Inter-Procedural Analysis for Parallelization of Java Programs

Antonio Magnaghi, Shuichi Sakai, Hidehiko Tanaka

The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan

1 Introduction

Parallelization of Java programs is a complex task due to inheritance, dynamic method dispatching and aliases. Our research [2] aims to perform static analysis of Java programs in order to identify implicit parallelism. In this paper, we discuss first the inter-procedural analysis technique we are studying and implementing to characterize data-dependency. And then we enhance this framework with type-based alias analysis.

2 Inter-Procedural Analysis

Some of the fundamental characteristics of Object Oriented Programming (OOP), such as information hiding, lead us to stress the role of inter-procedural analysis for Java programs because access to objects relies heavily on method dispatching. Through the example below we demonstrate how implicit parallelism can be identified in source programs. Let us focus on instructions `S1` and `S2` of method `p` in class `Y` (using a notation borrowed from C++: `Y::p()`). The member field `myX` of class `Y` is a reference object comprising two primitive member fields: `c` and `i`. In `S1` and `S2`, the actions performed by methods `m` and `n` on the receiver `myX` can alter the state of its constituent fields. Method `X::m()` performs an assignment that uses as input the value of member field `c`. Hence we can conclude that: 1.) no alteration is produced on the state of the receiver by `X::m()`; 2.) the execution of `X::m()` requires member field `c` as input information. We model this situation by associating two sets to method `X::m()`, a set `IN(X::m())={this.c}` containing objects externally visible to the method and used as input to the task performed by the method.

```
class Y {
private X myX;
void p() {
 S1:myX.m();
 S2:myX.n(3);}}
class X {
private char c;
private int i;
void m() {
 S3:String s="X"+c;}
void n(int e) {
 S4:i=e; }}
```

And similarly, `OUT(X::m())`$=\emptyset$ is the set of objects which are modified. The following sets are produced for `X::n(int)`: `IN(X::n(int))={e}`, `OUT(X::n(int))={this.i}`. Because it is not possible for the two member fields of `myX` (`c` and `i`) to be alias of one another, we conclude that `S1` and `S2` in `Y::p` do not interfere with each other even if they are invoked on the same object. Therefore `S1` and `S2` can be issued simultaneously in a multi-threaded manner.

3 Type-based Pointer Analysis

In more general contexts than the example above, alias problems arise [1]. We can formalize our type-based approach for aliases as follows. Let $Class$ and $Object$ respectively be the sets containing all program classes, and all program objects. Let $Type$ be a function that returns the class type of an object. Let $Comp$ be a function from $Class$ to 2^{Class}. It maps every class to the set of reference types encountered by recursively traversing the data structure of the input class. If $cls \in Class$, and S is the set of nodes of the sub-tree rooted in cls in the program taxonomy, then we designate $Comp^*(cls) = \bigcup_{i \in S} Comp(i)$. If we consider two objects in $Object$, obj_1 and obj_2, we assume that an alias may take place iff the following condition holds: $Comp^*(Type(obj_1)) \cap Romp^*(Type(obj_2)) \neq \emptyset$

4 Analyzer Structure

Current implementation produces dependency information based on the algorithms expressed above, conveniently addressing inheritance and method (constructor) overloading. The structure of the analyzer can be outlined as follows: *1.) First Pass:* the input program is parsed and information about call sides is collected. *2.) Second Pass:* it gathers additional information that becomes available only after parsing the whole input program. The following activities are carried out: call graph construction; topological sorting of methods (constructors) based on call graph analysis (the concept of $Transfer$ set is introduced as a generalization to OOP of the $Extension$ in [3]); evaluation of IN and OUT sets for every call site and method (constructor) by type-based alias analysis.

5 Conclusions

Preliminary evaluation has been carried out. On simple benchmarks like Linpack and Primes, we identified parallelism associated to immutable objects or stateless methods. Current research activity is improving the alias analysis algorithms by refining the type information associated with every program object.

References

1. A. Diwan, K. McKinley, E. Moss. Type-based Alias Analysis. In *ACM SIGPLAN98 Conference on Programming Language Design and Implementation*, pp. 106-117, 1998.
2. A. Magnaghi, S. Sakai, H. Tanaka. An Inter-procedural Approach for Optimizations of Java Programs. In *Proceedings of the Information Processing Society of Japan*, vol. 57, pp. 299-300, 1998.
3. M. Rinard, P. Diniz. Commutativity Analysis: a New Analysis Technique for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 6, pp. 942-991, 1997.
4. H. Zima, B. Chapman. Supercompilers for Parallel and Vector Computers. *ACM Press*, 1992.