Register Allocation

- Want to replace variables with some fixed set of registers if possible
- Main Idea: cannot allocate two variables to the same register if they are both live at some program point
- Need to know which variables are live at each instruction

Register Allocation

- For every node \( n \) in CFG now have \( \text{out}[n] \): which variables (temporaries) are live on exit from node.
- If two variables are in same live set, can’t be allocated to the same register – they interfere with each other
- How do we assign registers to variables?

Inference Graph

- Nodes of graph: variables
- Edges = variables that interfere with each other
- Register assignment is graph coloring

Graph Coloring

- Questions:
  - Can we efficiently find a coloring of the graph whenever possible?
  - Can we efficiently find the optimum coloring of the graph?
  - How can we choose registers to avoid \textit{mov} instructions?
  - What do we do when there aren’t enough colors (registers) to color the graph?

Coloring a Graph

- Simple algorithm for finding a \( K \)-coloring of a graph: (Assume \( K=3 \))
- \textbf{Step 1}: find some node with at most \( K-1 \) edges and cut it out of graph (simplify)
Simple Algorithm

- Once coloring is found for simplified graph, selected node can be colored using free color
- Step 2: simplify until graph contain no nodes, unwind adding nodes back & assigning colors

Failure of Heuristic

- If graph cannot be colored, it will reduce to a graph in which every node has at least K neighbors
- May happen even if graph is colorable in K
- Finding K-coloring is NP-hard problem (requires search)

Spilling

- Once all nodes have K or more neighbors, pick a node and mark it for spilling (storage on stack). Remove it from graph, continue as before
- Try to pick node not used much, not in inner loop

Optimistic Coloring

- Spilled node may be K-colorable; when assigning colors, try to color it and only spill if necessary.
- If not colorable, record this node as one to be spilled, assign it a stack location and keep coloring

Accessing Spilled Variables

- Need to generate additional instructions to get spilled variables out of stack and back in again
- Naive approach: always keep extra registers handy for shuttling data in and out
- Better approach: rewrite code introducing a new temporary, rerun liveness analysis and register allocation

Rewriting Code

- Example: add t1, t2
- Suppose that t2 is selected for spilling and assigned to stack location [ebp-24]
- Invent new variable t3 for just this instruction, rewrite:
  mov -24(%ebp), t3
  add t3, t1
- Advantage: t3 doesn’t interfere with as much as t2 did. Now rerun algorithm; fewer or no variables will spill.
Precolored Nodes

- Some variables are pre-assigned to registers
- `mul` instruction has
  `use[1] = eax, def[1] = { eax, edx }`
- Call instruction kills caller-save regs:
  `def[1] = { eax, ecx, edx }`
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes

Precolored Nodes

- Can’t simplify graph by removing a precolored node
- Precolored nodes: starting point of coloring process
- Once simplified graph is all colored nodes, add other nodes back in and color them

Optimizing Move Instructions

- Code generation produces a lot of extra mov instructions
  `mov t5, t9`
- If we can assign `t5` and `t9` to same register, we can get rid of the mov
- Idea: if `t5` and `t9` are not connected in interference graph, coalesce them into a single variable. mov will be redundant.

Coalescing

- Problem: coalescing two nodes can make the graph uncolorable
- High-degree nodes can make graph harder to color, even impossible
- Avoid creation of high-degree (>K) nodes (conservative coalescing)

Simplification + Coalescing

- Start by simplifying as much as possible without removing nodes that are either the source or destination of a mov (move-related nodes)
- Coalesce some pair of move-related nodes as long as low-degree node results; delete corresponding mov instruction (K)
- If can neither simplify nor coalesce, take a move-related pair and freeze the mov instruction, do not consider nodes move-related

High-level Algorithm

- Simplify, coalesce, and freeze
  - Spill node if necessary
  - Color graph optimistically
  - Rewrite code if necessary
Summary

- Register allocation pseudo-code in Appel, Chapter 11
- Now have seen all the machinery needed to produce optimized code:
  - Lexer, parser, semantic analysis
  - High IR to low IR
  - Control flow graphs
  - Dataflow analysis
  - Instruction selection
  - Register allocation