CSE P 501 – Compilers

Register Allocation
Hal Perkins
Winter 2008
Agenda

- Register allocation constraints
- Top-down and bottom-up local allocation
- Global allocation – register coloring
k

- Intermediate code typically assumes infinite number of registers
- Real machine has k registers available
- Goals
  - Produce correct code that uses k or fewer registers
  - Minimize added loads and stores
  - Minimize space needed for spilled values
  - Do this efficiently – O(n), O(n log n), maybe O(n^2)
Register Allocation

Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers and memory
  - No additional transformations – scheduling should have done its job
- Minimize inserted code, both dynamically and statically
Allocation vs Assignment

- Allocation: deciding which values to keep in registers
- Assignment: choosing specific registers for values
- Compiler must do both
Basic Blocks

- A *basic block* is a maximal length segment of straight-line code (i.e., no branches)

Significance

- If any statement executes, they all execute
  - Barring exceptions or other unusual circumstances
- Execution totally ordered
- Many techniques for improving basic blocks – simplest and strongest methods
Local Register Allocation

- Transformation on basic blocks
- Produces decent register usage inside a block
  - Need to be careful of inefficiencies at boundaries between blocks
- Global register allocation can do better, but is more complex
Allocation Constraints

- Allocator typically won’t allocate all registers to IR values.
- Generally reserve some minimal set of registers F used only for spilling (i.e., don’t dedicate to a particular value).
Liveness

- A value is *live* between its *definition* and *use*.
  - Find definitions ($x = \ldots$) and uses ($\ldots = \ldots \times \ldots$)
  - Live range is the interval from definition to last use
    - Can represent live range as an interval $[i,j]$ in the block
Top-Down Allocator

- **Idea**
  - Keep busiest values in a dedicated registers
  - Use reserved set, \( F \), for the rest

- **Algorithm**
  - Rank values by number of occurrences
  - Allocate first \( k-F \) values to registers
  - Add code to move other values between reserved registers and memory
Bottom-Up Allocator

- Idea
  - Focus on replacement rather than allocation
  - Keep values used “soon” in registers

- Algorithm
  - Start with empty register set
  - Load on demand
  - When no register available, free one

- Replacement
  - Spill value whose next use is farthest in the future
  - Prefer clean value to dirty value
  - Sound familiar?
Bottom-Up Allocator

- Invented about once per decade
  - Sheldon Best, 1955, for Fortran I
  - Laslo Belady, 1965, for analyzing paging algorithms
  - William Harrison, 1975, ECS compiler work
  - Chris Fraser, 1989, LCC compiler
  - Vincenzo Liberatore, 1997, Rutgers
- Will be reinvented again, no doubt
- Many arguments for optimality of this
Global Register Allocation

- A standard technique is *graph coloring*
- Use control and dataflow graphs to derive *interference graph*
  - Nodes are virtual registers (the infinite set)
  - Edge between \((t_1, t_2)\) when \(t_1\) and \(t_2\) cannot be assigned to the same register
    - Most commonly, \(t_1\) and \(t_2\) are both live at the same time
    - Can also use to express constraints about registers, etc.

- Then color the nodes in the graph
  - Two nodes connected by an edge may not have same color
  - If more than \(k\) colors are needed, insert spill code

- Disclaimer: this works great if there are “enough” registers – not as good on x86 machines
Coloring by Simplification

- Linear-time approximation that generally gives good results
  1. Build: Construct the interference graph
  2. Simplify: Color the graph by repeatedly simplification
  3. Spill: If simplify cannot reduce the graph completely, mark some node for spilling
  4. Select: Assign colors to nodes in the graph
1. Build

- Construct the interference graph using dataflow analysis to compute the set of temporaries simultaneously live at each program point
  - Add an edge in the graph for each pair of temporaries in the set
- Repeat for all program points
2. Simplify

- Heuristic: Assume we have K registers
- Find a node $m$ with fewer than K neighbors
- Remove $m$ from the graph. If the resulting graph can be colored, then so can the original graph (the neighbors of $m$ have at most K-1 colors among them)
- Repeat by removing and pushing on a stack all nodes with degree less than K
  - Each simplification decreases other node degrees – more simplifications possible
3. Spill

- If simplify stops because all nodes have degree $\geq k$, mark some node for spilling
  - This node is in memory during execution
  - $\therefore$ Spilled node no longer interferes with remaining nodes, reducing their degree.
- Continue by removing spilled node and push on the stack (optimistic – hope that spilled node does not interfere with remaining nodes)
4. Select

- Assign nodes to colors in the graph:
  - Start with empty graph
  - Rebuild original graph by repeatedly adding node from top of the stack
    - (When we do this, there must be a color for it)
  - When a potential spill node is popped it may not be colorable (neighbors may have $k$ colors already). This is an actual spill – no color assigned
5. Start Over

- If Select phase cannot color some node (must be a potential spill node), add to the program loads before each use and stores after each definition
  - Creates new temporaries with tiny live ranges

- Repeat from beginning
  - Iterate until Simplify succeeds
  - In practice a couple of iterations are enough
Complications

- Need to deal with irregularities in the register set
  - Some operations require dedicated registers (idiv in x86, split address/data registers in M68k and others)
  - Register conventions like function results, use of registers across calls, etc.
- Model by precoloring nodes, adding constraints in the graph
Coming Attractions

- Dataflow and Control flow analysis
- Overview of optimizations