TBDA: Trace Based Dependence Analyzer

http://www4.ncsu.edu/~rramase/CSC548/TBDA.html

Ravi Ramaseshan rramase@ncsu.edu

Abstract

Dependence graphs can be used as a vehicle for formulating and implementing compiler optimizations. Static compiler analysis of a program may lead to conservative dependence graphs, some of whose edges might be exercised with extremely low probability during most execution runs. Also, imprecise pointer analysis may add may-dependence edges that might prevent aggressive compiler optimizations from being applied on the program. TBDA proposes building a dynamic dependence graph (DDG) based on memory reference traces obtained during a typical run of the program. TBDA also proposes annotating the edges in the dependence graph with distance vectors, direction vectors and probability of the dependence edge being exercised, which can be used by a compiler to base heuristics for speculative or user-directed parallelization.

Introduction

The advent of multi-cores, the academic and industry must solve the parallel programming problem for a broad range of applications with your typical good programmers, not just rocket scientists. There exists a lot of code that has not not been parallelized and which has been written in languages like C/C++ on which pointer analysis is difficult. Standard compilers like gcc implements very limited and conservative pointer analysis. More powerful research compilers capable of performing more aggressive analysis and optimizations are not easily available or are broken. In that light, a tool that makes dependence information available to the user and compiler can guide the compiler to performing optimizations which it couldn't using it's own analysis. A user may also use the information from the dependence analyzer to quickly identify potentially parallel regions in the program.

Recent research in thread level speculation (TLS) has proposed methods for parallelizing serial code. TLS systems can tolerate code generation which ignores dependences, by having a run time system which detects and recovers from dependence violations. While TLS provides a method to aggressively or optimistically parallelize a program, careful selection of threads is required to see a performance improvement. Manually inspecting programs in order to identify speculative threads is not feasible for large and complex programs. In such a situation, a tool which inspects the memory reference trace of a program to automatically guide speculative thread selection, would be very useful to study the limits of thread level speculation.

Framework

Figure 1 shows the overall framework of the trace based analyzer with the interaction of the various systems and the flow of information through the framework.

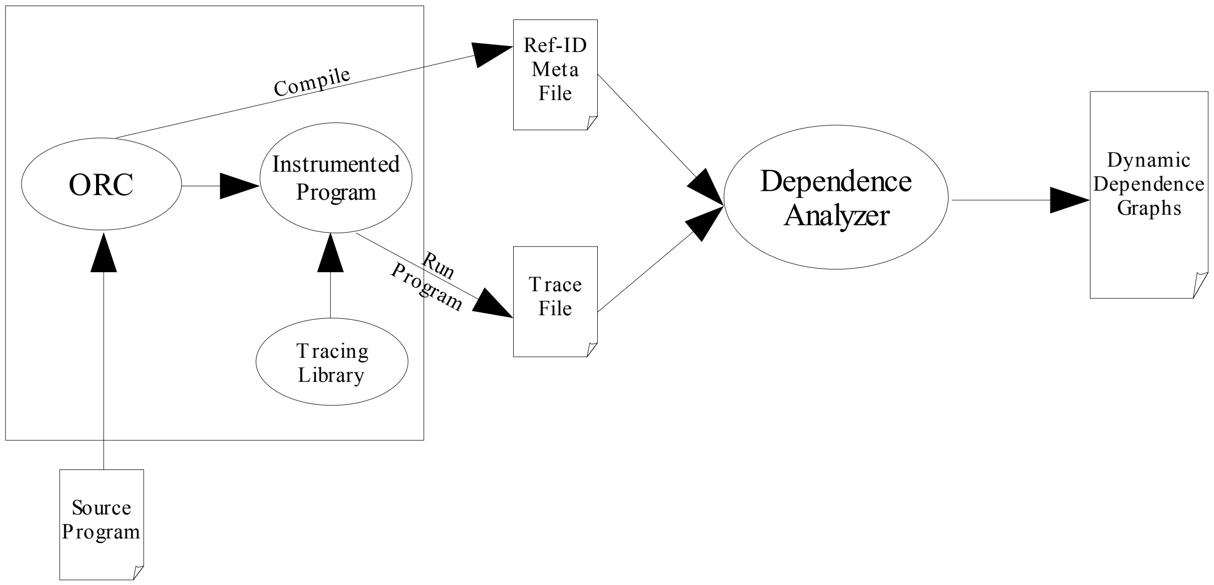

Figure 1: Trace Based Dependence Analyzer Framework

The trace based dependence analyzer operates in two distinct phases. The first phase is the tracing phase in which the target program is instrumented to generate a memory trace, loop iteration and function call markers. The instrumented program is then compiled with the tracing library and run with training input to generate a trace file of the memory references. In the second phase, the trace file is given as input to the dependence analyzer which builds the dynamic dependence graph of the program. In the following sections we describe the instrumentation and dependence analysis frameworks in detail.

