

Register Allocation

	live in:	live-out
g:=mem[j+12]	j, k	g, j, k
h:=k-1	g, j, k	h, g, j
f:=g*h	h, g, j	f, j
e:=mem[j+8]	f, j	e, f, j
m:=mem[j+16]	e, f, j	m, e, f
b:=mem[f]	m, e, f	b, e, m
c:=e+8	b, e, m	c, b, m
d:=c	c, b, m	d, b, m
k:=m+4	d, b, m	k, d, b
j:=b	k, d, b	j, d, k

Build

- Build interference graph
 - look at live ranges
 - for each $a \leftarrow b1 \dots bm$ put an edge for all live ranges in live-out
 - for move instructions $a \leftarrow c$, a and c are move-related, but do not add an edge in the graph

Simplify

- we want to K-color graph
 - if a node v has $< K$ neighbors, and all of its neighbors have assigned colors, then we can color node v as well
 - nodes with $< K$ edges – insignificant nodes
 - nodes with $\geq K$ edges – significant nodes
- Simplify:
 - remove all insignificant nodes from G
 - color the graph G'
 - add the insignificant nodes back and assign colors in the process

Note: each time you remove an insignificant node v , you may decrease the degree on remaining nodes in the graph

=> possibilities for further simplification

- maintain a stack of nodes removed from graph, then start to pop nodes of the stack
 - (you can always color insignificant nodes placed on the stack)
- you can stop removing nodes from graph, i.e. simplifying when there are $\leq K$ nodes remaining
 - just give each node a different color

Spill

- What if we cannot simplify graph to G' with $\leq K$ nodes (because all remaining nodes are significant)
 - we probably don't have enough registers => have to put something in memory
- Pick a node v and "spill" it!

Actual spill:

- if you really need to spill a node, need to re-write code to include load/store of value to memory location

Potential spill:

- maybe you will still be able to color the graph, even if you had to choose a victim to spill – (better!)
- Delay spilling a rewriting code.
- Pick a vertex to spill, but place it on stack, and hope that when you pop it back, you'll be able to assign it a color
- If not => have to spill! have to rewrite code and start over!!!

Steps:

- Build
- Simplify
- Choose Spill
- Select
- ? Really Spill – if no ReBuild
- Done

Coalesce

- move $a \leftarrow c$: if a and c don't interfere, move can be eliminated and a and c assigned to the same register
 - in general, any two non-interfering nodes can be coalesced
 - edges of new node are union of edges of original nodes =>
 - better to be conservative and not apply this always, as may end up with graph not colorable with K colors
 - conservative => moves are better than spills
- find conservative coalescing strategy
 - stick to move-related nodes
 - pick strategy so that resulting graph is still colorable with K colors

Coalescing strategies:

Briggs:

- coalesce a and b if ab has $< K$ neighbors of significant degree
- proof: if all insignificant nodes removed, and ab removed, remaining graph will have less than K neighbors and will be colorable

Coalescing strategies:

George:

- coalesce a and b if for every neighbor t of a , either t already interferes with b , or t is insignificant
- proof:
original $G \rightarrow G1$ after insign. nodes deleted.
after coalescing, and removal of nodes $\rightarrow G2$.
 a 's insignificant neighbors t will be removed in both cases.
plus, some t 's neighbors of both a and b may be removed from $G2$.
 $G2$ will be subset of $G1 \Rightarrow$ at least as easily colorable as $G1 \Rightarrow$ strategy is ok

Steps:

- Build
 - Mark move-related or non-move related
- Simplify
 - Remove insignificant non-move related nodes
- Coalesce
 - Remove constrained move edges
 - (interference(x,y), move(x,z), move(y,z))
- Freeze: move-related node of low degree
 - this will be separate register, don't need to keep track of move for it
- Choose Spill
- Select
- ? Really Spill – if no ReBuild
- Done

