Speedup Prediction for Selective Compilation of Embedded Java Programs ⋆ ⋆⋆ ⋆⋆⋆

Vincent Colin de Verdière[1], Sébastien Cros[1], Christian Fabre[1], Romain Guider[1], and Sergio Yovine[2]

[1] Silicomp Research Institute, 2 Ave de Vignate, F-38610, Gières, France
[2] Verimag, 2 Ave de Vignate, F-38610, Gières, France

Abstract. We propose a profile based code selection scheme for an AOT Java compiler. This scheme relies on a model that accurately predicts the speedup of a given selection. The model takes into account the cross-call patterns of the application. This approach allows us to reduce the size of compiled code significantly for several benchmarks.

1 Introduction

Java has been recognized as an attractive language and platform to program embedded systems. There are several reasons for this. Embedded systems use a large variety of processors. The portability of Java is very appealing in this context. Embedded systems have generally a limited memory size. Because Java has been designed to ease the communication of programs through the internet, one of it's design goals is the small size of executables (which is achieved by the use of a stack based bytecode). This feature is also beneficial to embedded systems.

However, in order to limit energy consumption, embedded systems are not built around high performance processors. So neither can they accommodate the low performance of purely interpreted Java nor the overhead of a JIT compiler [6, 10]. One solution to the performance problem is the use of Ahead of Time compilers (AOT compilers)[11–13]. Because the compilation is done offline with such compilers, there is no runtime overhead imposed by compilation.

There still is a problem with AOT compilers: compilation increases the size of code. To address this problem, we have developed the TurboJ[13] compiler that partially compiles an application. The runtime, which is based on a Java Virtual Machine (JVM), allows a mixed execution mode where both compiled code and interpreted code are executed.

The TurboJ compiler allows code to be selected for compilation at the method level. The compiler is fed with the entry point of an application (from which it builds the list of classes that the application uses) and a list of methods that

⋆ Partially supported by French RNTL Project Expresso and FONGECIF.
⋆⋆ Contact authors: guider@ri.silicomp.fr, Sergio.Yovine@imag.fr.
⋆⋆⋆ Proc. EMSOFT'02, Grenoble, 7-9 October, 2002. LNCS 2491, Springer-Verlag.

it must compile. Any other method remains interpreted. This feature is very important and useful to solve the problem of code-size increase. However, in practice, it is not an easy task to select the methods to be compiled. In fact, it is not easy at all to figure out which methods are the most interesting to compile in order to get the best performance without expanding too much the size of the code.

The general problem that we deal with is as follows: given a memory size limit l what is the set of methods that when compiled produces the most efficient program whose size is lower than l? This corresponds to an optimization of programs for performance. The problem can also be seen the other way round: given a speedup s what is the method set that, when compiled produces the smallest program with the expected performance. This corresponds to an optimization of programs for size.

A first answer is to use a profiler and select the set of methods where the program spends most of its time[3]. This approach may give good results for some applications; however, experience proved that there are programs for which the result of such an approach is extremely poor: we encountered programs that were slower than their interpreted version.

Experience also shows that an important factor of performance degradation is the context-switching done when calling compiled code from interpreted code or when calling interpreted code from compiled code(cross-calls). In this paper we present a model of performance of compiled Java programs that take the impact of cross-calls into account. An algorithm to select compiled methods that uses this model to guide its choice has been implemented in the tool TurboJ.

Section 2 is devoted to the description of TurboJ, our AOT compiler and to the description of the profilers we developed. In Section 2.2 we describe a first, naive, approach for partitioning programs and show its limits. In Section 3 we propose a model of program performance that takes more elements into account to circumvent the limitations of the first approach. In Section 4 we present a method selection algorithm that exploits the model presented in the previous section. We discuss related work in Section 5. Then we conclude and discuss possible enhancements of our work in Section 6

2 Existing framework

2.1 The TurboJ Compiler and Profiler

TurboJ[13] is a bytecode to native code compiler. It is an AOT compiler that supports mixed-mode execution (compiled/interpreted) of Java programs. Code can be selected for compilation at the method level. Mixed-mode execution is achieved by relying on a VM to execute interpreted code. The VM also serves as a runtime environment for the compiled code. Beside the possibility of mixed-mode execution, the advantage of this approach is that it makes it possible to cope with the full range of Java features such as reflection or dynamic loading.

