Low-Level Issues

Last lecture
- Liveness analysis
- Register allocation

Today
- More register allocation

Wednesday
- Common subexpression elimination for PA2

Logistics
- PA1 is due
- PA2 has been posted
- Monday the 15th, no class due to LCPC in Oregon

Register Allocation

Problem
- Assign an unbounded number of symbolic registers to a fixed number of architectural registers
- Simultaneously live data must be assigned to different architectural registers

Goal
- Minimize overhead of accessing data
  - Memory operations (loads & stores)
  - Register moves
Graph Coloring Approach to Register Allocation

Main Steps

– Create an interference graph
– Use a heuristic to color the interference graph
– Each color represents a register

Interference Graph Example (Variables)
Computing the Interference Graph

Use results of live variable analysis

for each symbolic-register/temporary \( t_i \) do
  for each symbolic-register/temporary \( t_j \) (\( j < i \)) do
    for each \( \text{def} \in \{ \text{definitions of } t_i \} \) do
      if \( t_j \) is live out at \( \text{def} \) then
        \( E \leftarrow E \cup (t_i,t_j) \)

Options
– treat all instructions the same
– treat MOVE instructions special
– which is better?

Allocating Registers Using the Interference Graph

\( K \)-coloring
– Color graph nodes using up to \( k \) colors
– Adjacent nodes must have different colors

Allocating to \( k \) registers \( \equiv \) finding a \( k \)-coloring of the interference graph
– Adjacent nodes must be allocated to distinct registers

But . . .
– Optimal graph coloring is NP-complete
  – Optimal register allocation is NP-complete, too (must approximate)
  – What if we can’t \( k \)-color a graph? (must spill)
Register Allocation: Spilling

If we can’t find a \( k \)-coloring of the interference graph

- Spill variables (nodes) until the graph is colorable

Choosing variables to spill

- Choose arbitrarily or
- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!

Spilling (Original CFG and Interference Graph)
Simple Greedy Algorithm for Register Allocation

\[
\text{for each } n \in N \text{ do} \quad \{ \text{select } n \text{ in decreasing order of weight} \}
\]
\[
\text{if } n \text{ can be colored then}
\]
\[
\text{do it} \quad \{ \text{reserve a register for } n \} \]
\[
\text{else}
\]
\[
\text{Remove } n \text{ (and its edges) from graph} \quad \{ \text{allocate } n \text{ to stack (spill)} \}
\]
Example

Attempt to 3-color this graph (  ,  ,  )

What if you use a different order?

Example

Attempt to 2-color this graph (  ,  )
Concepts from last week and today so far

Liveness
– Used in register allocation
– Generating liveness
– Flow and direction
– Data-flow equations and analysis

Control flow graph terminology

Register allocation
– scope of allocation
– granularity: what is being allocated to a register
– order that allocation units are visited in matters in all heuristic algorithms

Global approach: greedy coloring

Improvement #1: Simplification Phase [Chaitin 81]

Idea
– Nodes with \(< k\) neighbors are guaranteed colorable

Remove them from the graph first
– Reduces the degree of the remaining nodes

Must spill only when all remaining nodes have degree \( \geq k\)

Referred to as pessimistic spilling
Algorithm [Chaitin81]

\[
\textbf{while} \text{ interference graph not empty} \quad \textbf{do} \\
\quad \textbf{while} \exists \text{ a node } n \text{ with } < k \text{ neighbors} \quad \textbf{do} \\
\qquad \text{Remove } n \text{ from the graph} \\
\qquad \text{Push } n \text{ on a stack} \\
\quad \textbf{if} \text{ any nodes remain in the graph } \textbf{then} \quad \{ \text{ blocked with } \geq k \text{ edges } \} \\
\qquad \text{Pick a node } n \text{ to spill} \quad \{ \text{ lowest spill-cost or } \} \\
\qquad \text{Add } n \text{ to spill set} \quad \{ \text{ highest degree } \} \\
\quad \text{Remove } n \text{ from the graph} \\
\textbf{if} \text{ spill set not empty } \textbf{then} \\
\qquad \text{Insert spill code for all spilled nodes} \quad \{ \text{ store after def; load before use } \} \\
\qquad \text{Reconstruct interference graph } \& \text{ start over} \\
\textbf{while} \text{ stack not empty } \textbf{do} \\
\qquad \text{Pop node } n \text{ from stack} \\
\qquad \text{Allocate } n \text{ to a register} \\
\]
The Problem: Worst Case Assumptions

Is the following graph 2-colorable?

