Register allocation

**Motivation**
- Registers must last longer than memory
  - Limited number of physical registers
  - Keep values in registers as long as possible

**Register allocation**
- Bottom-up — split live ranges
- Top-down — color interference graph
- Global allocation
  - Top-down — spill registers by distance
  - Bottom-up — assign registers by frequency
  - Local allocation

**Approaches**
- Local allocation
  - Top-down — assign virtual registers by frequency
  - Bottom-up — assign registers by reuse distance
- Global allocation
  - Top-down — use color interference graph
  - Bottom-up — spill registers by reuse distance

**Example**

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

**Local register allocation**
- Problem — assignment fixed for entire block
- Solve simultaneously physical and virtual registers
- Assign virtual registers by rank
- Reserve sufficient registers for memory
- Rank virtual registers by number of uses in block
- Top-down
Global register allocation (top-down)

Register coloring

Example

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

Building the interference graph

Allocation justifies the goal of minimal coloring. However, the separation of optimization and allocation changes our goals.

Building the interference graph

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

Several live ranges

A single virtual register name may conflict

\( a = b = a \)
\( a = b = b \)
\( a = b = a \)
\( a = b = b \)

Note that the abstraction slightly changes our goals.

However, the separation of optimization and allocation changes our goals.

A coloring represents a register assignment

A coloring represents a register assignment

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)

\( a = b = a \)
\( a = b = b \)
Graph coloring

Given a graph, assign a color to each node such that no neighbors have the same color.

Determining whether a graph may be colored with a given number of colors is NP-hard for $k \geq 3$. Register coloring determines whether a graph may be colored with $k$ registers, and assigns colors to each node.

Chaitin et al.'s algorithm [1981]

1. Repeatedly remove nodes with degree $< k$ from the graph and push them on a stack.
2. If every remaining node is degree $\geq k$, spill node with lowest spill cost and remove it from the graph.
3. Reassemble the graph with nodes popped from the stack.

Given $k$ colors, determine the number of available registers $\leq k$.

Register coloring

Given a graph, find a legal coloring given $k$ colors.
Optimistic coloring

What if we have two registers? Christen et al.'s algorithm would spill a variable.

Briggs et al.'s algorithm [1989]

1. Repeatedly remove nodes with degree < \( r \) from the stack.
2. If every remaining node is degree \( \leq r \), select the graph, and push them on a stack.
3. Reassemble the graph with nodes popped from the stack, use/de-use order live range

\begin{align*}
\text{Effects} & \quad \text{Reduces chance of interference} \\
I_1 &= \text{load} \quad I_1 = a \\
I_2 &= \text{store} \quad I_2 = a \\
I_1 &= \text{load} \quad I_1 = b \\
I_2 &= \text{store} \quad I_2 = b \\
& \quad \text{insert load (before use) and store (after def)}
\end{align*}

Spilling

Spill code

\begin{align*}
\text{Case 1:30} & \quad \text{Constant value – compute value (temeratizing)} \\
& \quad \text{Read-only value – read from memory (clean)} \\
& \quad \text{Value modified – store register to memory (dirty)} \\
\text{Reducing spill code}
\end{align*}

Deferring spill decisions helps when neighbors of a node are the same color.

Reduced chance of interference.

Dbeg's et al.'s algorithm [1989]
Estimating Spill Costs

Which live ranges to spill? Two goals

Try to minimize cost of spill

Try to maximize decrease in interference

Live Range Splitting

Global Register Allocation (bottom-up)

Cost Estimate Heuristics

H1: cost \( v \) / degree \( v \)

H0: cost \( v \) / degree \( v \)

Approach

Apply different spilling heuristics' pick best

Cost functions

Reduce need for more spills

Try to maximize decrease in interference

Try to minimize cost of spill

Global Register Allocation (bottom-up)

Approach

Apply different spilling heuristics' pick best

Cost functions

Reduce need for more spills

Try to maximize decrease in interference

Try to minimize cost of spill

Global Register Allocation (bottom-up)
Live range splitting

One approach

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>spill $p$</td>
<td>spill $p$</td>
<td></td>
<td></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>reload $p$</td>
<td>reload $p$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p \leftarrow f()$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p \leftarrow p + 1$</td>
<td>$p \leftarrow p + 1$</td>
<td>$p \leftarrow p + 1$</td>
<td>$p \leftarrow p + 1$</td>
</tr>
<tr>
<td>spill $p$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Combining Scheduling and Allocation

Alloca tion before scheduling

Allocate registers after scheduling

This schedule to reduce local register pressure
Schedule instructions first as passes

Integrated press scheduling

Multi-pass - schedule, allocate, then schedule
Pre-pass - schedule, allocate
Post-pass - allocate, then schedule
Change ordering

Post-pass scheduling

Pre-pass scheduling

Multi-pass scheduling

Integrated postpass scheduling

What can we do?

We see that instruction scheduling and register allocation are interdependent

Allocation before scheduling

Integrated prepass scheduling

Bias scheduling to reduce local register pressure
Allocate registers after scheduling

Example

Load r1, a
Load r2, b
Load r3, c
Load r3, d
Load r3, e
Load r1, f
Load r1, g
Load r2, h
Load r3, i
Load r4, j
Load r1, k

C MSC 4/3/0
Lecture 2/0, Page 3/0

C MSC 4/3/0
Lecture 3/0, Page 1/3