Interprocedural 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

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 iff n_1 and n_2 are simultaneously live at some program point
- symmetric, not reflexive, not transitive

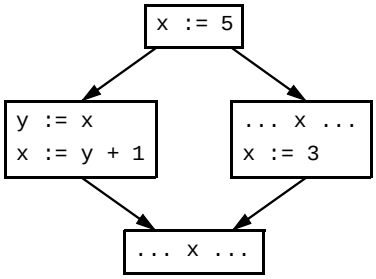
Two adjacent nodes must be allocated to distinct locations

Units of allocation

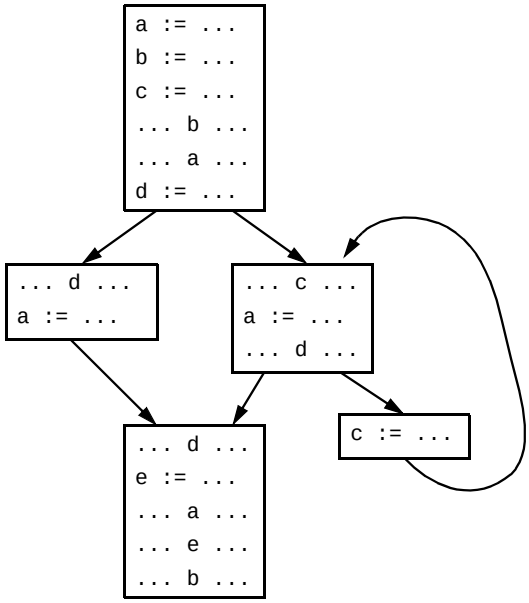
What are the units of allocation?

- option 1: variables
- option 2: distinct connected def/use chains (**live ranges**)

Example:



A bigger example



Computing interference graph

Construct interference graph as side-effect of
live variables analysis

- easy if variables are units of allocation

Construct incrementally as live vars sets modified

- when add a new var to live vars set,
  create edge from new var to all existing vars
- when merge two live vars sets,
  add one sets' vars to other set

Allocating registers using interference graph

Register allocation via graph coloring:
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

- need algorithms + heuristics
  to do a decent job in reasonable time

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 (nodes with highest out-degree)

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 and highest out-degree,
if forced to spill

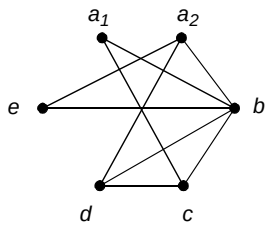
A simple greedy allocation algorithm

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 scratch 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, but still 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*

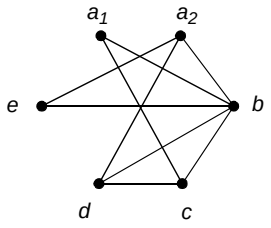
The algorithm

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 node with lowest weight/degree to spill
        remove node from graph
        push it on the stack

while stack not empty:
    pop node from stack
    put back in graph
    if possible, allocate to register different from all its neighbors
        otherwise, allocate to stack

Example

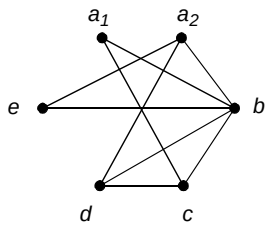


Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Example

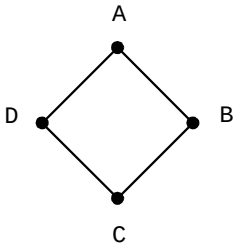


Weight Order:

c
d
a_2
b
a_1
e

Assume **2** registers available

Another example



Assume 2 registers available

Coalescing and preference hints

When generating code for copy statement like `x = y`,
if x and y were assigned same register,
then skip generating a move instruction

If register allocator sees `x = y`,
and x & y are not simultaneously live,
then it should **prefer** to assign x and y to same register

One implementation strategy:
coalesce x and y into same unit of allocation
(similar effect as copy propagation)

\+ avoids generating code for simple copies

– can cause more spilling

Another strategy: add preference hints that two things be allocated to same register

- can assign costs to (violating) preferences
- when picking a register,
  favor most preferred available register

Live range splitting

If a long live range cannot be allocated a register,
can split it into multiple separate live ranges,
linked by copy statements

- can allocate each separate piece separately
- since each piece is shorted, each may interfere with fewer things, and so be more allocatable

The reverse of coalescing

Pretty tricky to decide where to split, where to coalesce, etc. to come up with good overall allocations

Handling calling conventions

How should register allocator deal w/ calling conventions?

Simple: calling-convention-oblivious register allocation

- spill all live caller-save registers before call, restore after call
- save all callee-save registers at entry, restore at return

Better: calling-convention-aware register allocation

- add preferred registers for formals, actuals, results
- variables live across a call interfere with caller-save regs
  - allocator knows to avoid these registers,
    save/restore code turns into normal spills
  - live range splitting for before/during/after call could be good
- procedure entry "assigns" to all callee-save registers,
  procedure exit "reads" all callee-save registers
  - simultaneously live with all variables in procedure
    $\Rightarrow$ allocator knows must spill these registers if used

Gives limited form of interprocedural register allocation

- leaf routines (try to) use only caller-save registers
- routines with calls use callee-save registers for
  variables live across calls