Instrumentation Framework

The instrumentation framework was implemented in Intel's Open Research Compiler (ORC) for the Itanium architecture. Since one of the most promising sites for parallelism in a program is loops, the compilation phase chosen for inserting instrumentation into the program was the Loop Nest Optimizer (LNO). The internal representation of the compilation units in ORC is called WHIRL [6]. The WHIRL nodes in the IR that are of interest to be instrumented to generate the memory reference trace of the target program are scalar loads (LDID), scalar stores (STID), indirect loads (ILOAD) and indirect stores (ISTORE). In order to capture loop carried dependences, loop nodes (DO_LOOP) were inserted too. To capture dependences across different functions each call site is instrumented.

We walk the WHIRL tree of each function in pre-order and insert instrumentation code for each of the above nodes. Each instrumentation point is given a reference ID and information about the instrumentation point is stored in a file which is also fed to the dependence analyzer. The reference ID uniquely identifies a static instrumentation point and also contains information like the type of the instrumentation, the size of the memory reference and debugging information like line number of the program being instrumented. This strategy of instrumentation was chosen in order to reduce the size of each trace record to a reference ID, memory address tuple.

The exact instrumentation strategy for each type of node is explained below.

Memory References

For each memory reference (LDID, STID, ILOAD, ISTORE) we copy the subtree computing the address of memory being referenced and pass it to the tracing function. Consider the following source code line, it's simplified WHIRL statement and simplified generated assembly code.

```
X = X + 1;      (STID                     , X)          mov r1 = &X

                     (ADD          , 1)                 ld r2 = [r1]

                          (LDID X)                      add r3 = r2,1

                                                        st [r1] = r3
```

Source code WHIRL tree Assembly instructions

As can be seen, the order in which instructions encountered in the tree-representation is the reverse of the lowered order. So when inserting instrumentation for a statement, the calls to the tracing library are inserted in reverse order so that the order of the memory reference trace is correct.

Loops

In order to be able to generate distance vectors it is necessary to distinguish between accesses made in different iterations of the same loop. We thus need a mechanism to mark beginning and ends of each loop iteration. Instead of generating *loop_begin* and *loop_end* markers at the beginning and end of the loop body, instrumentation is just added for *loop_begin* at the beginning of the loop body. Thus all the accesses in each particular loop iteration, apart from the last iteration, is bounded by two *loop_begin* markers. For the last iteration and to denote the end of a set of loop iterations, we insert a *loop_end* marker just after the loop. This has the advantage that the instrumented code for *loop_end* is executed only once for a loop. This also means that fewer tracing records are written out. The instrumentation is kept minimal and induction variable values are not traced. As will be seen later in the dependence analyzer, induction variable values are not required to compute distance vectors.

Calls

Calls cause a unique problem in that they aggregate a number of loads and stores which occur in the function body and any calls subsequently made by the called function. The dependence analyzer needs to be able to aggregate any such loads and stores into a single call node when detecting a dependence edge. For that reason each call site is instrumented with a *function_begin* and *function_end* markers. Similar to the scheme described in the previous section about loops, it might be possible to insert the *function_end* instrumentation just before the return of the callee and reap the benefit that there would be at most one *function_end* per function and is independent of the number of call sites.

Dependence Analysis Framework

The instrumented binary is then run to produce an execution trace. The dependence analyzer uses the reference ID meta-file, generated by ORC while instrumenting the program, and the trace file generated by running the instrumented program and computes the dynamic dependence graph of the program. It annotates dependence edges with calling context information and distance vectors using the *loop_begin, loop_end, function_begin* and *function_end* markers. In the following sections we describe how the framework calculates dependences and their corresponding distance and direction vectors. Figure 2 shows a high level view of the dependence analyzer.

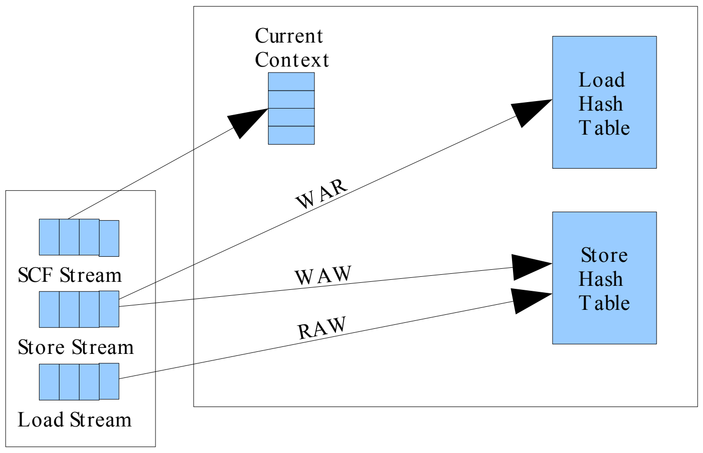

Figure 2: Dependence Analyzer

Detecting Dependences

The dependence analyzer maintains two hash tables for detecting dependences. One is for store references and the other is for load references. The memory reference stream is read in by the dependence analyzer and each load and store instruction is hashed into its corresponding hash table using the address of the reference as the key. All the references to the same address are stored in the hash bucket as a linked list which is traversed when detecting dependences.

