Plan for Today

• Announcements
  – PM2 due next Monday
  – Wash U Tech Entrepreneurs meeting tonight
  – No lecture on Wednesday; meet in groups to finish PM2

• Questions

• Today’s discussion
  – Compiler techniques for exposing ILP
Interested in Entrepreneurship?

Join **WUTE**: the WashU Tech Entrepreneurs!

- All students, faculty, staff are welcome

Next meeting **Tonight @ 6PM**

- WashU alum Kevin Haar will talk about his cloud computing startup, Appistry
- Refreshments provided!

1. Join us on 10/26 @ 6PM in McDonnell 162!
2. Details w/ map: [wute.wustl.edu](http://wute.wustl.edu)
3. (Optionally) RSVP at Facebook
Compiler techniques for exposing ILP

- Loop unrolling and scheduling
- Static branch prediction
- Software pipelining
- Trace Scheduling
- Superblock
Loop unrolling & scheduling

for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;

Loop:  L.D   F0,0(R1)
       ADD.D  F4,F0,F2
       S.D   F4,0(R1)
       DADDUI R1,R1, #−8
       BNE   R1,R2,Loop

Assumptions: latencies
- F-P Op → F-P Op: 3
- F-P Op → S.D: 2
- L.D → F-P Op: 1
- L.D → S.D: 0
- Branch penalty: 1

Loop:  L.D   F0,0(R1)
       stall
       ADD.D  F4,F0,F2
       stall
       stall
       S.D   F4,0(R1)
       stall
       DADDUI R1,R1, #−8
       stall
       BNE   R1,R2,Loop
       stall
Loop unrolling & scheduling (2)

Step 1: scheduling

Loop:

- L.D  F0,0(R1)  
- **stall**  
- ADD.D  F4,F0,F2  
- **stall**  
- **stall**  
- S.D  F4,0(R1)  
- DADDUI  R1,R1, #-8  
- **stall**  
- BNE  R1,R2,Loop  
- **stall**

Loop:

- L.D  F0,0(R1)  
- DADDUI  R1,R1, #-8  
- ADD.D  F4,F0,F2  
- **stall**  
- BNE  R1,R2,Loop  
- S.D  F4, 8(R1)
Loop unrolling & scheduling (3)

Step 2: unrolling

Loop:

L.D F0,0(R1)
DADDUI R1,R1, #-8
ADD.D F4,F0,F2

stall
BNE R1,R2,Loop
S.D F4,8(R1)

branch & overhead

Loop:

L.D F0,0(R1) ; (i)
ADD.D F4,F0,F2
S.D F4,0(R1)
L.D F6,−8(R1); (i+1)
ADD.D F8,F6,F2
S.D F8,−8(R1)
L.D F10,−16(R1); (i+2)
ADD.D F12,F10,F2
S.D F12,−16(R1)
DADDUI R1,R1, #−24
BNE R1,R2,Loop

branch & overhead
Loop unrolling & scheduling (4)

Step 3: scheduling

Loop:
```
L.D    F0,0(R1); (i)
ADD.D  F4,F0,F2
S.D    F4,0(R1)
L.D    F6,-8(R1); (i+1)
ADD.D  F8,F6,F2
S.D    F8,-8(R1)
L.D    F10,-16(R1); (i+2)
ADD.D  F12,F10,F2
S.D    F12,-16(R1)
DADDUI R1,R1, #-24
BNE    R1,R2,Loop
```

Loop:
```
L.D    F0,0(R1)
L.D    F6,-8(R1)
L.D    F10,-16(R1)
ADD.D  F4,F0,F2
ADD.D  F8,F6,F2
ADD.D  F12,F10,F2
S.D    F4,0(R1)
S.D    F8,16(R1)
DADDUI R1,R1, #-24
BNE    R1,R2,Loop
S.D    F12,8(R1)
```
Loop-carried dependences

• Loop level parallelism
  – Instructions belonging to different loop iterations are parallelizable

• Loop-carried dependences
  – Data accessed in a *later iteration* is dependent on data value produced in *earlier iteration*
  – *Successive iterations* of the loop must execute *in series*
Loop-carried dependences (2)

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Dependences?
  - $x[i]$ not loop-carried
  - $i$ loop-carried BUT on the induction variable: can be eliminated

- Loop parallelizable
Loop-carried dependences (3)

for (i=1; i<100; i=i+1){
    A[i+1] = A[i] + C[i];    /*S1*/
    B[i+1] = B[i] + A[i+1];    /*S2*/
}

• Dependences?
  – i: loop-carried BUT on the induction variable
  – S1: A, loop-carried
  – S2: B, loop-carried
  – S2: A, not loop-carried

• Loop carried, circular dependences
  – Total ordering on the statements: successive iterations executed in series
Detecting Loop-level Parallelism

• Easier to find in source code than in assembly

• Compiler must recognize
  – Loops
  – Array references
  – Induction variable computations (for index calculations)

• Compiler steps (all are inexact and 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)

Iteration i+1:  
L.D  F0,0(R1)
ADD.D  F4,F0,F2
S.D  F4,0(R1)

Iteration i+2:  
L.D  F0,0(R1)
ADD.D  F4,F0,F2
S.D  F4,0(R1)

