Local Register Allocation & Programming Project

CSE 421
Penn State
University Park
2004
Where are we (and why)?

Register Allocation

• Chapter 13 in EAC
• Read Chapter 1, look at Appendix A
• Details will appear on the web
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
  \( O(n), O(n \log_2 n), \text{maybe } O(n^2), \text{but not } O(2^n) \)
Register Allocation

Consider a fragment of assembly code (or ILOC)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadl</td>
<td>2</td>
<td>r1</td>
</tr>
<tr>
<td>loadA1</td>
<td>r0,@y</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r3</td>
</tr>
<tr>
<td>loadA1</td>
<td>r0,@x</td>
<td>r4</td>
</tr>
<tr>
<td>sub</td>
<td>r4,r3</td>
<td>r5</td>
</tr>
</tbody>
</table>

// r1 ← 2, r2 ← y, r3 ← 2 x y, r4 ← x, r5 ← x - (2 x y)

The Problem

- At each instruction, decide which values to keep in registers
- Simple if |values| ≤ |registers|
- Harder if |values| > |registers|
- The compiler must automate this process

Register allocation is described in Chapters 1 & 13 of EAC
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
• Minimize inserted code — both dynamic & static measures
• Make good use of any extra registers

Allocation versus assignment
• Allocation is choosing what 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
  > Your project 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
Your project will do register allocation on basic blocks in “ILOC”
- Pseudo-code for a simple, abstracted RISC machine
- Simple, compact data structures
- You will use a tiny subset of ILOC

**Naïve Representation:**

<table>
<thead>
<tr>
<th>loadI</th>
<th>2</th>
<th>r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadAI</td>
<td>r0</td>
<td>@y</td>
</tr>
<tr>
<td>add</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0</td>
<td>@x</td>
</tr>
<tr>
<td>sub</td>
<td>r4</td>
<td>r3</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*
Register Allocation

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

<table>
<thead>
<tr>
<th>Local Allocation</th>
<th>Local Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Simplified cases ⇒ O(n)</td>
<td>• Single size, no spilling ⇒ O(n)</td>
</tr>
<tr>
<td>• Real cases ⇒ NP-Complete</td>
<td>• Two sizes ⇒ NP-Complete</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global Allocation</th>
<th>Global Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>• NP-Complete for 1 register</td>
<td>• NP-Complete</td>
</tr>
<tr>
<td>• NP-Complete for k registers</td>
<td></td>
</tr>
</tbody>
</table>

Real compilers face real problems
**Observations**

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
- Use these registers only for spilling

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

- Remember that the underlying problem is NP-Complete
- The allocator can either
  - Check for this situation
  - Adopt a more complex strategy
  - Accept the fact that the technique is an approximation

**Notation:**

$k$ is the number of registers on the target machine
Observations

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*”
- Can represent live range as an interval $[i,j]$ (in block)

Let $\text{MAXLIVE}$ be the maximum, over all the instructions in the block, of the number of values live at that instruction

- 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 case
Top-down Versus Bottom-up Allocation

Top-down allocator

- Work from external notion of what is important
- Assign registers in priority order
- Save some registers for the values relegated to memory

Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Handle all values uniformly

You will implement one of each
Top-down Allocator

The idea:
- Keep busiest values in a register
- Use the 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

Seem familiar?
- C’s register declaration
- Common technique of 60’s and 70’s (You will implement a variant)
**Bottom-up Allocator**

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 used farthest in the future
- Prefer clean value to dirty value
- Sound familiar? Think page replacement ...
**Bottom-up Allocator**

This algorithm should sound familiar

Decade algorithm

→ Sheldon Best, 1955, for Fortran I
→ Laslo Belady, 1965, for paging studies
→ William Harrison, 1975, in ECS compiler work
→ Chris Fraser, 1989, in the LCC compiler
→ Anonymous student, 1995, COMP 412 at Rice
→ Vincenzo Liberatore, 1997, Rutgers

• It will be reinvented again
• Many authors have argued for its optimality
Programming Project

The task:

• Implement a version of the top-down allocator
  (see question 2 on page 774)
• Implement a version of the bottom-up allocator
• Run them on a collection of test blocks
• Write up a brief report
  > Describe the experience
  > Compare the two allocators

Test & report blocks will be available from the web site