Register allocation

Motivation
- registers much faster than memory
- limited number of physical registers
- keep values in registers as long as possible (minimize number of load/stores executed)

Register allocation
1. for simplicity, assume an infinite set of virtual registers during optimization & code gen.
2. map virtual registers onto finite # of registers
3. assign virtual registers to physical registers

Approaches
- local allocation
  - top-down — assign registers by frequency
  - bottom-up — spill registers by reuse distance
- global allocation
  - top-down — color interference graph
  - bottom-up — split live ranges

Local register allocation

Top-down
- rank virtual registers by # uses in block
- reserve sufficient registers for memory operations
- assign remaining physical registers by rank
- problem — assignment fixed for entire block

Bottom-up
- put physical registers on free list
- for each instruction in order
  - if operand not in register, assign free register
  - if free list empty, reclaim register holding value whose use is furthest in future

Example

<table>
<thead>
<tr>
<th></th>
<th>Top-down</th>
<th>Bottom-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) a = b + c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) c = a + d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3) a = a + c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Global register allocation (top-down)

Register coloring
- map register allocation to graph coloring
- major steps
  1. global data-flow analysis to find live ranges
  2. build and color interference graph
  3. if unable to color, spill registers and repeat

Live ranges

Definition
- all definitions which reach a use, plus all uses reached by these definitions
  \[
  a =
  b =
  = a
  = b
  = b
  = a
  = b
  \]
- a single virtual register name may comprise several live ranges

Example
- a = b + c
- c = a + d
- a = a + c

Live ranges delineate when variables need to be stored in the same physical register to avoid extra code
Interference

Using live ranges, an interference graph is constructed where

- vertices represent live ranges
- edges represent interferences between two live ranges, i.e., both ranges are live at some point and cannot occupy the same register
- a coloring represents a register assignment (one color per register)

Note that the abstraction subtly changes our goals. However, the separation of optimization and allocation justifies the goal of minimal coloring.

Building the interference graph

- at each point \( p \) in the program, add edge \((x, y)\) for all pairs of live ranges \( x, y \) live at \( p \)

Example

\[
\begin{align*}
a &= a \\
b &= a\\
c &= b \\
d &= b
\end{align*}
\]

Simplification Example

Optimistic coloring

What if we have two registers? Coloring algorithm would spill a variable

Optimistic coloring algorithm

1. Repeatedly remove nodes with degree < \( k \) from the graph and push them on a stack.
2. If every remaining node is degree \( \geq k \), select node with lowest spill cost and remove it from the graph.
3. Reassemble the graph with nodes popped from the stack. If node cannot be colored, spill it.

Deferring spill decisions helps when

- neighbors of a node are the same color
- a neighboring node has already been spilled
**Spill Costs**

Which live ranges to spill? Two goals

- try to minimize cost of spill
- try to maximize decrease in interference (reduce need for more spills)

<table>
<thead>
<tr>
<th>Cost functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>degree(v)</td>
<td># of edges for v in interference graph</td>
</tr>
<tr>
<td>depth(I)</td>
<td>loop nesting depth of instruction I</td>
</tr>
<tr>
<td>cost(v)</td>
<td>( \sum_{v \text{ live at instr } I} 1^{\text{depth}(I)} )</td>
</tr>
</tbody>
</table>

Cost estimate heuristics

- \( \text{cost}(v) / \text{degree}(v) \) [Chaitin et al.]
- \( \text{cost}(v) / \text{degree}(v)^2 \)
- etc...

**Allocation with Spilling**

Approach

- apply different cost estimate heuristics, pick best result
- most expense is in building interference graph
- spill cost estimates can be calculated efficiently

Reducing spill code

- value modified – store register to memory (dirty)
- read-only value – reload from memory (clean)
- constant value – recompute value (rematerialize)

Recognizing special cases can reduce need to spill

**Global register allocation (bottom-up)**

Live range splitting

- insert copies to split up live ranges
- hope to reduce spilling
- also controls spill code placement

Coalescing (Subsumption)

- allocate source and destination of copy to same register to eliminate register-to-register copies
- combines live ranges
- can clean up unnecessary splits

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>r1</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>=</td>
<td>r1</td>
</tr>
</tbody>
</table>
Live range splitting

One approach [Chow & Hennessy 90]
1. locally allocate registers for each basic block
2. prioritize live ranges by estimated spill cost
3. allocate registers to live ranges
4. split live range if no colors available

Enhancement examples

<table>
<thead>
<tr>
<th>Original</th>
<th>Chaitin</th>
<th>Splitting</th>
<th>Rematerialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \leftarrow f() )</td>
<td>( p \leftarrow f() )</td>
<td>( p \leftarrow f() )</td>
<td>( p \leftarrow f() )</td>
</tr>
<tr>
<td>( y \leftarrow y + [p] )</td>
<td>( y \leftarrow y + [p] )</td>
<td>( y \leftarrow y + [p] )</td>
<td>( y \leftarrow y + [p] )</td>
</tr>
<tr>
<td>...regs...</td>
<td>...regs...</td>
<td>...regs...</td>
<td>...regs...</td>
</tr>
<tr>
<td>( p \leftarrow p + 1 )</td>
<td>( p \leftarrow f() )</td>
<td>( p \leftarrow f() )</td>
<td>( p \leftarrow f() )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Combining Scheduling and Allocation

Allocation before scheduling
- register assignment introduces dependences
- reduces freedom of scheduler
- example
  
  \[
  \begin{align*}
  \text{load vr1,a} & \quad \text{load r1,a} \\
  \text{load vr1,a} & \quad \text{load r1,a} \\
  \text{vr3 = vr1} & \quad \text{r3 = r1} \\
  \text{load vr2,b} & \quad \text{load r1,b} \quad \text{r3 = r1} \\
  \text{vr4 = vr2} & \quad \text{r4 = r1} \quad \text{r4 = r2}
  \end{align*}
  \]

Scheduling before allocation
- lengthens live range of virtual registers
- increases register pressure, causes spills
- need to schedule spill code after allocation
- example
  
  \[
  \begin{align*}
  \text{load vr1,a} & \quad \text{load vr1,a} \\
  \text{load vr2,b} & \quad \text{load vr3,c} \\
  \text{vr4 = vr1} & \quad \text{load vr1,b} \quad \text{vr5 = vr1} \\
  \text{load vr2,b} & \quad \text{load vr3,c} \quad \text{vr6 = vr2} \\
  \text{vr6 = vr3} & \quad \text{vr6 = vr3}
  \end{align*}
  \]
Combining Scheduling and Allocation

We see that instruction scheduling and register allocation are interdependent. Some possible solutions to problem are:

Assigning registers
- first-fit — lowest number available register (reduces total number of registers assigned)
- round-robin — cycle through all registers (reduces memory-related dependences)

Change ordering
- postpass – allocate then schedule
- prepass – schedule then allocate
- multipass – schedule, allocate, then schedule

Integrated prepass scheduling
- schedule instructions first as prepass
- bias schedule to reduce local register pressure
- allocate registers after scheduling