An Inter-procedural Approach for Optimizations of Java Programs

Antonio Magnaghi, Shuichi Sakai, Hidehiko Tanaka
The University of Tokyo

1 Introduction

Java is capturing an increasing interest both in academic and industrial contexts. Code portability ("Write once-Run everywhere") is one of the most relevant aspects of the language, and it relyes on the interpretation of Java byte-code by the JVM. As a drawback, program execution can be considerably slower compared to native code. Our research activity aims at improving execution performance of Java programs through the extraction of implicit parallelism [1, 2, 3]. This paper reports on the source-to-source Java optimizer that we are developing in order to reconstruct source code into a semantically equivalent multi-threaded program. As original contribution, the adopted approach directly addresses the rich and articulated Object Oriented features of Java. Source code static analysis is performed on an inter-procedural level through the characterization of method invocation effects in order to achieve a better description of program dependencies.

2 Target Optimizations

We are focusing on task-level parallelism extraction [1, 2], hence we are developing a framework for statically analyzing program behavior beyond method invocation boundaries. As follows a Java code fragment is proposed as a working example that will be considered also in subsequent sections to illustrate the tasks performed by the optimizer in different passes. Before proceeding, let us discuss how we intend to identify implicit parallelism in source programs.

```
class Y {
S1:     private X myX;
S2:     public void p() {
S3:             int myValue;
                ...
S4:             myX.m();
S5:             myX.n(myValue); }
        }
class X {
S6:     private char c;
S7:     private int i;
S8:     public void m() {
S9:             System.out.println("X"+c); }
S10:    public void n(int e) {
S11:            this.i = e; }
S12:    public void n(long j) {
                ... }
        }
class XX extends X {
S13:    public void m() {
S14:            System.out.println("XX"+c); }
        }
```

Let us focus on the code of method p in class Y (using a notation borrowed from $C++$: $X :: p()$) and on its body. In order to characterize the potential dependencies between instructions $S4$ and $S5$ (true, output or anti dependency) it is necessary to describe the effect produced in the program execution environment as a consequence of invocation of methods m and n respectively. The member field myX of class Y is a nested object whose type is X, hence it comprises two primitive member fields: c and i. In $S4$ and $S5$, the actions performed by methods m and n on the receiver object myX can alter the state of its constituent fields. Therefore we need to analyze $X :: m()$ and $X :: n(int)$. Method $X :: m()$ performs an output operation that uses as input the value of member field c. Hence we can conclude that:

- No alteration is produced on the state of the receiver by $X :: m()$.
- The execution of $X :: m()$ requires the bit of information associated to member field c.

This situation can be conveniently modeled by associating two sets to method $X :: m()$, a set $IN(X :: m()) = \{this.c\}$ containing the member fields used as input to the task performed by the method (methods instructions are considered as a whole), and a set $OUT(X :: m()) = \{\}$ containing the member fields modified by the method invocation. Proceeding in the same way for the other method ($X :: n(int)$) invoked in $S5$, the following sets are produced: $IN(X :: n(int)) = \{e\}$, $OUT(X :: n(int)) = \{this.i\}$. After collecting this necessary information, let us return to our original matter: the dependency analysis of instructions $S4$ and $S5$. It is possible to produce more detailed considerations. The call site represented by instruction $S4$ operates on receiver myX, and we know that $myX.c$ is used by $m()$, because $IN(X :: m())$ contains "$this.c$". In $S5$ the receiver is still myX, and the method invocation produces an alteration of $myX.i$ because $OUT(X :: n(int))$ contains "$this.i$". As it is not possible for the two member fields of myX object to be alias of one another, these considerations lead us to conclude that $S4$ and $S5$ do not interfere with each other even if they are invoked on the same object. Therefore $S4$ and $S5$ can be issued simultaneously in a multi-threaded manner.

3 Compiler First Pass

In the previous paragraph we exposed ideas underlying the implementation of an analysis framework. In the following sections an overview is presented of current implementation status and of the design choices necessary to face the great deal of situations arising when optimizing a real program. While parsing the input Java code, the following attributes are evaluated:

1) *Construction of the Symbol Table (ST).* The ST stores objects declared in the program according to the language scope rules. Compared to C language, in Java the task is simplified by the absence of "*typedef*" declarations. On the other hand, visibility extends also to class member fields. Moreover in Java an object can be declared in any point where an instruction can appear, and not only at the beginning of a new scope marker.

