

Relevance To Global Register Allocation



- In global register allocation, we want to assign temporary variables to a fixed set of registers.
- To do this, we first need to know which variables are live after each instruction.
 - Two simultaneously live variables cannot be allocated to the same register.
 - Such variables are said to **interfere**.
- For the CFG $G = (V, E)$, for every node (i.e., basic block) $b \in V$, we have computed $Out(b)$, the set of names live out of b .
- Two names m and n interfere if:
 - Both names are initially live (e.g., function arguments), or
 - $\exists b \in V : \{m, n\} \subseteq Out(b)$.
- Given this interference information, how should we assign registers to names?

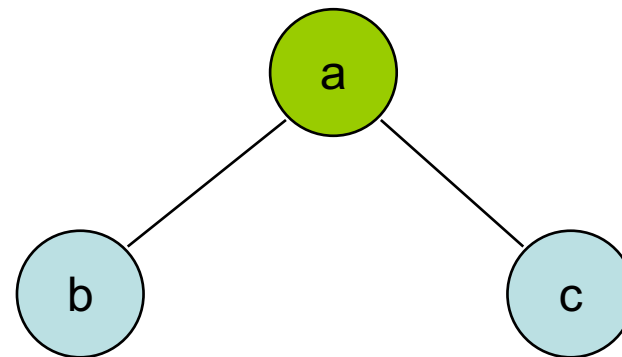
YAG: The Interference Graph

- Given the CFG $G = (V, E)$, its interference graph $I_G = (V', E')$ is defined as follows.
 - V' is the set of variable names in G .
 - $(m, n) \in E'$ iff m and n interfere in G .
- Registers will be assigned to nodes of the interference graph.
 - Registers are thought of as “colors”, hence the link to graph coloring.
 - Two nodes that are adjacent in I_G must be colored with different colors, i.e., allocated to different registers.

The Interference Graph: Simple Example

Instructions	Live variables
	{a}
b = a+2;	
	{a, b}
c = b*b;	
	{a, c}
b = c+1;	
	{a, b}
return b*a;	

