Register Allocation

Hal Perkins

Autumn 2011
Agenda

- Register allocation constraints
- Local methods
  - Faster compile, slower code, but good enough for lots of things (JITs, ...)
- Global allocation – register coloring
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
      - But we will usually rerun scheduling after this
  - 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
Local Register Allocation

- Apply to basic blocks
- Produces decent register usage inside a block
  - But can have inefficiencies at boundaries between blocks
- Two variations: top-down, bottom-up
Top-down Local Allocation

- Principle: keep most heavily used values in registers
  - Priority = # of times register referenced in block
- If more virtual registers than physical,
  - Reserve some registers for values allocated to memory
    - Need enough to address and load two operands and store result
  - Other registers dedicated to “hot” values
    - (But are tied up for entire block with particular value, even if only needed for part of the block)
Bottom-up Local Allocation (1)

- Keep a list of available registers (initially all registers at beginning of block)
- Scan the code
- Allocate a register when one is needed
- Free register as soon as possible
  - In `x:=y op z`, free `y` and `z` if they are no longer needed before allocating `x`
If no registers are free when one is needed for allocation:

- Look at values assigned to registers – find the one not needed for longest forward stretch in the code
- Insert code to spill the value to memory and insert code to reload it when needed later
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 live ranges (not registers!)
  - Edge between (t1,t2) when t1 and t2 cannot be assigned to the same register
    - Most commonly, t1 and t2 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 (i.e., be allocated to same register)
  - If more than k colors are needed, insert spill code
Live Ranges (1)

- A live range is the set of definitions and uses that are related because they flow together
  - Every definition can reach every use
  - Every use that a definition can reach is in the same live range
Live Ranges (2)

- The idea relies on the notion of *liveness*, but not the same as either the set of variables or set of values
  - Every value is part of some live range, even anonymous temporaries
  - Same name may be part of several different live ranges
Live Ranges: Example

1. loadi ... → rfp
2. loadai rfp, 0 → rw
3. loadi 2 → r2
4. loadai rfp,xoffset → rx
5. loadai rfp,yoffset → ry
6. loadai rfp,zoffset → rz
7. mult rw, r2 → rw
8. mult rw, rx → rw
9. mult rw, ry → rw
10. mult rw, rz → rw
11. storeai rw → rfp, 0

<table>
<thead>
<tr>
<th>Register</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfp</td>
<td>[1,11]</td>
</tr>
<tr>
<td>rw</td>
<td>[2,7]</td>
</tr>
<tr>
<td>rw</td>
<td>[7,8]</td>
</tr>
<tr>
<td>rw</td>
<td>[8,9]</td>
</tr>
<tr>
<td>rw</td>
<td>[9,10]</td>
</tr>
<tr>
<td>rw</td>
<td>[10,11]</td>
</tr>
<tr>
<td>r2</td>
<td>[3,7]</td>
</tr>
<tr>
<td>rx</td>
<td>[4,8]</td>
</tr>
<tr>
<td>ry</td>
<td>[5,9]</td>
</tr>
<tr>
<td>rz</td>
<td>[6,10]</td>
</tr>
</tbody>
</table>
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
- Find live ranges – SSA!
  - Build SSA form of IR
  - Each SSA name is initially a singleton set
  - A $\Phi$-function means form the union of the sets that includes those names (union-find algo.)
  - Resulting sets represent live ranges
  - Either rewrite code to use live range names or keep a mapping between SSA names and live-range names
1. Build

- Use dataflow information to build interference graph
  - Nodes = live ranges
  - Add an edge in the graph for each pair of live ranges that overlap
  - But watch copy operations. MOV ri \rightarrow rj does not create interference between ri, rj since they can be the same register if the ranges do not otherwise interfere
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 – may make more simplifications possible
Example with $k = 3$
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 – Briggs allocator)
3. Spill

- Spill decisions should be based on costs of spilling different values

- Issues
  - Address computation needed for spill
  - Cost of memory operation
  - Estimated execution frequency
    - (e.g., inner loops first)
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 if it didn’t represent a potential spill – pick a different color from any adjacent node)
  - When a potential spill node is popped it may not be colorable (neighbors may have k colors already). This is an actual spill.
Example with $k = 3$

Stack

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
</tbody>
</table>
```
5. Start Over

- If Select phase cannot color some node (must be a potential spill node), add load instructions 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
Coalescing Live Ranges

- Idea: if two live ranges are connected by a copy operation (MOV ri → rj) do not otherwise interfere, then the live ranges can be coalesced (combined)
  - Rewrite all references to rj to use ri
  - Remove the copy instruction
- Then need to fix up interference graph
Advantages?

- Makes the code smaller, faster (no copy operation)
- Shrinks set of live ranges
- Reduces the degree of any live range that interfered with both live ranges \( ri, rj \)
- But: coalescing two live ranges can prevent coalescing of others, so ordering matters
  - Best: Coalesce most frequently executed ranges first (e.g., inner loops)
- Can have a substantial payoff – do it!
Overall Structure

1. Find live ranges
2. Build int. graph
3. Coalesce
4. Spill Costs
5. Find Coloring
6. Insert Spills

More Coalescing Possible

No Spills
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, etc.
Graph Representation

- The interference graph representation drives the time and space requirements for the allocator (& maybe the compiler)
- Not unknown to have $O(5K)$ nodes and $O(1M)$ edges
- Dual representation works best
  - Triangular bit matrix for efficient access to interference information
  - Vector of adjacency vectors for efficient access to node neighbors
And That’s It

- Modulo all the picky details, that is...