Clearly 2-colorable
- But Chaitin’s algorithm leads to an immediate block and spill
- The algorithm assumes the worst case, namely, that all neighbors will be assigned a different color
**Improvement #2: Optimistic Spilling [Briggs 89]**

**Idea**
- Some neighbors might get the same color
- Nodes with \( k \) neighbors **might** be colorable
- Blocking does not imply that spilling is necessary
  - Push blocked nodes on stack (rather than place in spill set)
  - Check colorability upon popping the stack, when more information is available

**Algorithm [Briggs et al. 89]**

```
while interference graph not empty do
  while \( \exists \) a node \( n \) with \(< k \) neighbors do
    Remove \( n \) from the graph
    Push \( n \) on a stack
    if any nodes remain in the graph then \( \{ \) blocked with \( \geq k \) edges \( \} \)
      Pick a node \( n \) to spill \( \{ \) lowest spill-cost/highest degree \( \} \)
      Push \( n \) on stack
      Remove \( n \) from the graph
  while stack not empty do
    Pop node \( n \) from stack
    if \( n \) is colorable then \( \{ \) make decision \( \} \)
      Allocate \( n \) to a register
    else \( \{ \) Store after def; load before use \( \} \)
      Insert spill code for \( n \)
    Reconstruct interference graph & start over
```
**Example**

Attempt to 2-color this graph ( , )

* blocked node

---

**Improvement #3: Coalescing**

**Move instructions**
- Code generation can produce unnecessary move instructions
  - mov t1, t2
- If we can assign t1 and t2 to the same register, we can eliminate the move

**Idea**
- If t1 and t2 are not connected in the interference graph, coalesce them into a single variable

**Problem**
- Coalescing can increase the number of edges and make a graph uncolorable
- Limit coalescing to avoid uncolorable graphs
Coalescing Logistics

Rule
– When building the interference graph, do NOT make virtual registers interfere due to copies.
– If the virtual registers s1 and s2 do not interfere and there is a copy statement s1 = s2 then s1 and s2 can be coalesced.
– Example

<table>
<thead>
<tr>
<th>Before Coalescing</th>
<th>After Coalescing</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = t + u</td>
<td>ab = t + u</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>b = a</td>
<td>...</td>
</tr>
<tr>
<td>c = a</td>
<td>c = ab</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x = b + w</td>
<td>x = ab + w</td>
</tr>
<tr>
<td>z = c + y</td>
<td>z = c + y</td>
</tr>
</tbody>
</table>

Register Allocation: Spilling

If we can’t find a k-coloring of the interference graph
– Spill variables (nodes) until the graph is colorable

Choosing variables to spill
– Choose least frequently accessed variables
– Break ties by choosing nodes with the most conflicts in the interference graph
– Yes, these are heuristics!
More on Spilling

Chaitin’s algorithm restarts the whole process on spill
– Necessary, because spill code (loads/stores) uses registers
– Okay, because it usually only happens a couple times

Alternative
– Reserve 2-3 registers for spilling
– Don’t need to start over
– But have fewer registers to work with

Weighted Interference Graph

Goal
– Weight(s) = \sum_{\forall \text{references } r \text{ of } t} f(r)\ f(r) \text{ is execution frequency of } r

Static approximation
– Use some reasonable scheme to rank variables
– Some possibilities
  – Weight(t) = num of times \ t \ is used in program
  – Weight(t) = 10 \times (# \text{ uses in loops}) + (# \text{ uses in straightline code})
  – Weight(t) = 20 \times (# \text{ uses in loops}) + 2 \times (# \text{ uses in straightline code}) +
    (# \text{ uses in a branch statement})
Register Allocation and Procedure Calls

Problem
– Register values may change across procedure calls
– The allocator must be sensitive to this

Two approaches
– Work within a well-defined calling convention
– Use interprocedural allocation (not covering this)

Calling Conventions

Goals
– Fast calls (pass arguments in registers, minimal register saving/restoring)
– Language-independent
– Support debugging, profiling, garbage collection, etc.

