# CS243 Winter 2004 Final solutions

April 18, 2004

1. Loop level parallelism

   (a) Is the following a DoAll loop?

   ```
   for (i = 0; i < n ; ++i) {
   a [i-2] = a [i-1] + 1;
   }
   ```

   *Answer:* No, this is not a DoAll loop, since for `n` $> 1$, the is a dependency across loop iteration: an iteration has a WAR dependency with the previous iteration.

   (b) Is the following a DoAll loop?

   ```
   for (i = 0; i < n ; ++i) {
   a [2*i-1] = a [2*i] + 1;
   }
   ```

   *Answer:* Yes, this is a DoAll loop, all iterations are independent: if $i \neq j$, then $2i \neq 2j$, $2i \neq 2j - 1$, and conversely if we switch $i$ and $j$.

   (c) What is the complexity of data-dependence analysis?

   *Answer:* Data-dependence analysis, in all generality, is undecidable. The limited cases we are interested in (array indices) are solvable, but NP-complete. They can be solved using integer linear programming, which usually takes time $n^n$, where $n$ is the number of variables.

2. Is a preorder of a reverse graph always a post-order of the graph?

   *Answer:* No. In figure 1, the only post-order for the graph is 3,2,1. However 3,1,2 is a preorder for the reverse graph.

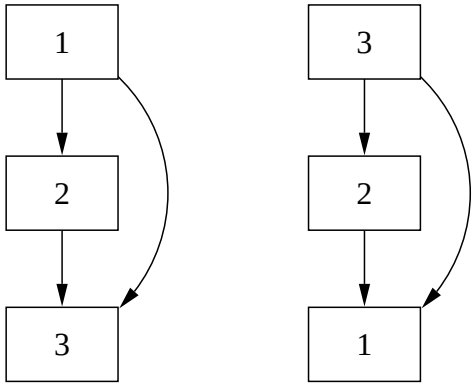


Figure 1: A control flow graph and its reverse

3. Is there a scenario where a mark-and-compact garbage collector outperforms a generational garbage collector?

   *Answer:* Yes. If objects don't die young, a generational garbage collector will do a lot of useless collections, since when the nursery becomes full, it is collected, and if the objects aren't dead, the will all be copied to an older generation. It also suffers a write barrier overhead. The mark-and-compact GC, however, will wait until the heap is full to collect (which might not even ever happen).

4. Points-to analysis

   One thing to remember is that points-to analysis has to respect typing (if the types are totally incompatible). Since there is only one location where a `Node` object is created (`a6`), `Node` objects can only point to this (or not point to anything).

   Results:

   (a) `a1.head`: `a6`

   (b) `a2.head`: `a6`

   (c) `a3.head`: `a6`

   (d) `a6.next`: `a6`

(e) `a6.elem`: `a3,a4,a5`

5. Graph optimization

   Several conflicting assumptions have been made about this graph. All were accepted. The "expected" assumptions were that:

   - all variables are defined at the entry (e.g. parameters);
   - we don't optimize tests out (as a matter of fact the `t==0` was supposed to be `t!=0`)

   The result of the optimization is on figure 2.

6. Loop parallelization

   (a) Dependence graph
   See figure 3.

   (b) The minimum initiation interval due to resource constraints is 2 (6 memory operations, 3 slots)

   (c) The minimum initiation interval due to precedence constraints is 3 (only cyclic path has length 9, and iteration delay 3)

   (d) We were able to reach our lower bound of 3.

   (e) Modulo resource table (A = ALU, M = Memory, L = loopback)

| A | A | A | M | M | M | L |
|---|---|---|---|---|---|---|
| | | | (1) | (4) | (7) | |
| (2) | (5) | (8) | | | | |
| | | | (3) | (6) | (9) | (10) |

   (f) Loop schedule

   The pipelined loop is very straightforward:

| Clock | Instructions |
|---|---|
| 0 | (1) (4) (7) |
| 1 | (2) (5) (8) |
| 2 | (3) (6) (9) (10) |

7. LOCK / UNLOCK dataflow analysis.

   There are a lot of ways to do this analysis. The most simple is probably to use several analyses.

