CSE P 501 – Compilers

Register Allocation

Hal Perkins

Spring 2018
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 if we insert spill code
  – 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
Bottom-up Local Allocation (2)

• 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
    • If a copy already exists in memory, no need to spill
Local "bottom-up" Register Allocation, -1

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>; load v2 from memory</td>
<td>2.</td>
<td>; load v3 from memory</td>
<td>3.</td>
</tr>
<tr>
<td>4.</td>
<td>; load v5, v6 from memory</td>
<td>5.</td>
<td>v4 = v5 - v6</td>
<td>6.</td>
</tr>
<tr>
<td>7.</td>
<td>; load v9 from memory</td>
<td>8.</td>
<td>v8 = - v9</td>
<td>9.</td>
</tr>
<tr>
<td>10.</td>
<td>v11 = v10 - v3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Still in LIR. So lots (too many!) virtual registers required (v2, etc).
- Grey instructions (1,2,4,7) load operands from memory into virtual registers.
- We will ignore these going forward. Focus on mapping virtual to physical.
Local "bottom-up" Register Allocation, 0

1. \( v_1 = v_2 + v_3 \)
2. \( v_4 = v_5 - v_6 \)
3. \( v_7 = v_2 - 29 \)
4. \( v_8 = -v_9 \)
5. \( v_{10} = v_6 \times v_4 \)
6. \( v_{11} = v_{10} - v_3 \)

<table>
<thead>
<tr>
<th>pReg</th>
<th>vReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>-</td>
</tr>
<tr>
<td>R2</td>
<td>-</td>
</tr>
<tr>
<td>R3</td>
<td>-</td>
</tr>
<tr>
<td>R4</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vReg</th>
<th>NextRef</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>1</td>
</tr>
<tr>
<td>v2</td>
<td>1</td>
</tr>
<tr>
<td>v3</td>
<td>1</td>
</tr>
<tr>
<td>v4</td>
<td>2</td>
</tr>
<tr>
<td>v5</td>
<td>2</td>
</tr>
<tr>
<td>v6</td>
<td>2</td>
</tr>
<tr>
<td>v7</td>
<td>3</td>
</tr>
<tr>
<td>v8</td>
<td>4</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
<tr>
<td>v_{10}</td>
<td>5</td>
</tr>
<tr>
<td>v_{11}</td>
<td>6</td>
</tr>
</tbody>
</table>
Local "bottom-up" Register Allocation, 1

1. \( v1 = v2 + v3 \)
2. \( v4 = v5 - v6 \)
3. \( v7 = v2 - 29 \)
4. \( v8 = - v9 \)
5. \( v10 = v6 \times v4 \)
6. \( v11 = v10 - v3 \)

\[
\begin{array}{c|c}
\text{pReg} & \text{vReg} \\
\hline
R1 & v2 \\
R2 & v3 \\
R3 & v1 \\
R4 & - \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{vReg} & \text{NextRef} \\
\hline
v1 & 1 \infty \\
v2 & 1 3 \\
v3 & 1 6 \\
v4 & 2 \\
v5 & 2 \\
v6 & 2 \\
v7 & 3 \\
v8 & 4 \\
v9 & 4 \\
v10 & 5 \\
v11 & 6 \\
\end{array}
\]

\( R3 = R1 + R2 \)
Local "bottom-up" Register Allocation, 2

1. \( v1 = v2 + v3 \)
2. \( v4 = v5 - v6 \)
3. \( v7 = v2 - 29 \)
4. \( v8 = -v9 \)
5. \( v10 = v6 \times v4 \)
6. \( v11 = v10 - v3 \)

<table>
<thead>
<tr>
<th>pReg</th>
<th>vReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>v2</td>
</tr>
<tr>
<td>R2</td>
<td>v3</td>
</tr>
<tr>
<td>R3</td>
<td>v4</td>
</tr>
<tr>
<td>R4</td>
<td>v5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vReg</th>
<th>NextRef</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>( \infty )</td>
</tr>
<tr>
<td>v2</td>
<td>3</td>
</tr>
<tr>
<td>v3</td>
<td>6</td>
</tr>
<tr>
<td>v4</td>
<td>2 5</td>
</tr>
<tr>
<td>v5</td>
<td>2 ( \infty )</td>
</tr>
<tr>
<td>v6</td>
<td>2 5</td>
</tr>
<tr>
<td>v7</td>
<td>3</td>
</tr>
<tr>
<td>v8</td>
<td>4</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
<tr>
<td>v10</td>
<td>5</td>
</tr>
<tr>
<td>v11</td>
<td>6</td>
</tr>
</tbody>
</table>

\[ R3 = R1 + R2 \]
; spill R3
; spill R2? - no - still clean
\[ R2 = R4 - R3 \]
Local "bottom-up" Register Allocation, 3

1. \( v1 = v2 + v3 \)
2. \( v4 = v5 - v6 \)
3. \( v7 = v2 - 29 \)
4. \( v8 = - v9 \)
5. \( v10 = v6 * v4 \)
6. \( v11 = v10 - v3 \)