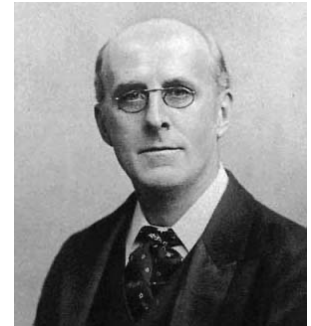
color	register
	%rax
	%rbx



Questions for Graph Coloring Allocator

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

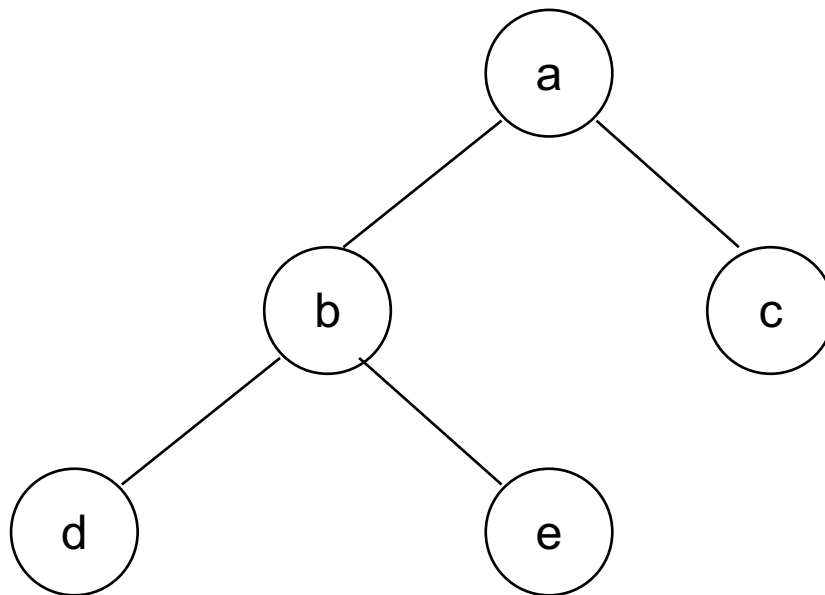
Kempe's Heuristic



- Kempe's algorithm (1879) for K -coloring 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 steps.
 - Intuition: Once a coloring is found for the simpler (and smaller) graph, we can always color the stack we saved on the stack.
 - Failure mode: No such low-degree node exists.
- Step 2 (color):
 - When the simplified graph has been colored, add back the node on the top of the stack and color it a color not taken by any of the adjacent colors.
 - Such a color exists by construction.
- This heuristic was applied by Chaitin in 1981 in the context of register allocation and is known as Chaitin's algorithm in the compiler literature.

Graph Coloring Allocator: Simple Example

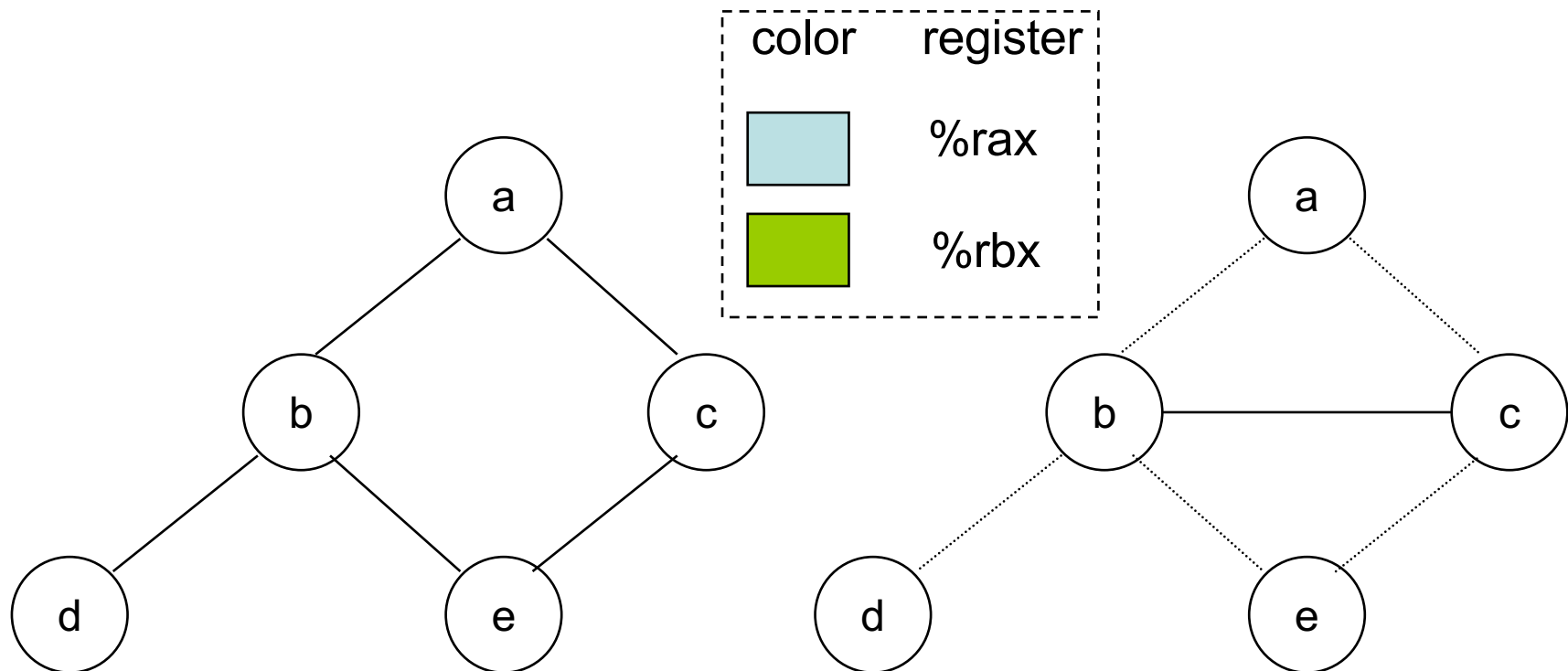
color	register
	%rax
	%rbx



stack:

When Kempe's Heuristic Fails

- The simplification step fails if every node has at least K neighbors.
- Sometimes, this simplified graph may still be K -colorable.
 - Determining K -colorability *in all situations* is an NP-complete problem.
 - So we will need to approximate.



Spilling

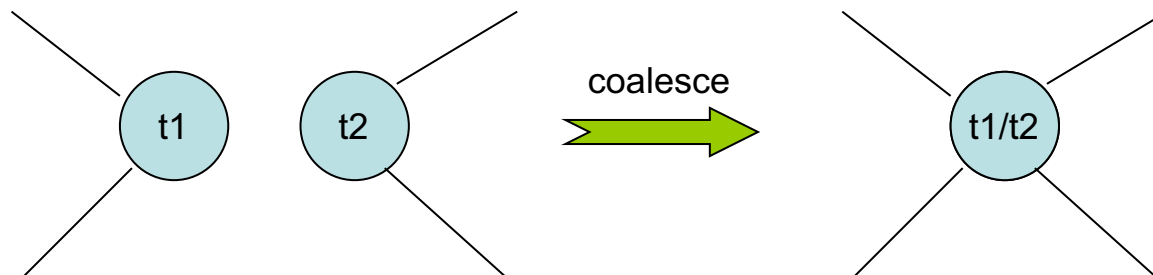
- Step 3 (spill):
 - Once all nodes have K or more neighbors, pick a node to “spill” to the run-time stack.
 - Many heuristics can be used to pick a spill candidate.
 - Chaitin: Spill the variable v with the smallest value of $\text{SpillCost}(v) / \text{iDegree}(v)$.
 - Spill a variable that is not in an inner loop.
- How to spill
 - Need to generate extra instructions to load variables from the stack and to store them. These instructions themselves use registers.
 - Rather than reserving registers for this purpose, simply re-write the code, introducing a new temporary, and then re-run liveness analysis and register allocation. Fewer variables will spill; the process usually converges rapidly.
 - Intuition: You weren’t 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 the variable.

Pre-Coloring

- Some variables are pre-assigned to registers.
 - E.g., method arguments and return values.
- Treat these names as special temporaries. Add them to the interference graph with their colors at initialization time.
- Such a pre-colored node cannot be removed in an attempt to simplify the interference graph.
- Once the interference graph has been simplified down to pre-colored nodes only, start adding back other nodes as before.

Optimizing Moves: Coalescing

- Code generation tends to produce a lot of extra move instructions (generically, think `MOV t1, t2`).
 - If we can allocate t1 and t2 to the same register, then the move becomes unnecessary and can be eliminated.
- Idea
 - If two such move-related nodes t1 and t2 are not connected in the interference graph, **coalesce** them into a single variable.
- Problem
 - Coalescing can increase the degree of the coalesced node and make a graph uncolorable.



Simplify and Coalesce

- Step 1 (simplify):
 - Designate those nodes that are source or destination of a move as **move-related nodes**.
 - Simplify the interference graph as much as possible without removing move-related nodes.
- Step 2 (coalesce):
 - Coalesce move-related nodes provided such coalescing results in low-degree nodes.
 - Multiple possible approaches to control degree of coalesced node:
 - [Briggs] Avoid creation of nodes of degree $\geq K$.
 - [George] Node a can be coalesced with node b if every neighbor t of a already interferes with b or has degree $< K$.
- Step 3 (freeze):
 - If neither steps 1 or 2 apply, freeze a move instruction, i.e., mark the involved nodes as **not move-related**.
 - Try step 1 again.

Structure of Graph Coloring Allocator

