CS415 Compilers
Register Allocation

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

• Recitations start this week (today)
  Attendance is mandatory
  You have to attend your section’s recitation

• Office hours have been posted
  Office hours start this week

• Reminder: get ilab account

• SP numbers (need pre-requisites!)?

• Lecture on Monday, January 29, cancelled
  possible makeup: review session for midterm exam

Local: within single basic block
Global: across procedure/function
Register Allocation

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 \( O(2^n) \)
Source code

```
A = 5;
B = 6;
C = A + B;
```
Register allocation on basic blocks in “ILOC”

- Pseudo-code for a simple, abstracted RISC machine generated by the instruction selection process
- Simple, compact data structures
- Here: we only use a small subset of ILOC

**Naïve Representation:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>2</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>loadAl</td>
<td>r0</td>
<td>@y</td>
<td>r2</td>
</tr>
<tr>
<td>add</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0</td>
<td>@x</td>
<td>r4</td>
</tr>
<tr>
<td>sub</td>
<td>r4</td>
<td>r3</td>
<td>r5</td>
</tr>
</tbody>
</table>

**Quadruples:**

- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

*ILOC is described in Appendix A of EAC*
Memory Model / Code Shape

ILOC: EaC Appendix A

Source code

\[
\begin{align*}
A &= 5; \\
B &= 6; \\
C &= A + B;
\end{align*}
\]

ILOC code

\[
\begin{align*}
\text{loadI } 5 & \Rightarrow r1 \\
& \quad \text{// compute address of } A \text{ in } r2 \\
\ldots \\
\text{store } r1 & \Rightarrow r2 \quad \text{// content}(A) = r1 \\
\text{loadI } 6 & \Rightarrow r3 \\
& \quad \text{// compute address of } B \text{ in } r4 \\
\ldots \\
\text{store } r3 & \Rightarrow r4 \quad \text{// content}(B) = r3 \\
\text{add } r1, r3 & \Rightarrow r5 \\
& \quad \text{// compute address of } C \text{ in } r6 \\
\ldots \\
\text{store } r5 & \Rightarrow r6 \quad \text{// content}(C) = r1 + r3
\end{align*}
\]

Is this code correct?
Memory Model / Code Shape

ILOC: EaC Appendix A

Source code

foo (var A, B)
A = 5;
B = 6;
C = A + B;
end foo;
call foo(x,x);
print x;

ILOC code

loadI 5  ⇒ r1
// compute address of A in r2
... 
store r1  ⇒ r2  // content(A) = r1
loadI 6  ⇒ r3
// compute address of B in r4
... 
add r1, r3  ⇒ r5
// compute address of C in r6
... 
store r5  ⇒ r6  // content(C) = r1 + r3

Incorrect for call-by-reference!

Is this code correct?
**Aliasing Problem**

**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”
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”
Consider a fragment of assembly code (or ILOC)

\[
\begin{align*}
\text{loadI} & \quad 2 \implies r1 \quad // r1 \leftarrow 2 \\
\text{loadAI} & \quad r0, @y \implies r2 \quad // r2 \leftarrow y \\
\text{mult} & \quad r1, r2 \implies r3 \quad // r3 \leftarrow 2 \cdot y \\
\text{loadAI} & \quad r0, @x \implies r4 \quad // r4 \leftarrow x \\
\text{sub} & \quad r4, r3 \implies r5 \quad // r5 \leftarrow x - (2 \cdot y)
\end{align*}
\]

The Problem

- At each instruction, decide which values to keep in registers
  - Note: a value is a pseudo-register (virtual register)
  - Simple if $|\text{values}| \leq |\text{registers}|$
  - Harder if $|\text{values}| > |\text{registers}|$
- The compiler must automate this process
Register Allocation

Consider a fragment of assembly code (or ILOC)

\[
\begin{align*}
\text{loadI} & \quad 2 & \Rightarrow r1 & \quad // \ r1 \leftarrow 2 \\
\text{loadAI} & \quad r0, @y & \Rightarrow r2 & \quad // \ r2 \leftarrow y \\
\text{mult} & \quad r1, r2 & \Rightarrow r3 & \quad // \ r3 \leftarrow 2 \cdot y \\
\text{loadAI} & \quad r0, @x & \Rightarrow r4 & \quad // \ r4 \leftarrow x \\
\text{sub} & \quad r4, r3 & \Rightarrow r5 & \quad // \ r5 \leftarrow x - (2 \cdot y)
\end{align*}
\]

The Problem

- At each instruction, decide which \textit{values} to keep in registers
  - Note: a value is a \textit{pseudo-register (virtual register)}
  - Simple if \(|values| \leq |registers|
  - Harder if \(|values| > |registers|
- The compiler must automate this process
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
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
  → The first project (register allocation) assumes that the block is the entire program

• How many passes can the allocator make?
  → This is an off-line problem (not done during program execution)
  → As many passes as it takes

• memory-to-memory vs. register-to-register model
  → code shape and safety issues
Can we do this optimally? (on real code?)

**Local Allocation**
- Simplified cases $\Rightarrow O(n)$
- Real cases $\Rightarrow$ NP-Complete

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

**Local Assignment**
- Single size, no spilling $\Rightarrow O(n)$
- Two sizes $\Rightarrow$ NP-Complete

**Global Assignment**
- NP-Complete

Real compilers face real problems
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, \( F \)
  \( \rightarrow \) \( F \) depends on target architecture
• \( F \) contains registers to make spilling work
  (set \( F \) registers “aside”, i.e., not available for register assignment)

What if \( k - F < |values| < k \) ?

• The allocator can either
  \( \rightarrow \) Check for this situation
  \( \rightarrow \) 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 \(\text{MAXLIVE}\) be the maximum, over each instruction \(i\) in the block, of the number of values (pseudo-registers) live at \(i\).

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

*Finding live ranges is harder in the global case*
Next class

More registers allocation (bottom-up) and Instruction Scheduling

Read EaC: Chapter 13 and 12