[3] It is a well known rule of thumb that most programs spend 80% of the time in 20% of its code.

TurboJ is used along with a profiler that allows the extraction of information about Java programs that is useful to partition applications (e.g., number of bytecodes executed by a method, frequency of the call of a method at a call site, etc.). This profiler works by instrumenting the bytecode so that it is fully portable: it can be used with any VM and on any platform. Another advantage of our profiler is that the results reported are precise (no sampling is done) and are not influenced by the profiling because we count events instead of measuring time (i.e., no probe effect).

2.2 A naive model of performance of programs

The best possible selection is the set of methods that have the best speedup up to a given size of bytecode. Compiling in isolation each method of a program and measure the speedup it provides would not be tractable for obvious reasons. Instead we use a model of the speedup that allows to predict the speedup provided by a method without actually compiling it and running the program. It only requires an interpreted run of the instrumented program to extract execution profiles.

The first model that we tried is very simple: we consider a constant acceleration factor among all bytecodes. Under this model, the best selection of size l is the selection that maximizes the number of executed bytecodes.

We did not consider useful in practice to use an optimization package to solve this problem. Instead, we use a very simple heuristic: we present the methods sorted by decreasing number of executed bytecodes and select the first ones up to a given static size.

Our selection tool has a graphical interface that presents the results of the instrumented runs. Results are presented with sorted methods on the x-axis and their respective number of executed bytecodes on the y-axis.

2.3 Experiments (limitations of the approach)

Under the hypothesis that most of the execution time is spent in a small portion of the program, the speedup curves that result from the application of the above selection strategy should have a high slope at the beginning and become quickly almost horizontal. The initial high slope would also mean that, in most cases, the first sorted methods provide a good ratio of executed bytecode over size.

However, in practice the model of performance does not always predict the speedup of the partially compiled applications. In order to assess the problem, we ran experiments on several real-world applications. For lack of space we present here two of the most significant ones:

- `compress`, from the SpecJvm98 benchmark suite [5] and
- `xalan`, the Apache XSL processor [8].

Compress is a computation kernel code composed of 41 methods which is representative of code that can be found in embedded systems. Xalan is a large object-oriented application comprising 2372 methods.

Our model asserts that the more bytecodes are compiled (whatever bytecode they are) the faster the program is. To assess this, our experiment consists in the observation of the speedup progression in parallel with the progression of the number of compiled bytecodes. We generated, for each of our benchmarks, a suite of growing selections of methods - the methods being sorted by decreasing number of executed bytecodes. For each of these selections, we compiled the selected methods and ran the program. The reported results are the speedups of each of these runs as compared to the fully interpreted program. We ran our tests on a Intel 730MHz-Pentium III running Linux.

