Advanced Compilers

CMPSCI 710
Spring 2003
Lecture 2

Emery Berger
University of Massachusetts, Amherst

Control-Flow Analysis

- Motivating example: **identifying loops**
  - majority of runtime
  - ⇨ focus optimization on loop bodies!
    - remove redundant code, replace expensive operations) speed up program
- Finding loops:
  - easy…
  - or harder (GOTOs)

```
for i = 1 to 1000
  for j = 1 to 1000
    for k = 1 to 1000
     do something
```

```
i = 1; j = 1; k = 1;
A1: if i > 1000 goto L1;
A2: if j > 1000 goto L2;
A3: if k > 1000 goto L3;
do something
k = k + 1; goto A3;
L3: j = j + 1; goto A2;
L2: i = i + 1; goto A1;
L1: halt
```

Steps to Finding Loops

1. Identify basic blocks
2. Build control-flow graph
3. Analyze CFG to find loops

Control-Flow Graphs

- **Control-flow graph**:
  - Node: an instruction or sequence of instructions (a **basic block**)
    - Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
  - Directed edge: *potential* flow of control
  - Distinguished start node *Entry*
    - First instruction in program

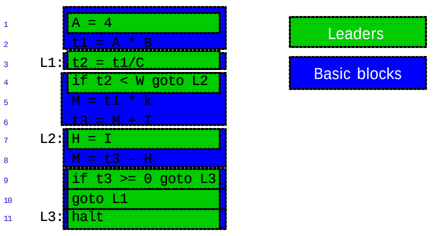
Identifying Basic Blocks

- Input: sequence of instructions *instr(i)*
- Identify **leaders**:
  first instruction of basic block
- Iterate: add subsequent instructions to basic block until we reach another leader

Basic Block Partition Algorithm

```
leaders = 1                // start of program
for i = 1 to |n|           // all instructions
  if instr(i) is a branch
     leaders = leaders [ targets of instr(i)
worklist = leaders
While worklist not empty
  x = first instruction in worklist
  worklist = worklist – {x}
  block(x) = {x}
  for i = x + 1; i <= |n| && i not in leaders; i++
     block(x) = block(x) [ {i}
```

Basic Block Example



```
    A = 4
    t1 = A * B
L1: t2 = t1/C
    if t2 < W goto L2
    M = t1 * k
    t3 = M + I
L2: H = I
    M = t3 - H
    if t3 >= 0 goto L3
    goto L1
L3: halt
```

Control-Flow Edges

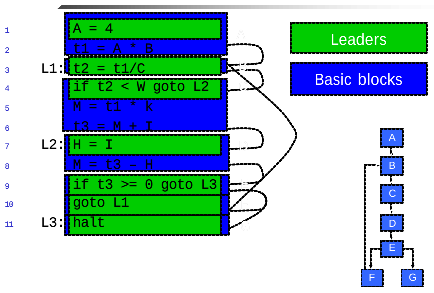
- Basic blocks = nodes
- Edges:
  - Add directed edge between B1 and B2 if:
    - Branch from last statement of B1 to first statement of B2 (B2 is a leader), or
    - B2 immediately follows B1 in program order and B1 does not end with unconditional branch (goto)

Control-Flow Edge Algorithm

```
Input: block(i), sequence of basic blocks
Output: CFG where nodes are basic blocks

for i = 1 to the number of blocks
   x = last instruction of block(i)
   if instr(x) is a branch
       for each target y of instr(x),
               create edge block i ! block y
   if instr(x) is not unconditional branch,
       create edge block i ! block i+1
```

CFG Edge Example



```
    A = 4
    t1 = A * B
L1: t2 = t1/C
    if t2 < W goto L2
    M = t1 * k
    t3 = M + I
L2: H = I
    M = t3 - H
    if t3 >= 0 goto L3
    goto L1
L3: halt
```

Steps to Finding Loops

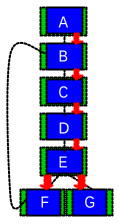
1. Identify basic blocks
2. Build control-flow graph
3. Analyze CFG to find loops
   - Spanning trees, depth-first spanning trees
   - Reducibility
   - Dominators
   - Dominator tree
   - Strongly-connected components

Spanning Tree

- Build a tree containing every node and some edges from CFG

```
procedure Span (v)
  for w in Succ(v)
    if not InTree(w)
      add w, v !w to ST
      InTree(w) = true
      Span(w)

for v in V do inTree = false
InTree(root ) = true
Span(root )
```



