1 Loops, dominators, natural loops, control trees, dominator analysis

Loop optimizations are a fruitful class of optimizations because most execution time in programs is spent in loops: a 90/10 split between loops and non-loops is typical. Consequently, many loop optimizations have been developed: for example, loop-invariant code motion, loop unrolling, loop peeling, strength reduction using induction variables, removing bounds checks, and loop tiling.

When should we do loop optimizations? In source or high-level IR form, loops are easy to recognize, but there may be many kinds of loops, all of which can benefit from the same optimizations. Furthermore, loop optimizations often benefit from other optimizations that we want to do on a lower-level representation. We want to be able to interleave loop optimizations with these other optimizations.

In order to do loop optimizations, the first problem we must tackle is to define what we mean by "a loop" at the IR level, and to efficiently find these loops.

2 Definition of a loop

At the level of a control-flow graph, a *loop* is a set of nodes that form a strongly connected subgraph: every loop node is reachable from every other node following edges within the subgraph. In addition, there is one distinguished node called the *loop header*. There are no entering edges to the loop from the rest of the CFG except to the loop header. There may be any number (including 0) nodes with outgoing edges, however; these are *loop exit nodes*.

For example, the CFG in Figure 1 contains three loops as indicated, with header nodes are marked in the corresponding color.

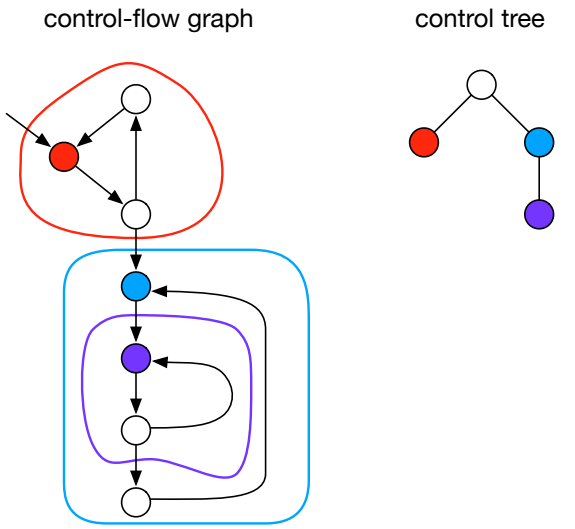

Figure 1: Loops in a CFG

A given CFG may contain multiple loops, and loops, as sets of nodes, may contain each other. If the nodes in loop 1 are a strict superset of the nodes in loop 2, we say that these are nested loops, with loop 2 nested inside loop 1.

Assuming that any two loops only intersect when one is nested inside the other, the loops in the CFG form a *control tree* in which the nodes are loops and the edges define the nesting relationship.

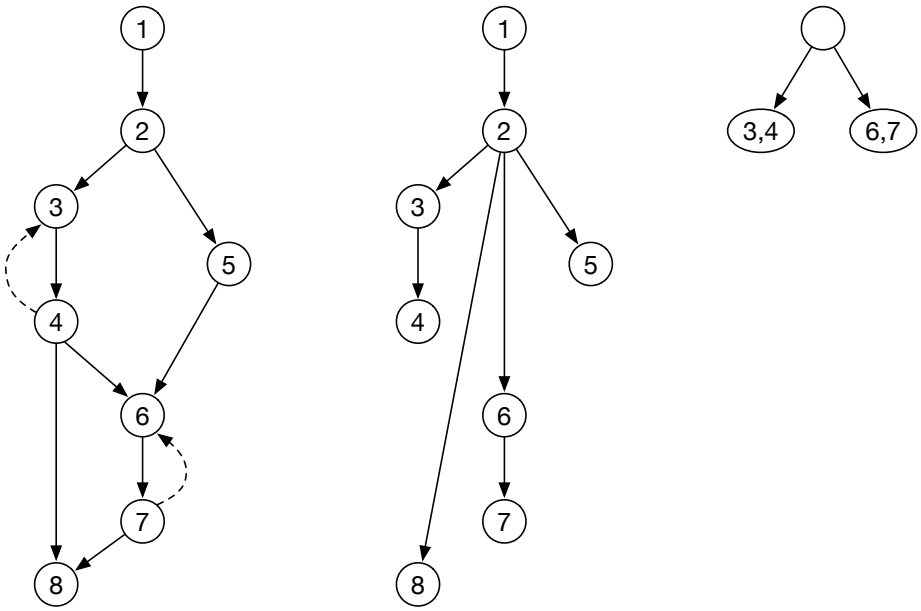

Figure 2: A control-flow graph (left), its dominator tree (middle), and its control tree (right). Back edges in the control-flow graph, indicating loops, are dashed.

