Exam in Optimising Compilers (DAT230/EDA230)

October 22, 2009, 8.00 — 13.00

Examiner: Jonas Skeppstedt

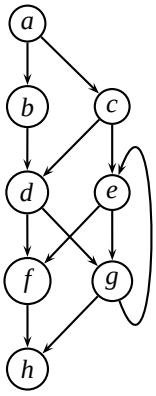


Figure 1: Control flow graph.

1. (10p) Explain how the Lengauer-Tarjan algorithm (the $O(N^2)$-version) finds the dominator tree in the control flow graph in Figure 1. For each vertex, your solution should explain:
   - when is the vertex put in a bucket?
   - in which bucket?
   - when is it deleted from the bucket?
   - when does the algorithm find the immediate dominator for the vertex?
2. (5p) What is the dominance frontier of a vertex?

   Answer: see the book.
3. (5p) What is control dependence and how is it computed?

   Answer: see the book.
4. (10p) Consider again the control flow graph in Figure 1. Suppose there is a use of variable x in each vertex and an assignment to x in vertices a, c and e. **In vertices a and c the definition is before the use and in vertex e the definition is after the use.**

Translate the program to SSA form. Show the contents of the rename stack and when the stack is pushed and popped. *You do not have to show how you compute the dominance frontiers.*

Answer: see the book.

5. (10p) List scheduling is often inferior to software pipelining. Explain why. To get full points you must show an example of when software pipelining produces better code than list scheduling.

   Answer: because list scheduling only can hide the latency of instructions in one loop iteration and there may not be any independent instructions to execute between a particular producer and consumer if only instructions from the same loop iteration are considered. With software pipelining, instructions from other loop iterations can be used to hide the latency. Consider:

```
float   a[100];
float   b[100];
float   c[100];
int     i;

/* ... */

for (i = 0; i < 100; ++i)
        a[i] = b[i] + c[i];
```

   List scheduling will not be able to hide more than perhaps one or two clock cycles while software pipelining fully can hide the latency of the floating point add and — assuming L1 cache hits — the array accesses.

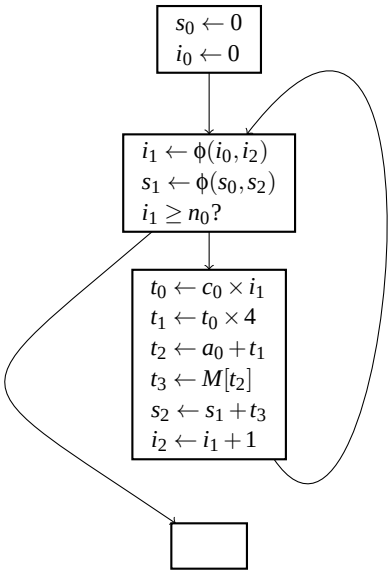
```
int f(int* a, int n, int c)
{
        int     i, s;

        s = 0;
        for (i = 0; i < n; i++)
                s += a[c * i];
        return s;
}
```

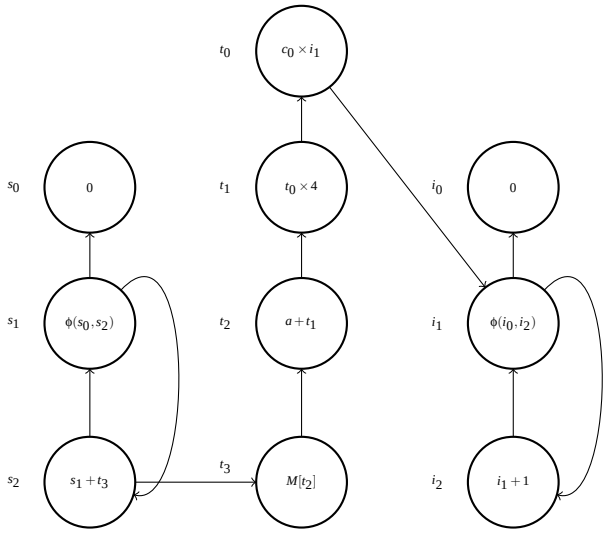
Figure 2: C function for question on operator strength reduction.

6. (10p) Explain in principle how operator strength reduction on SSA form optimises the loop in Figure 2. Your description should be based on the SSA-graph of the code, but you don't have to explain every detail of the algorithm.

Answer:

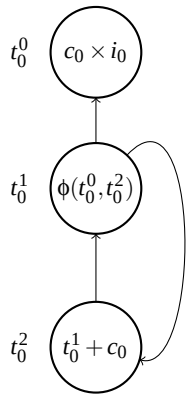


The SSA-graph becomes:

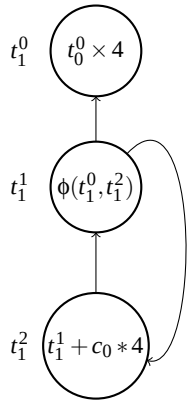


During the execution of Tarjan's algorithm, i is classified as an induction vari-

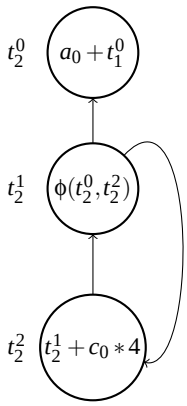
able, which leads to its strongly connected component is copied and modified for t_0 *as follows:*



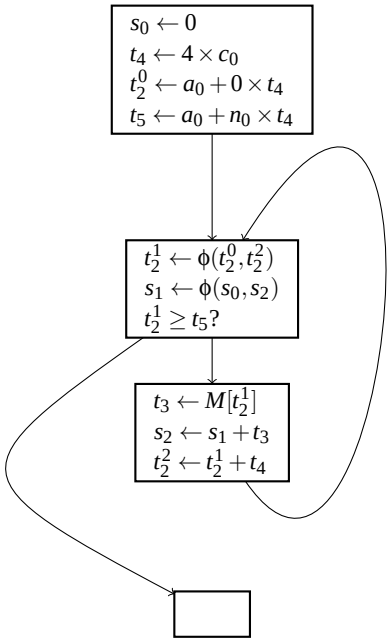
The use of t_0 *is changed to instead use* t_0^1*. The computation of* t_1 *now also is a multiplication of an induction variable and a region constant and the SCC of* t_0 *is copied and modifed for* t_1*:*



Then the use of t_1 *is changed to instead use* t_1^1*. The computation of* t_2 *now is the sum of an induction variable and a region constant and the SCC of* t_1 *is copied and modifed for* t_2*:*



The multiplication $c_0 \times 4$ is performed before the loop and saved in a new temporary variable. The resulting program — after DCE — will look as follows:



7. (10p) Why should a loop transformation matrix be invertible?

   Answer: it needs to be invertible when the new loop bounds are computed.