CS415 Compilers
Register Allocation

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

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been less successful in the back end.
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 \( \text{spilled values} \)
• Operate efficiently
  \( O(n), O(n \log_2 n), \) maybe \( O(n^2) \), but not \( O(2^n) \)

Local: within single basic block
Global: across procedure/function
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>Instruction</th>
<th>r0</th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>2</td>
<td></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>store</td>
<td>r0</td>
<td></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 Simulator at ~zz124/cs415_2014/ILOC_Simulator on ilab cluster**
Source code

```c
A = 5;
B = 6;
A = A + B;
```
**Source code**

A = 5;
B = 6;
A = A + B;

**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

... store r3 ⇒ r4 // content(B) = r3
add r1, r3 ⇒ r5
// compute address of A in r6

... store r5 ⇒ r6 // content(A) = r1 + r3

Is this code correct?
Memory Model / Code Shape

Source code

```plaintext
foo (var A, B)
    A = 5;
    B = 6;
    A = A + B;
end foo;

call foo(x,x);
print x;
```

ILOC code

```plaintext
loadI 5 ⇒ r1
    // compute address of A in r2

. . .

store r1 ⇒ r2  // content(A) = r1
loadI 6 ⇒ r3
    // compute address of B in r4

. . .

store r3 ⇒ r4  // content(B) = r3
add r1, r3 ⇒ r5
    // compute address of C in r6

. . .

store r5 ⇒ r6  // content(C) = r1 + r3
```

Is this code correct?
Memory Model / Code Shape

Source code

foo (var A, B)
A = 5;
B = 6;
A = 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

...  

store r3 ⇒ r4  
// content(B) = r3

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?
• **register-register model**
  → Values that may safely reside in registers are assigned to a unique virtual register
  → 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 level or 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  
  - 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 level or program abstraction  
  - Even without register allocation/assignment, code will “work”
Register Allocation

Consider a fragment of assembly code (or ILOC)

- loadI 1024 ⇒ r0 // r0 ← 1024
- loadI 2 ⇒ r1 // r1 ← 2
- loadAI r0, @y ⇒ r2 // r2 ← y
- mult r1, r2 ⇒ r3 // r3 ← 2 · y
- loadAI r0, @x ⇒ r4 // r4 ← x
- sub r4, r3 ⇒ r5 // r5 ← x - (2 · y)

The Problem

- At each instruction, decide which values to keep in registers
  - Note: a value is a pseudo-register
- Simple if |values| ≤ |registers|
- Harder if |values| > |registers|
- The compiler must automate this process

Register allocation is described in Chapters 1 & 13 of EAC
ILOC is discussed in Appendix A of EAC
Register Allocation

Consider a fragment of assembly code (or ILOC)

- loadI 1024 ⇒ r0  // r0 ← 1024
- loadI 2 ⇒ r1  // r1 ← 2
- loadAI r0, @y ⇒ r2  // r2 ← y
- mult r1, r2 ⇒ r3  // r3 ← 2 · y
- loadAI r0, @x ⇒ r4  // r4 ← x
- sub r4, r3 ⇒ r5  // r5 ← x - (2 · y)

The Problem

- At each instruction, decide which values to keep in registers
  - Note: a value is a pseudo-register
- Simple if |values| ≤ |registers|
- Harder if |values| > |registers|
- The compiler must automate this process

Register allocation is described in Chapters 1 & 13 of EAC
ILOC is discussed in Appendix A of EAC
The General Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
  - No instruction reordering (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 (instruction scheduling) assumes that the block is the entire program

• 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 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></th>
<th>2</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>r0</td>
<td>@y</td>
</tr>
<tr>
<td>loadAI</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>add</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>storeAI</td>
<td>r0</td>
<td>@x</td>
</tr>
<tr>
<td>sub</td>
<td>r4</td>
<td>r3</td>
</tr>
<tr>
<td></td>
<td></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 Simulator at ~zz124/cs415_2014/ILOC_Simulator on ilab cluster_
Register Allocation

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

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

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

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

Global Assignment
- NP-Complete

Real compilers face real problems
F - Set of Feasible Registers

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)

Notation:

$k$ is the number of registers on the target machine
A value is live between its definition and its uses
• Find definitions (x ← ...) and uses (y ← ... x ...)
• 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
• If \( \text{MAXLIVE} \leq k \), 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 (non local) 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 (different register assignments for different parts of live range)
- Save some registers for the values relegated to memory (feasible set F)
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
(add LOADs & STOREs)
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
• Sound familiar? Think page replacement ...
An Example

