```
==================================
Trace Based Dependence Analyzer

Ravi Ramaseshan (rramase@ncsu.edu)

Progress Report 1 - 7th Nov 2006
==================================

Timeline
--------

26 Oct 2006     Project Statement:
                Write problem description and set up a webpage.

07 Nov 2006     Project Report I:

                Generate trace based dependence graph with ILOADs,
                ISTOREs, LDIDs, STIDs and CALLs for micro-benchmarks.

13 Nov 2006:
                Run the tool over standard benchmarks, test code and fix bugs.

20 Nov 2006:
                Map output of tool to source code lines. Identify TLS
                regions in the benchmarks.

27 Nov 2006:
                Plug dependence graph into ORC, run optimization passes and
                generate speedup results.

04 Dec 2006     End of Project:
                Class presentation.

Project Tasks
-------------

1. Generating a trace-based dependence graph.
   * Handle the following references:
     - Array references: ILOADs and ISTOREs.
     - Scalar references: LDIDs and STIDs.
     - Call references: CALL.
   * Generating dependence and direction vectors.

2. Running the tool over benchmark suites.
   * Test the tool and fix bugs.
   * Analyze the benchmarks for speculative optimizations.
     - TLS regions.

3. Speculative optimizations.
   * Plug trace-based dependence graph into ORC.
   * Run loop optimization passes.
   * Generate speedup results.

Progress
--------

The tool now generates a trace based dependence graph for all the
instruction types.

Instrumented call sites and inserted tracing calls into the IR to dump
call trace information. The dependence analyzer uses the call trace in
the following way:
```

The iteration vectors generated for loads and stores, besides having the normalized iteration counts of the loops, now also have call information.

Consider the following code:

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        A[i] = 5;               // A
        A[i+1] = 6;             // B
        foo ();                 // C

foo () {
    for (k = 0; k < N; k++)
        A[0] = A[i] + A[i+1];   // D
}
```

Consider the end of the first iterations of the ith and jth loops. The iteration vectors for A and B are (Loop-i 0, Loop-j 0) and the iteration vectors for the two loads and the store at D are (Loop-i 0, Loop-j 0, Call-foo, Loop-k 0).

There might be dependences between the loads and stores in the for-i,j loop nest and foo. This iteration vector is used to aggregate all the load stores in foo to the call node at C and show such dependences between the call node and load-stores in the for-i,j loop nest.

Open Issues

The instrumentation point in ORC that I've used currently is invoked only for loop nests. So straight line code doesn't get instrumented. This poses a problem when dealing with calls. Unless the procedure being called has a loop nest, it isn't instrumented.

Next Steps

1. Workaround the no-loop function tracing problem.
2. Run the tool for the NAS benchmark.