Register Allocation

The problem:

assign machine resources (registers, stack locations)
to hold run-time data

Constraint:

simultaneously live data allocated to different locations

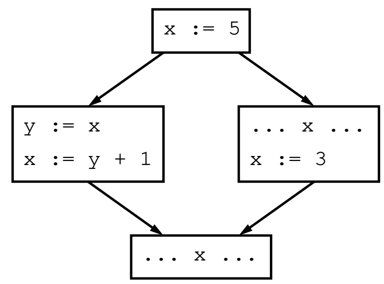
Goal:

minimize overhead of stack loads & stores
and register moves

Granularity of allocation

What is allocated to registers/stack locations?

- variables?
- values?
- separate def/use chains (**live ranges**)?



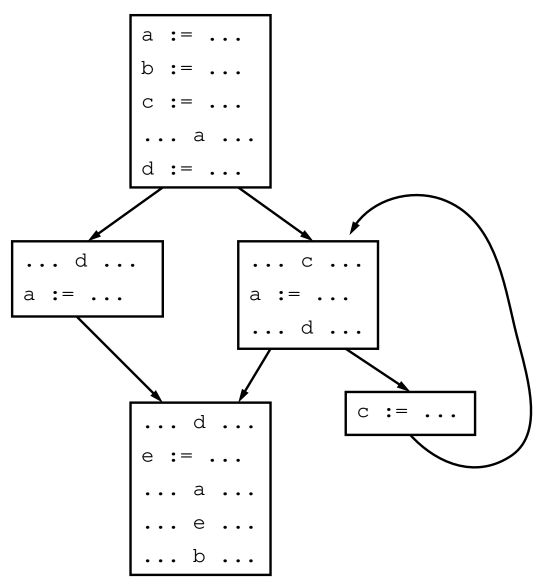
Interference graph

Represent notion of “simultaneously live” using
interference graph

- nodes are units of allocation
- n_1 is linked by an edge to n_2 if n_1 and n_2 are simultaneously live at some program point
- symmetric, not reflexive, not transitive

Two adjacent nodes must be allocated to distinct locations

Example



Computing interference graph

Construct as side-effect of live variables analysis

- backwards iterative dfa algorithm

Flow function: identify defs & last uses

E.g.

```
a := ... b ...
```

Merge function:

E.g.

```
liveVars1 merge liveVars2
```


Allocating registers using interference graph

Allocating variables to *k* registers is equivalent to finding a *k*-coloring of the interference graph

k-coloring: color nodes of graph using up to *k* colors, adjacent nodes have different colors

- optimal graph coloring: NP-complete

Spilling

If can't find *k*-coloring of interference graph,
must **spill** some variables to stack,
until the resulting interference graph is *k*-colorable

Which to spill?

- least frequently accessed variables
- most conflicting variables

Weighted interference graph:

weight(*n*) =
sum over all references (uses and defs) *r* of *n*:
execution frequency of *r*

Try to spill nodes with lowest weight, if forced to spill

Static frequency estimates

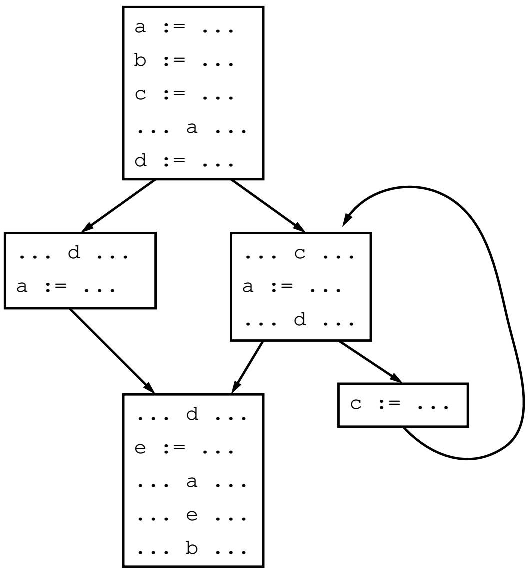
Initial node: weight = 1

Nodes after branch: 1/2 weight of branch

Nodes in loop: 10x nodes outside loop

Just need heuristic ranking of variables

Example



Simple greedy allocation algorithm

For all nodes, in decreasing order of weight:

- try to allocate node to a register, if possible
- if not, allocate to a stack location

Reserve 2-3 temp registers to use when manipulating nodes allocated to stack locations