Here is a sample code sequence

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value(s)</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>1028</td>
<td>r1</td>
<td>r1 ← 1028</td>
</tr>
<tr>
<td>load</td>
<td>r1</td>
<td>r2</td>
<td>r2 ← MEM(r1) == y</td>
</tr>
<tr>
<td>mult</td>
<td>r1, r2</td>
<td>r3</td>
<td>r3 ← 1028 · y</td>
</tr>
<tr>
<td>loadI</td>
<td>5</td>
<td>r4</td>
<td>r4 ← 5</td>
</tr>
<tr>
<td>sub</td>
<td>r4, r2</td>
<td>r5</td>
<td>r5 ← 5 - y</td>
</tr>
<tr>
<td>loadI</td>
<td>8</td>
<td>r6</td>
<td>r6 ← 8</td>
</tr>
<tr>
<td>mult</td>
<td>r5, r6</td>
<td>r7</td>
<td>r7 ← 8 · (5 - y)</td>
</tr>
<tr>
<td>sub</td>
<td>r7, r3</td>
<td>r8</td>
<td>r8 ← 8 · (5 - y) - (1028 · y)</td>
</tr>
<tr>
<td>store</td>
<td>r8</td>
<td>r1</td>
<td>MEM(r1) ← 8 · (5 - y) - (1028 · y)</td>
</tr>
</tbody>
</table>
An Example: Top-Down

Live Ranges

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
8. sub r7, r3 ⇒ r8 // r1 r8
9. store r8 ⇒ r1 //

NOTE: live sets on exit of each instruction
An Example

Top down (3 physical registers (k-F): ra, rb, rc)

<table>
<thead>
<tr>
<th></th>
<th>Operation</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>

Consider statements with MAXLIVE > (k-F)
- number of occurrences of virtual register
- length of live range
An Example

Top down (3 physical registers (k-F): ra, rb, rc)

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
8. sub r7, r3 ⇒ r8 // r1 r8
9. store r8 ⇒ r1 //

Consider statements with MAXLIVE > (k-F)
- number of occurrences of virtual register (most important)
- length of live range (tie breaker)
- Note: This is different from the algorithm discussed in EAC!
An Example

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

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Value</th>
<th>Register</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadI</td>
<td>1028</td>
<td>ra</td>
<td>// r1</td>
</tr>
<tr>
<td>2</td>
<td>load</td>
<td>ra</td>
<td>rb</td>
<td>// r1 r2</td>
</tr>
<tr>
<td>3</td>
<td>mult</td>
<td>ra, rb</td>
<td>f1</td>
<td>// r1 r2 r3</td>
</tr>
<tr>
<td></td>
<td>store*</td>
<td>f1</td>
<td>10</td>
<td>// spill code</td>
</tr>
<tr>
<td>4</td>
<td>loadI</td>
<td>5</td>
<td>rc</td>
<td>// r1 r2 r3 r4</td>
</tr>
<tr>
<td>5</td>
<td>sub</td>
<td>rc, rb</td>
<td>rb</td>
<td>// r1 r3 r5</td>
</tr>
<tr>
<td>6</td>
<td>loadI</td>
<td>8</td>
<td>rc</td>
<td>// r1 r3 r5 r6</td>
</tr>
<tr>
<td>7</td>
<td>mult</td>
<td>rb, rc</td>
<td>rb</td>
<td>// r1 r3 r7</td>
</tr>
<tr>
<td></td>
<td>load*</td>
<td>10</td>
<td>f1</td>
<td>// spill code</td>
</tr>
<tr>
<td>8</td>
<td>sub</td>
<td>rb, f1</td>
<td>rb</td>
<td>// r1 r8</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>rb</td>
<td>ra</td>
<td>//</td>
</tr>
</tbody>
</table>

- **Insert spill code for every occurrence of spilled virtual register in basic block**
A virtual register is spilled by using only registers from the feasible set (F), not the allocated set (k-F).

How to insert spill code, with $F = \{f_1, f_2, \ldots\}$?

→ 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:

\[
\begin{align*}
\text{add } r1, r2 & \Rightarrow f1 \\
\text{loadI } @f & \Rightarrow f2 \quad \text{// value lives at memory location } @f \\
\text{store } f1 & \Rightarrow f2
\end{align*}
\]

→ For the use of the spilled value, load value from memory into a feasible register:

\[
\begin{align*}
\text{loadI } @f & \Rightarrow f1 \\
\text{load } f1 & \Rightarrow f1 \\
\text{add } f1, r2 & \Rightarrow r1
\end{align*}
\]

How many feasible registers do we need for an add instruction?
More Register Allocation
Instruction Scheduling

Read EaC: Chapters 12.1 - 12.3

Recitation starts TODAY!

First homework will be posted tomorrow.

http://www.cs.rutgers.edu/~zz124/cs415_spring2014/