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

- Office hours have been posted
  Office hours start this week

- Reminder: get ilab account

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
  \[ O(n), O(n \log_2 n), \text{maybe } O(n^2), \text{but not } O(2^n) \]
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

<table>
<thead>
<tr>
<th>Naïve Representation:</th>
<th>Quadruples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 2 r1</td>
<td>• table of ( k \times 4 ) small integers</td>
</tr>
<tr>
<td>loadAI r0 @y r2</td>
<td>• simple record structure</td>
</tr>
<tr>
<td>add r1 r2 r3</td>
<td>• easy to reorder</td>
</tr>
<tr>
<td>loadAI r0 @x r4</td>
<td>• all names are explicit</td>
</tr>
<tr>
<td>sub r4 r3 r5</td>
<td></td>
</tr>
</tbody>
</table>

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

ILOC: EaC Appendix A

Source code

A = 5;
B = 6;
C = 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 C in r6

... 

store r5 ⇒ r6 // content(C) = r1 + r3

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);
paint x;

ILOC code

loadI 5  ⇒ r1
// compute address of A in r2
loadI 6  ⇒ r3
// compute address of B in r4
add r1, r3 ⇒ r5
// compute address of C in r6

store r3 ⇒ r4  // content(B) = r3
store r5 ⇒ r6  // content(C) = r1 + r3

Incorrect for call-by-reference!  Is this code correct?
Aliasing: Two variables or source-code names may refer to the same memory location.

Examples:
- formal call-by-reference parameters \( a \) and \( b \)
- pointers \( a \rightarrow f \) and \( b \rightarrow f \)
- 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)

```plaintext
loadI 2 \Rightarrow r1 // r1 ← 2
loadAI r0, @y \Rightarrow r2 // r2 ← y
mult r1, r2 \Rightarrow r3 // r3 ← 2 · y
loadAI r0, @x \Rightarrow r4 // r4 ← x
sub r4, r3 \Rightarrow r5 // r5 ← x - (2 · y)
```

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
Consider a fragment of assembly code (or ILOC)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value(s)</th>
<th>Register(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loadI</code></td>
<td>2</td>
<td><code>r1</code></td>
<td><code>r1 ← 2</code></td>
</tr>
<tr>
<td><code>loadAI</code></td>
<td><code>r0, @y</code></td>
<td><code>r2</code></td>
<td><code>r2 ← y</code></td>
</tr>
<tr>
<td><code>mult</code></td>
<td><code>r1, r2</code></td>
<td><code>r3</code></td>
<td><code>r3 ← 2 · y</code></td>
</tr>
<tr>
<td><code>loadAI</code></td>
<td><code>r0, @x</code></td>
<td><code>r4</code></td>
<td><code>r4 ← x</code></td>
</tr>
<tr>
<td><code>sub</code></td>
<td><code>r4, r3</code></td>
<td><code>r5</code></td>
<td><code>r5 ← x − (2 · y)</code></td>
</tr>
</tbody>
</table>

The Problem

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

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 compile-time (“off-line”) problem (not done during program execution); typically, 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

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
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, $F$
  \[ \rightarrow F \text{ depends on target architecture} \]
- $F$ contains registers to make spilling work
  \[ \rightarrow \text{set } F \text{ registers "aside" for address computation & instruction execution, i.e. these are not available for register assignment} \]
- Note: $F$ physical registers need to be able to support the pathological case where all virtual registers are spilled

What if $k - F < |values| < k$?

- The allocator can either
  \[ \rightarrow \text{Check for this situation} \]
  \[ \rightarrow \text{Accept the fact that the technique is an approximation} \]
More register allocation (bottom-up)

Instruction Scheduling

Read EaC: Chapter 12.1 – 12.3

Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;