NTNU – Trondheim
Norwegian University of
Science and Technology

Control flow and loop detection

Where we are

- We have a handful of different analysis instances
- None of them are optimizations, in and of themselves
- The objective now is to
  - Show how loop detection is a simple instance of the same ideas
  - Suggest how a combination of different analysis results enable a loop optimization (loop-invariant code motion)

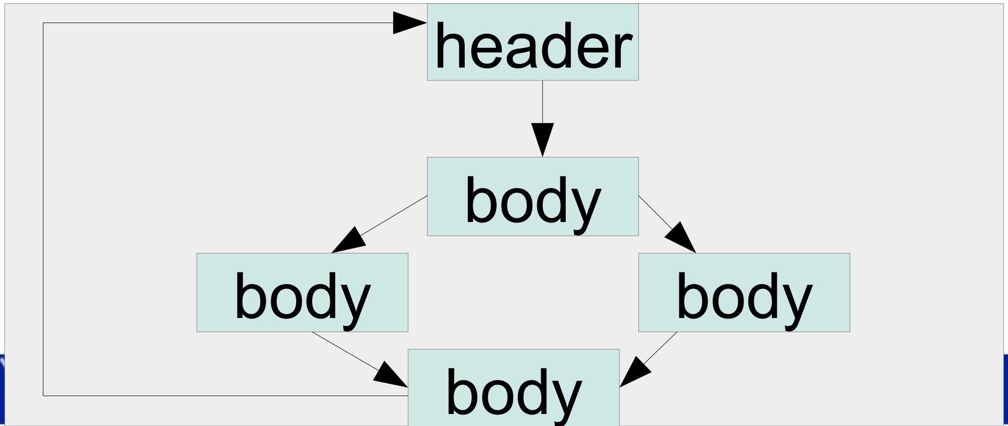
Detecting loops

- It's easy to detect loops at the syntactic level
  - Unless there are free-form jump instructions in the language, loops are explicitly written in the source code
- It's not as easy to detect loops at lower levels
  - Low-level code has only jump instructions
  - General control flow graphs have only edges
- Language-independent optimizations need to elucidate loops implicit in the control flow

Control flow analysis

In a Control Flow Graph,

- A loop is a set of blocks that should be grouped together
- There is a *loop header* every control flow that enters the loop must go through
- There is a *back edge* from one of the blocks that leads back to the header

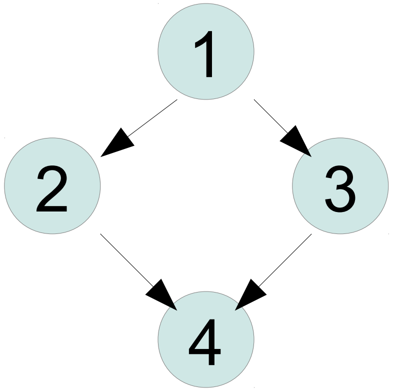


NTNU – Trondheim
Norwegian University of
Science and Technology

Dominator relation

- Introduce the idea that a node X *dominates* a node Y if every path to Y must go through X
- Every node dominates itself
- 1 dominates 1,2,3,4
- Neither 2 nor 3 dominate 4

    (There are paths to 4 which bypass them)



Immediate dominators

- The first node in a CFG dominates all the other ones
  - That's not so useful to know
- If both A and B dominate C, then either

