CS 473: COMPILER DESIGN
REGISTER ALLOCATION
Register Allocation Problem

• Given: an IR program that uses an unbounded number of temporaries

• Find: a mapping from temporaries to machine registers such that
  – the program behaves the same when temps are replaced with registers
  – as many temporaries as possible are in registers
  – moves between registers are minimized
  – architecture requirements are obeyed (e.g., zero register always has the value 0)

• Stack Spilling
  – If there are $k$ registers available and $m > k$ temporaries are live at the same time, then not all of them will fit into registers
  – So we must "spill" the excess temporaries to the stack
Register Allocation by Graph Coloring

• Liveness analysis tells us which variables are live at each point in a CFG

• We can’t assign two variables to the same register if they’re live at the same time

• So we want to assign registers to all variables such that no two variables that are live at the same time get the same register

• This is a graph coloring problem! (where colors are registers)
Interference Graphs

• Nodes of the graph are variables
• Edges connect variables that *interfere* with each other
  – Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph on which they are both live)
• Register assignment is a *graph coloring*
  – A graph coloring assigns each node in the graph a color (register)
  – Any two nodes connected by an edge must have different colors.
• Example:

```
// live = \{a\}
b1 = addi a, 2
// live = \{a, b1\}
c = mul b1, b1
// live = \{a, c\}
b2 = addi c, 1
// live = \{a, b2\}
ans = mul b2, a
// live = \{ans\}
return ans
```
Register Allocation Questions

• How many colors do we have?
  – As many as there are registers

• Can we efficiently find a $k$-coloring of the graph whenever possible?
  – Answer: in general the problem is NP-complete
  – But we can do an efficient approximation using heuristics

• How do we assign registers to colors?
  – If we do this in a smart way, we can eliminate redundant MOV instructions

• What do we do when there aren’t enough colors/registers?
  – We have to use stack space, but how do we do this effectively?
Coloring a Graph: Kempe’s Algorithm

- Algorithm for $k$-coloring a graph by Kempe [1879]
- Recursive algorithm with three steps:
  
  - **Step 1:** Find a node with degree $< k$ and cut it out of the graph
    - Remove the node and all its edges
    - This is called *simplifying* the graph
  
  - **Step 2:** Recursively $k$-color the remaining subgraph
  
  - **Step 3:** When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was $< k$). Color the node with one of those free colors.
Example: 3-coloring a graph

1 and 2: recurse down the simplified graphs
Example: 3-color this Graph

3: assign colors on the way back up
Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least \( k \) neighbors.
  - This can happen even when the graph is \( k \)-colorable!
  - This is a symptom of NP-hardness – we’d need to try every possibility to always get the best answer

- Example: When trying to 3-color this graph:

```
\[ \text{Graph image} \]
```
Questions
Spilling

• Idea: If we can’t $k$-color the graph, we need to store (at least) one temporary variable on the stack.

• Which variable to spill?
  – Pick one that isn’t used very frequently
  – Pick one that isn’t used in a (deeply nested) loop
  – Pick one that has high interference (since removing it will make the graph easier to color)

• In practice: some weighted combination of these criteria

• When coloring:
  – Mark the node as spilled
  – Remove it from the graph
  – Keep recursively coloring
Spilling Example

- If no nodes have degree \(< k\), select a node to spill
- Mark it and remove it from the graph
- Continue coloring
Optimistic Coloring

• Sometimes it is possible to color a node marked for spilling.
  – If we get “lucky” with the choices of colors made earlier.

• Example: When 2-coloring this graph:

• Even though the node was marked for spilling, we can color it.
• So: on the way down, mark for spilling, but don’t actually spill...
**Accessing Spilled Registers**

- If optimistic coloring fails, we need to generate code to move the spilled temporary to and from memory

- **Option 1: Reserve registers specifically for moving to/from memory**
  - Pro: Only need to color the graph once
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2
  - Not good on x86 (especially 32-bit) because there are too few registers & too many constraints on how they can be used.

- **Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory**
  - Pro: Need to reserve fewer registers
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph
Example Spill Code

• Suppose temporary $t$ is marked for spilling to stack slot $[rbp+offs]$

• Rewrite the program like this:

  $$ t = a \text{ op } b $$

  $$ x = t \text{ op } c $$

  $$ y = d \text{ op } t $$

  $$ t = a \text{ op } b \quad // \text{ defn. of } t $$

  $$ \text{move } [rbp+offs], t $$

  $$ \text{move } t_{37}, [rbp+offs] // \text{ use 1 of } t $$

  $$ x = t_{37} \text{ op } c $$

  $$ \text{move } t_{38}, [rbp+offs] // \text{ use 2 of } t $$

  $$ y = d \text{ op } t_{38} $$

• Here, $t_{37}$ and $t_{38}$ are freshly generated temporaries that replace $t$ for different uses of $t$.