Spilling cont'd

- Rewrite when actual spill detected:
 - rewrite the code to incorporate the spilling of t :
 - replace t with a new temp t' and store: $n(\$fp) := t'$ (where n is a new frame slot allocated using `allocLocal`)
 - at each node using t , replace t with a new temp t'' and load: $t'' := n(\$fp)$
 - Note that t' and t'' will be live-out for only one instruction.
 - redo liveness analysis, construction of interference graph, and coloring using the rewritten code

Improvements

- Split the live range if possible to avoid spilling
 - Consider choice of color based on move-related nodes: if `move(x,y)`, pick
 - $color_x = color_y$ if y colored; or
 - $color_x \neq color_y$ 's_neighbors if y not colored
 - coalescing does this
 - Spilling adds registers to stack frame; introduces new temporaries which can further cause spilling
 - can we limit/reuse the stack frames?
- Can have stack-slot coloring algorithm, based on liveness info => can reuse stack slots

Candidates for spilling

What to spill?

- *Spill the least used temp*
 - statically least used (fewest occurrences in the code)
 - dynamically least used (weight occurrences in loops higher)
 - this minimizes runtime cost of spills (number of loads and stores)
- *Spill the temp with the most interferences (largest number of adjacent nodes in interference graph)*
 - this removes the most edges, decreasing likelihood of further spills

- Apple uses combination:
 - spill priority = (uses&defs) / degree
 - uses&defs within loops have larger weight

Precolored Nodes

- Register Allocator should consider interference between temporaries and machine registers
 - special registers which may not be available – FP, SP
 - also instruction requires use of specific register
- Assign colors to machine registers.

- Use precolored nodes in the interference graph
 - cannot simplify, spill, coalesce
 - “infinite” degree
- machine registers/precolored registers interfere among each other
 - can have one of each color only

Caller- & Callee-Save Reg

- Caller-save are dead at CALL
 - interfere with anything live across call
- Callee-save are live across entire call
 - CALL “defines” them
 - are “used” by return of CALL
 - interfere with everything in procedure!

- on Proc entry:

```
t100 := r7

// r7 is available here

r7 := t100
ret
```

- now precolored register r7 has short live-range and can be used in the meantime.
- if necessary, t100 will be spilled
- if possible, will be coalesced, and moves removed

example:

```
int f(int a, int b) {

    int d=0;           r1, r2 - caller-save
    int e=a;           r3 - callee-save
    do (d = d + b;
        e = e - 1;
    } while (e > 0);
    return d;
}
```

RegAlloc Implementation

- Need efficient:
 - get nodes adjacent to X (list per node)
 - used in select, don't need it for precolored nodes
 - are X,Y adjacent – matrix or hash table
 - lists of candidates for simplify, spill, coalesce...
 - efficient implementation for frequent insert/delete
 - which list does X belong to
 - move-related/non-move-related
 - degree
 - color assigned
 - aliases
 - stack
 -

RegAlloc for Trees

- concern is registers at tile roots for source/destination allocation
 - $r3 := r1 \text{ op } r2$ or load/store
 - assume stack/memory for other vars.
 - can have some bottom-up algorithm jointly with instruction selection/MaxMunch
 - but....
 - $a + (b + (c + d))$ emit lhs vs rhs first...

- numReg(e)

if e is trivial => 1

if e is unop => $\max\{1, \text{numReg}(e.\text{arg})\}$

if e is binop =>

if numRegs(e.leftOp) == numRegs(e.rightOp)

e.numRegs = 1 + numRegs(e.leftOp)

else

e.numRegs = $\max\{1, \text{regs}(\text{leftOp}), \text{regs}(\text{rightOp})\}$

Sethi-Ulman alg.

- have 1 to R_{avail} regs for E
- if E is fixed reg, like FP, then use it
- else give R_{avail} to E for its result
- now only $R_{\text{avail}} - 1$ regs are available

- looking at numRegs for subtrees needed for SU algorithm
- useful for graph-coloring too – can determine for which tile to emit instruction first (if possible) -> less spills later

