COMPILERS
Register Allocation

hussein suleman
uct csc305w 2004

Register Allocation

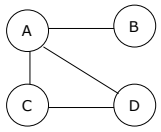
- Want to maximise use of registers for temporaries.
- Build interference graph for each program point
  - compute set of temporaries simultaneously live
  - add edge to graph for each pair in set
- Then K-colour the graph by assigning a maximum of K colours such that interfering variables always have different colours.

Inference Graph

- Start with final live-in and live-out sets from liveness analysis. Add an edge for any simultaneous pairs of nodes.

out	in
A	AB
AB	D
D	ACD
ACD	AC

steady state from liveness analysis



inference graph

Simplify

- Simplify: Colour graph using a simple heuristic
  - suppose G has node m with degree < K
  - if G' = G - {m} can be coloured then so can G, since nodes adjacent to m have at most K - 1 colours
  - each such simplification will reduce degree of remaining nodes leading to more opportunity for simplification
  - leads to recursive colouring algorithm

Spill and Select

- Spill: suppose there are no nodes of degree < K
  - target some node (temporary) for spilling (optimistically, spilling node will allow coloring of remaining nodes)
  - remove and continue simplifying
- Select: assign colours to nodes
  - start with empty graph
  - if adding non-spill node there must be a colour for it as that was the basis for its removal
  - if adding a spill node and no colour available (neighbors already K-coloured) then mark as an actual spill
  - repeat select

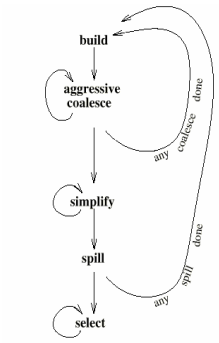
Repeat

- Start over: if select has no actual spills then finished, otherwise
  - rewrite program to fetch actual spills before each use and store after each definition – i.e., convert a spill from a "register temporary" to a "memory temporary"
  - recalculate liveness and repeat

Coalescing

- Can delete a move instruction when source s and destination d do not interfere:
  - coalesce them into a new node whose edges are the union of those of s and d
- In principle, any pair of non-interfering nodes can be coalesced
  - unfortunately, the union is more constrained and new graph may no longer be K-colourable
  - overly aggressive

Aggressive Coalescing



Conservative Coalescing

- Apply tests for coalescing that preserve colourability.
- Suppose a and b are candidates for coalescing into node ab.
- Briggs: coalesce only if ab has < K neighbours of significant degree ≥ K
  - simplify will first remove all insignificant-degree neighbours
  - ab will then be adjacent to < K neighbours
  - simplify can then remove ab

Conservative Coalescing

- George: coalesce only if all significant-degree neighbours of a already interfere with b
  - simplify can remove all insignificant-degree neighbors of a
  - remaining significant-degree neighbours of a already interfere with b so coalescing does not increase the degree of any node

Iterated Register Coalescing 1/2

- Interleave simplification with coalescing to eliminate most moves while without extra spills
  - Build interference graph G; distinguish move-related from non-move-related nodes
  - Simplify: remove non-move-related nodes of low degree one at a time
  - Coalesce: conservatively coalesce move-related nodes
    - remove associated move instruction
    - if resulting node is non-move-related it can now be simplified
    - repeat simplify and coalesce until only significant-degree or uncoalesced moves

Iterated Register Coalescing 2/2

- Freeze: if unable to simplify or coalesce
  - look for move-related node of low-degree and freeze its associated moves (no hope of coalescing them)
  - now treat as a non-move-related and resume iteration of simplify and coalesce
- Spill: if no low-degree nodes
  - select candidate for spilling
  - remove to stack and continue simplifying
- Select: pop stack assigning colours (including actual spills)
- Start over: if select has no actual spills then finished, else
  - rewrite code to fetch actual spills before each use and store after each definition
  - recalculate liveness and repeat

Spilling

- Spills require repeating build and simplify on the whole program.
- To avoid increasing number of spills in future rounds of build can simply discard coalescences.
- Alternatively, preserve coalescences from before first potential spill, discard those after that point.
- Spilled temporaries can be graph-coloured to reuse activation record slots.
  - Coalescing can be aggressive, since (unlike registers) there is no limit on the number of stack-frame locations

Precoloured Nodes

- Precoloured nodes correspond to machine registers (e.g., stack pointer, arguments)
  - select and coalesce can give an ordinary temporary the same colour as a precoloured register, if they don't interfere
    - e.g., argument registers can be reused inside procedures for a temporary
  - simplify, freeze and spill cannot be performed on them
  - precoloured nodes interfere with other precoloured nodes
    - Treat precoloured nodes as having infinite degree - this also avoids large adjacency lists for precoloured nodes

Temporary Copies

- Since precoloured nodes don't spill, their live ranges must be kept short:
  - use move instructions
  - move callee-save registers to fresh temporaries on procedure entry, and back on exit, spilling between as necessary
  - register pressure will spill the fresh temporaries as necessary, otherwise they can be coalesced with their precoloured counterpart and the moves deleted