For flow dependences, whenever a load is read from the trace, the store hash table entry for the same address is checked. If the bucket is not empty, then the list of stores are traversed and flow dependence edges are added for each store. In a similar way, for anti and output dependences, whenever a store is read from the trace, the load and store hash tables are checked respectively, and dependence edges are added to the dynamic dependence graph.

Extended Iteration Vectors

In order to detect loop carried dependences and dependences across functions, calling context information and loop nesting information. Loop nesting information can be derived using iteration vectors. We extend the concept of an iteration vector to also contain calling context information due to which cross iteration as well as cross-function dependences can be detected transparently. This section describes the generation of these extended iteration vectors.

The dependence analyzer maintains current context information by maintaining a stack of descriptors for each loop and function call. When a *loop_begin* is encountered and the reference ID is different from the current context, a new loop descriptor is pushed onto the stack with zero as the current iteration number. If the reference ID of the *loop_begin* marker is the same as that of the current context, the the current context's descriptor's current iteration is incremented by 1. Thus without using the value of induction variable, we maintain loops in canonical form. Whenever we encounter a *loop_end* we pop the descriptor off the stack of the current execution context. Similar action without the complexities of iteration numbers is performed for the *function_begin* and *function_end* markers. This stack of descriptors forms an extended iteration vector.

Distance and Direction Vectors

Distance vectors characterize dependences by the distance between he source and sink of a dependence in the iteration space of the loop nest containing the statements involved in the dependence. In some situations it is useful to work with distance vector that is expressed in terms of umber of loop iterations that dependence causes.[5]. Extended iteration vectors are stored along with each load/store access. When a dependence is detected. The extended iteration vectors are aligned for a common prefix and the difference of their iteration numbers of loop descriptors is computed and the distance vector is stored in a list along with the dependence edge. For each distance vector computed, the direction vector of the edge is updated.

Solved Issues

1. Generation of dynamic dependence graph of functions with and without loops
   - ● The Pre-Optimizer did not build DU-information for the function because of which it was not possible to instrument the IR at LNO.
   - ● Tweaked the Pre-Optimizer to build DU-information for all functions.
2. Switched to a reference ID instrumentation strategy
   - o The trace files being generated were too large. In an attempt to reduce the size of the trace file, this strategy was employed.
   - o Along with reducing each trace record to two integers, packing structures together to save on padding space was also employed to reduce the size of the trace file.

Testing

The trace based dependence analyzer has been tested with a few micro benchmarks. The test cases were designed to handle the following cases:

1. Dependences with both memory references inside a single loop nest.
   a) Unique distance vectors per edge.
   b) Multiple distance vectors per edge.
2. Dependence with one reference in a loop nest and the other in a function call.
3. Dependence with both references in disjoint function calls.

The `equake` benchmark has also been instrumented with ORC and been traced.

Open Issues

1. Getting `equake` to run through the dependence analyzer.
2. Generating dependence probabilities similar to the scheme mentioned in [7].
3. Correlating the reference IDs with source code line number information.
4. Running the dependence analyzer over the NAS benchmark set.
5. Using the profile information inside ORC.
   a) Correlating the dependence graph generated by the dependence analyzer with the dependence graphs inside ORC.
   b) Using the dependence probabilities to speculatively remove dependence edges.
   c) Perform a limit study of speculative parallelization using the trace based analyzer.

References

1. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages
2. Lin, J., Chen, T., Hsu, W., Yew, P., Ju, R. D., Ngai, T., and Chan, S. 2003. A compiler framework for speculative analysis and optimizations. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation
3. Austin, T. M. and Sohi, G. S. 1992. Dynamic dependency analysis of ordinary programs. In Proceedings of the 19th Annual international Symposium on Computer Architecture
4. Prabhu, M. K. and Olukotun, K. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*
5. Kennedy, K. and Allen, J. R. 2002 Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc.
6. WHIRL Intermediate Language Specification
7. Tong Chen, Jin Lin, Xiaoru Dai,Wei-Chung Hsu, and Pen-Chung Yew, Data Tong Chen, Jin Lin, Xiaoru Dai,Wei-Chung Hsu, and Pen-Chung Yew, Data Dependence Profiling for Speculative Optimizations
8. Rauchwerger, L. and Padua, D. A. 1999. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. Parallel Distrib. Syst.* 10