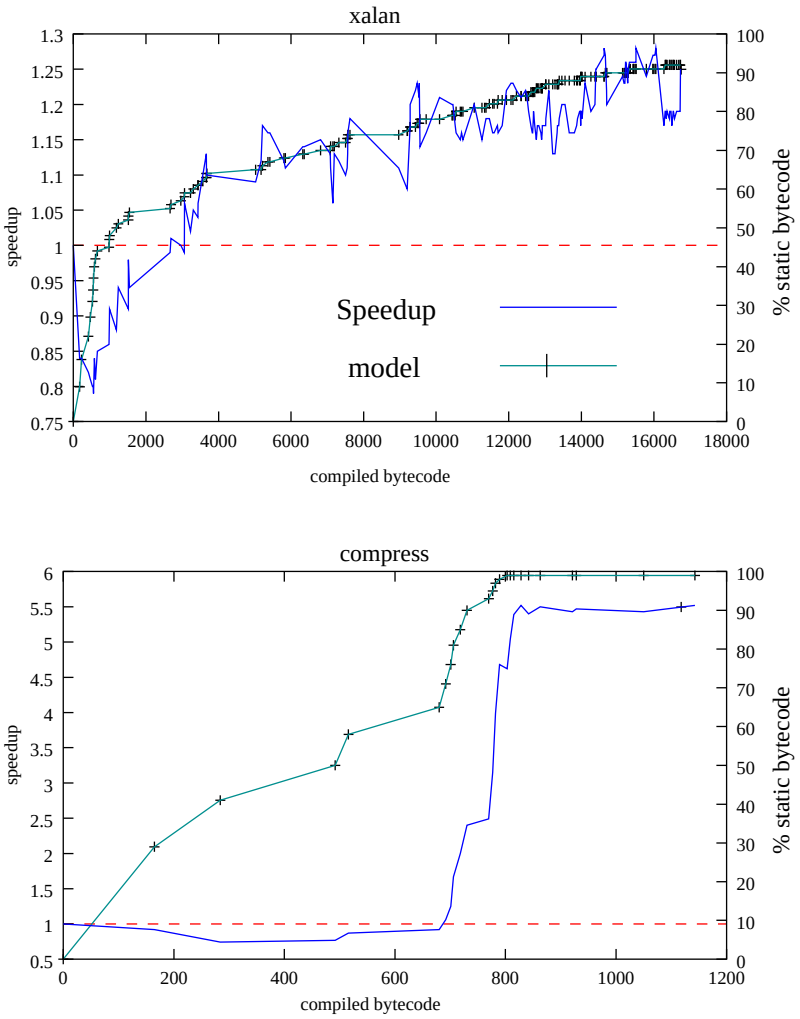

Fig. 1. Progression of the Speedup with the number of Compiled Bytecodes

The results of our experiments (see Fig. 1) showed that the model is far from being accurate. The speedup obtained in the case of Compress for up to 700 compiled bytecodes has an inverse progression as compared to the model. For Xalan, the speedup follows on average the model. However, it makes a lot of local oscillations that are not negligible. Moreover, there is a regression of performance when compiling up to 3000 bytecodes.

These results are not acceptable for mainly two reasons. First, the selection procedure would lead to selecting a set of methods which will slow down the application when compiled (e.g., Compress). Second, it shows unpredictable oscillations, that is, selecting one more method can dramatically reduce the speedup (e.g., Xalan). The major drawback of this approach is that it does not provide any convenient way to detect such problems. Therefore, it is unable to faithfully predict the speedup of a selection after compilation.

This raises some questions. First, how can we explain that compiling some bytecodes slows down the application? Second, are there bytecodes that provide more speedup than others when compiled? Or are there bytecodes that slow down the application when compiled until a small number of other bytecodes are compiled? In the next section we propose a model that solves the problems we observed on the benchmarks.

3 Taking calls/context-switching into account

Our investigations showed that cross-calls (calls from interpreted code to compiled code or from compiled code to interpreted code) are responsible for the slow down of applications by the compilation. Cross-calls are done through stubs that convert the argument passing convention and install exception handlers. These stubs imply costly (yet necessary) computations that are not done in case of direct calls. We made several experiments to assess the relative cost of the calls and its impact on the speedup provided by a selection.

3.1 The Cost and Impact of Cross-calls

We ran micro-benchmarks that allowed us to isolate the speedup gained for each of the four kinds of call in various situations. The results of these measurements are summarized in the Table 1.

Speedup	Int/Int	Int/Compl	Compl/Int	Compl/Compl
`invokevirtual`	**1**	0.3	0.6	5
`invokeinterface`	**1**	0.5	0.6	2.5
`invokestatic`	**1**	0.4	0.6	5
`invokespecial`	**1**	0.4	0.5	5

Table 1. Speedups for the 4 kinds of calls.

There is no relation between the lines of the table. For each line, the interpreted-interpreted cost is taken as reference. The other figures are the speedup of the call as compared to the performance of the interpreted-interpreted call. It is clear from these figures that the cross-calls can actually slow down an application. For instance an interpreted `invokevirtual` is 70% slower when it calls compiled code than when it calls interpreted code.

To further convince the reader that the cross-calls are responsible for the slow down, we ran experiments on our 2 benchmarks. The experiment consists in measuring the number of cross-calls executed for each selection. We report (see Fig. 2), for each selection, its execution time and the amount of cross-calls in this number of compiled-bytecode executed. We observe that the amount of cross-calls and the execution time have related progressions. In particular, peaks of cross-calls coincide with peaks of execution time.

