Lecture 22: Register Allocation
18 March 2016
Review

• Goal: replace all variables (including temporaries) with some fixed set of registers if possible
• Best case: keep variable in same register for whole lifetime of that variable
  • don’t need to store in memory
  • variable has been allocated to that register
• Register may be used for multiple vars
  • But simultaneously live variables cannot be allocated to same register
  • Therefore: need to know which variables are live at each point in the program (live variable analysis)
Register allocation

- live variable analysis gives us:
- For every node $n$ in CFG, $out[n]$ is which variables (temporaries) are live on exit from node.

  ![Diagram]

- 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 connect all variables that *interfere* with each other
- Register assignment is graph coloring

```plaintext
b = a + 2;
c = b*b;
b = c + 1;
return b*a;
```
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 mov instructions?
  – What do we do when there aren’t enough colors (registers) to color the graph?
Coloring a Graph

- Kempe’s algorithm [1879] for finding a K-coloring of a graph: (Assume K=3)
- Step 1: find a node with at most K–1 edges and cut it out of graph (simplify)
Kempe’s 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-complete problem (requires search)
Spilling

• Once all nodes have K or more neighbors, pick a node and mark it for \textit{spilling} (storage on stack). Remove it from graph, continue as before.

• Heuristic: pick node not used much, not in inner loops.
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. Problem: uses up 3 registers!

• Better approach: rewrite code introducing a new temporary, rerun liveness analysis and register allocation
Rewriting code for spilling

• Assign spilled variables to stack locations
  (can share stack locations across variables that don’t interfere)
• Replace each use/def of a spilled variable with a new fresh temporary.
• Add code around use/def to shuttle data from/to stack location.

Example: t2 is selected for spilling and assigned to stack location -24(%rbp) in this code:

\[
\begin{align*}
\text{addq } t1, & \ t2 \\
... \\
\text{movq } 8(t2), & \ t3
\end{align*}
\]

• Invent temporary variables t90, t91 for just these instructions, rewrite:

\[
\begin{align*}
\text{movq } -24(%rbp), & \ t90 \\
\text{addq } t1, & \ t90 \\
\text{movq } t90, & \ -24(%rbp) \\
... \\
\text{movq } -24(%rbp), & \ t91 \\
\text{movq } 8(t91), & \ t3
\end{align*}
\]

(May be able to avoid introducing new temps via fancy addressing modes)

• Adds nodes in general, but new nodes have fewer edges \(\Rightarrow\) easier to color
**Precolored nodes**

- Some variables are pre-assigned to registers
- `mul` instruction has
  \[ \text{use}(n) = \text{rax}, \text{def}(n) = \{ \text{rax}, \text{rdx} \} \]
- `call` instruction kills caller-save regs:
  \[ \text{def}(n) = \{ \text{rax}, \text{rcx}, \text{rdx}, \ldots \} \]
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as *precolored nodes*
- Defs of precolored nodes interfere with variables live-out at that def
  - prevents assigning a var to, e.g., `rax` across a call.
Caller-save registers

• Interfere with all temporaries live across function calls

• Temporaries not live across function call ⇒ caller-save registers
Simplifying graph with precolored nodes

• Can’t simplify graph by removing a precolored node, so...
• Precolored nodes are starting point of coloring process
• Once simplified graph is all colored nodes, add other nodes back in and color them
Optimizing mov instructions

• Code generation produces a lot of extra mov instructions
  
  \[
  \text{mov } t5, \ t9
  \]

• 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 inference graph, coalesce them into a single variable. mov will be redundant.
Callee-save registers

- Must save if used.
- Idea: copy to temporary at beginning of code, code back at end.
- If register is needed, becomes spill to stack.
- If register is not needed, temporary will be assigned to the same callee-save register, move coalescing removes the save/restore moves.
- Temporaries live across function calls ⇒ callee save registers
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(s)

• 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 acceptable code!
• But still not up to level of reasonably good optimizing compilers
• Next few lectures: optimizations, analyses allowing performance to approach or surpass assembly-coded programs