Improvement #1: add simplification phase

[Chaitin 82]

Key idea:

nodes with < *k* neighbors can be allocated after all their neighbors, and guaranteed a register

So remove them from the graph first

- reduces the degree of the remaining nodes

Must resort to spilling only when all remaining nodes have degree ≥*k*

Chaitin allocates live ranges to registers

The algorithm

while interference graph not empty:
  while there exists a node with < *k* neighbors:
    remove it from the graph
    push it on a stack
  if all remaining nodes have *k* neighbors, then **blocked**:
    pick a node to spill
      (choose node with lowest spill cost/degree)
    remove node from graph
    add to spill set

if any nodes in spill set:
  insert spill code for all spilled nodes
    (insert stores after defs, loads before uses)
  reconstruct interference graph, start over

while stack not empty:
  pop node from stack
  allocate to register

Spilling

Chaitin restarts algorithm if anything spilled

- loads & stores of spilled value need registers for short time

Alternative:
reserve a couple temp registers for use by spill code

+ no need to start over if anything spilled
– lose 2-3 registers that might help avoid spilling

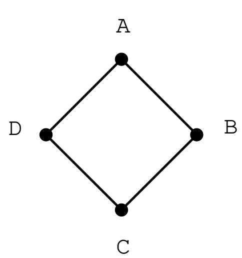
Subsumption

Twist in Chaitin's algorithm:
if see `x := y`, where `x` & `y` not simultaneously live,
then merge live ranges & eliminate all such copies

- can introduce extra spilling, though

If allocate values instead of variables,
then subsumption happens automatically

An annoying case



If only 2 registers available ⇒ blocked immediately, must spill

Improvement #2: blocked doesn't mean spill

[Briggs *et al.* 89]

Key idea:
just because a node has *k* neighbors
doesn't mean it will need to be spilled
(neighbors may get overlapping colors)

Algorithm:
Like Chaitin, except:

- when removing blocked node, just push onto stack
- when done removing nodes:
  - pop nodes off stack and see if they can be allocated
  - really spill only if it can't be allocated at this stage

Other miscellaneous enhancements

Results:
~ 1% execution time improvement over Chaitin

Improvement #3: live range splitting

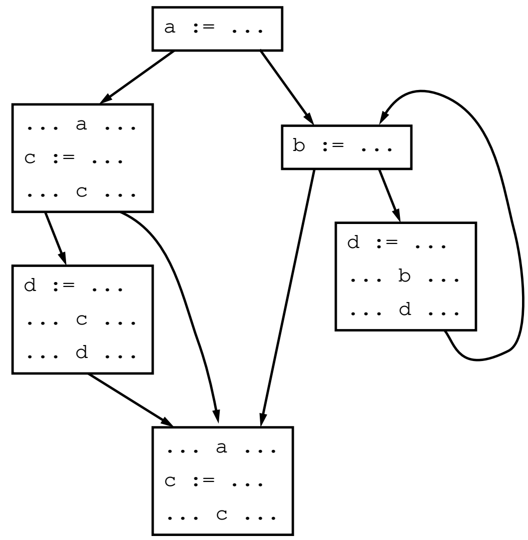
Priority-Based Coloring [Chow & Hennessy 86]

Key idea: if a variable can't be allocated to a register,
try to split it into multiple subranges that can be allocated separately

Several miscellaneous differences from Chaitin/Briggs:

- start out assuming variables in memory, not registers
  - better for CISCs
- **live range**: set of (perhaps disconnected) **live units**
  - live unit = basic block
- prioritize using size of range, not degree of range
- when blocked during simplification phase,
  allocate ranges highest priority first

Example



The algorithm

while there exists a node with < *k* neighbors:
remove it from the graph; push it on a stack

if all remaining nodes have *k* neighbors, then **blocked**:
pick node with highest priority (spill cost/# of units)
if uncolorable, remove from graph
if colorable, allocate a register
then visit all adjacent nodes
if uncolorable, call **split**

while stack not empty:
pop node from stack
allocate to register

Splitting algorithm

Try to divide up a live range into two pieces

First piece:

- start with a live unit that's colorable
  - prefer one starting with a def
- include successor live units as long as
  range remains colorable
  - breadth-first traversal

Second piece:

- everything else

When done, adjust unconstrained stack, priorities

Performance

~ 30% faster, vs. no register allocation

Vs. Chaitin?