Caller and callee save registers

- Variables whose live ranges span calls should go to callee-save registers, otherwise to caller-save
- This is easy for graph coloring allocation with spilling
  - calls define/interfere with caller-save registers
  - a cross-call variable interferes with all precoloured caller-save registers, as well as with the fresh temporaries created for callee-save copies, forcing a spill
  - choose nodes with high degree but few uses, to spill the fresh callee-save temporary instead of the cross-call variable
  - this makes the original callee-save register available for colouring the cross-call variable

Example

```
enter:
        c = r3
        a = r1
        b = r2
        d = 0
        e = a
loop:
       d = d + b
       e = e - 1
       if e > 0 goto loop
       r1 = d
       r3 = c
       return [ r1, r3 live out ]
```

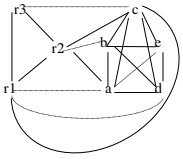
```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do (d = d + b;
      e = e - 1;
     ) while (e > 0) ;
     return d;
}
```

Example

- Temporaries are a,b,c,d,e
- Assume target machine with K = 3 registers:
  - r1, r2 (caller-save/argument/result
  - r3 (callee-save)
- The code generator has already made arrangements to save r3 explicitly by copying into temporary a and back again

Example

- Inference graph



  - No opportunity for simplify or freeze (all non-precoloured nodes have significant degree > K)
  - Any coalesce will produce a new node adjacent to > K significant-degree nodes

Example

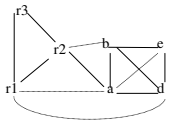
- Must spill based on priorities:

Node	uses + defs outside loop		uses + defs inside loop	degree	priority
a	(2	+ 10 *	0	) / 4	= 0.50
b	(1	+ 10 *	1	) / 4	= 2.75
c	(2	+ 10 *	0	) / 6	= 0.33
d	(2	+ 10 *	2	) / 4	= 5.50
e	(1	+ 10 *	3	) / 3	= 10.30

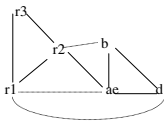
- Node c has lowest priority so spill it

Example

- Inference graph with c removed

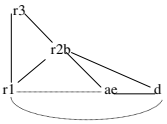


- Only possibility is to coalesce a and e: ae will have fewer than K significant degree neighbors (after coalescing d will be low-degree, though high-degree before)

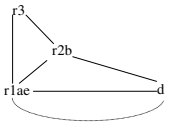


Example

- Can now coalesce b with r2 (or coalesce ae with r1):

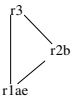


- Coalescing ae with r1 (could coalesce d with r1):



Example

- Cannot coalesce r1ae with d because the move is constrained: the nodes interfere. Must simplify d:



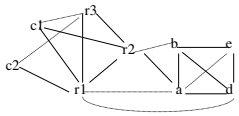
- Graph now has only precoloured nodes, so pop nodes from stack colouring along the way
  - d = r3
  - a,b, and e have colours by coalescing
  - c must spill since no colour can be found for it
- Introduce new temporaries c1 and c2 for each use/def, add loads before each use and stores after each def

Example

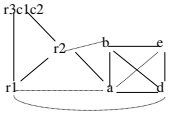
```
enter:
    c1 = r3
    M[c_loc] = c1
    a = r1
    b = r2
    d = 0
    e = a
loop:
    d = d + b
    e = e - 1
    if e > 0 goto loop
    r1 = d
    c2 = M[c_loc]
    r3 = c2
    return [ r1, r3 live out ]
```

Example

- New inference graph

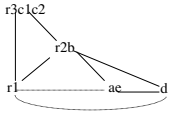


- Coalesce c1 with r3 then c2 with r3



Example

- Coalesce a with e then b with r2:



- Coalesce ae with r1 and simplify d:

Example

- Pop d from stack: select r3. All other nodes were coalesced or precoloured. So, the colouring is:
  - a = r1
  - b = r2
  - c = r3
  - d = r3
  - e = r1

Example

- Rewrite the program with this assignment:

```
enter:
        r3 = r3
        M[c_loc] = r3
        r1 = r1
        r2 = r2
        r3 = 0
        r1 = r1
loop:
        r2 = r3 + r2
        r1 = r1 - 1
        if r1 > 0 goto loop
        r1 = r3
        r3 = M[c_loc]
        r3 = r3
        return [ r1, r3 live out ]
```

Example

- Delete moves with source and destination the same (coalesced):

```
enter:
        M[c_loc] = r3
        r3 = 0
loop:
        r2 = r3 + r2
        r1 = r1 - 1
        if r1 > 0 goto loop
        r1 = r3
        r3 = M[c_loc]
        return [ r1, r3 live out ]
```

- One uncoalesced move remains