Register Allocation

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 / store statements executed

Register allocation & assignment

- For simplicity
  - Assume infinite number of virtual registers
- Decide which values to keep in finite # of virtual registers
- Assign virtual registers to physical registers

Critical properties

- Produce correct code that uses k (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold spilled values
- Operate efficiently
  - O(n), O(n log n), maybe O(n^2), but not O(2^n)

Register Allocation Approaches

Local allocation

- Top-down
  - Assign registers by frequency
  - Spill registers by reuse distance

Global allocation

- Top-down
  - Color interference graph
- Bottom-up
  - Split live ranges

Local Register Allocation

- What's "local"? (as opposed to "global")
  - A local transformation operates on basic blocks
  - Many optimizations are done locally
- Does local allocation solve the problem?
  - It produces decent register use inside a block
  - Inefficiencies can arise at boundaries between blocks
- How many passes can the allocator make?
  - This is an off-line problem
  - As many passes as it takes
- Memory-to-memory vs. register-to-register model
  - Code shape and safety issues
Register Allocation

Can we do this optimally? (on real code?)

Local Allocation
- Simplified cases ⇒ O(n)
- Real cases ⇒ NP-Complete

Global Allocation
- NP-Complete for 1 register
- NP-Complete for k registers
  (most sub-problems are NPC, too)

Real compilers face real problems

Observations

Allocator may need to reserve registers to ensure feasibility
- Must be able to compute addresses
- Requires some minimal (feasible) set of registers, F
  - F depends on target architecture
- Use these registers only for spilling
  (set them aside, i.e., not available for register assignment)

What if k - F < |values|? k?
- The allocator can either
  → Check for this situation
  → Accept the fact that the technique is an approximation

ILOC Instruction Set

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 ⇒ r2</td>
<td>MEM(r1) − r2</td>
<td>2</td>
</tr>
<tr>
<td>store r1 ⇒ r2</td>
<td>r1 − MEM(r2)</td>
<td>2</td>
</tr>
<tr>
<td>load 0 ⇒ r1</td>
<td>0 ⇒ r1</td>
<td>1</td>
</tr>
<tr>
<td>add r1, r2 ⇒ r3</td>
<td>r1 + r2 ⇒ r3</td>
<td>1</td>
</tr>
<tr>
<td>sub r1, r2 ⇒ r3</td>
<td>r1 − r2 ⇒ r3</td>
<td>1</td>
</tr>
<tr>
<td>mult r1, r2 ⇒ r3</td>
<td>r1 × r2 ⇒ r3</td>
<td>1</td>
</tr>
<tr>
<td>lshift r1, r2 ⇒ r3</td>
<td>r1 &lt;&lt; r2 ⇒ r3</td>
<td>1</td>
</tr>
<tr>
<td>rshift r1, r2 ⇒ r3</td>
<td>r1 &gt;&gt; r2 ⇒ r3</td>
<td>1</td>
</tr>
<tr>
<td>output 0</td>
<td>print out MEM(e)</td>
<td>1</td>
</tr>
</tbody>
</table>

Assume a register-to-register memory model, with 1 class of registers.
Latencies are important for instruction scheduling, not register allocation and assignment

ILOC Example

Sample code sequence

load 1028 ⇒ r1 // r1 − 1028
load r1 ⇒ r2 // r2 ← MEM(r1) ⇒ y
mult r1, r2 ⇒ r3 // r3 ← 1028 · y
load 5 ⇒ r4 // r4 ← 5
sub r4, r2 ⇒ r5 // r5 ← 5 · y
load 0 ⇒ r6 // r6 ← 0
mult r5, r6 ⇒ r7 // r7 ← 0 · (5 · y)
sub r7, r3 ⇒ r8 // r8 ← 8 · (5 · y) − (1028 · y)
store r8 ⇒ r1 // MEM(r1) ← 8 · (5 · y) − (1028 · y)

ILOC Example - Live Ranges

Live range for r1

1. load 1028 ⇒ r1 // r1
2. load r1 ⇒ r2 // r2, r3
3. mult r1, r2 ⇒ r3 // r3, r4
4. load 5 ⇒ r4 // r4, r5
5. sub r4, r2 ⇒ r5 // r5, r3
6. load 0 ⇒ r6 // r6, r7
7. mult r5, r6 ⇒ r7 // r7, r3
8. sub r7, r3 ⇒ r8 // r1, r8
9. store r8 ⇒ r1 //