3 Dominators

Control-flow analysis builds on the key idea of *dominators*. A node A *dominates* another node B (written A dom B) if every path from the start node of the CFG to node B includes A. An edge from A to B is called a *forward edge* if A dom B and a *back edge* if B dom A. Every loop must contain at least one back edge.

The domination relation has some interesting properties. It is reflexive, because a node dominates itself. It is also obviously transitive. Finally, it is antisymmetric. If A dom B and B dom A, they must be the same node. If they were different, then A dom B means you can get to A without going through B, so B can't dominate A. These three properties mean that domination is a partial order.

In addition, two nodes cannot both dominate a third node without there being some domination relationship between the two dominating nodes. Suppose for purpose of contradiction that A dom C and B dom C but neither A dom B nor B dom A. But getting to C requires going through both A and B in some order. Suppose it's A and then B. But in that case if A doesn't dominate B, there must be a path from start to B to C that doesn't go through A. So A couldn't dominate C.

These properties of the domination relationship imply that domination is essentially tree-like. In particular, the Hasse diagram for a CFG is always a tree rooted at the start node. For example, Figure 2 shows a control-flow graph on the left and its corresponding dominator tree on the right.

4 Natural loops

Each back edge from node n to node h in the CFG defines a *natural loop* with h as its header node. The natural loop is a strongly connected subgraph that contains both n and h. It consists of the nodes that are dominated by h and that can reach n without going through h. Thus, the CFG in Figure 2 contains two natural loops.

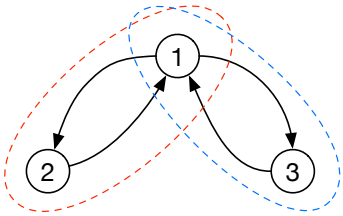

Figure 3: Overlapping natural loops

5 Finding natural loops and control trees

The first step in construction of the program control tree is to compute the domination relation, as described previously. Then, for each back edge $n \to h$ (that is, h dom n), we run a depth-first search starting from n in the transposed CFG. However, the search is stopped when node h is encountered. The set of reached nodes that are dominated by h are the natural loop.

Note that it's possible in a general CFG to reach nodes that are not dominated by h, but in this case the nodes are unreachable code that doesn't need to be included in the loop (or in the program).

Having found the natural loops, we can assemble them into a control tree. Observe that two natural loops can be disjoint or can be nested. Or they can intersect without being nested, but only if they share the same header node. In this case, we can treat them as a single loop for the purpose of constructing the control tree. Figure 3 shows an example of two natural loops that share a header and can be merged together in this fashion.

Once overlapping natural loops are merged, all loops are either disjoint or nested. The control tree then falls out directly from subset inclusion on the various loops.

6 Loop-invariant code motion

Loop-invariant code motion is an optimization in which computations are moved out of loops, making them less expensive. The first step is to identify *loop-invariant expressions* that take the same value every time they are computed.

An expression is loop-invariant if:

1. It contains no memory operands that could be affected during the execution of the loop (i.e., that do not alias any memory operands updated during the loop). To be conservative, we could simply not allow memory operands at all, though fetching array lengths is a good example of a loop-invariant computation that can be profitably hoisted before the loop.

2. And, the definitions it uses (in the sense of reaching definitions) either come from outside the loop, or come from inside the loop but are loop-invariant themselves.

6.1 Analysis

The recursive nature of this definition suggests that we should use an iterative algorithm to find the loop-invariant expressions, as a fixed point. The algorithm works as follows:

1. Run a reaching definitions analysis.

2. Initialize $INV := \{$all expressions in loop, including subexpressions$\}$.

3. Repeat until no change:

    - Remove all expressions from INV that:

- use a memory operand whose value might change in the loop
- might cause a side effect that prevents hoisting: exceptions, nontermination, etc.
- use variables x with more than one definition inside the loop, or whose single definition $x = e$ in the loop has $e \notin INV$.

(The final case is simplified if the code is in SSA form.)

6.2 Code transformation

There are actually two kinds of loop-invariant code motion. The first hoists an assignment to a variable before the loop, the other some computation done inside the loop.

In the first version, we can move the assignment $x = e$ with loop-invariant expression e before the loop header if:

1. it is the only definition of x in the loop,
2. it dominates all loop exits where x is live-out, and
3. it is the only definition of x reaching uses of x in the loop: it is not live-in at the loop header.

If these conditions are not satisfied, we may still be able to perform the second kind of loop-invariant code motion. Here the idea is to hoist the computation of loop-invariant expression (or subexpression) e out of the loop and assign it to a new variable t. Then the occurrences of the expression e are replaced with t.

In either case we need to avoid hoisting expressions e that might generate an exception or other error, because hoisting them ensures they are evaluated, even though they might not have been evaluated in the original execution.