CS415 Compilers
More Register Allocation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Recitations started this week
  Attendance is mandatory
  You have to attend your section’s recitation

• Office hours have been posted
  Office hours started this week

• Reminder: get ilab account

• First homework will be handed out by tomorrow
  Deadline: Friday, February 2

• Lecture on Monday, January 29, is cancelled
  Possible makeup: review session for midterm exam
Memory Model / Code Shape

• **register-register model**
  - Values that *may safely reside* in registers are assigned to a unique virtual register (*alias analysis*)
  - Register allocation/assignment maps virtual registers to limited set of physical registers
  - Register allocation/assignment pass needed to make code “work”

• **memory-memory model**
  - All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
  - Register allocation/assignment has to try to identify cases where values can be safely kept in registers
  - Safety verification is hard at the low levels of program abstraction
  - Even without register allocation/assignment, code will “work”
Memory Model / Code Shape

• **register-register model**  
  → Values that *may safely reside* in registers are assigned to a unique virtual register (*alias analysis*)
  → Register allocation/assignment maps virtual registers to limited set of physical registers
  → Register allocation/assignment pass needed to make code “work”

• **memory-memory model**  
  → All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
  → Register allocation/assignment has to try to identify cases where values can be safely kept in registers
  → Safety verification is hard at the low levels of program abstraction
  → Even without register allocation/assignment, code will “work”
The Task
• At each point in the code, pick the values to keep in registers
• Insert code to move values between registers & memory
  → No transformations (leave that to scheduling)
• Minimize inserted code — both dynamic & static measures
• Make good use of any extra registers

Allocation versus assignment
• Allocation is deciding which values to keep in registers
• Assignment is choosing specific registers for values
• This distinction is often lost in the literature

The compiler must perform both allocation & assignment
Basic Blocks

Definition

→ A basic block is a maximal length segment of straight-line (i.e., branch free) code

Importance (assuming normal execution)

• Strongest facts are provable for branch-free code
• If any statement executes, they all execute
• Execution is totally ordered

Optimization

• Many techniques for improving basic blocks
• Simplest problems
• Strongest methods
Basic Approach of Allocators

Allocator may need to reserve registers to ensure feasibility

- Must be able to compute addresses
- Requires some minimal set of registers, $F$
  - $F$ depends on target architecture
- $F$ contains registers to make spilling work
  (set $F$ registers “aside”, i.e., not available for register assignment)

What if $k - F < |\text{values}| < k$?

- The allocator can either
  - Check for this situation
  - Accept the fact that the technique is an approximation
A value is live between its definition and its uses

- Find definitions ($x \leftarrow \ldots$) and uses ($y \leftarrow \ldots x \ldots$)
- From definition to last use is its live range
  - How does a second definition affect this?
- Can represent live range as an interval $[i,j]$ (in block)
  - live on exit

Let $MAXLIVE$ be the maximum, over each instruction $i$ in the block, of the number of values (pseudo-registers) live at $i$.

- If $MAXLIVE \leq k$, allocation should be easy
  - no need to reserve $F$ registers for spilling
- If $MAXLIVE > k$, some values must be spilled to memory

Finding live ranges is harder in the global case
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 $F$)
The idea:

- Machine has $k$ physical registers
- Keep “busiest” values in an assigned register
- Use the feasible (reserved) set, $F$, for the rest
- $F$ is the minimal set of registers needed to execute any instruction with all operands in memory:
  - Move values with no assigned register from/to memory by adding LOADs and STOREs (SPILL CODE)

Basic algorithm:

- Rank values by number of occurrences (or some other metric)
- Allocate first $k - F$ values to registers
- Rewrite code (with spill code) to reflect these choices
ILOC instructions subset (see Appendix A in EaC):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>MEM(r1) → r2</td>
<td>2</td>
</tr>
<tr>
<td>store</td>
<td>r1 → MEM(r2)</td>
<td>2</td>
</tr>
<tr>
<td>loadl</td>
<td>c → r1</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>r1 + r2 → r3</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>r1 – r2 → r3</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>r1 x r2 → r3</td>
<td>1</td>
</tr>
<tr>
<td>lshift</td>
<td>r1 &lt;&lt;&lt; r2 → r3</td>
<td>1</td>
</tr>
<tr>
<td>rshift</td>
<td>r1 &gt;&gt;&gt; r2 → r3</td>
<td>1</td>
</tr>
<tr>
<td>output</td>
<td>print out MEM(c)</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.
### Live Ranges

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadI</td>
<td>1028</td>
<td>r1</td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>3</td>
<td>mult</td>
<td>r1, r2</td>
<td>r3</td>
</tr>
<tr>
<td>4</td>
<td>loadI</td>
<td>5</td>
<td>r4</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>r4, r2</td>
<td>r5</td>
</tr>
<tr>
<td>6</td>
<td>loadI</td>
<td>8</td>
<td>r6</td>
</tr>
<tr>
<td>7</td>
<td>mult</td>
<td>r5, r6</td>
<td>r7</td>
</tr>
<tr>
<td>8</td>
<td>sub</td>
<td>r7, r3</td>
<td>r8</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>r8</td>
<td>r1</td>
</tr>
</tbody>
</table>