NOTE: live sets on exit of each instruction
ILOC Example - Live Ranges

- Live range for r2
  1. load r 1028 \[\Rightarrow r1\] // r1
  2. load r 1 \[\Rightarrow r2\] // r2
  3. mul r 1,2 \[\Rightarrow r3\] // r1\[r2\] r3
  4. load r 3 \[\Rightarrow r4\] // r3
  5. sub r 1, r2 \[\Rightarrow r5\] // r1\[r2\] r5
  6. load r 1 \[\Rightarrow r6\] // r1
  7. mul r 5, r6 \[\Rightarrow r7\] // r5\[r6\] r7
  8. sub r 7, r3 \[\Rightarrow r8\] // r1
  9. store r 8 \[\Rightarrow r1\] //

NOTE: live sets on exit of each instruction

ILOC Example - Live Ranges

- Live range for r3
  1. load r 1028 \[\Rightarrow r1\] // r1
  2. load r 1 \[\Rightarrow r2\] // r2
  3. mul r 1, r2 \[\Rightarrow r3\] // r1\[r2\] r3
  4. load r 3 \[\Rightarrow r4\] // r3
  5. sub r 1, r2 \[\Rightarrow r5\] // r1\[r2\] r5
  6. load r 8 \[\Rightarrow r6\] // r8
  7. mul r 5, r6 \[\Rightarrow r7\] // r5\[r6\] r7
  8. sub r 7, r3 \[\Rightarrow r8\] // r1
  9. store r 8 \[\Rightarrow r1\] //

NOTE: live sets on exit of each instruction

Top-down Versus Bottom-up Allocation

**Top-down allocator**
- Work from external notion of what is important
- Assign registers in priority order
- Register assignment remains fixed for entire basic block
- Save some registers for the values relegated to memory (feasible set F)

**Bottom-up allocator**
- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Register assignment may change across basic block
- Save some registers for the values relegated to memory (feasible set F)

Bottom-up Allocator

**The 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 is available, free one

**Replacement:**
- **Spill** the value whose next use is **farthest in the future**
- Prefer clean value to dirty value
- Sound familiar? Think cache line / page replacement ...

Spill code

- A virtual register is **spilled** by using only registers from the feasible set (F), not the allocated set (k-F)
  - For the definition of the spilled value (assignment of the value to the virtual register), use a feasible register as the target register and then use an additional register to load its address in memory, and perform the store:
    - add r 1, r 2 \[\Rightarrow r3\] add r 1, r 2 \[\Rightarrow r3\]
    - load r 1 \[\Rightarrow f1\] // value lives at memory location \(f\)
    - store f 1 \[\Rightarrow f2\]
  - For the use of the spilled value, load value from memory into a feasible register:
    - load r 1 \[\Rightarrow f1\] // value lives at memory location \(f\)
    - add r 1, r 2 \[\Rightarrow r3\]

- How many feasible registers do we need for an add instruction?

ILOC Example - Bottom-up Allocation

**Bottomup (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>Source code</th>
<th>Life ranges</th>
<th>register allocation and assignment (on exit)</th>
</tr>
</thead>
</table>
| load r 1028 \[\Rightarrow r1\] | ra \[
| load r 1 \[\Rightarrow r2\] | rb \[
| mul r 1, r 2 \[\Rightarrow r3\] | rc \[
| load r 8 \[\Rightarrow r8\] | ra \[
| store r 8 \[\Rightarrow r1\] | rb \[

Note: this is only one possible allocation and assignment!

Part of r1 live range spilled
### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>source code</th>
<th>register allocation and assignment(on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 1028  ⇒ r2</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load r2 1028  ⇒ r3</td>
<td></td>
</tr>
<tr>
<td>load r3 1028  ⇒ r4</td>
<td></td>
</tr>
<tr>
<td>load r4 1028  ⇒ r5</td>
<td></td>
</tr>
<tr>
<td>load r5 1028  ⇒ r6</td>
<td></td>
</tr>
<tr>
<td>load r6 1028  ⇒ r7</td>
<td></td>
</tr>
<tr>
<td>load r7 1028  ⇒ r8</td>
<td></td>
</tr>
<tr>
<td>store r8 1028  ⇒ r9</td>
<td></td>
</tr>
</tbody>
</table>

**Let's generate code now!**

For read registers, use previous register assignment.

