CS415 Compilers
Register Allocation
(Part 2)

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

• First homework problem will be posted by tomorrow, Thursday January 31

• No office hours tomorrow. Please stay warm. Instead, there will be office hours on Monday 2/4, from 9:30 – 10:30am, CoRE 318 (my office)

• Posted paper on local register allocation (if you are interested in the topic)

• Last reminder: get ilab account
Part of the compiler’s back end

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
  - $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$
**Aliasing**: Two variables or source-code names may refer to the same memory location.

Examples:
- formal call-by-reference parameters a and b
- pointers a->f and b->f
- array elements: a(i, j) and a(k, l)

**Challenge**: When is it safe to keep a variable’s value in a register across STORE instructions, i.e., while other STORE instructions are executed?
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”
Basic Approach of Allocators

Allocator may need to reserve physical registers to ensure feasibility

- Must be able to compute memory addresses
- Requires some minimal set of registers, $F$
  $\rightarrow$ $F$ depends on target architecture
- $F$ contains registers to make spilling work
  $\rightarrow$ set $F$ registers “aside” for address computation & instruction execution, i.e., these are not available for register assignment
- Note: $F$ physical registers need to be able to support the pathological case where all virtual registers are spilled

What if $k - F < |\text{values}| < k$?
- The allocator can either
  $\rightarrow$ Check for this situation
  $\rightarrow$ Accept the fact that the technique is an approximation

Notation:
$k$ is the number of physical registers on the target machine
Top-down allocator
- May use notion of “live ranges” of virtual registers
- 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)
Assume $i$ and $j$ are two instructions in a basic block

A value (virtual register) is live between its definition and its uses

• Find definitions ($x \leftarrow ...$) and uses ($y \leftarrow ... x ...$)
• From definition to last use is its live range
  → How many (static) definitions can you have for a virtual register?
• 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 (virtual 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
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>r1 ( \Rightarrow ) r2</td>
<td>MEM(r1) ( \Rightarrow ) r2</td>
</tr>
<tr>
<td>store</td>
<td>r1 ( \Rightarrow ) r2</td>
<td>r1 ( \Rightarrow ) MEM(r2)</td>
</tr>
<tr>
<td>loadl</td>
<td>c ( \Rightarrow ) r1</td>
<td>c ( \Rightarrow ) r1</td>
</tr>
<tr>
<td>add</td>
<td>r1, r2 ( \Rightarrow ) r3</td>
<td>r1 + r2 ( \Rightarrow ) r3</td>
</tr>
<tr>
<td>sub</td>
<td>r1, r2 ( \Rightarrow ) r3</td>
<td>r1 - r2 ( \Rightarrow ) r3</td>
</tr>
<tr>
<td>mult</td>
<td>r1, r2 ( \Rightarrow ) r3</td>
<td>r1 \times r2 ( \Rightarrow ) r3</td>
</tr>
<tr>
<td>lshift</td>
<td>r1, r2 ( \Rightarrow ) r3</td>
<td>r1 \ll r2 ( \Rightarrow ) r3</td>
</tr>
<tr>
<td>rshift</td>
<td>r1, r2 ( \Rightarrow ) r3</td>
<td>r1 \gg r2 ( \Rightarrow ) r3</td>
</tr>
<tr>
<td>output</td>
<td>c ( \Rightarrow ) r3</td>
<td>print out MEM(c)</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>Live Ranges</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
An Example: Top-Down

Live Ranges

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Destinations</th>
<th>Live Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>1028</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>load</td>
<td>r1</td>
<td>r2</td>
<td>r1 r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1, r2</td>
<td>r3</td>
<td>r1 r2 r3</td>
</tr>
<tr>
<td>loadI</td>
<td>5</td>
<td>r4</td>
<td>r1 r2 r3 r4</td>
</tr>
<tr>
<td>sub</td>
<td>r4, r2</td>
<td>r5</td>
<td>r1 r3 r5</td>
</tr>
<tr>
<td>loadI</td>
<td>8</td>
<td>r6</td>
<td>r1 r3 r5 r6</td>
</tr>
<tr>
<td>mult</td>
<td>r5, r6</td>
<td>r7</td>
<td>r1 r3 r7</td>
</tr>
<tr>
<td>sub</td>
<td>r7, r3</td>
<td>r8</td>
<td>r1 r8</td>
</tr>
<tr>
<td>store</td>
<td>r8</td>
<td>r1</td>
<td></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 \texttt{MAXLIVE} > \((k-F)\) \textit{basic algorithm}

- Spill heuristic: - 1. number of occurrences of virtual register
- 2. 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. \( \text{loadI} 1028 \Rightarrow r1 \)  // r1
2. \( \text{load} r1 \Rightarrow r2 \)  // r1 r2
3. \( \text{mult} r1, r2 \Rightarrow r3 \)  // r1 r2 r3
4. \( \text{loadI} 5 \Rightarrow r4 \)  // r1 r2 r3 r4  -- MAXLIVE = 4
5. \( \text{sub} r4, r2 \Rightarrow r5 \)  // r1 r3 r5
6. \( \text{loadI} 8 \Rightarrow r6 \)  // r1 r3 r5 r6  -- MAXLIVE = 4
7. \( \text{mult} r5, r6 \Rightarrow r7 \)  // r1 r3 r7
8. \( \text{sub} r7, r3 \Rightarrow r8 \)  // r1 r8
9. \( \text{store} r8 \Rightarrow r1 \)  // 

- 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
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 \( \text{MAXLIVE} > (k-F) \) 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
Finish up Register Allocation

Instruction Scheduling

Read EaC: Chapter 12.1 - 12.3

Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;