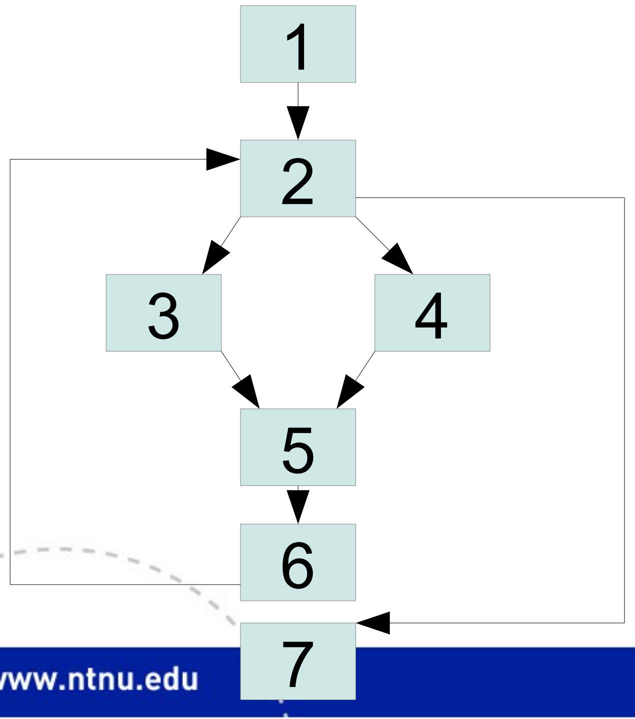
  A dominates B, or

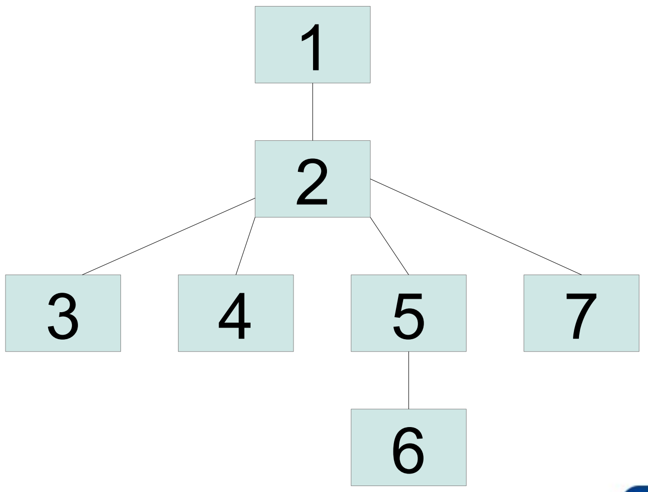
  B dominates A
- A *strictly* dominates B if they're separate (A != B)
- The *immediate* dominator of a node n is the last strict dominator on any path to n
  - There can only be one
  - If there were multiple last strict dominators, they would not be dominators

NTNU – Trondheim
Norwegian University of
Science and Technology

Dominator tree

- Dominators form a hierarchy, so we can represent them as a tree
  - The root is the entry node
  - Children attach to their immediate dominator





Control flow as a set of things

This can be seen as a data flow problem:

- dom(n) = set of nodes that dominate n
- $dom(n) = \cap \{ dom(m) \mid m \in pred(n) \} \cup \{n\}$

- That is:

    dominators of n are the dominators of n-s predecessors, as well as n itself

NTNU – Trondheim
Norwegian University of
Science and Technology

Control flow as a DF problem

Collecting sets of CFG nodes as the problem domain, we have the makings of another framework instance:

- $out[B] = in[B] \cup B$
- $in[B] = \cap \{ out[B'] \mid B' \in pred(B) \}$

- Transfer function is
  - Monotonic
  - Distributive
  - Practically trivial, all it represents is a collection of predecessors

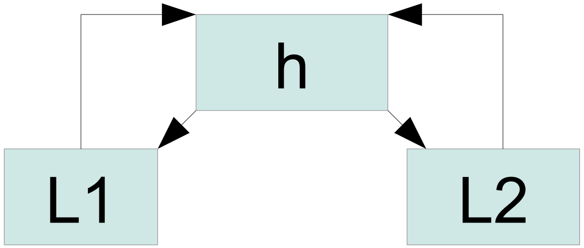
NTNU – Trondheim
Norwegian University of
Science and Technology

Natural loops

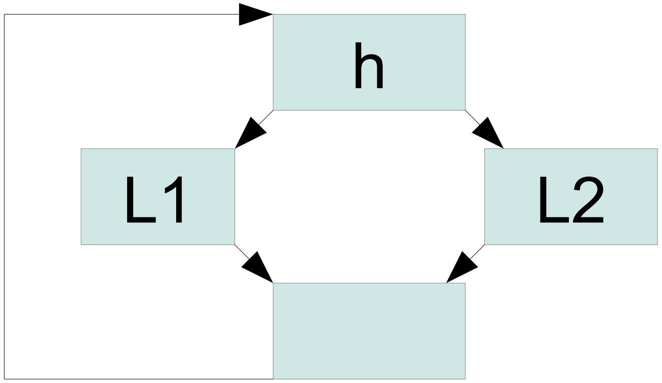
- A *back edge* is an edge where a node has a successor which dominates it
- A natural loop has a back edge n→h such that
  - h is the loop header
  - nodes that can reach n without going through h are the loop body
- To detect natural loops
  - Compute the dominator relation
  - Find back edges (use the dominator relation)
  - Find the loop body (predecessors of n that are dominated by h)
- Starting with a back edge from n, traverse its predecessors until reaching h