---

### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>source code</th>
<th>register allocation and assignment(on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 1028  ⇒ r2</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load r2 1028  ⇒ r3</td>
<td></td>
</tr>
<tr>
<td>load r3 1028  ⇒ r4</td>
<td></td>
</tr>
<tr>
<td>load r4 1028  ⇒ r5</td>
<td></td>
</tr>
<tr>
<td>load r5 1028  ⇒ r6</td>
<td></td>
</tr>
<tr>
<td>load r6 1028  ⇒ r7</td>
<td></td>
</tr>
<tr>
<td>load r7 1028  ⇒ r8</td>
<td></td>
</tr>
<tr>
<td>store r8 1028  ⇒ r9</td>
<td></td>
</tr>
</tbody>
</table>

**For written registers, use current register assignment**

---

### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>source code</th>
<th>register allocation and assignment(on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 1028  ⇒ r2</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load r2 1028  ⇒ r3</td>
<td></td>
</tr>
<tr>
<td>load r3 1028  ⇒ r4</td>
<td></td>
</tr>
<tr>
<td>load r4 1028  ⇒ r5</td>
<td></td>
</tr>
<tr>
<td>load r5 1028  ⇒ r6</td>
<td></td>
</tr>
<tr>
<td>load r6 1028  ⇒ r7</td>
<td></td>
</tr>
<tr>
<td>load r7 1028  ⇒ r8</td>
<td></td>
</tr>
<tr>
<td>store r8 1028  ⇒ r9</td>
<td></td>
</tr>
</tbody>
</table>

**Insert spill code**

---

### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>source code</th>
<th>register allocation and assignment(on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 1028  ⇒ r2</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load r2 1028  ⇒ r3</td>
<td></td>
</tr>
<tr>
<td>load r3 1028  ⇒ r4</td>
<td></td>
</tr>
<tr>
<td>load r4 1028  ⇒ r5</td>
<td></td>
</tr>
<tr>
<td>load r5 1028  ⇒ r6</td>
<td></td>
</tr>
<tr>
<td>load r6 1028  ⇒ r7</td>
<td></td>
</tr>
<tr>
<td>load r7 1028  ⇒ r8</td>
<td></td>
</tr>
<tr>
<td>store r8 1028  ⇒ r9</td>
<td></td>
</tr>
</tbody>
</table>

**Spill Code**

---

### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>source code</th>
<th>register allocation and assignment(on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1 1028  ⇒ r2</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load r2 1028  ⇒ r3</td>
<td></td>
</tr>
<tr>
<td>load r3 1028  ⇒ r4</td>
<td></td>
</tr>
<tr>
<td>load r4 1028  ⇒ r5</td>
<td></td>
</tr>
<tr>
<td>load r5 1028  ⇒ r6</td>
<td></td>
</tr>
<tr>
<td>load r6 1028  ⇒ r7</td>
<td></td>
</tr>
<tr>
<td>load r7 1028  ⇒ r8</td>
<td></td>
</tr>
<tr>
<td>store r8 1028  ⇒ r9</td>
<td></td>
</tr>
</tbody>
</table>

**Spill Code**
### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Register Allocation and Assignment (on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td>1028 =&gt; ra</td>
</tr>
<tr>
<td>load ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>mult ra, rb</td>
<td>ra =&gt; rc</td>
</tr>
<tr>
<td>store ra</td>
<td>ra =&gt; 10</td>
</tr>
<tr>
<td>loadI 5</td>
<td>5 =&gt; rc</td>
</tr>
<tr>
<td>sub ra, rb</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>loadI 8</td>
<td>8 =&gt; rc</td>
</tr>
<tr>
<td>mult rb, ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>sub rb, rc</td>
<td>rb =&gt; rc</td>
</tr>
</tbody>
</table>

