Compilers Course
Lecture 18: Basic Blocks, Control Flow Graphs (CGF), Liveness, Graph Coloring

Basic Blocks

A basic block (BB) is a maximal sequence of instructions with:

- No jumps out of the sequence, except in the last instruction
- No jumps into the sequence, except to the first instruction (no labels in the sequence except at the first instruction)

A BB has a single entry, a single exit, and is strictly sequential in between.

Identifying BBs:

- A jump terminates the current BB
- A label starts a new BB

Example:

```
-------------------------
 I1. L1: a := b+c
 I2.     d := -a                BB1
 I3.     e := d + f
 I4.     if e > 0 goto L2
-------------------------
 I5.     f := 2*e               BB2
 I6.     goto L3
-------------------------
 I7. L2: b := d+e               BB3
 I8.     e := e-1
-------------------------
 I9. L3: b := f+c               BB4
 I10.    if b > 0 goto L1
-------------------------
 I11.    return b               BB5
-------------------------
```

BB1 starts at *I1* and ends at *I4*, because *I4* is a jump.
BB2 starts at *I5* and ends at *I6*, because *I6* is a jump.
BB3 starts at *I7* and ends at *I8*, because *I9* has a label.
BB4 starts at *I9* and ends at *I10*, because *I10* is a jump.
BB5 starts at *I11* and ends at *I11*, because *I11* is a jump.

Control-Flow Graphs

A control-flow graph (CFG) is a directed graph:

- Each node is a BB
- There is an edge from BB ***A*** to BB ***B*** if there is a jump from **A**'s last instruction to **B**'s first instruction, or if **A** "falls through" to **B** (implicit jump)

In the example:
BB1 has edges to *BB2* and *BB3*
BB2 has an edge to *BB4*
BB3 has an edge to *BB4*
BB4 has edges to *BB1* and *BB5*

For traversing the CFG, two functions are used:
```
succ[n] = { the set of nodes n' where n → n' }
pred[n] = { the set of nodes n' where n' → n }

succ[BB1] = { BB2, BB3 }
succ[BB2] = { BB4 }
succ[BB3] = { BB4 }
succ[BB4] = { BB1, BB5 }
succ[BB5] = {}

pred[BB1] = { BB4 }
pred[BB2] = { BB1 }
pred[BB3] = { BB1 }
pred[BB4] = { BB2, BB3 }
pred[BB5] = { BB4 }
```

Liveness Analysis

Consider:

```
I1:  x := y op z
```

x is alive (needed) after I1 (LIVEOUT) if

- There is an instruction *I2* that uses *x*, and
- There is an execution path *P* from *I1* to *I2*, and
- On path *P* there is no other assignment to *x*.

Instruction-local functions:
```
DEF(I) = { x | instruction I assigns to temporary x }
USE(I) = { x | instruction I uses the value of temporary x }
```

Now consider an instruction *I*:
x is alive (needed) before *I* (LIVEIN) if

- *x* is used by *I* (USE), or
- *x* is alive after *I* (LIVEOUT) and not assigned by *I* (DEF)

x is alive after I (LIVEOUT) if *x* is alive before (LIVEIN) any successor instruction *I'* to *I* (succ), so:

```
LIVEIN(I) = USE(I) ∪ (LIVEOUT(I) \ DEF(I))
LIVEOUT(I) = ∪{I' ∈ succ(I)} LIVEIN(I')
```

Computing Liveness

- LIVEIN and LIVEOUT are mutually recursive due to loops in the CFG.
- To compute them perform a least fixpoint iteration:
  - Compute DEF and USE for every instruction.
  - Initialize LIVEIN = LIVEOUT = empty for all instructions I.
  - Update LIVEIN and LIVEOUT for all instructions I.
  - Repeat step 3 until no more changes occur.

To compute liveness efficiently, observe that:

- liveness is a property where data flows backwards in the program's control flow

To update/recompute LIVEIN for a basic block BB = [I1; I2; ...; In]:

```
LIVEOUT(In) = LIVEOUT(BB)
for every instruction Ij from n down to 1:
    LIVEIN(Ij) = USE(Ij) ∪ (LIVEOUT(Ij) \ DEF(Ij))
    if j > 1 then LIVEOUT(Ij-1) = LIVEIN(Ij)
LIVEIN(BB) = LIVEIN(I1)
```

To compute liveness for an entire CFG:

1. Compute *DEF(I)* and *USE(I)* for every instruction *I*.
2. Set *LIVEIN(I) = LIVEOUT(I) = empty* for every instruction *I*
3. Set *LIVEIN(BB) = LIVEOUT(BB) = empty* for every basic block *BB*
4. Set *LIVEOUT(BB) = bottom* for every basic block *BB*. *Bottom* is a special value distinct from any set.
5. Traverse CFG in depth-first order and append BBs to a worklist *WKL*. *WKL* contains BBs whose *LIVEOUT* may have changed.
6. While *WKL* is non-empty:
   - Let *BB* be the first element in *WKL*. Remove it from *WKL*.
   - Recompute LIVEOUT(BB) = ∪{LIVEIN(BB') for BB' ∈ succ(BB)}.