• Rewriting the code in this way breaks $t$’s live range up:

  $$ t, t_{37}, t_{38} \text{ are only live across one edge } $$
Questions

Top
Precolored Nodes

• Some variables must be pre-assigned to registers
  – Most processors have a dedicated frame pointer register for fp
  – Most instruction sets reserve certain registers for passing function arguments (a0-a3 in MIPS)
  – Note that we can still assign other variables to these registers, too!

• To properly allocate temporaries, we can treat registers as nodes in the interference graph with pre-assigned colors
  – Pre-colored nodes can’t be removed during simplification, and should never be spilled
  – Trick: Treat pre-colored nodes as having infinite degree in the interference graph (so their degree is always > k)
  – When the graph is empty except the pre-colored nodes, then we start coloring the rest of the nodes.
Picking Good Colors

• When choosing colors during the coloring phase, any choice is semantically correct, but some choices are better for performance.

• In particular, if we have \texttt{move t1, t2} and assign \texttt{t1} and \texttt{t2} to the same register, the move is redundant and can be eliminated
  – Note that \texttt{t1} and \texttt{t2} probably don’t interfere with each other – there’s no reason to keep using both

• A simple color choosing strategy that helps eliminate such moves:
  – Add a new kind of “move-related” edge between the nodes for \texttt{t1} and \texttt{t2} in the interference graph
  – When choosing a color for \texttt{t1} (or \texttt{t2}), if possible pick a color of an already-colored node reachable by a move-related edge
Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a move from one temporary to another.

- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.
Coalescing Interference Graphs

• A more aggressive strategy is to *coalesce* nodes of the interference graph into a single node if they are connected by move-related edges.
  – Coalescing the nodes *forces* the two temporaries to be assigned the same register

![Coalescing Interference Graphs](image)

• Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated

• Problem: coalescing can sometimes increase the degree of a node
Conservative Coalescing

• Two strategies are guaranteed to preserve the $k$-colorability of the interference graph:

• *Brigg’s strategy*: It's safe to coalesce $x$ and $y$ if the resulting node will have fewer than $k$ neighbors (with degree $\geq k$).

• *George’s strategy*: We can safely coalesce $x$ and $y$ if for every neighbor $t$ of $x$, either $t$ already interferes with $y$ or $t$ has degree $< k$. 
Conservative Coalescing

- Two strategies are guaranteed to preserve the \( k \)-colorability of the interference graph:
  
  - **Brigg’s strategy**: It's safe to coalesce \( x \) and \( y \) if the resulting node will have fewer than \( k \) neighbors (with degree \( \geq k \)).

- **George’s strategy**: We can safely coalesce \( x \) and \( y \) if for every neighbor \( t \) of \( x \), either \( t \) already interferes with \( y \) or \( t \) has degree \( < k \).
Two strategies are guaranteed to preserve the $k$-colorability of the interference graph:

- **Brigg’s strategy**: It's safe to coalesce $x$ and $y$ if the resulting node will have fewer than $k$ neighbors (with degree $\geq k$).

- **George’s strategy**: We can safely coalesce $x$ and $y$ if for every neighbor $t$ of $x$, either $t$ already interferes with $y$ or $t$ has degree $< k$. 
Questions
Complete Register Allocation Algorithm

1. Build interference graph, with precolored nodes and move-related edges
2. Reduce the graph (building a stack of nodes to color)
   1. Simplify the graph by removing nodes with degree < $k$ that aren’t move-related; remaining nodes are high degree or move-related
   2. Coalesce move-related nodes using Brigg’s or George’s strategy
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced
   4. If no nodes can be coalesced, freeze (remove) a move-related edge and keep trying to simplify/coalesce
3. If there are non-precolored nodes (with degree $\geq k$) left, mark one for spilling, remove it from the graph, and go back to step 2
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack)
   1. If a node must be spilled, insert spill code and rerun the whole register allocation algorithm starting at step 1
5. After register allocation, the compiler should do an optimization pass to remove redundant moves (like move r1, r1)
Register Allocation: Summary

- Once we have liveness information, we can build an *interference graph* showing which variables are live at the same time
- Coloring the graph (so that no two connected nodes have the same color) with $k$ colors corresponds to allocating the variables to $k$ machine registers
- If we can’t color the graph fully, we have to *spill* a variable to the stack and then try again
- Graph coloring is NP-complete, so we might end up spilling more variables than necessary
- There are a few tricks for getting efficient allocations (move-related edges, coalescing)

- Once we’ve done this to the output of instruction selection, we’ve translated all the way from source language to real assembly!
Questions

Top