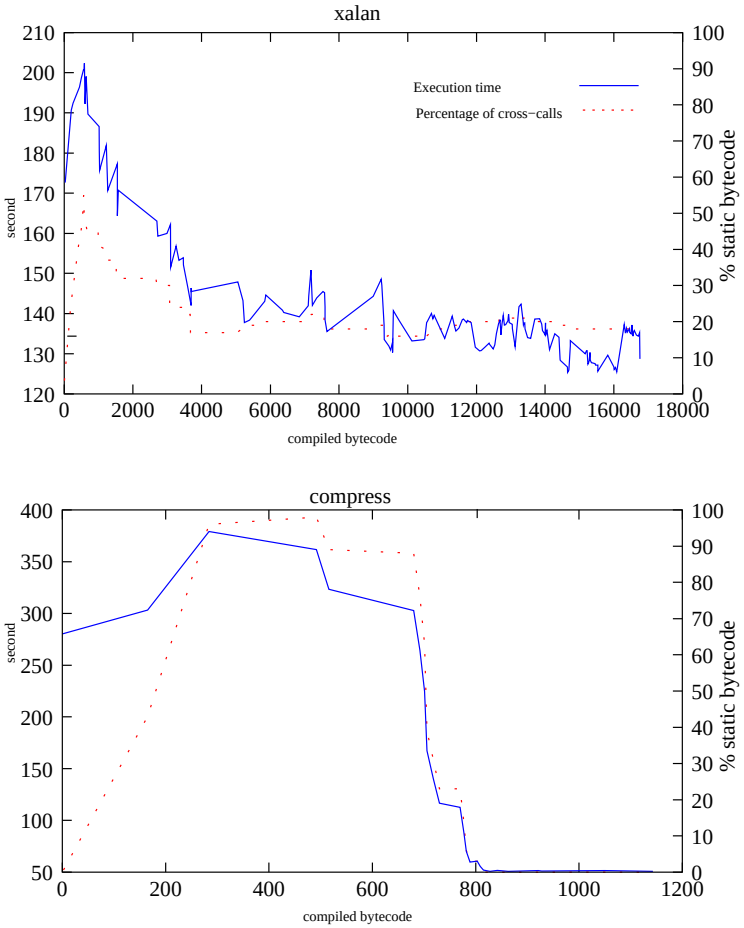

Fig. 2. Impact of Cross-Calls

3.2 A new model of application performance

From the experiments above, it is clear that the amount of cross-calls needs to be small enough for the compilation to provide some speedup. Moreover, compilation not only does not provide the same speedup for all the bytecodes but it may provide an anti-speedup when calls become cross-calls. This suggests that the speedup is inversely proportional to the number of cross-calls.

Given a set of methods M and a sub-set s of these methods, we note N_s the number of bytecodes executed by s on a run and C_s the number of cross-calls executed during a run when s is compiled. The model we propose to use is then

$$R_s = \frac{N_s}{C_s}$$

The intended use of this model is not to get an absolute measure of the speedup obtained when compiling a selection. Instead, we simply use it to compare selections so as to be able to select the best one with respect to our model.

3.3 Experiments

To assess our new model, we re-ran our benchmarks and for each selection we computed the value of R_s and measured the corresponding speedup. Results of these measurements are reported in Fig. 3. What we observe on these curves is that the model follows most of the progression of the speedup. That is most of the peaks of the speedup curves coincide with peaks of the model curve. This is particularly the case for Compress where the model and the speedup progression are very similar.

Thus, the model we propose is much better than the naive one that is commonly used in dynamic compilation systems. Moreover, this model has several important qualities:

- it is simple so that its evaluation can be done at low cost;
- its evaluation can be done in an incremental fashion as the selection evolves, there is no need to re-compute the model for the whole selection when a method is added;
- it does not contain any constant related to the platform.

The last point is quite important for us. In fact, our profilers instrument the bytecode so as to be independent of the JVM. With this model, the whole chain (instrumentation, profiling and method selection) remains independent of the platform.

4 A partitioning heuristics

4.1 A Greedy Algorithm to Select Methods

Having a model, we need an algorithm that finds good method selections using the model. While the naive model was linear, the model with cross-calls is not.

