Subroutine optimization

- most subroutine optimizations rely on some form of dataflow analysis having been performed first
- will revisit dataflow analysis in more detail later
- some key useful sets:
- liveout(b): variables live on exit from block b, roughly the variables whose values are set in/before b and get used after b (not getting overwritten first)
- uevar(b): upward-exposed variables of block b (b uses these variables' values as they were set by some previous block)
- varkill(b): the variables whose values b sets at some point (i.e. killing their old values)

liveout(b) for basic blocks

- the liveout set for a block will be used repeatedly in our optimizations
- algorithm to compute liveout(b):

```
initialize liveout(b) to { }
for each successor, m, of block b
   add each variable of uevar(m) to liveout(b)
   for each variable, v, in liveout(m)
       if v is not in varkill(m) then add it to liveout(b)
```

computing liveout for subroutines

- use our old algorithm to identify the blocks within the subroutine
  - two passes: all labelled entry instructions as entry points, then second pass to identify exit points
- compute uevar(b) and varkill(b) for individual blocks
  - one pass: anything set gets added to varkill, anything used beforehand gets added to uevar
- now keep repeating liveout(b) computations across all blocks, until results stabilize (see next slide)

compute all liveout(b)'s

```
for each block, b: set liveout(b) to { }
changed=true
while (changed)
   changed=false
   for each block, b:
       recompute liveout(b) // for just the block itself
       if new result for b differs from old
          then changed=true
// uevar(b) and varkill(b) don't ever change
// before first pass each liveout starts as { }
// each pass some liveouts may have changed,
// so the following pass their predecessor liveouts can change
```

liveout implications

- hypothetically, anything in liveout(n) might be used uninitialized in b, since we don't know if it was correctly set in some previous block/subroutine
- in fact, that's overly conservative: sometimes as programmers we can see logically that uninitialized use is impossible
- the compiler has to take a conservative approach when generating warnings

Example: conservative warning

```
int f(int i) {
    if ((i % 2) == 0) {
        return 10;
    } else if ((i % 2) == 1) {
        return 20;
    }
}
```

- complains that you're missing a return
- we can tell that logically it is fine, one of the two branches will always run

Uses of liveout

- can be used for subroutine-level register allocation (only live values need allocation)
- can be used for SSA construction (skipping steps for values in blocks where they're not live)
- eliminating needless stores to memory (if not live then the value is not needed further)

Code placement of blocks

- the order in which blocks are stored in a subroutine (in the executable) can impact memory performance (paging/caching) and instruction caching
- we want to identify blocks that are frequently used in sequence, and store them together in the executable
- this is more important for commonly used sequences of blocks than for rarely used sequences
- will identify paths of blocks, associate frequencies with each (how often is that sequence used)

Hot paths

- hot paths are those most frequently used
- often identified by profiling the code then optimizing further
  - run gprof or other utilities
  - store results where compiler can make use of it
  - re-compile for optimization
- will determine a good code layout in two steps:
  - finding hot paths
  - adjusting code layout

Building paths

- start with each block is just a path by itself, with a infinite weight (representing frequency of use)
- keep joining paths together pairwise using edges from the control flow graph, start with heaviest weighted edges
- new path's weight is minimum of weights of the joined paths and the weight of the edge
  - e.g. paths p1 weight 5, p2 weight 7
  - edge e with weight 4 connects end of p1 to front of p2
  - can connect p1,p2 together into a single path
  - new path weight is min(5,7,4)

Path-building cont.

- algorithm keeps making passes through the edge list until there is no change in the set of paths (i.e. can no longer find a p1,p2,e combo we can combine)
- at end of algorithm we have the blocks divided across a collection of paths, and each path has a weight
- now can use the possible paths to determine the order we should use for blocks in the subroutine's code segment of the executable

Layout algorithm

```
collection = { path-containing-routine's-start-block }
while collection not empty
   take out lowest priority path, p
   for each block, b, on path p
       place b at the end of the remaining code space
   for each block, b, on path p
       for each edge (b, n) where n is not yet placed
           if a path using (b,n) has not yet been placed
              in the collection, then add it to collection
```