- **Insert spill code**

### ILOC Example - Bottom-up Assignment

**Bottom up (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Register Allocation and Assignment (on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td>1028 =&gt; ra</td>
</tr>
<tr>
<td>load ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>mult ra, rb</td>
<td>ra =&gt; rc</td>
</tr>
<tr>
<td>store* ra</td>
<td>ra =&gt; 10</td>
</tr>
<tr>
<td>loadI 5</td>
<td>5 =&gt; rc</td>
</tr>
<tr>
<td>sub ra, rb</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>loadI 8</td>
<td>8 =&gt; rc</td>
</tr>
<tr>
<td>mult rb, ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>sub rb, rc</td>
<td>rb =&gt; rc</td>
</tr>
</tbody>
</table>

- **Insert spill code**

### Top-down Allocator

**The idea:**
- Keep busiest values in a register
- Use the feasible (reserved) set, F, for the rest

**Algorithm:**
- Rank values by number of occurrences
- Allocate first k - F values to registers
- Rewrite code to reflect these choices

**SPILL:** Move values with no register into memory

### ILOC Example - Top-down Allocation

**Top down (3 physical registers: ra, rb, rc)**

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Register Allocation and Assignment (on exit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td>1028 =&gt; ra</td>
</tr>
<tr>
<td>load ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>mult ra, rb</td>
<td>ra =&gt; rc</td>
</tr>
<tr>
<td>loadI 5</td>
<td>5 =&gt; rc</td>
</tr>
<tr>
<td>sub ra, rb</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>loadI 8</td>
<td>8 =&gt; rc</td>
</tr>
<tr>
<td>mult rb, ra</td>
<td>ra =&gt; rb</td>
</tr>
<tr>
<td>sub rb, rc</td>
<td>rb =&gt; rc</td>
</tr>
</tbody>
</table>

- **Consider**
  - More occurrences of virtual register (most important)
  - Fewer → better spill candidate
  - More → less suitable register

- **Note that this assumes that an extra register is not needed for save/restore**
Global Register Allocation - Top Down

- Register coloring
  - Maps 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 find coloring, spill registers & repeat

Global Live Ranges

- Definition
  - All definitions which reach a use...
  - ...plus all uses reached by these definitions

- A single virtual register may comprise several live ranges

- Live ranges estimate 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 live ranges
    - Both ranges are live at same point
    - Cannot occupy the same register
  - Coloring represents register assignment
    - One color per register

- Using a graph coloring abstraction subtly changes the problem
  - Justified by need to separate optimization and allocation

Building the Interference Graph

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

- Example

Coloring

- Graph coloring
  - Given graph, find assignment of colors to each node
    - Such that no neighbors have the same color
    - Determining whether a graph has a k-coloring
    - Is NP-hard for k ≥ 2

- Register coloring
  - Find a legal coloring given k colors
  - Where k is the number of available registers

Graph Coloring Through Simplification

- Coloring algorithm [Chaitin et al., 1981]
  1. Repeatedly remove nodes with degree ≤ k from graph
    - Push nodes onto stack
  2. If every remaining node is degree ≤ k
    - Spill node with lowest spill cost
    - Remove node from graph
  3. Reassemble graph with nodes popped from stack
    - As each node is added to graph
    - Choose a color differing from its neighbors
Graph Coloring Example

- Given interference graph and 3 registers
- Simplify graph by removing nodes with < 3 neighbors
  → Push nodes onto stack
  → Reassemble graph by popping nodes from stack
    → Assigning colors not used by neighbors

Graph Coloring

- Given interference graph and 2 registers
- Simplify graph by removing nodes with < 2 neighbors
  → No such node, must spill node with lowest spill cost
- Remaining nodes can then be simplified & colored
  → Can we do better?
    → Yes!