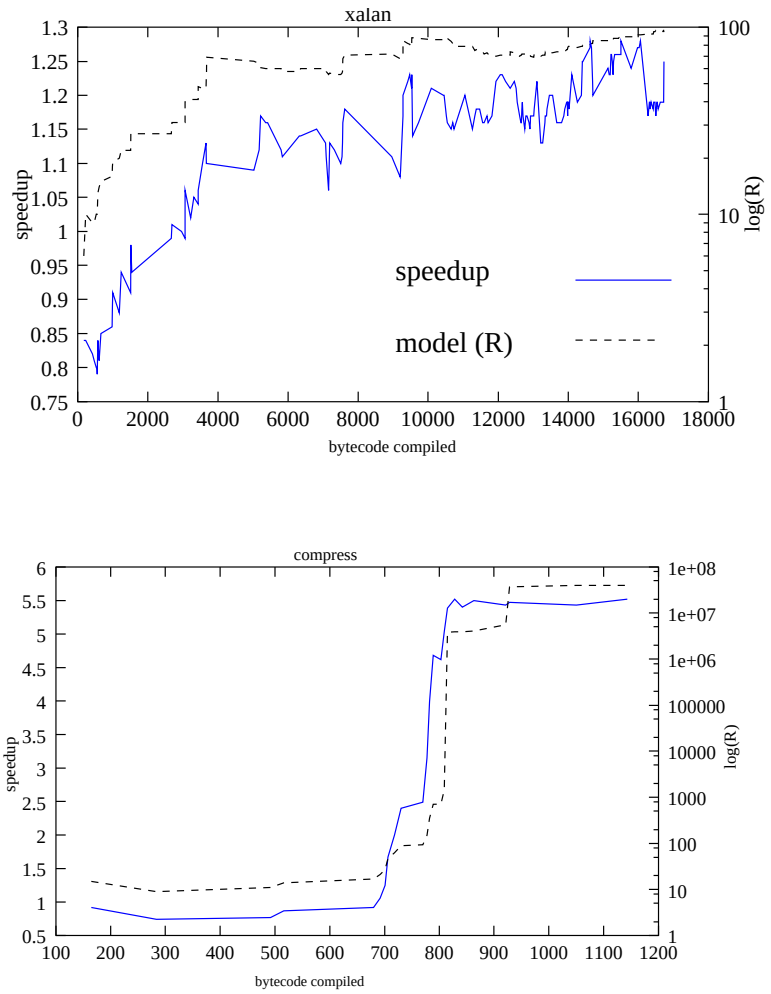

Fig. 3. The new model of speedup

Adding a method influences both N_s and C_s. Furthermore, it is not monotone with respect to N_s: adding a method to the selection can turn direct calls to cross-calls and so lower the model value. We propose a heuristic based on a greedy algorithm to find solutions.

Starting from an initial selection, our algorithm enhances the value of R_s by adding a method to the selection at each step. The algorithm selects the method that, if selected maximizes the increase of R_s. At each step, the search for a method is restricted to the set of methods that have an incident edge with some selected method. The values N_s and C_s are maintained at each step so that the value of R_s can be evaluated in an incremental fashion without looking at the entire selection.

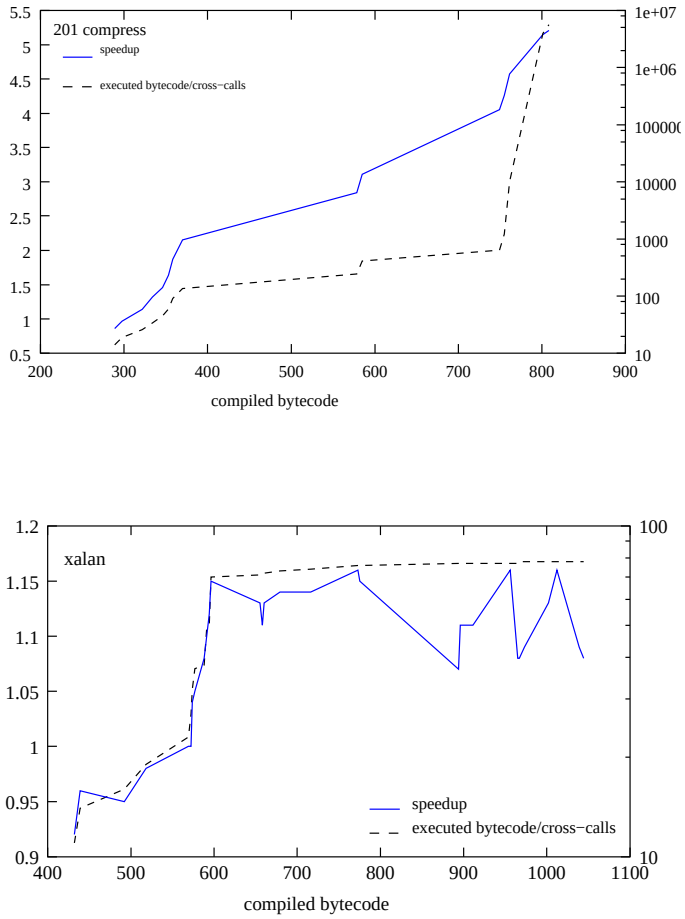

Fig. 4. Evolution of the speedup as compared to its model following the greedy heuristic

