problem resolution has an exponential factor. Nevertheless, through the use of the above dependency analysis technique, the problems resolved are of a lower complexity.

2.2.2. Benefit revaluation

The benefit corresponding to each node is progressively revised during tree construction, by taking into account the variation in the number of patterns occurring. The branching factor is then lower or equal to the number estimated previously. Thus, tree width is reduced. Tree algorithm construction starts with all the elements of a semi-free pattern set as parameters, and it continues through the following stages:

Repeat

While the eligible pattern set of the current node is not empty, do

-1- choose a pattern among the eligible patterns;

-2- re-estimate the pattern benefit;

-3- if the benefit is null, return to stage 1;

-4- if the benefit is not null, build an offspring node from the current one; compress the program according to the selected pattern. Eliminate these patterns from the eligible pattern set of the parent node. Then, the current node becomes the offspring node;

End while

-5- go back to the parent node and decompress the program according to the offspring node;

Until the root eligible pattern set is completely treated.

We reiterate this algorithm, as many times as there are semi-free pattern subsets.

2.3. Code transformation

We replace patterns in the original bytecode by the macros that are assigned to them. Code annotations, that allow the embedded JVM to interpret compressed bytecode, are added to the macro table. This table is forwarded with the compact code to the embedded support.

3. Experimental results

In order to show the effectiveness of our compression technique, we implemented the factorization algorithm as well as a performance assessment series. The programs used to support our experiments are part of a benchmark called CaffeineMark, version 3.0. They implement a test series that measures the speed of the Java programs. This supposes that the application is optimally implemented so that the encumbrance of the embedded system on which it is supposed to be run is minimal. This feature gave us a credible reference point for comparing our approach with other compression techniques. The following table represents the experimental results obtained by the application of our factorization program to CaffeineMark package (TAB.1).

Class name	Original size (KB)	Size after compression (KB)	Compression benefit (%)
BenchmarkAtom	0.38	0.38	0
BenchmarkMonitor	0.14	0.14	0
BenchmarkUnit	4.74	3.45	27
CaffeineMarkEmbeddedApp	2.62	2.33	11
CaffeineMarkEmbeddedBenchmark	12.3	7.97	35
FloatAtom	4.99	3.79	24
LogicAtom	5.85	3.89	33
LoopAtom	3.21	2.46	23
MethodAtom	2.77	2.48	10
SieveAtom	2.48	2.22	11
StopWatch	2.22	1.54	31
StringAtom	2.93	2.82	4
Total size	44.63	33.47	25

TAB.1 - Summary table of the results obtained

An analysis of Table 1 shows that the highest compression ratio (35%) was recorded for a relatively large class size. This observation proves that the bigger the class size, the greater the probability of finding redundant code sequences in the bytecode. The last line of the table establishes a balance assessment for all the classes. Our factorization algorithm reduces the overall program size from 44.11 KB to 33.47 KB. This size decrease represents a 25% compression benefit, on average.

These numerical results underscore the value of such a compression technique since the benefit achieved is greater than other solutions that do not violate the Java virtual machine specification.

4. Conclusion

The studied approach has allowed us to show the significant improvement that can result from using a specific bytecode technique. The experimental results obtained by the application of our algorithm on a benchmark and an input test demonstrate a reduction of the compression rate by an average of 25% and maximum of 35%. These results reflect the relevance of the solutions suggested to the problems encountered during the bytecode compression process. Given its compression rate, this technique represents a high-performance solution.

In the following phase of our project, we intend to seek out ways to reduce this loss of speed. Aggressive acceleration techniques will be added to our overall optimization strategy.