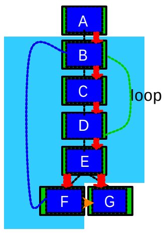
CFG Edge Classification

Tree edge:
in CFG & ST

Advancing edge:
(v,w) not tree edge but w is descendant of v in ST

Back edge:
(v,w): v=w or w is proper ancestor of v in ST

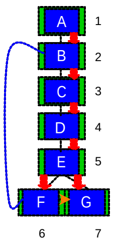
Cross edge:
(v,w): w neither ancestor nor descendant of v in ST



Depth-first spanning tree

```
procedure DFST (v)
  pre(v) = vnum++
  InStack(v) = true
  for w in Succ(v)
    if not InTree(w)
      add v!w to TreeEdges
      InTree(w) = true
      DFST(w)
    else if pre(v) < pre(w)
      add v!w to AdvancingEdges
    else if InStack(w)
      add v!w to BackEdges
    else
      add v!w to CrossEdges
  InStack(v) = false

for v in V do inTree = false
vnum = 0
DFST(root)
```

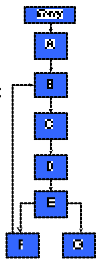


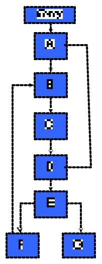
Reducibility

- **Natural loops**:
  - no jumps into middle of loop
  - entirely disjoint or nested
- **Reducible**: hierarchical, "well-structured"
  - flowgraph reducible *iff* all loops in it natural

Reducibility Example

- Some languages only permit procedures with reducible flowgraphs (e.g., Java)
- "GOTO Considered Harmful": introduces irreducibility
  - FORTRAN
  - C
  - C++
- DFST does not find unique header in irreducible graphs


reducible graph

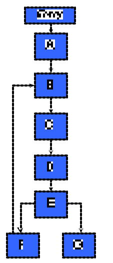

irreducible graph

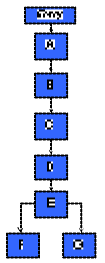
Dominance

- Node *d* **dominates** node *i* ("*d dom i*")
  if every path from *Entry* to *i* includes *d*
  - Reflexive: *a dom a*
  - Transitive: *a dom b, b dom c* ! *a dom c*
  - Antisymmetric: *a dom b, b dom a* ! *b*=*a*
- **Immediate dominance**:
  - *a idom b* iff *a dom b*
    Æ no *c* such that *a dom c, c dom b* (*c* ¹ *a*, *c* ¹ *b*)
  - Idom's:
    - each node has unique idom
    - relation forms tree

Dominance Example

- Immediate and other dominators: (excluding *Entry*)
  - *a idom b; a dom a, c, d, e, f, g*
  - *b idom c; b dom b, d, e, f, g*
  - *c idom d; c dom c, e, f, g*
  - *d idom e; d dom d, f, g*
  - *e idom f, e idom g; e dom e*


control-flow graph


dominator tree

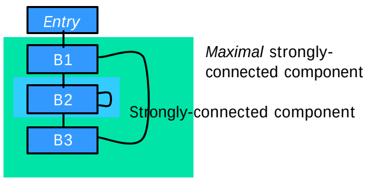
Dominance and Loops

- Redefine *back edge* as one whose head dominates its tail
  - Slightly more restrictive definition
- Now we can (finally) find natural loops!
  - for back edge *m* ! *n*, natural loop is subgraph of nodes containing *n* (*loop header*) and nodes from which *m* can be reached without passing through *n* + connecting edges

Strongly-Connected Components

- What about irreducible flowgraphs?
- Most general loop form = **strongly-connected component (SCC)**:
  - subgraph S such that every node in S reachable from every other node by path including only edges in S
- **Maximal SCC**:
  - S is maximal SCC if it is the largest SCC that contains S.
- Now: Loops = all maximal SCCs

SCC Example


Entry
B1
B2
B3
Maximal strongly-connected component
Strongly-connected component

Computing Maximal SCCs

- Tarjan's algorithm:
  - Computes all maximal SCCs
  - Linear-time (in number of nodes and edges)
- CLR algorithm:
  - Also linear-time
  - Simpler:
    - Two depth-first searches and one "transpose": reverse all graph edges
- Unlike DFST, neither distinguishes inner loops

Conclusion

- Introduced control-flow analysis
  - Basic blocks
  - Control-flow graphs
- Discussed application of graph algorithms: loops
  - Spanning trees, depth-first spanning trees
  - Reducibility
  - Dominators
  - Dominator tree
  - Strongly-connected components

Next Time

- Dataflow analysis
  - Read ACDI Chapter 8, pp. 217-251
    photocopies should be available soon