Combine loops with shared header

- Unstructured jumps, one can make multiple loops with the same header



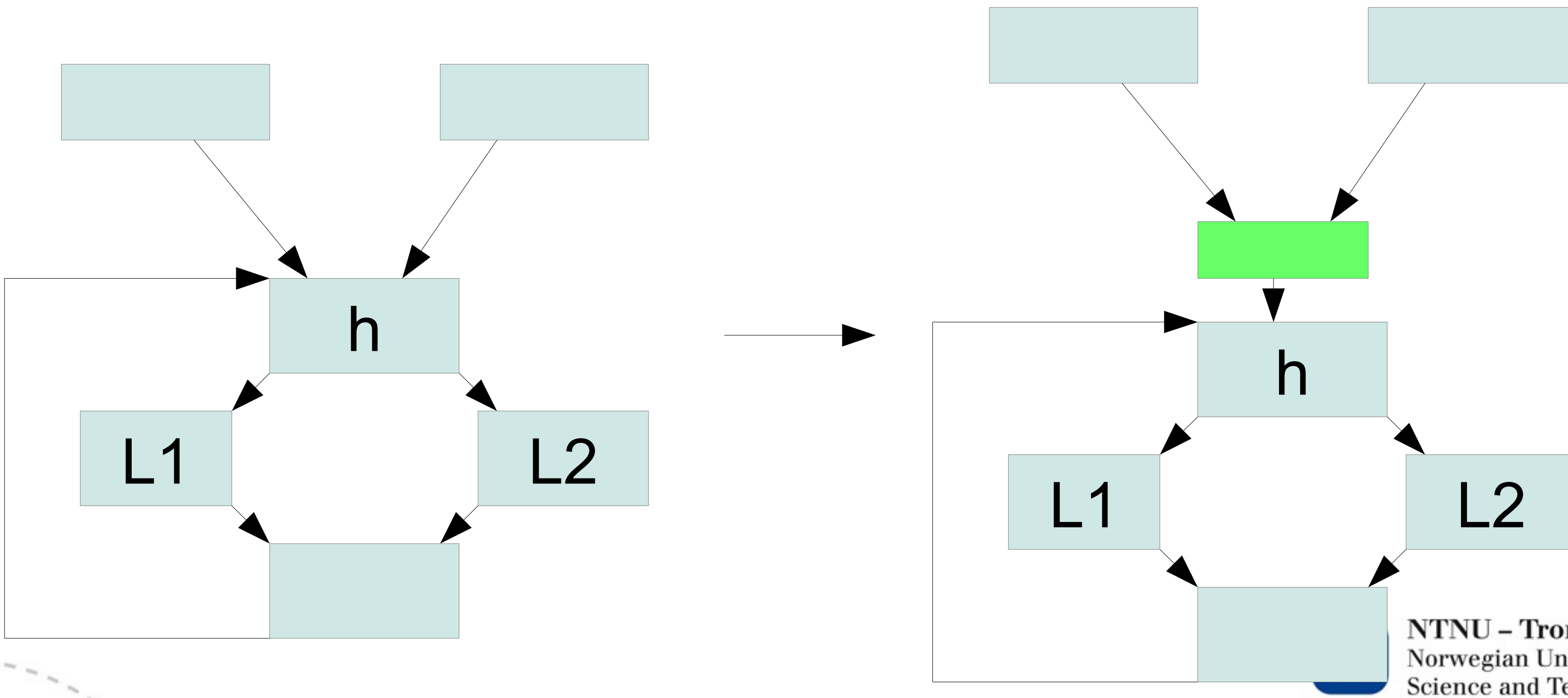
- These can be combined



- This leaves only disjoint and nested loops

Preheader insertion

- If an optimization needs to add code before the header, insert another basic block



...and now, *an application* (finally!)

- Optimizations can combine the results of several analyses
- *Loop invariant code motion* aims to find statements that produce the same result in every iteration, and move it outside of the loop

```
for ( i=0; i<n; i++ )
    buffer[i] = 10*i + x*x;
```

might as well be

```
tmp = x*x
for (i=0; i<n; i++ )
    buffer[i] = 10*i + tmp;
```

Identifying invariant code

- An instruction a = b OP c is loop-invariant if
  - b and c are constants, or
  - all definitions of b and c are outside the loop, or
  - b and c are defined once, and their defs are loop-invariant
- The invariant property for an instruction can be derived from
  - Finding that it's inside a loop (using the dominator relation)
  - Finding the definitions that reach it (using reaching defs)

Moving invariant code

- Introduce a pre-header and move a = b OP c there if
  - The definition a = b OP c dominates every loop exit where a is live
    - Search nodes dominated by this one, consult live variables
  - There is no other definition of a in the loop
    - Consult dominators for loop body, scan them for definitions of a
  - Every use of a in the loop can only be reached by this def.
    - Consult reaching definitions at every use of a, to see if there are others than this one which can influence it