COS320 Register allocation

- Informed by program analysis:

live variables analysis

- Var x is live at point n if ∃ path starting from n that reads value of x before writing to it
  - x is live if it uses later computation
- If x isn't live we can free/reuse memory associated w/ x
- If two variables aren't live at the same time, we can store them in the same memory

Live Variables

- backwards dataflow analysis problem

Compute least IN, OUT such that

① $OUT[n] = \top$ ∀ return blocks n

② $\forall n \in N,\ pre_L(n, OUT[n]) \sqsubseteq IN[n]$

③ $\forall (n \to s) \in E,\ IN[s] \sqsubseteq OUT[n]$

pre instead of post.

Local specification:

① Abstract domain: 2^{vars} where vars is variables

↖ gen/kill on vars

Existential analysis

order $\subseteq$, join $\cup$, $\top$ is vars, $\bot$ is $\emptyset$

$$pre(elt, L) = (L \setminus kill(elt)) \cup gen(elt)$$

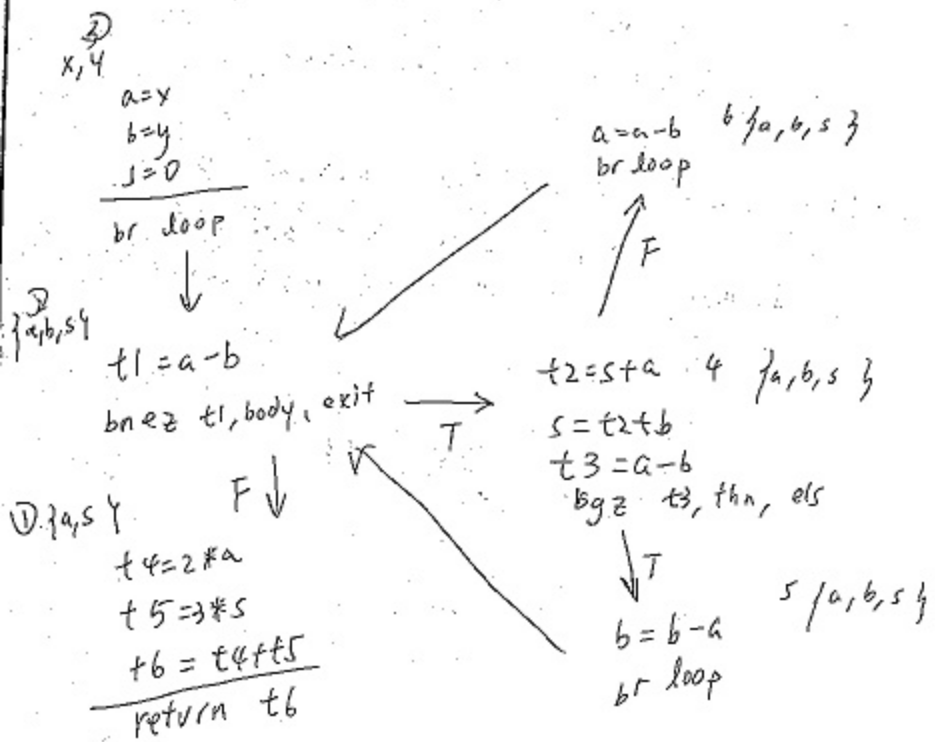
gen(x=e) → everything else

$kill(x := e) = \{x\}$ ← Assignment = kill

$gen(br\ x) = \{x\}, kill(cbr\ x, l_1, l_2) = \emptyset$ ← Account for terms

↑ generate x if cbr.

Example


2
x, y
a = x
b = y
s = 0
br loop
3
{a, b, s}
t1 = a - b
bnez t1, body, exit
T
F
1
{a, s}
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s + a
4
{a, b, s}
s = t2 + b
t3 = a - b
bgz t3, thn, els
T
b = b - a
br loop
5
{a, b, s}
a = a - b
br loop
6
{a, b, s}
F

Next phase: Interference graph thing of

- undirected graph (V, I) where

  Vertices V = program variables

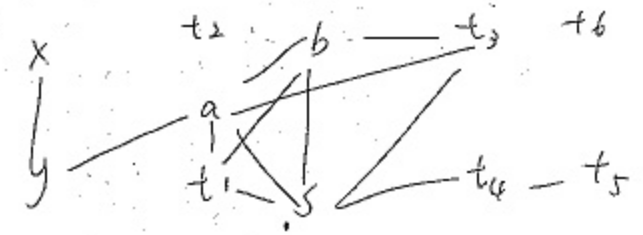
  Edges I connect variables x and y

  $\iff$ x, y are simultaneously live

Encapsulates constraint for register allocation

$\iff$ Vertex coloring: Variables that are simultaneously live can't be in same coloring

ex



Def k-coloring of interference graph

$C : V \longrightarrow \{1 \ldots k\}$

such that for adjacent vertices V_i, V_j

$C(V_i) \neq C(V_j)$

Idea (Chaitin) Processor w/ k ge registers

$\longleftrightarrow$ k-coloring of interference graph = valid memory layout

☆ Idea: Reduce to low-level optimization problem ☆

Problem: k-Coloring is NPC

But we don't need optimal coloring (any coloring does)

If use more colors than regs, then we spill (place var in memory rather than register)

Greedy Coloring - Approximation!

