Register Allocation

Code generation strategies

- Simple: stack code
  - all variables and user-defined temporaries are allocated on stack
  - use registers as temporaries when evaluating expressions
    - x86: and to return values from function calls
- Better strategy: use registers to reduce loads/stores
  - local variables and user-defined temporaries can be allocated to registers if you have enough registers
    - in our discussion, we will focus on user-defined temporaries
  - also need registers to return values from function calls, to perform some instructions (eg. MUL in x86)
- Approach:
  - generate "abstract" assembly code in which you assume you have an unbounded number of registers
  - perform register allocation
- Reality is a little more complex but this is the high-level idea

Main idea

- Want to replace temporary variables with some fixed set of registers
- First: need to know which variables are live after each instruction
  - Two simultaneously live variables cannot be allocated to the same register

Register allocation

- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if
  - both initially live (ie: function args), or
  - both appear in out[n] for any n
- How to assign registers to variables?

Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can't be next to one another in the graph

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

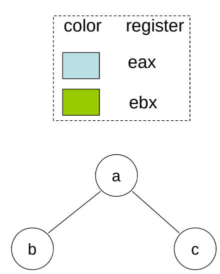
Instructions	Live vars
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
	a
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

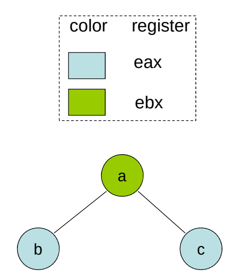
Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	



Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	



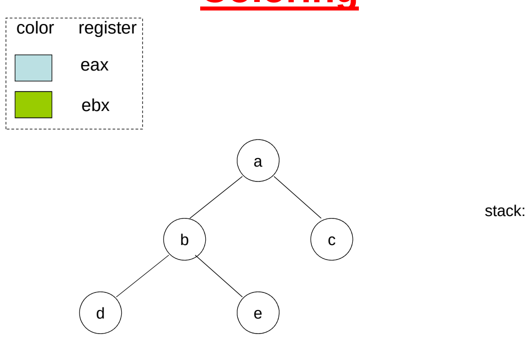
Graph coloring

- Questions:
  - Can we efficiently find a coloring of the graph whenever possible?
  - Can we efficiently find an optimal coloring of the graph?
  - How do we choose registers to avoid move instructions?
  - What do we do when there aren't enough colors (registers) to color the graph?

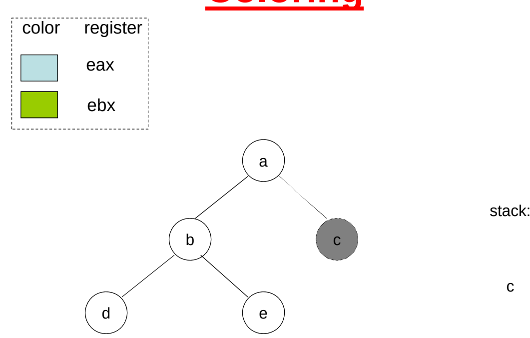
Kempe's heuristic

- Kempe's algorithm [1879] for finding a K-coloring of a graph
- Step 1 (simplify):
  - find a node with at most K-1 edges and cut it out of the graph.
  - remember this node on a stack for later stages.
- Intuition: once a coloring is found for the simpler graph, we can always color the node we saved on the stack
- Step 2 (color): when the simplified graph has been colored, add back the node on the top of the stack and assign it a color not taken by one of the adjacent nodes

Coloring

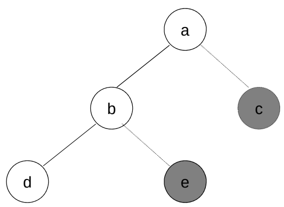


Coloring



Coloring

color	register
	eax
	ebx

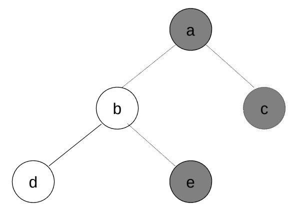


stack:

e
c

Coloring

color	register
	eax
	ebx

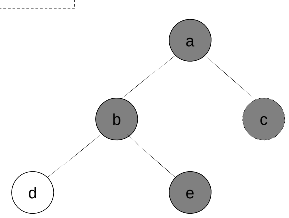


stack:

a
e
c

Coloring

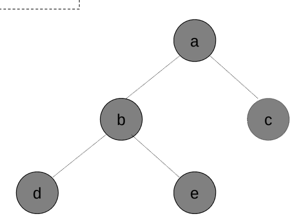
color	register
	eax
	ebx



stack:
b
a
e
c

Coloring

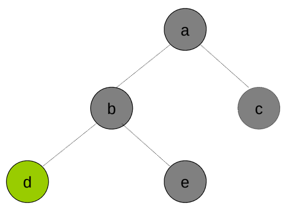
color	register
	eax
	ebx



stack:
d
b
a
e
c

Coloring

color	register
	eax
	ebx

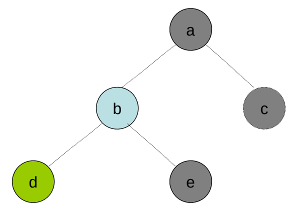


stack:

b
a
e
c

Coloring

color	register
	eax
	ebx

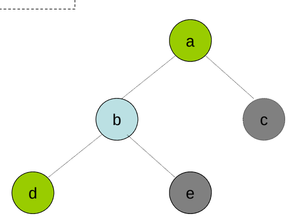


stack:

a
e
c

Coloring

color	register
	eax
	ebx

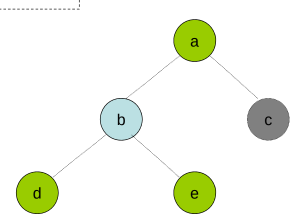


stack:

e
c

Coloring

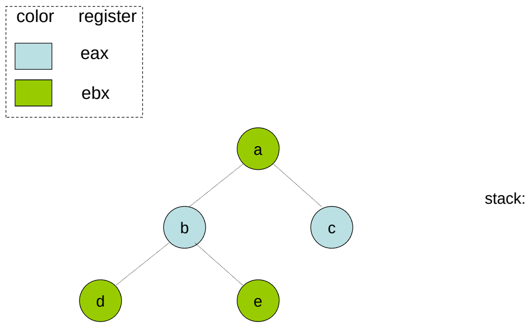
color	register
	eax
	ebx



stack:

c

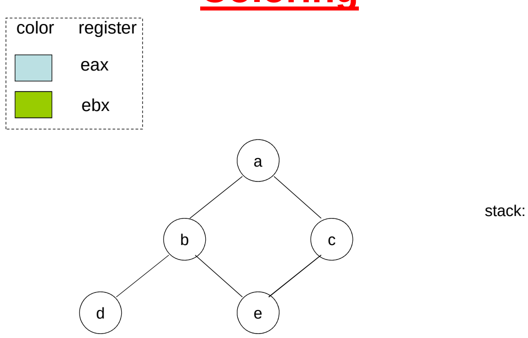
Coloring



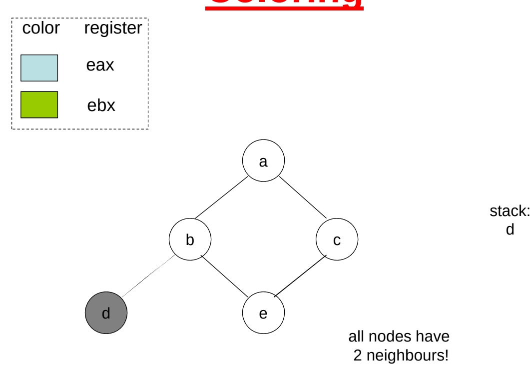
Failure

- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors
- Sometimes, the graph is still K-colorable!
- Finding a K-coloring in all situations is an NP-complete problem
  – We will have to approximate to make register allocators fast enough

Coloring

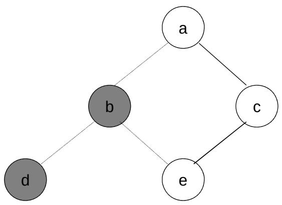


Coloring



Coloring

color	register
	eax
	ebx

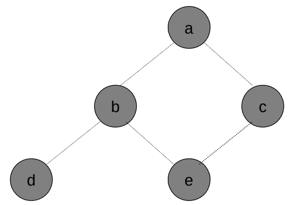


stack:

b
d

Coloring

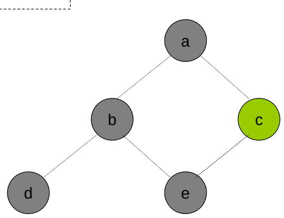
color	register
	eax
	ebx



stack:
c
e
a
b
d

Coloring

color	register
	eax
	ebx

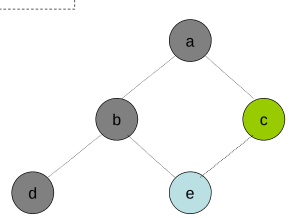


stack:

e
a
b
d

Coloring

color	register
	eax
	ebx

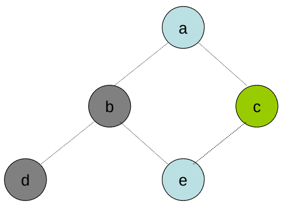


stack:

a
b
d

Coloring

color	register
	eax
	ebx

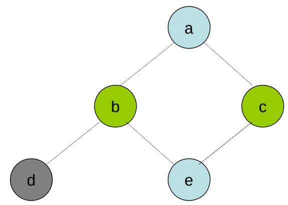


stack:

b
d

Coloring

color	register
	eax
	ebx

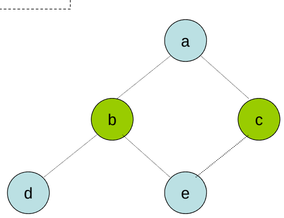


stack:

d

Coloring

color	register
	eax
	ebx



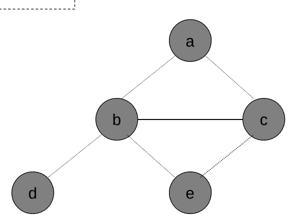
stack:

We got lucky!

Coloring

color	register
	eax
	ebx

Some graphs can't be colored in K colors:



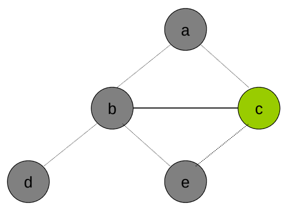
stack:

c
b
e
a
d

Coloring

color	register
	eax
	ebx

Some graphs can't be colored in K colors:



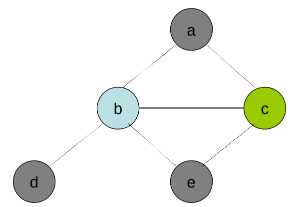
stack:

b
e
a
d

Coloring

color	register
	eax
	ebx

Some graphs can't be colored in K colors:



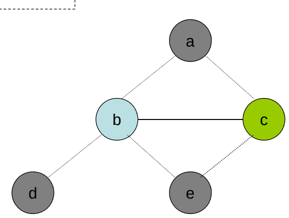
stack:

e
a
d

Coloring

color	register
	eax
	ebx

Some graphs can't be colored in K colors:



stack:

e
a
d

no colors left for e!

Spilling

- Step 3 (spilling): once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack
- There are many heuristics that can be used to pick a node
  - not in an inner loop

Spilling code

- We need to generate extra instructions to load variables from stack and store them
- These instructions use registers themselves. What to do?
  - Stupid approach: always keep extra registers handy for shuffling data in and out
  - Better approach: rewrite code introducing a new temporary; rerun liveness analysis and register allocation
    - Intuition: you were not able to assign a single register to the variable that was spilled but there may be a free register available at each spot where you need to use the value of that variable

Rewriting code

- Consider: add t1 t2
  - Suppose t2 is selected for spilling and assigned to stack location [ebp-24]
  - Invent new temporary t35 for just this instruction and rewrite:
    - mov t35, [ebp – 24];
    - add t1, t35
  - Advantage: t35 has a very short live range and is much less likely to interfere.
  - Rerun the algorithm; fewer variables will spill

Precolored Nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines (trashes) caller-save registers eax, ecx, edx
- Treat these registers as special temporaries; before beginning, add them to the graph with their colors

Precolored Nodes

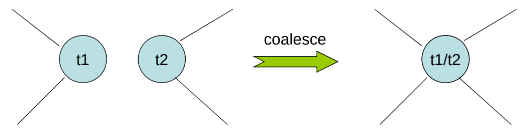
- Can't simplify a graph by removing a precolored node
- Precolored nodes are the starting point of the coloring process
- Once simplified down to colored nodes start adding back the other nodes as before

Optimizing Moves

- Code generation produces a lot of extra move instructions
  - mov t1, t2
  - If we can assign t1 and t2 to the same register, we do not have to execute the mov
  - Idea: if t1 and t2 are not connected in the interference graph, we coalesce into a single variable

Coalescing

- Problem: coalescing can increase the number of interference edges and make a graph uncolorable



- Solution 1 (Briggs): avoid creation of high-degree (>= K) nodes
- Solution 2 (George): a can be coalesced with b if every neighbour t of a:
  - already interferes with b, or
  - has low-degree (< K)

Simplify & Coalesce

- Step 1 (simplify): simplify as much as possible without removing nodes that are the source or destination of a move (move-related nodes)
- Step 2 (coalesce): coalesce move-related nodes provided low-degree node results
- Step 3 (freeze): if neither steps 1 or 2 apply, freeze a move instruction: registers involved are marked not move-related and try step 1 again

Overall Algorithm