4.2 Results

One first result is that our heuristic finds successive selections with a growing ratio R_s. We represent the evolution of the model and the corresponding speedup for our two examples Fig. 4. These are two representative results out of several other experiments. The figure shows that these curves (the evolution of the model following the greedy algorithm) are useful tools in the search for a good selection. In the case of Compress, the progression of the ratio and of the real speedup are similar in that both are only growing.

Our model and heuristic provide a reliable tool to explore the tradeoff between application size and execution performance. The model does not reflect some oscillations in the case of Xalan, but the choice that the model leads to turns out to be the correct one with respect to the speedup curve. More generally, we observed in all our experiments (including some that are not reported in this paper) that when the model is growing then the real speedup is always growing, but when the value of the model is stable, there may be some oscillations of the real speedup. This makes our tool reliable because there is no reason to compile more methods if the model reports that they will not provide any additional speedup.

To compare with the naive approach, we report in Table 2 the relative static quantities of bytecode that need to be compiled in both approaches to reach a given speedup. The figures for Compress and Xalan are drawn from the curve represented Fig. 4. We take, as a reference, the best solution found with the naive approach. Speedups are given as a percentage of the reference speedup (i.e., 100% of speedup corresponds to the maximum speedup achieved with the naive method). Static quantities of bytecodes are reported as a percentage of the reference static quantity of bytecodes (which corresponds to the reference speedup).

application	% Speedup	% Compiled bytecode	
		First method	Greedy
Xalan	89	35.6	4.0
	97	98.4	5.2
	100	100	7.8
	108		24.6
Compress	27	61.3	18.1
	39	62.4	32.1
	56	67.5	50.8
	82	69.8	66.1

Table 2. Comparison of the results of the naive approach and the greedy algorithm

For Xalan, only 4% of the reference quantity of bytecode is selected to be compiled by the greedy algorithm to get 89% of the reference speedup. As compared to the 35% of bytecodes selected with the first approach to get the same

speedup this is an improvement of 88%. To reach the reference speedup, only 7.8% of the reference quantity of bytecode needs to be compiled which yields 92% of improvement. Results are a bit less spectacular for Compress but we still observe that our new model significantly reduces the total amount of bytecode to be compiled.

The figures of Table 2 also show that we can improve the result both in terms of size and of performance: for Xalan, by compiling 24.6% of the bytecode needed to get 100% of the reference speedup we actually obtain 108% of this speedup. This means that we go beyond the best speedup observed with the naive approach while (at the same time) reducing by 75.4% the size of the compiled bytecode.

5 Related work

Profiling information has been used for a long time to guide optimizers in static compilers. One trend of work [3, 4] uses profiling information to detect frequently executed scenarios and transform the program so as to be able to optimize the frequently executed code. More recently, profiling information has been used in dynamic optimization systems. Some work only uses it to detect so-called hot spots [6, 10] while others uses it to select the optimizations applied to the code [1, 2]

There are many AOT compilers that mix interpreted and compiled Java code. Harissa [11] integrates an interpreter in its runtime environment to implement dynamic loading of code. TowerJ [12] also integrates an interpreter in its runtime environment and allows selective compilation but there is no mention of an assistance tool to partition applications. The company OTI developed a JVM called J9 and an AOT compiler that generates code that relies on the JVM as a runtime environment, much like TurboJ. They also have the possibility to selectively compile methods and they use profiling information to select methods that are worth compiling [7]. Their profiling approach is based on the Java interface for profiling [9] so that they implement a sampling profiler that records execution time and call frequency of each method. They select methods for compilation on the basis of their respective execution time. This is close to our naive approach. However, we see some limitations to this approach because they have no convenient way of identifying the kind of bytecodes executed in the methods that run for long times. As a consequence they cannot figure out the call patterns. Besides, our profiling method is less biased because we count events instead of measuring time (no probe effect). Moreover, the OTI approach is not portable since it requires writing JVMPI native code.