<table>
<thead>
<tr>
<th>pReg</th>
<th>vReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>v2</td>
</tr>
<tr>
<td>R2</td>
<td>v4</td>
</tr>
<tr>
<td>R3</td>
<td>v6</td>
</tr>
<tr>
<td>R4</td>
<td>( v5 ) v7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vReg</th>
<th>NextRef</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>( \infty )</td>
</tr>
<tr>
<td>v2</td>
<td>3 ( \infty )</td>
</tr>
<tr>
<td>v3</td>
<td>6</td>
</tr>
<tr>
<td>v4</td>
<td>5</td>
</tr>
<tr>
<td>v5</td>
<td>( \infty )</td>
</tr>
<tr>
<td>v6</td>
<td>5</td>
</tr>
<tr>
<td>v7</td>
<td>3 ( \infty )</td>
</tr>
<tr>
<td>v8</td>
<td>4</td>
</tr>
<tr>
<td>v9</td>
<td>4</td>
</tr>
<tr>
<td>v10</td>
<td>5</td>
</tr>
<tr>
<td>v11</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
R3 &= R1 + R2 \\
\text{; spill } R3 \\
\text{; spill } R2? - \text{ no!} \\
R2 &= R4 - R3 \\
\text{; spill } R4? - \text{ no!} \\
R4 &= R1 - 29
\end{align*}
\]

And so on . . .
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 by Graph Coloring

- How to convert the infinite sequence of temporary data references, t1, t2, ... into assignments to finite number of actual registers
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete ... it is equivalent to chromatic number – minimum colors needed to color nodes of a graph so no edge connects same color
Begin With Data Flow Graph

• procedure-wide register allocation
• only live variables require register storage

**dataflow analysis**: a variable is *live* at node N if *the value* it holds is used on some path further down the control-flow graph; otherwise it is *dead*

• two variables(values) interfere when their live ranges overlap
Live Variable Analysis

```
a := read();
b := read();
c := read();
d := a + b*c;
d < 10
  e := c+8;
  print(c);
if (d < 10 ) then
  e := c+8;
  print(c);
else
  f := 10;
  e := f + d;
  print(f);
fi
print(e);```

```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
  e := c+8;
  print(c);
else
  f := 10;
  e := f + d;
  print(f);
fi
print(e);
```
Register Interference Graph

\[
\begin{align*}
a &:= \text{read();} \\
b &:= \text{read();} \\
c &:= \text{read();} \\
d &:= a + b \times c; \\
d &< 10 \\
\end{align*}
\]

\[
\begin{align*}
e &:= c + 8; \\
\text{print}(c); \\
\end{align*}
\]

\[
\begin{align*}
f &:= 10; \\
e &:= f + d; \\
\text{print}(f); \\
\text{print}(e); \\
\end{align*}
\]
Graph Coloring

• NP complete problem

• Heuristic: color easy nodes last
  – find node N with lowest degree
  – remove N from the graph
  – color the simplified graph
  – set color of N to the first color that is not used by any of N’s neighbors

• Basics due to Chaitin (1982), refined by Briggs (1992)
Apply Heuristic
Apply Heuristic

UW CSE P 501 Spring 2018
Apply Heuristic
Apply Heuristic
Continued
Continued
Continued

Diagram showing connections between nodes labeled a, b, c, d, e, and f.
Continued

![Graph Diagram]

- Vertices: a, b, c, d, e, f
- Edges: (a, b), (b, c), (c, d), (d, e), (e, f)

The graph is transformed by removing edge (c, d) and adding edge (a, b).
Continued

```
a -- b
|   |   |
|   |   |
c    d
|   |   |
|   |   |
e    f
```
Continued

\[
\begin{align*}
&\text{a} \quad \text{b} \\
&\text{c} \quad \text{d} \\
&\text{e} \quad \text{f}
\end{align*}
\quad \rightarrow \quad 
\begin{align*}
&\text{a} \quad \text{b} \\
&\text{c} \quad \text{d} \\
&\text{e} \quad \text{f}
\end{align*}
\]
Continued
Continued

![Graph Diagram]

UW CSE P 501 Spring 2018
Continued
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
    e := c+8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
Some Graph Coloring Issues

• May run out of registers
  – Solution: insert spill code and reallocate

• Special-purpose and dedicated registers
  – Examples: function return register, function argument registers, registers required for particular instructions
  – Solution: “pre-color” some nodes to force allocation to a particular register
Global Register Allocation (for real)

• Graph coloring is the standard technique, but
• Nodes are *live ranges* not variables
• Use control and dataflow (actually SSA) graphs to derive interference graph
  – Edge between (t1,t2) when live ranges 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., cannot allocate 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 algorithm)
  - 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 → 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
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.
5. Start Over

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

• Idea: if two live ranges are connected by a copy operation (MOV ri → rj) but 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
Coalescing 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 \( r_i, r_j \)
• 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

- Find live ranges
- Build int. graph
- Coalesce
- Spill Costs
- Find Coloring
- More Coalescing Possible

- Insert Spills
- No Spills

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 overlap (AH, AL, AX, EAX, RAX) in x86 and x86-64
  – 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 (and 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...