Register Allocation (wrapup) &
Code Scheduling

CS2210
Lecture 22

Constructing and Representing
the Interference Graph

- Construction alternatives:
  - as side effect of live variables analysis (when variables are units of allocation)
  - compute du-chains & webs (or ssa form), do live variables analysis, compute interference graph
- Representation
  - adjacency matrix: \( A_{min(i,j), \max(i,j)} = true \) iff (symbolic) register \( i \) adjacent to \( j \) (interferes)

Adjacency List

- Used in actual allocation
- \( A[i] \) lists nodes adjacent to node \( i \)
- Components
  - color
  - disp: displacement (in stack for spilling)
  - spcost: spill cost (initialized 0 for symbolic, infinity for real registers)
  - nints: number of interferences
  - adjnds: list of real and symbolic registers currently interfere with \( i \)
  - rmvadj: list of real and symbolic registers that interfered with \( i \) but have been removed
Register Coalescing

- Goal: avoid unnecessary register to register copies by coalescing register
  - ensure that values are in proper argument registers before procedure calls
  - remove unnecessary copies introduced by code generation from SSA form
  - enforce source / target register constraints of certain instructions (important for CISC)
- Approach:
  - search for copies $s_j := s_i$ where $s_i$ and $s_j$ do not interfere (may be real or symbolic register copies)

Computing Spill Costs

- Have to spill values to memory when not enough registers can be found (can’t find k-coloring)
- Why webs to spill?
  - least frequently accessed variables
  - most conflicting
- Sometimes can rematerialize instead:
  - = recompute value from other register values instead of store / load into memory (Briggs: in practice mixed results)

Spill Cost Computation

\[
\text{defwt} \times \sum 10^{\text{depth(def)}} + \text{usewt} \times \sum 10^{\text{depth(use)}} - \text{copywt} \times \sum 10^{\text{depth(copy)}}
\]

- defwt / usewt / copywt costs relative weights assigned to instructions
- def, use, copy are individual definitions /uses/ copies
- frequency estimated by loop nesting depth
Coloring the Graph

Graph Pruning

Improvement #1 (Chaitin, 1982)
- Nodes with <k edges can be colored after all other nodes and still be guaranteed registers
- So remove <k-degree nodes first
  - this may reduce the degree of remaining graph and make it colorable

Algorithm

```plaintext
while interference graph not empty
  while there is a node with <k neighbors
    remove it from graph, push on stack
  if all remaining nodes have >= k neighbors, then blocked:
    pick a node to spill (lowest cost)
    remove from graph, add to spill set
  if any nodes in spill set:
    insert spill codes for all spilled nodes, reconstruct interference graph and start over
  while stack not empty
    pop node from stack, allocate to register
```
Coloring the Graph with pruning

![Graph Diagram]

1. Assume 3 registers available
2. Assume 2 registers available

An Annoying Case

![Graph Diagram]

If only 2 registers available: blocked must spill!

Improvement #2: blocked != spill (Briggs et al. 1989)

- Idea: just because node has k neighbors doesn’t mean it will be spilled (neighbors can have overlapping colors)
- Algorithm: like Chaitin, except
  - when removing blocked node, just push onto stack (“optimistic spilling”)
  - when done removing nodes, pop nodes off stack and see if they can be allocated
  - really spill if cannot be allocated at this stage
Improvement #3: Priority-based Coloring (Chow & Hennessy 1984)

- **Live-range splitting**
  - when variable cannot be register-allocated, split into multiple subranges that can be allocated separately
  - move instructions inserted at split points
  - some live ranges in registers, some in memory
    - selective spilling
- Based on **variable live ranges**
  - can result in more conservative interference graph than webs
  - live range = set of basic blocks a variables is live in

---

Live Range Example

```plaintext
x := a+b
y := x+c
if y = 0 goto L1
z := y+d
w := z
L1:
variables x,y,z,w
```

---

Improvement #4: Rematerialization

- Idea: instead of reloading value from memory, **recompute** instead if recomputation cheaper than reloading
- Simple strategy choose rematerialization over spilling if
  - can recompute a value in a single instruction, and
  - all operands will always be available
- Examples
  - constants, address of a global, address of variable in stack frame
Evaluation

- Rematerialization
  - showed a reduction of -26% to 33% in spills
- Optimistic spilling
  - showed a reduction of -2% to 48%
- Priority-based coloring
  - may often not be worthwhile
  - appears to be more expensive in practice than optimistic spilling

Instruction Scheduling

Reading & Topics

- Chapter 17
- Topics
  - Instruction scheduling methods
    - list scheduling
    - software pipelining
    - trace scheduling
    - scheduling-related optimizations (e.g., loop unrolling and variable expansion)
Instruction Scheduling

- Goal: reorder instructions to fit processor pipeline better
  - important for pipelined processors (almost all)
  - especially important for superscalar processors