- If *LIVEOUT(BB)* changed, recompute *LIVEIN(BB)*.
- If *LIVEIN(BB)* changed, add every BB' ∈ pred[BB] to *WKL* if they aren't already there.

Computing liveness for the example:

1) WKL = [BB5, BB4, BB3, BB2, BB1].

2) BB=BB5, WKL = [BB4, BB3, BB2, BB1].
LIVEOUT(BB5) is currently bottom.
Recompute LIVEOUT(BB5) = empty.
LIVEOUT(BB5) changed so recompute LIVEIN(BB5).
LIVEIN(BB5) is currently empty.
LO(I11) = empty, LI(I11) = {b} = LIVEIN(BB5).
LIVEIN(BB5) changed so add pred(BB4) = BB4 to WKL (already there).

3) BB=BB4, WKL = [BB3, BB2, BB1].
LIVEOUT(BB4) is currently bottom.
Recompute LIVEOUT(BB4) = LIVEIN(BB5) ∪ LIVEIN(BB1)
= {b} ∪ {} = {b}.
LIVEOUT(BB4) changed so recompute LIVEIN(BB4).
LIVEIN(BB4) is currently empty.
LO(I10) = {b}, LI(I10) = {b}, LO(I9) = {b}, LI(I9) = {c,f}
= LIVEIN(BB4).
LIVEIN(BB4) changed so add pred(BB4) = {BB3,BB2} to WKL (already there).

4) BB=BB3, WKL = [BB2, BB1].
LIVEOUT(BB3) is currently bottom.
Recompute LIVEOUT(BB3) = LIVEIN(BB4) = {c,f}.
LIVEOUT(BB3) changed so recompute LIVEIN(BB3).
LIVEIN(BB3) is currently empty.
LO(I8) = {c,f}, LI(I8) = {c,e,f}, LO(I7) = {c,e,f}, LI(I7) = {c,d,e,f}
= LIVEIN(BB3).
LIVEIN(BB3) changed so add pred(BB3) = BB1 to WKL (already there).

5) BB=BB2, WKL = [BB1].
LIVEOUT(BB2) is currently bottom.
Recompute LIVEOUT(BB2) = LIVEIN(BB4) = {c,f}.
LIVEOUT(BB2) changed so recompute LIVEIN(BB2).
LIVEIN(BB2) is currently empty.
LO(I6) = {c,f}, LI(I6) = {c,f}, LO(I5) = {c,f}, LI(I5) = {c,e}
= LIVEIN(BB2).
LIVEIN(BB2) changed so add pred(BB2) = BB1 to WKL (already there).

6) BB=BB1, WKL = [].
LIVEOUT(BB1) is currently bottom.
Recompute LIVEOUT(BB1) = LIVEIN(BB2) ∪ LIVEIN(BB3) = {c,d,e,f}.

LIVEOUT(BB1) changed so recompute LIVEIN(BB1).
LIVEIN(BB1) is currently empty.
LO(I4) = {c,d,e,f}, LI(I4) = {c,d,e,f}, LO(I3) = {c,d,e,f},
LI(I3) = {c,d,f}, LO(I2) = {c,d,f}, LI(I2) = {a,c,f},
LO(I1) = {a,c,f}, LI(I1) = {b,c,f} = LIVEIN(BB1).
LIVEIN(BB1) changed so add pred(BB1) = BB4 to WKL. WKL = [BB4].

7) BB=BB4. WKL=[].
LIVEOUT(BB4) is currently {b}.
Recompute LIVEOUT(BB4) = LIVEIN(BB5) ∪ LIVEIN(BB1) = {b,c,f}.
LIVEOUT(BB4) changed so recompute LIVEIN(BB4).
LIVEIN(BB4) is currently {c,f}.
LO(I10) = {b,c,f}, LI(I10) = {b,c,f}, LO(I9) = {b,c,f}, LI(I9) = {c,f}
= LIVEIN(BB4).
LIVEIN(BB4) did not change, so no BBs added to WKL.

8. WKL is empty. We're done.

In the example:

```
                                USE     DEF  LIVEIN        LIVEOUT
-------------------------
 1. L1: a := b+c       | BB1    b,c     a    b,c,f         a,c,f
 2.  d := -a           |        a       d    a,c,f         c,d,f
 3.  e := d+f          |        d,f     e    c,d,f         c,d,e,f
 4.  if e > 0 goto L2  |        e            c,d,e,f       c,d,e,f
-------------------------
 5.  f := 2*e          | BB2    e       f    c,e           c,f
 6.  goto L3           |                     c,f           c,f
-------------------------
 7. L2: b := d+e       | BB3    d,e     b    c,d,e,f       c,e,f
 8.  e := e-1          |        e       e    c,e,f         c,f
-------------------------
 9. L3: b := f+c       | BB4    c,f     b    c,f           b,c,f
10.  if b > 0 goto L1  |        b            b,c,f         b,c,f
-------------------------
11. return b           | BB5    b            b             -
-------------------------
```

Interference Graph (IG)

A (register) interference graph, IG:

- Each node is a temporary (virtual register)
- There is an undirected edge between nodes v1 and v2, written (v1,v2), if v1 and v2 are live at the same time, which means that they cannot use the same physical register

Constructing the IG:
At each instruction *I*:

- Let DEF(I) = {a} and LIVEOUT(I) = {b_1,...,b_n}
- For each b_j != a, add an edge (a, b_j)

In the example:
(a, f) and (a, c) from I1
(d, f) and (d, c) from I2
(e, d), (e, c), and (e, f) from I3
(f, c) from I5
(b, e), (b, c), and (b, f) from I7
(e, c) and (e, f) from I8
(b, c) and (b, f) from I9

Colorings

A valid coloring of an IG:

- Each node has some color
- No two adjacent nodes have the same color

A K-coloring of an IG:

- A valid coloring of the IG
- Using exactly K distinct colors

In a compiler:

- Color == physical register
- K == number of available registers

An Observation

Finding K-colorings is expensive (NP-hard).

Kempe [1879] observed the following:

- Let G be a graph.
- Assume node n in G has fewer than K neighbors.
- Let G' = G \ n.
- Theorem: if G' has a K-coloring, then so does G. Because node n has < K neighbors, we can always find a color for it.

Simple Graph Coloring

Simplify:

- Find low-degree (< K neighbors) nodes, remove them from the graph, and push them on a stack
- Repeat until we end up with the empty graph

Select:

- Pop topmost node n from the stack
- Add n to the graph
- Assign n a color (it is guaranteed to have < K neighbors in the current graph)
- Repeat until the stack is empty

Trying to perform a 4-coloring of the example:

Simplify:
push a (degree 2)
push d (degree 3)
push c (degree 3)
push b (degree 2)
push e (degree 1)
push f (degree 0)

Select:
pop f, assign f = r1
pop e, assign e = r2
pop b, assign b = r3
pop c, assign c = r4
pop d, assign d = r3
pop a, assign a = r2

Graph Coloring with Spills

The heuristic can fail: simplify can find an IG with only significant-degree (>= K neighbors) nodes.

Optimistic coloring:

- Simplify: if only significant-degree nodes remain, pick one, remove it from the graph and push it on the stack anyway. This is called a "potential spill".
- Select: coloring a potentially spilled node may succeed or fail: if it failed, mark the node as an "actual spill", remove it from the graph, and continue.

An actual spill means that some temporary couldn't be permanently assigned to a specific register. We must simplify and rewrite the program to reduce its "register pressure":

```
- Some set of nodes N became actual spills
- Select one of them, say n
- Allocate a memory area for n in the stack
- Rewrite any instruction using n
    x := m op n
  as:
    t := load n's value from the stack frame
    x := m op t
  where t is a new temporary
- Rewrite any instruction assigning n
    n := E
  as
    t := E
    store t in n's area in the stack frame, where t is a new temporary
```

Then we must rebuild the CFG, redo the liveness analysis, rebuild the IG, and repeat the graph coloring process. The new temporaries t should be marked so that they cannot be spilled.

1. Build IG.
2. Simplify loop.
3. Potential spill. Resume at 2.
4. Select loop with optimistic coloring.
5. No actual spills? If so, we're done.
6. Handle actual spills. Rebuild CFG. Resume at 1.

Trying to perform a 3-coloring of the example:
Simplify:
push a (degree 2)

Potential spill:
push f (degree 4), resume simplify

Simplify:
push b, d, e, c

Select:
pop c (r1), e (r2), d (r3), b (r3), f (no color), a (r2)

f was an actual spill:
rewrite I3 as I3a: f3 := load FP-offset_f; I3b: e := d+f3
rewrite I5 as I5a: f5 := 2*e; I5b: store f5 to FP-offset_f
rewrite I9 as I9a: f9 := load FP-offset_f; I9b: b := f9+c

Update liveness:
f is no longer LIVEOUT or LIVEIN for any instruction

LIVEOUT(I3b) = {e, d, c}
LIVEIN(I3b) = {d, f3, c} = LIVEOUT(I3a)
LIVEIN(I3a) = {d, c}

LIVEOUT(I5b) = {c}
LIVEIN(I5b) = {c, f5} = LIVEOUT(I5a)
LIVEIN(I5a) = {e, c}

LIVEOUT(I9b) = {b, c}
LIVEIN(I9b) = {c, f9} = LIVEOUT(I9a)
LIVEIN(I9a) = {c}

New IG:
(a, c) from I1
(d, c) from I2
(f3, d) and (f3, c) from I3a
(e, d) and (e, c) from I3b
(f5, c) from I5a
(b, e) and (b, c) from I7
(e, c) from I8
(f9, c) from I9a
(b, c) from I9b

Simplify:
push a, f5, f9, b, f3, c, d, e

Select:
pop e (r1), d (r2), c (r3), f3 (r1), b (r2), f9 (r1), f5 (r1), a (r1).

Handling spills is an important but complex part of an optimizing compiler:

- Heuristics for selecting which spill candidate to spill
- Heuristics for partitioning a spilled temporary: above we spilled it permanently, it's also possible to split it into two or more temporaries used in different parts of the program (live range splitting) and to color those independently