Register Allocation

The problem:

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

Constraint:

simultaneously live data allocated to different locations

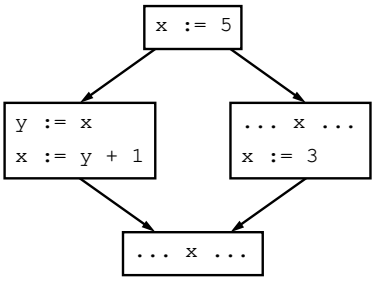
Goal:

minimize overhead of stack loads & stores and register moves

Granularity of allocation

What is allocated to registers/stack locations?

- variables?
- values?
  - same as variables after SSA form + copy propagation
- separate def/use chains (**live ranges**)?



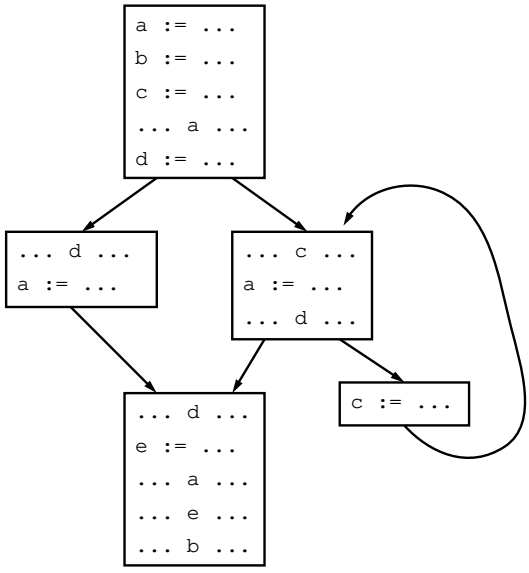
Interference graph

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

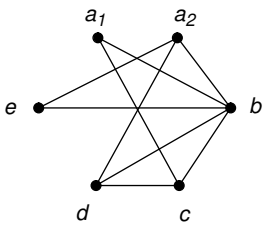
Simple greedy allocation algorithm

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

Example



Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

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

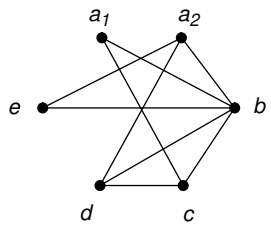
The algorithm

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

Example

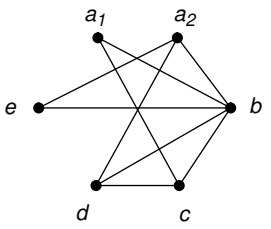


Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Example



Weight Order:

c
d
a_2
b
a_1
e

Assume **2** registers available

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

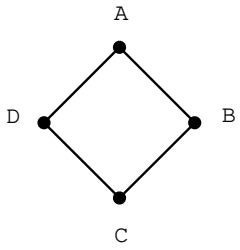
Subsumption

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 ("optimistic spilling")
- 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

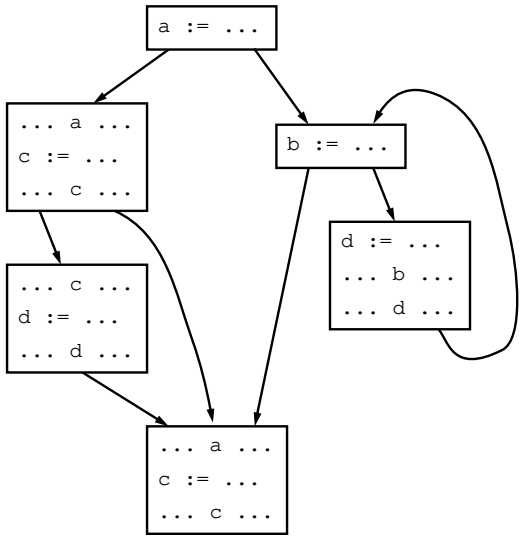
Improvement #3: live range splitting

Priority-Based Coloring [Chow & Hennessy 84]

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

- some live range pieces in registers, some in memory
  ⇒ selective spilling
- insert load, store, register move instructions at boundaries

Example



Improvement #4: rematerialization

Idea: instead of *reloading* value from memory,
recompute it instead,
if recomputation is cheaper than reloading

Easy case:

- can recompute a value in a single instruction (or so)
  and all operands will always be available

Examples:

- constants
- address of global var
- address of var in stack frame

Performance results

[Briggs *et al.* 94]

E.g.

For some procedure:

XXX spill instructions before
YYY spill instructions after

YYY is Z% smaller than XXX

- Z ranges between -2% and 48% for "optimistic spilling"
- Z ranges between -26% and 33% for rematerialization

What's the conclusion?