**NOTE:** live sets on exit of each instruction
**Live Ranges**

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Value</th>
<th>Range Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadI</td>
<td>1028</td>
<td>r1</td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td>r1</td>
<td>r1, r2</td>
</tr>
<tr>
<td>3</td>
<td>mult</td>
<td>r1, r2</td>
<td>r1, r2, r3</td>
</tr>
<tr>
<td>4</td>
<td>loadI</td>
<td>5</td>
<td>r1, r2, r3, r4</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>r4, r2</td>
<td>r1, r3, r5</td>
</tr>
<tr>
<td>6</td>
<td>loadI</td>
<td>8</td>
<td>r1, r3, r5, r6</td>
</tr>
<tr>
<td>7</td>
<td>mult</td>
<td>r5, r6</td>
<td>r1, r3, r7</td>
</tr>
<tr>
<td>8</td>
<td>sub</td>
<td>r7, r3</td>
<td>r1, r8</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>r8</td>
<td>r1</td>
</tr>
</tbody>
</table>

**NOTE:** live sets on exit of each instruction
An Example: Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - k = 4, F = 1, (k-F) = 3

1. loadI 1028 \(\Rightarrow r1\) // r1
2. load r1 \(\Rightarrow r2\) // r1 r2
3. mult r1, r2 \(\Rightarrow r3\) // r1 r2 r3
4. loadI 5 \(\Rightarrow r4\) // r1 r2 r3 r4
5. sub r4, r2 \(\Rightarrow r5\) // r1 r3 r5
6. loadI 8 \(\Rightarrow r6\) // r1 r3 r5 r6
7. mult r5, r6 \(\Rightarrow r7\) // r1 r3 r7
8. sub r7, r3 \(\Rightarrow r8\) // r1 r8
9. store r8 \(\Rightarrow r1\) //

- Consider statements with MAXLIVE > (k-F) basic algorithm
  - Spill heuristic: - number of occurrences of virtual register
  - length of live range (tie breaker)
3 physical registers to allocate: ra, rb, rc

1 selected register: f1 (feasible set)

- \( k = 4, F = 1, (k-F) = 3 \)

1. loadI 1028 \( \Rightarrow r1 \) // r1
2. load r1 \( \Rightarrow r2 \) // r1 r2
3. mult r1, r2 \( \Rightarrow r3 \) // r1 r2 r3
4. loadI 5 \( \Rightarrow r4 \) // r1 r2 r3 r4 \( -- MAXLIVE = 4 \)
5. sub r4, r2 \( \Rightarrow r5 \) // r1 r3 r5
6. loadI 8 \( \Rightarrow r6 \) // r1 r3 r5 r6 \( -- MAXLIVE = 4 \)
7. mult r5, r6 \( \Rightarrow r7 \) // r1 r3 r7
8. sub r7, r3 \( \Rightarrow r8 \) // r1 r8
9. store r8 \( \Rightarrow r1 \) //

Consider statements with \( MAXLIVE > (k-F) \) basic algorithm

Spill heuristic: - number of occurrences of virtual register
- length of live range (tie breaker)
An Example: Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - \( k = 4, F = 1, (k-F) = 3 \)

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

- Consider statements with \( \text{MAXLIVE} > (k-F) \) basic algorithm
  - Spill heuristic:
    - number of occurrences of virtual register
    - length of live range (tie breaker)

Note: EAC Top down algorithm does not look at live ranges and \( \text{MAXLIVE} \), but counts overall occurrences across entire basic block
3 physical registers for allocation: ra, rb, rc

- 1 physical register **designated** to be in the feasible set $F$

```
1   loadI  1028  ⇒ ra  // r1
2   load   ra  ⇒ rb  // r1 r2
3   mult   ra, rb ⇒ f1  // r1 r2 r3
   store*  f1  ⇒ 10  // spill code  *NOT ILOC
4   loadI  5    ⇒ rc  // r1 r2 r3 r4
5   sub    rc, rb ⇒ rb  // r1 r3 r5
6   loadI  8    ⇒ rc  // r1 r3 r5 r6
7   mult   rb, rc ⇒ rb  // r1 r3 r7
   load*   10   ⇒ f1  // spill code  *NOT ILOC
8   sub    rb, f1 ⇒ rb  // r1 r8
9   store  rb   ⇒ ra  //
```

- Insert spill code for every occurrence of spilled virtual register in basic block using feasible register $f1$
More register allocation (bottom-up)

Instruction Scheduling

Read EaC: Chapter 12.1 - 12.3

Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;