A new Dependence Test based on Shape Analysis for Pointer-based Codes

Sandeep Dasgupta Barnali Basak Amey Karkare

Indian Institute of Technology,
Kanpur, India
{dsand,barnali,karkare}@iitk.ac.in

1 Introduction and Motivation

In the recent arena of multi-core architecture, software side lags behind that of hardware in terms of parallelism. Parallelization of code, without violating correctness, is a key step in increasing the code performance and efficiency. The former, whether manual or compiler-assisted, is not a trivial task as aliased objects make it even more difficult. In the context of data structures, arrays and pointer directed expressions incorporate aliases in programming codes. Over the past years, lot of work had been done on parallelizing codes but they mainly focused on static data structures and stack directed pointers. Program codes for real time applications, scientific researches, etc. are dynamic data structure intensive. However, optimizing and parallelizing such irregular dynamic data structure based codes for multi-core architecture is very complex and requires a great deal of analysis.

In this paper, we focus on automatically parallelizing heap-intensive dynamic and recursive data structures. A data structure is dynamic if it is allocated at execution time and accessed through heap directed pointers. More precisely, these dynamic structures are recursive in the sense that each heap object can access other heap objects via pointer fields, thus resulting in structures like trees, Directed Acyclic Graphs (DAG), linked lists, graphs, etc. Such data structures can be accessed by either loop based iterative methods or recursive function calls. Hence in this work, we have developed a novel store-less data flow analysis technique to detect parallelism hidden in loops and function calls accessing data structures pointed by heap directed pointers.

The particular goal of our approach is to detect loop level and function call level parallelisms, i,e. coarse-grain parallelism in the context of dynamic recursive data structures. This kind of data flow analysis requires information at compile time regarding the shape of the data structures to be allocated at run time. To meet such requirements, we have also devised a technique for doing field-sensitive shape analysis that gives more precise information about the shape of the data structure pointed to by heap directed pointers. Our technique for finding loop carried dependencies (henceforth, referred as LCD) extends the work of [2], which focuses on detecting LCD (i.e., flow, anti or output dependencies) for static data structures mainly arrays, and thus helps in disambiguating heap accesses originating from heap directed pointers in different iterations of a loop.

2 Summary Of Work

The rest of the paper is organized as follows: Section 2.1 “Shape Analysis framework” briefly describes the key ideas under our shape analysis framework. With this background, in Section 2.2 “Loop Carried Dependence Detection” we present our technique to detect LCD in codes based on dynamic data structures.

2.1 Shape Analysis Framework

The goal of our shape analyzer is to detect the shape of the data structure pointed to by the heap directed pointers at each program point. Our approach is an extension of Ghiya [1] with the additional information of abstract path between two heap nodes. This helps in evaluating precise shape of the data structures pointed to by the heap directed pointers in cases when Ghiya [1]'s approach gives conservative shape information.
Our shape analysis takes into account the sequence of field pointers (hence the analysis is called field-sensitive) between two heap nodes, and hence able to provide precise information about the following:

1. Detecting actual shape of the data structure even when some links between nodes are broken (due to statements like p –> f = NULL or p –> f = q).

2. Our analysis can detect the shape transition when a cyclic node become acyclic at some program point.

3. Cyclic links of a node n, where the shape attribute of n is cycle. We define cyclic links as the set of abstract paths such that when starting at node n and following the path, the node n is reached again.

4. Whether or not two paths p.S and q.S' (where p, q are heap directed pointers and S, S' are sequences of field pointers) reach the same heap node.

The first two points are an improvement on Ghiya [1]'s approach as in there approach once cycle attribute is associated with a data structure, it persists for the rest of the analysis even when the structure becomes acyclic in the rest of the code. Again the last two informations will later be used in detecting LCD.

2.2 Loop Carried Dependence Detection

We have devised a novel LCD detection approach that first symbolically executes at compile time the statements of the loops iteratively until a fixed-point is reached. During symbolic execution, we keep the abstract path traversed by the pointers within the loop, at each program point. Depending on the nature of access (Read or Write) of the value field of the pointers, the abstract paths are marked with Read or Write tags. These abstract paths can be normalized to simpler ones by using the cyclic links information from the shape analyzer. The fixed-point is reached by merging certain abstract paths depending upon the tagging and the normalization technique. Once the fixed-point is reached pairs of conflicting tuples (i.e. either of them must be tagged with "Write") are identified and fed to the shape analyzer which concludes if the two tuples (or paths) are accessing the same heap node. Depending on the order of tags of the tuples in a conflicting pair and the conclusion of the shape analyzer the type of dependence is declared. Two key points of our approach are:

1. Less Space Overhead: The model used for shape analyses is store less(does not deal with real memory locations)

2. Fast Analysis: As the proposed technique does not deal with manipulating any shape graph while traversing the loop to reach a fixed-point(as done by [3]), so it is faster in reaching fixed-point than any such store based approach.

References

[1] Rakesh Ghiya and Laurie J. Hendren. "Is it a Tree, Dag, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C". In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pages 1-15, St. Petersburg, Florida, January 1996.

[2] Allen, R. and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Transactions on Programing Languages and Systems* **9**:4 (1987), pp. 491-542.

[3] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata and E.L. Zapata, "A New Dependence Test Based on Shape Analysis for Pointer-Based Codes," *LANGUAGES AND COMPILERS FOR HIGH PERFORMANCE COMPUTING,* LNCS, 2005, Volume 3602/2005, 394-408.