A Hierarchical Task Level Representation of Java Programs for Parallelization

Antonio Magnaghi and Hidehiko Tanaka
The University of Tokyo

1 Introduction

Extraction of parallelism from source programs is an important and active area of research in the field of compiler design. By actual trends, code parallelization is accredited as a key technology for an efficient utilization of computational resources. However, direct detection of parallelism is generally a difficult task, whose scope spans different conceptual levels [4]. In particular, there exists a strict mutual interdependency between the specific type of parallelism to consider in automatic reconstruction and the target architecture. Therefore status-of-art architecture technology impacts on compiler design. The scenery is basically dominated by two major complementary strategies. On one hand, dedicated ad-hoc hardware components are produced and assembled into complex systems. The remarkable development costs in such a case are justified by the intent to aggressively utilize a precise form of parallelism. Numerical application programs are one of the most relevant examples, also for historical reasons due to the popularity of Fortran for scientific computations. However for most commercial applications it is not admissible such an investment of capital on new components, yet advantages are relevant deriving from extraction of concurrency. In such a stream, the employment of off-the-shelf components has led to those kinds of architectures such as PC-networks, cluster of workstation or hybrid configurations. The new perspectives revealed by this approach constitute the framework to our research project about Java language.

2 Java and the Hierarchical Task Graph

Our research project aims at the development of proper techniques to optimize programs written in Java. The choice of Java can be motivated by the consideration that it includes many of the most meaningful and advanced features of programming languages such as C and C++. Java introduces new elements [3], in particular the thread paradigm for parallel program execution is part of the language specification. Nonetheless, also some important restrictions are imposed. The most relevant ones do not allow memory address arithmetic, the *goto* statement and multiple inheritance. Our approach consists in restructuring source programs, based on the introduction of the Hierarchical Task Graph (HTG). Previously the HTG was introduced only for C language [2]. As original contribution, we aim at extending it to Java and analyzing how inheritance mechanism can impact on such a program representation model. The hierarchy among classes makes additional information available about program structure that could be used in order to improve performance. This paper will not provide a formal specification of the concept of HTG for Java because of space limitations. Rather, we will proceed intuitively through an example that will further be developed in following sections. Let m be a method defined as follows:

```
void m()
    { 1: obj_3.m_1(13);
      2: obj_1.m_3();
      3: obj_1.m_2();
      4: for(...)
      5:      obj_1.m_1(17);
      6: obj_2=obj_1; }
```

The relationship between classes and methods is expressed by the taxonomy of figure(1). obj_1, obj_2, obj_3

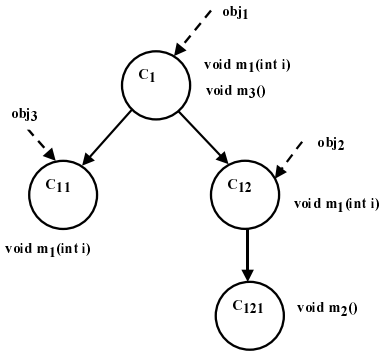

Figure 1: Class Taxonomy

are instances of classes C_1, C_{12}, C_{11} respectively. m_1, m_2, m_3 are methods. The HTG of input programs is built from control flow and data dependency. It provides an intermediate representation which decomposes the program in constituents tasks of different granularity. Let us focus on method m. We assume the existence of a class containing a static variable v: $v \in use(m_1 \ of \ C_{11}), v \in def(m_3 \ of \ C_1)$. Moreover methods m_1, m_2 do not alter the internal state of the object they use as input. The HTG of m is showed in figure (2). The instructions of m have been represented by single nodes because they contains control flow transfer due to method invocations, and each node should be linked to the HTG of the associated method. Solid arcs correspond to control flow and dotted ones represent data dependency. In particular, in node 2, m_3 assigns a new value to variable v; in node 5 that value could be used, hence the dependency edge from 2

to 5 is required. Figure (2) shows that the HTG is obtained by merging control flow and data dependency in a structured manner. Nodes 4 and 5 are grouped together to form a loop-component, labeled 4-5, whose granularity level is the same as nodes 1, 2, 3 and 6, if it is considered as an atomic unit. On the other hand, if we need to concentrate on finer grain parallelism, it is possible to refine the information associated to node 4-5, analyzing its inner structure. Hence we can conclude that the represented task graph contains two distinct levels, hierarchically related to each other. However in previous considerations the taxonomy among classes has not been properly utilized. Our investigation aims at optimizing the information encapsulated in the HTG of a Java program by detecting and consequently deleting potential dependency constraints that actually do not subsist because of program semantics and inheritance structure.