Complicating Issues
– Varargs
– Passing/returning aggregates
– Exceptions, non-local returns
  – setjmp() / longjmp()
Partition registers into two categories
– Caller-saved
– Callee-saved

Caller-saved registers
– Caller must save/restore these registers when live across call
– Callee is free to use them

Example
```c
foo()
{
    r_caller = 4
    save r_caller
    goo()
    restore r_caller
}
```

Callee-saved registers
– Callee must save/restore these registers when it uses them
– Caller expects callee to not change them

Example
```c
foo()
{
    r_callee = 4
    goo()
    r_callee?
}
```
```c
goo()
{
    save r_callee
    r_callee = 99
    restore r_callee
    goo () promises not to modify r_callee
}
```
Architectures with Callee and Caller Registers

**SPARC**
- hardware-saved %i0-%i7, %o0-%o8

**Alpha**
- 7 callee-saved out of 32 registers

**MIPS**
- caller-saved: $t0-$t9, $a0-$a3, $v0-$v1
- callee-saved: $s0-$s7, $ra, $fp

**PPC**
- 18 callee-saved
- 14 caller-saved

**StarCore EABI**
- 4 callee-saved
- 28 caller-saved

Register Allocation and Calling Conventions

**Insensitive register allocation**
- Save all live caller-saved registers before call; restore after
- Save all used callee-saved registers at procedure entry; restore at return
- Suboptimal
  ```
  foo ()
  {
      t = ...  // A variable that is not live across calls should go in
      ... = t  // callee-saved registers
      s = ...
  
      f ()  // A variable that is live across multiple calls should
      g ()  // go in callee-saved registers
      ...
  }
  ```

**Sensitive register allocation**
- Encode calling convention constraints in the IR and interference graph
- How? Use precolored nodes
Precolored Nodes

Add architectural registers to interference graph
- Precolored (mutually interfering)
- Not simplifiable
- Not spillable

Express allocation constraints
- Integers usually can’t be stored in floating point registers
- Some instructions can only store result in certain registers
- Caller-saved and callee-saved registers . . .

Precolored Nodes and Calling Conventions

Callee-saved registers
- Treat entry as def of all callee-saved registers
- Treat exit as use of them all
- Allocator must “spill” callee-saved registers to use them (really temp)

```
foo ()
{
    def (r3)
    t1 = r3

    r3 = t1
    use (r3)
}
```

Live range of callee-saved registers

Caller-saved registers
- Variables live across call interfere with all caller-saved registers
Example

```plaintext
defo(\texttt{r3})
t1 := r3
a := ...
b := ...

... a ...
call \texttt{goo}
... b ...
r3 := t1
use(r3)
return
```

\textbf{r1, r2} caller-saved
\textbf{r3} callee-saved

Tradeoffs

\textbf{Callee-saved registers}
\begin{itemize}
\item Decreases code size: one procedure body may have multiple calls
\item Small procedures tend to need fewer registers than large ones; callee-save makes sense because procedure sizes are shrinking
\item May increase execution time: For long-lived variables, may save and restore registers multiple times, once for each procedure call, instead of a single end-to-end save/restore
\end{itemize}

\textbf{The larger “problem”}
\begin{itemize}
\item We’re making local decisions for policies that require global information
\end{itemize}
Concepts

Decision tree for register allocation

Global approaches centered around an interference graph
- greedy coloring
- coloring with simplification [Chaitin]
- coloring with simplification and optimistic spilling [Briggs]
- coloring with simplification, coalescing, and optimistic spilling

Register allocation across procedure calls
- precolored nodes in the interference graph