2) *Construction of method abstract syntax trees.* Syntax trees provide a structured representation of source code, allowing successive analysis phases. The adopted syntax tree representation is designed to facilitate inter-procedural analysis. In this initial phase we are not directly addressing more standard transformations as, for example, expression optimizations, hence expressions are represented in a compact way in syntax trees.

3) *Identification of call sites.* Call sites represent discontinuity points in program control flow and the points of interest in task-level optimizations. Every method keeps a list whose elements point to tree nodes where a method is invoked. Every time a new call site is met, it is added to the call site list, and a call site descriptor is compiled with information that includes the receiver and actual parameters. If these are locally declared variables or method parameter, their descriptors are already allocated on the ST. However, not all information required to completely characterize a call site can be available in this phase. For example, call site descriptors need to contain also information about the invoked method, but it can belong to a different class that maybe textually follows the invocation point in the source file. Let us consider method $Y :: p()$. Its call site list contains the syntax tree nodes that represent instructions $S4$ and $S5$. The call site descriptor of $S5$ links the identifier $myValue$ to the corresponding variable descriptor stored in ST; on the other hand the link to the code of method n is still dangling and it will be resolved successively.

4 Compiler Second Pass

The second pass of the compiling process aims at gathering additional information that is available only after parsing the whole input program. In particular, the following activities are carried out:

1) *Resolution of name references.* All name references that were dangling from previous pass are fixed. This concerns, for example, member field accesses: the reference to a class member field is linked to the appropriate descriptor in this phase.

2) *Construction of the call-graph.* A method invocation in a call site descriptor is associated to the corresponding syntax tree. Inheritance and method overloading play a relevant role in this context. When identifying the appropriate method, at first the type of the receiver object provides a candidate class from which to start the matching procedure. It can be necessary to inspect one or more ancestor classes of the candidate class, moving upward in the program taxonomy. Inside a class is then required to resolve method overloading: the more specific method is identified that matches the descriptor of the call site under analysis. Also in this case it is required to consider the program taxonomy when matching actual invocation parameters against formal parameters in method signatures. Hence, considering our example, in the call site descriptor of S5 the method reference is set to $X :: n(int)$.

3) *Evaluation of Extent set for every method.* The $Extent$ of a method m is defined as the set of all other methods that, at run time, m might invoke directly or indirectly. Such information corresponds to the identification of the portion of call graph reachable from m. Current implementation does not support recursion yet. Given a call site descriptor D, $Extent$ evaluation requires to inspect also all descendant classes of the class containing the method linked to D. If, for example, after declaration S3, field myX is assigned an object of type XX and if this definition reaches $S4$, then the execution of $S4$ would activate method $m()$ in class XX ($XX :: m()$) because of dynamic binding mechanism. Therefore, $Extent(X :: p()) = \{X :: m(); XX :: m(); X :: n(int)\}$.

4) *Topological sorting of program methods.* Based on the $Extent$ set, all methods are topologically sorted in a list L. If $X_q :: m_h(...)$ is the k-th element in L, then every method in $Extent(X_q :: m_h(...))$ is included in the (k-1)-length prefix of L. In our example a possible method sorting is $L =< XX :: m(); X :: n(int); Y :: p(); X :: n(long) >$. Processing methods in this order guarantees that all information necessary in a method is available when it is processed.

5 Current Implementation Activity

Current activity is concluding the program static analysis in order to characterize method behavior. Following steps will include the identification of parallelizable code portions and multi-threaded code generation.

References

[1] C. Brownhill, A. Nicolau, S. Novack, C. Polychronopoulos. *Achieving Multi-level Parallelization.* Proc. Of International Symposium ISHPC'97, Springer, pp. 183-194

[2] M. Rinard, P. Diniz. *Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers.* ACM, Trans. on Programming Languages and Systems, vol. 19, No.6, Nov. 1997, pp. 942-991

[3] H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1992.