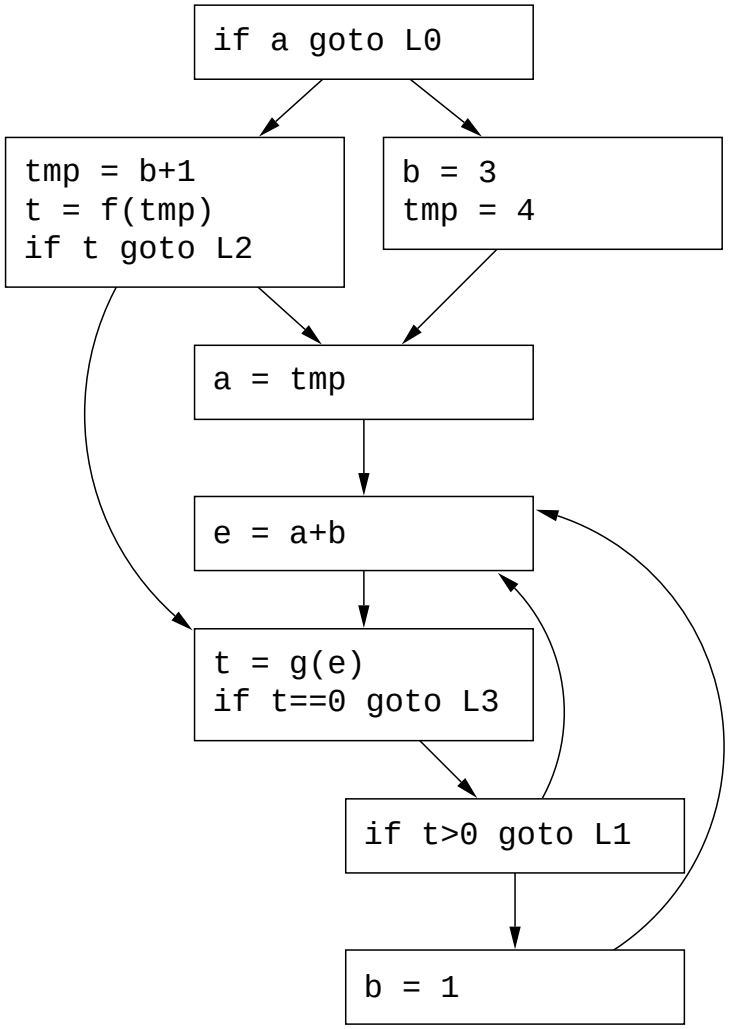


Figure 2: Optimized graph

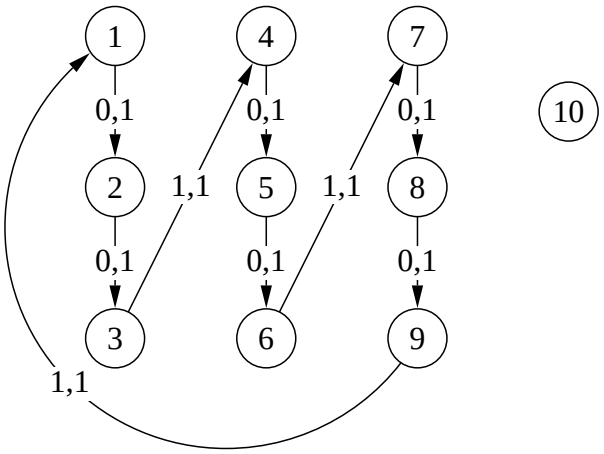


Figure 3: Dependence graph

The first one will be a constant-propagation style analysis, to issue warnings I and II.

(a) The analysis is forward.

(b) The values are the subsets of $\{U, L\}$.

(c) You should know what a set lattice looks like by now; the meet is union.

(d) As usual, interior points are set to $\top$ (empty)

(e) Transfer function: If B = LOCK, out(B) = {L}; if B = UNLOCK, out(B) = {U}; otherwise out(B)=in(B)

(f) out(Entry) = $\top$

(g) The framework is monotone (transfer function = constant or identity)

(h) The framework is distributive

(i) The algorithm will converge (monotone + finite lattice)

After running the dataflow, for each UNLOCK block if its IN contains U issue warning II (UNLOCK→UNLOCK); for each LOCK block if its IN contains L issue warning I (LOCK→LOCK).

Second analysis: warning III (Entry→UNLOCK)

(a) The analysis is forward.

(b) The values are $\top$ and $\bot$.

(c) Given the values it's trivial.

(d) As usual, interior points are set to $\top$

(e) Transfer function: If B = LOCK or UNLOCK, out(B) = $\top$; otherwise out(B)=in(B)

(f) out(Entry) = $\bot$

(g) The framework is monotone (transfer function = constant or identity)

(h) The framework is distributive

(i) The algorithm will converge (monotone + finite lattice)

After running the dataflow, for each UNLOCK block if its IN is $\bot$ issue warning III.

Third analysis: warning IV (LOCK→Exit)

(a) The analysis is backward.

(b) The values are $\top$ and $\bot$.

(c) Given the values it's trivial.

(d) As usual, interior points are set to $\top$

(e) Transfer function: If B = LOCK or UNLOCK, in(B) = $\top$; otherwise in(B)=out(B)

(f) in(Exit) = $\bot$

(g) The framework is monotone (transfer function = constant or identity)

(h) The framework is distributive

(i) The algorithm will converge (monotone + finite lattice)

After running the dataflow, for each LOCK block if its OUT is $\bot$ issue warning IV.

h. Reference counting

There is a way to extend reference counting to deal with circular references.

When an object is to be deleted (reference count down to 0), compute its reachable variable graph (following all the pointers). In this graph, find the minimal graph containing the original object, such that no edge points to it (at worst, it's the whole graph). Inside this graph, count all the pointers to the objects of this graph, if this count corresponds to each object's reference count, you can delete all the elements of the graph.