Optimistic Graph Coloring

- Optimistic coloring algorithm [Briggs et al., 1989]
  → Remove nodes with degree > k from graph (pop onto stack)
  → If every node has degree <= k
    → Remove node with lowest spill cost (pop onto stack)
    → Reassemble graph with nodes popped from stack
      → Spill node if it cannot be colored
  → Optimal coloring defers spilling decision
    → Helps if neighbors of node
      → Are the same color
      → Have already been spilled

Spill Code

- Inserted when too few registers to hold all live ranges
  → Insert load before use
  → Insert store after definition
- Effects
  → Breaks live range into many small live ranges
  → Reduces chance of interference
  → Expensive
    → Introduces load / store instructions
    → For each use / def instruction in live range

Spill Cost

- Need to decide which live range to spill if needed
- Two metrics to consider
  → Cost of spill
    → Cost of load / store instructions inserted
  → Decrease in interference
    → Reduce need for more spills
Spill Cost

- Possible cost functions
  - \( \text{degree}(v) \) # of edges for \( v \) in interference graph
  - \( \text{depth}(I) \) loop nesting depth of instruction \( I \)
  - \( \text{cost}(v) = \sum_{v \in \text{instr}} \text{degree}(v) \)

Assumes 10 loop iterations

- Possible cost estimate heuristics
  - Cost / degree
  - Cost / (degree * degree)
  - Etc...

[Chaitin et al., 1981]

Allocation with Spilling

- One approach
  - Apply different cost estimate heuristics
  - Pick best result
  - Assumptions
  - Building interference graph is highest expense
  - Spill cost estimates can be calculated inexpensively

- Reducing spill code by recognizing special cases
  - Value modified (dirty)
  - Store register value to memory, reload for use
  - Read-only value (clean)
  - Reload from memory for use
  - Constant value (rematerializable)
  - Recompute value (no need for memory load)

Global Register Allocation - Bottom Up

- Live range splitting
  - Insert copies to split up live ranges
  - Hopefully reduces need for spilling
  - Also controls spill code placement
  - Spill code generated at copies

Examples

\[
\begin{align*}
a &= \ldots \\
\ldots &= a
\end{align*}
\]

Global Register Allocation - Bottom Up

- Coalescing (subsumption)
  - Allocate source and destination of copy to same register
  - To eliminate register-to-register copies
  - Combines live ranges
  - Can reverse unnecessary splits

Examples

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

Live Range Splitting

- Approach [Chow & Hennessy 1990]
  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

Example

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

Register Allocation Examples

[Assuming only 1 register available, must spill \( p \)]
Combining Instruction Scheduling & Register Allocation

- Allocation before scheduling
  - Register assignment introduces dependences
  - Anti- & output dependences
  - Reduces freedom of instruction scheduling

- Example
  
<table>
<thead>
<tr>
<th>Load</th>
<th>Load</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr1,a</td>
<td>r1,a</td>
<td>r1,a</td>
</tr>
<tr>
<td>vr3 = vr1</td>
<td>r3 = r1</td>
<td>load r2,b</td>
</tr>
<tr>
<td>load vr2,b</td>
<td>load r1,b</td>
<td>r3 = r1</td>
</tr>
<tr>
<td>vr4 = vr2</td>
<td>r4 = r1</td>
<td>r4 = r2</td>
</tr>
</tbody>
</table>

Combining Instruction Scheduling & Register Allocation

- Scheduling before allocation
  - Lengthens live range of virtual registers
  - Increases register pressure
  - May cause spills
  - Still need to schedule spill code after allocation

- Example
  
<table>
<thead>
<tr>
<th>Load</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>vr1,a</td>
<td>vr1,a</td>
</tr>
<tr>
<td>load vr2,b</td>
<td>load r2,b</td>
</tr>
<tr>
<td>vr4 = vr1</td>
<td>load vr2,b</td>
</tr>
<tr>
<td>load vr3,c</td>
<td>vr5 = vr2</td>
</tr>
<tr>
<td>vr6 = vr3</td>
<td>vr6 = vr3</td>
</tr>
</tbody>
</table>

Scheduling and Allocation Are Interdependent

- Conflicting goals for scheduling & allocation

- Some possible solutions
  - Assigning registers
    - First fit
      - Lowest available register number
    - Round robin
      - Cycle through all registers
      - Reduces memory-related dependences

Scheduling and Allocation Are Interdependent

- More possible solutions
  - Change ordering
    - Postpass - allocate then schedule
    - Prepass - schedule then allocate
    - Multipass - schedule, allocate, schedule
  - Integrated prepass scheduling
    - Schedule instructions first as preparation
    - Bias schedule to reduce local register pressure
    - Allocate registers after scheduling