6 Conclusion

We propose a profile based code selection scheme for an AOT compiler that takes into account cross-call patterns of an application. We have shown that this approach allowed us to reduce the size of the compiled code significantly

as compared to the algorithm that relies on the relative quantities of bytecode executed.

The greedy heuristic that we propose needs to be inseminated with an initial selection. On the one hand, a rich initial selection will offer the greedy algorithm more possibilities of selection at each step and, therefore, lead to better selections. On the other hand, a too large initial selection may contain a method that cannot be optimized by the greedy algorithm and compromise the results. Currently, we use the list of methods sorted by growing number of executed bytecode to build the initial selection. This strategy turned out to provide good results. Besides, if the prefixes of the list do not contain interesting an initial selection, the programmer can select methods individually through a graphical user interface that represents the classes of the program as well as the call graph.

One of the key features of our heuristic is that it is incremental and, therefore, fast. The greedy algorithm considers a sub-set of the methods of a program that depends on the size of the selection, not on the size of the program. Because the heuristic is fast, the programmer gets quickly feedback in the form of the curve of the evolution of the model. This allows testing several solutions for the initial selection in a short amount of time.

Due to the expected large size of the call graph (e.g., 2372 methods and 5008 edges in the case of Xalan), we do not envision the use of global and exact optimization algorithms. The greedy algorithm allows us to propose a suite of solutions that are built incrementally and that can be used by the programmer to explore the size/performance tradeoff. It is unclear whether a global approach would provide the same possibilities.

Our results are promising but can be enhanced in several ways.

One direction consists in taking recursion into account. Recursion corresponds to cycles in the call graph. A possible and simple approach would be to reduce the call graph to its strongly connected components and consider strongly connected components as compilation units in place of methods.

In our greedy algorithm, the choice of a method is restricted, at each step, to the set of methods that have an incident edge to a selected method. This allows us to reduce the size of the search space at each step. It would be actually possible to consider all the methods. A method that has no incident edge with a selected method, if selected, will only introduce cross-calls so that it is possible to evaluate the increase of the value of R_s for each of such methods independently of the selection and of other methods: it is simply the ratio of its quantity of executed bytecode and of the number of calls it makes. So if we maintain a list of non-candidate methods sorted by their local ratio, the head of the list (the method that has the maximum local ratio) is the best choice among all the non-candidate methods. This allows to open the choice of a candidate to all the methods at each step while increasing the size of the search space by only one. This approach would eventually lead to better selections where the ratio becomes flat (Fig. 4).

Finally, beside calls, we consider that all other bytecodes are accelerated in the same way. This is actually not true since, for instance, runtime type checks

are not accelerated. We believe that we can still refine our model to take this into account.

References

1. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proc. of ACM-SIGPLAN OOPSLA'00*, 2000.
2. Matthew Arnold, Michael Hind, and Barbara G. Ryder. An empirical study of selective optimization. In *Proc. International Workshop on Languages and Compilers for Parallel Computing*, 2000.
3. Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.
4. W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W.-M. Hwu, R. Hank, and J. Gyllenhaal. Using profile information to assist advanced compiler optimization and scheduling. In *Proc. of Advances in Languages and Compilers for Parallel Processing*. Pitman Publishing, 1993.
5. Standard Performance Evaluation Corporation. Spec jvm98 benchmarks. http://www.spec.org/osg/jvm98/.
6. B. Delsart, V. Joloboff, and Eric Paire. Jcod: A lightweight modular compilation technology for embedded java. In *Accepted for publication in EMSOFT'02*, 2002.
7. Aldo H. Eisma. Feedback directed ahead-of-time compilation for embedded java applications. In Uwe Assmann, editor, *JOSES Workshop at ETAPS'01*, Genova, 2001.
8. The Apache Software Foundation. Xalan-java version 2. http://xml.apache.org/xalan-j/index.html.
9. Sun Microsystems Inc. Java virtual machine profiling interface (jvmpi), java 2 sdk, standard edition documentation, version 1.2.2. java.sun.com/products/jdk/1.2/docs/index.html.
10. Sun Microsystems. The java hotspot performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html, april 1999.
11. Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proc. of Usenix COOTS'97*, Berkeley, 1997.
12. Tower Technology. Towerj 3.0. www.towerj.com.
13. Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler. Turboj, a java bytecode-to-native compiler. In *Proc. of LCTES'98*, volume 1474 of *LNCS*, 1998.