Loop overhead

Loop:  
L.D  F0,0(R1)
stall
ADD.D  F4,F0,F2
stall
stall
stall
S.D  F4,0(R1)
stall
DADDUI  R1,R1,#-8
stall
BNE  R1,R2,Loop
stall
Example (2)

<table>
<thead>
<tr>
<th>iteration $i$</th>
<th>iteration $i+1$</th>
<th>iteration $i+2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F0,0(R1)</td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F4,0(R1)</td>
<td>S.D F4,0(R1)</td>
</tr>
</tbody>
</table>

software pipelined iteration
### Example (cont’d)

#### Original code

<table>
<thead>
<tr>
<th>iteration i</th>
<th>iteration i+1</th>
<th>iteration i+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F0,0(R1)</td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F4,0(R1)</td>
<td>S.D F4,0(R1)</td>
</tr>
</tbody>
</table>

Loop:

- L.D F0,0(R1)
- ADD.D F4,F0,F2
- S.D F4,0(R1)
- DADDUI R1,R1,#-8
- BNE R1,R2,Loop

#### Software Pipelined Version

<table>
<thead>
<tr>
<th>iteration i+1</th>
<th>iteration i+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.D F4,16(R1)</td>
<td>L.D F0,0(R1)</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F4,0(R1)</td>
</tr>
</tbody>
</table>

Loop:

- S.D F4,16(R1) ; store M[i]
- ADD.D F4,F0,F2 ; add to M[i-1]
- L.D F0,0(R1) ; loads M[i-2]
- DADDUI R1,R1,#-8
- BNE R1,R2,Loop
## Start-up code

### Iteration 1
- L.D F0,0 (R1)
- ADD.D F4, F0, F2
- S.D F4, 0 (R1)

### Iteration 2
- L.D F0, 0 (R1)
- ADD.D F4, F0, F2

### Iteration 3
- L.D F0, 0 (R1)
- ADD.D F4, F0, F2
- S.D F4, 0 (R1)

## Software Pipelined Version

**Loop:**
- S.D F4, 16 (R1)
- ADD.D F4, F0, F2
- L.D F0, 0 (R1)
- DADDUI R1, R1, #8
- BNE R1, R2, Loop
- L.D F0, 16 (R1)
- ADD.D F4, F0, F2
- LD F0, 8 (R1)
## Finish-up code

### iteration n-2

- **L.D** F0,0(R1)
- **ADD.D** F4,F0,F2
- **S.D** F4,0(R1)

### iteration n-1

- **L.D** F0,0(R1)
- **ADD.D** F4,F0,F2
- **S.D** F4,0(R1)

### iteration n

- **L.D** F0,0(R1)
- **ADD.D** F4,F0,F2
- **S.D** F4,0(R1)

## Software Pipelined Version

**Loop:**
- **S.D** F4,16(R1)
- **ADD.D** F4,F0,F2
- **L.D** F0,0(R1)
- **DADDUI** R1,R1,#-8
- **BNE** R1,R2,Loop

## Finish-up code

- **S.D** F4,16(R1)
- **ADD.D** F4,F0,F2
- **S.D** F4,8(R1)
Additional Considerations

• Register allocation can be difficult
  – e.g., if values must be passed through several iterations
    (our example used all values in the next iteration)

• Software pipelining vs. loop unrolling
  – Software pipelining: less code space
  – Overhead:
    • Loop unrolling: branch and counter updates less frequent
    • Software pipelining: the loop does not run at the peak (because
      of stalls) only at the beginning and at the end
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

L.D R4,0(R1); load A
L.D R5,0(R1); load B
DADDU R4,R4,R5; Add to A
SD R4,0(R1); Store A
...
BNEZ R4,elsepart; Test A
...
; Then part
SD ...,0(R2); Stores to B
...
J join; jump else
elsepart:...; else part
X ; code for X
...
join: ...; after if
SD ...,0(R3); store C[i]

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

• How can we raise the store to B above BNEZ?
  L.D  R4,0(R1); load A
  L.D  R5,0(R1); load B
  DADDU R4,R4,R5; Add to A
  SD   R4,0(R1); Store A

• How can we raise the store to C above BNEZ?
  BNEZ R4,elsepart; Test A
  ... ; Then part
  SD   ...,O(R2); Stores to B

• Tradeoffs are complex
  – How many options can the compiler consider?
  – How much will these help?
  – Cost of compensation code?
  J     join ; jump else
  elsepart:... ; else part
  X     ; code for X
  ...  ; after if
  join:  ...
  SD   ...,O(R3); store C[i]

• Instead rely on simple, focused approaches
Trace Scheduling

- Idea: schedule across basic blocks by focusing only on the critical path
  - Relies on accurate profile information
  - View this as a process to guide global code scheduling

- Method
  - Trace selection
    - Identify the most frequent path, i.e., the critical path
    - Use loop unrolling to generate trace
    - Use profile-driven static branch prediction to construct the straight-line code
  - Trace compaction
    - Reduce trace size via code scheduling
      - For VLIW machines, organize instructions into groups
    - 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
Assignment

• Readings
  – For Wednesday (no lecture; meet in groups)
    • H&P: Read C.1-C.2, 5.1
  – For Monday
    • PM2 Due