Idea: Assign colors to nodes in some order

For each node, assign color that isn't already assigned to one of its neighbors (No color available ⇒ spill)

If node has < k neighbors, a color is always available

(each color corresponds to a maximal independent set)

Process

Simplify: Choose node w/ < K neighbors. Add to stack. Remove from graph.

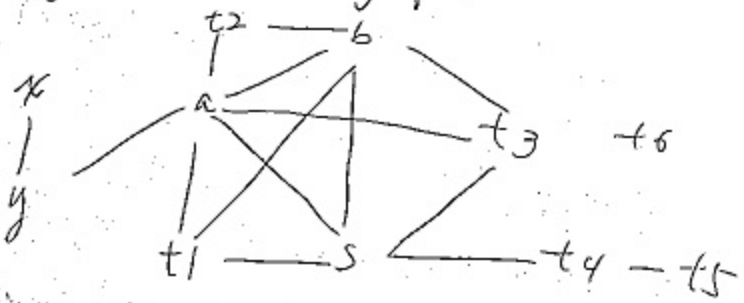
Spill: if all nodes have ≥ K neighbors, choose one to potentially spill. Add it to the stack & remove from graph. (Either color or spill)

Color: Traverse the stack, assign colors to simplified vertices.

Not optimal, but works well in practice.

Example:

3-coloring the interference graph



① Simplify:

Stack: t6, x, y, t5, t4, t2

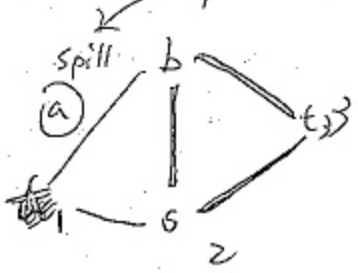
② Spill:

t1, t3, b, s

③ Graph empty — traverse graph

cannot color = spill!

Small example:



Accessing spilled registers.

- Problem: Need to use regs to access stack slots which we use to store them

Easy option — Reserve some regs for memory operation

Better option — Generate spill code, (using virtual regs) re-run register allocator

Spill code will take new virtual registers: e.g. if x is spilled in xloc, y is spilled in yloc.

x = y ⇝ t = load yloc;
store t xloc

When re-running register allocator, we allocate registers to these virtual regs. (Albeit might generate additional spills)

- live range for new virtual reg is very short.
- Use some bookkeeping to prevent ∞ loop.

- Boundary case exists! we might spill again. so bookkeeping is necessary.

Pre-colored nodes.

- Some instructions require use of certain registers
  eg, call must pass parameters in rdi rsi
  rdx rcx r08 r09
- Virtual registers must then be assigned to particular ~~register~~ color, consider them 'pre-colored'
- Terminate register allocator when no uncolored nodes remain.

Graph coloring isn't end of story

- Spill selection: (Which reg is spilled?)
  - Priority based on loops, etc.

- live range splitting
  - might be desirable to allocate a single variable in different registers in different code sections
  - SSA does some of this implicitly

Graph Coalescing

May be desirable to place two variables inside the same register

ex if we have assignment $x := y$ and x, y are in same register, we can elide the mov instruction.

Graph coalescing: collapses two non adjacent vertices into one vertex w/ neighborhood of both.

- Creates more register pressure
- Strategies to preserve K-colorability:

Briggs: Coalesce only when resulting node has $< K$ neighbors w/ degree $\geq K$.

George's (Appel): Coalesce x, y only when each neighbor of x is either a neighbor of y or has degree $< K$.

COS320 Control Flow Theory

Static Single Assignment Form

SSA: Each %uid appears on the lhs of at most one assignment in a CFG

ϕ-Nodes: SSA in conditional branching

```
if (x<0) {
  y := y - x;
} else {
  y := y + x;
}
return y
```

$\Rightarrow$

```
if (x0<0) {
  y1 := y0 - x0;
} else {
  y2 := y0 + x0;
}
y3 = φ(y1, y2);
return y3
```

Well-formedness condition: uids must be defined before use.