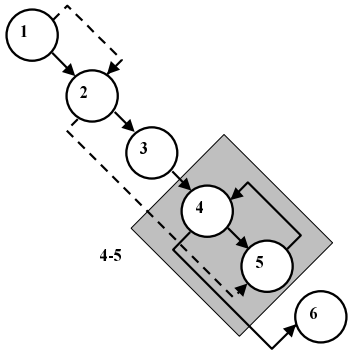

Figure 2: HTG of method m

3 Optimizing the HTG

Object-oriented technology relies on the concept that every data type is defined not only by its internal structure, but also by the operations we can perform on it. And inheritance allows to represent the relationships among instances of different classes. The taxonomy of Java code provides additional information whose utilization can reveal insights about the program behavior. Let us consider the HTG of figure (2), and the data dependency arc connecting nodes 2 and 5. It imposes to delay the execution of instruction 5 until instruction 2 has completed. The following analysis of inheritance structure and program semantics will show that such and arc can be safely suppressed improving performance. Let H=(C,E) be the tree of classes in the program, where C is the set of classes and E the set of connecting edges. Let $P(obj)$ denote the collection of all classes that are descendants to the class which object obj is declared of, including the class of obj. In our example, $P(obj_1) = \{C_1, C_{11}, C_{12}, C_{121}\}$; $P(obj_2) = \{C_{12}, C_{121}\}$; $P(obj_3) = \{C_{11}\}$. The mechanism of dynamic binding allows an object pointer to point also any other instances of inner classes in the inheritance taxonomy. Hence a generic instance obj can point to any element in $P(obj)$. However, being able to reduce the cardinality of the pointed-to instance set can make parallelism extraction more effective. Let us focus on object obj_1. In instruction 3 method m_2 is invoked, which is a member only of class C_{121}. obj_1 is never assigned a different object in m, therefore its actual object instantiation can not change and it is restricted to C_{121}. We can statically foresee that method activation in instruction 5 will dynamically be linked to method m_1 which class C_{121} inherits from its parent. As previously assumed, m_1 in C_{12} does not use variable v, hence the potential data dependency of node 5 from node 2 does not actually occur, and the corresponding arc can be safely deleted. This guarantees that node 5 (or loop-level node 4-5) can be executed before node 2. In particular, the restriction of possible pointed-to objects in the case of obj_1 propagates to obj_2 also: instruction 6 assures that obj_2 can identify only instances of class C_{121}.

The discussed example underlines that in Java polymorphism and dynamic binding produce a situation where inter-procedural optimizations are complex. An object can potentially point to several class types, therefore a conservative approach causes the loss of important information. An analogy can be drawn with pointer handling in C language: the difficulty of tracing pointed-to data by a pointer limits the extraction of parallelism. However Java couples each program to its taxonomy of classes, where data types are merged with their associated methods. Hence new information is made available to the compiler, that, as shown above, can refine the HTG topology.

4 Conclusions

Our research aims at introducing the concept of HTG to Java programs, in order to identify different task levels for concurrency extraction. The interest in Java springs from its state-of-art language design features, among which portability is a key factor for software development. The interest in the concept of HTG is motivated by its high flexibility in dynamically tuning the granularity of program concurrency: Java applications can be executed on a wide category of architectures and the HTG representation can properly address optimizing reconstructions for such systems.

References

[1] C. Brownhill, A. Nicolau, S. Novack, C. Polychronopoulos. *Achieving Multi-level Parallelization.* Proc. of International Symposium ISHPC'97 Japan. Springer, pp. 183-194, 1997.

[2] C. Polychronopoulos. *The Hierarchical Task Graph and its Use in Auto-Scheduling.* Proc. of the 1991 International Conference on Supercomputing. ACM, pp. 252-263, 1991

[3] P. van der Linden. *Just Java and Beyond, third edition.* The Sunsoft Press, 1997

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