Improvement #3: live range splitting

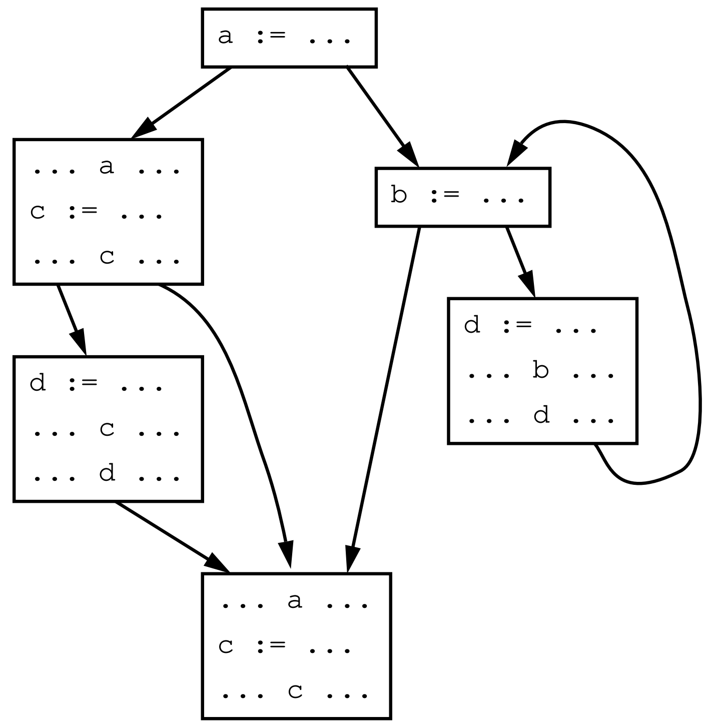
Priority-Based Coloring [Chow & Hennessy 90]

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?

Calling conventions

[Johnson & Ritchie 81]

Goals:

- fast calls
- language-independent rules
- works with debugger, profiler, etc.
  - need to find contents registers for particular stack frame

Problematic language features:

- varargs
- passing/returning aggregates
- returning multiple values
- non-LIFO procedure activations (closures, continuations)

Impact on register allocator

Calls impact register allocation

- imply desired location of args, results
- require register saving/restoring overhead

Allocation preferences due to calls

Calls affect desired location of variables

- want args to calls to get into registers/stack locations assigned by calling convention
  - result is similar
- want to use formals in place where they enter procedure
  - result is similar

How to incorporate preferences into allocation process?

Callee-save vs. caller-save registers

Need a convention at calls for which registers managed by caller (**caller-save**) and which managed by callee (**callee-save**)

- SPARC has **hardware-save** registers, too

Caller-save:

- caller must save/restore any caller-save registers live across calls
- callee is free to use these registers

Callee-save:

- callee must save/restore any callee-save registers it uses
- caller is free to use these registers

Hardware-save:

- caller and callee can use freely

Exploiting calling convention

Register allocator can customize its usage to use "cheaper" registers

- leaf routines use only caller-save registers
- routines with calls use callee-save registers for variables live across calls

Poor man's interprocedural register allocation

Problems with callee-save registers

Run-time utilities (e.g. longjmp) and programming environment tools (e.g. debugger) need to be able to find contents of registers relative to a particular stack frame

Caller-save registers are on stack in stack frame

Callee-save registers?

Modelling caller-/callee-save register status

Treat call as a kill of all caller-save registers

- can be a pre-colored node in interference graph
- allocator knows to avoid these registers, inserts save/restore as normal spills

Treat procedure entry as a def of all callee-save registers, procedure exit as a use of all callee-save registers

- allocator knows must spill these registers if used in proc

Save/restore code inserted, at cheapest place

- works best if calls can split live ranges