Phase Ordering Problem

- Should register allocation follow instruction scheduling or vice versa?
  - allocation first: constrains scheduler (dependences created by allocation)
  - scheduling first: may increase register pressure

In practice:
- 1st scheduling pass on symbolic registers
  - preceded by a resource limiting pass (e.g., special address calculation instructions, like s4addq), coalescing
  - register allocation
  - 2nd scheduling pass (if spills happened)

List Scheduling

- Schedule instructions in a basic block

Approach:
- topological sort of instructions (based on data dependences)
  - have to construct dependence DAG
- choose order with minimal execution time of basic block
List Scheduling Example

\[
\begin{align*}
r_8 &:= [r_{12}+8](4) \\
r_1 &:= r_8 + 1 \\
r_2 &:= 2 \\
call & r_{14}, r_{31} \\
nop & \\
r_9 &:= r_1 + 1
\end{align*}
\]

Algorithm

- Compute delay for all nodes in postorder
  - \(\text{delay}(n) = \text{ExecTime}(n)\) if \(n\) a leaf node else max \(\text{Late Delay}(n,m)\) for all \(m\) successors of \(n\) in \(\text{DAG}\)
  - \(\text{Late Delay}(n,m) = \text{Latency}(n,m) + \text{delay}(m)\)
- Use a "clock" \(\text{curTime}\) and find which instructions are schedulable at that time
  - schedule a node at \(\text{Etime}(n)\) the earliest time it is schedulable

Example

<table>
<thead>
<tr>
<th>Node</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

execTime(6) = 2
execTime(1-5) = 1
Software Pipelining

Goal: improve loop execution performance by taking advantage of ILP capabilities in processors (superscalar or VLIW)

Approach:
- overlap execution of instructions from different loop iterations

Implementing Software Pipelining

Window scheduling
- create two copies of dependence DAG, connect and slide window across

Unroll and compact
- produce multiple loop bodies and search for repeating pattern

Loop Unrolling

Unroll loop by unrolling factor $n$: replace loop body by $n$ copies and adjust loop control code accordingly
- original loop called rolled loop

Benefits:
- expose more ILP
- cross-iteration optimization opportunities (e.g., CSE)
- reduce loop overhead (fewer control instructions per loop body instructions)
Example

\[
\text{do } i = l_0, h_i \\
\text{ body } \\
\text{ enddo}
\]

\[
\text{do } i = l_0, h_i - u + 1, u \\
\text{ body } \\
\text{ body } / i + 1 / i \\
\text{ ... } \\
\text{ body } / i + u - 1 / i \\
\text{ enddo} \\
\text{ do } j = i, h_j \\
\text{ body } / j / i \\
\text{ enddo}
\]

CS2210 Compiler Design 2004/5

Loop Unrolling for Non-counting Loops

\[
\text{while } (a[i] > 0) \\n\text{ b[i] = c[i] + a[i]; } \\
\text{ i++; } \\
\}
\]

\[
\text{while } (a[i] > 0) \\n\text{ b[i] = c[i]+a[i]; } \\
\text{ i++; } \\
\text{ if } (a[i] <= 0) \text{ break; } \\
\text{ b[i]=d[i]+a[i]; } \\
\text{ i++; } \\
\text{ if } (a[i] <= 0) \text{ break; } \\
\text{ b[i] = c[j]+a[i]; } \\
\text{ i++; } \\
\text{ if } (a[i] <= 0) \text{ break; } \\
\text{ b[i] = c[l]+a[i]; } \\
\text{ i++; } \\
\}
\]

\[\text{can overlap addition with load of a[i] (given non-aliasing)}\]

CS2210 Compiler Design 2004/5

Variable Expansion

- Transformation in conjunction with loop unrolling:
  - expand variables used in loop body to n copies
  - combine values at end of loop
  - reduces # of dependences
    - most useful for loop induction variables, accumulator variables

CS2210 Compiler Design 2004/5
Variable Expansion Example

\[
\begin{align*}
\text{acc} & := 10 \\
\text{max} & := 0 \\
\text{imax} & := 0 \\
\text{for } i & := 1 \text{ to } 100 \text{ do} \\
\text{acc} & := \text{acc} + a[i] \times b[i] \\
\text{if } (a[i] > \text{max}) \text{ then} \\
\text{max} & := a[i]; \\
\text{imax} & := i; \\
\text{endif} \\
\text{endfor}
\end{align*}
\]

Trace Scheduling

- Schedule sequence of basic blocks at a time
  - to find large ILP (e.g., for VLIW machines)
- Trace = sequence of instructions including branches (w/o backedges)
  - multiple entry & multiple exit
  - use compensation code for side entries and exits
- Trace formation = identify sequence of basic blocks likely to result in good schedule
  - usually based on profiling
  - pioneered by the Multiflow compiler (Fisher 1981)