Computer Systems Organization

CSE 521/560M
Lecture 13
Prof. Patrick Crowley

Plan for Today

- Distribute HW2
- Questions
- Today’s discussion
Finish Lecture 12

Detecting Loop-level Parallelism

- Easiest to find in source code rather than assembly
- Compiler must recognize
  - Loops
  - Array references
  - Induction variable computations (for index calculations)
- Compiler steps (all inexact, yield ‘maybe’ info)
  - Find loop-carried dependences
  - Give up on all but affine array index calculations
    • $a[i] + b$, with $a, b$ constant and $i$ loop index variable
  - Schedule provably independent instructions
- Pointers and indirect references are difficult
- Strongly typed languages (e.g., Java) are easier
Software Pipelining

- Build a new loop body consisting of instructions from different iterations of the original loop
  - Creates a loop with independent instructions
  - Does not increase code size
- Can be used along with loop unrolling
Example

Loop:  L.D  F0,0(R1)  ADD.D  F4,F0,F2  S.D  F4,0(R1)  DADDUI  R1,R1,#-8  BNE  R1,R2,Loop
       Iteration i: L.D  F0,0(R1)  ADD.D  F4,F0,F2  S.D  F4,0(R1)  DADDUI  R1,R1,#-8  BNE  R1,R2,Loop
       Iteration i+1: L.D  F0,0(R1)  ADD.D  F4,F0,F2  S.D  F4,0(R1)  DADDUI  R1,R1,#-8  BNE  R1,R2,Loop
       Iteration i+2: L.D  F0,0(R1)  ADD.D  F4,F0,F2  S.D  F4,0(R1)  DADDUI  R1,R1,#-8  BNE  R1,R2,Loop

Example (cont’d)

<table>
<thead>
<tr>
<th>Original code</th>
<th>Software Pipelined Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: L.D F0,0(R1)</td>
<td>Loop: S.D F4,16(R1) ; store M[i]</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2 ; add to M[i-1]</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>L.D F0,0(R1) ; loads M[i-2]</td>
</tr>
<tr>
<td>DADDUI R1,R1,#-8</td>
<td>DADDUI R1,R1,#-8</td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td>BNE R1,R2,Loop</td>
</tr>
</tbody>
</table>
Additional Considerations

- Startup and clean up code are required to replace missing instructions
  - Before first iteration
  - After last $n$ iterations
- Register allocation can be difficult
  - e.g., if values must be passed through several iterations (our example used all values in the next iteration)

Global Code Scheduling

- So far, our code scheduling has been within one (sometimes large) basic block
  - Local code scheduling
- What if the loop body had internal control flow?
  - Would require code movement across control flow instructions
- Moving code across branches is *global code scheduling*
  - Must preserve both data and control dependences
Global Code Scheduling

- Objective is unchanged: try to reduce code sequence to smallest, fastest schedule
- Conditional (non-loop) branches represent a decision
  - Must know/assume path frequencies, especially the critical path
  - No improvement guarantees
- Finding path frequencies
  - Programmer annotation
  - Profiling
- Cannot change program semantics, only program performance
  - Such code movement can be speculative, and we can only move exception-generating instructions (e.g., memory instructions) if we have hardware support for speculative exceptions

Example

© 2003 Elsevier Science (USA). All rights reserved.
Example

- How can we raise the store to B?
- How can we raise the store to C?
- Tradeoffs are complex
  - How much will these help?
  - Cost of compensation code?
- Instead rely on simple, focused approaches

```
L.D R4,0(R1); load A
L.D R5,0(R1); load B
ADDDU R4,R4,R5; Add to A
SD R4,0(R1); Store A
...
BNEZ R4,elsepart; Test A
...
; Then part
SD ...,O(R2); Stores to B
...
J join ; jump else
elsepart:...

X ; code for X
...
join: ...

; after if
SD ...,O(R3); after C[1]
```

Trace Scheduling

- Idea: schedule across basic blocks by focusing only on the critical path
  - Relies on accurate profile information
- Method
  - Trace selection
    - Identify the most frequent path, i.e., the critical path
    - Use loop unwrapping to generate trace
    - Use profile-driven static branch prediction to construct the straight-line code
  - Trace compaction
    - Reduce trace size via code scheduling
    - Organize instructions into groups for VLIW
    - Must insert book-keeping code when instructions are moved past trace entry and exit points
    - Directive: Move code if it makes the main trace faster!
  - Repeat for the next most likely path until whole program is scheduled
Trace Example

Superblocks

- Address trace weaknesses
  - Complexity consequences of trace entries and exits
- Idea: A superblock can have multiple exits but only one entry point
  - Simplifies compaction since only exits need attention when moving code
- Each unrolled loop iteration creates an exit
- A secondary exit loop performs any remaining iterations
  - Code size might be larger than original trace-based approach
Superblock Example

Assignment

- Readings
  - For Monday
    • H&P: 4.5-4.6
    • Ash: Skim Ch. 8, Read Ch. 9
  - For Wednesday
    • H&P: 6.9
    • H&P: 5.1-5.2
    • Ash: Ch. 10