CMSC 411
Computer Systems Architecture
Lecture 8
Instruction Level Parallelism 1
(Compiler Techniques)

Outline

• Instruction-level parallelism (ILP)
• Compiler techniques to increase ILP
  – Loop Unrolling
• Branch Prediction
• Dynamic Scheduling
  – Tomasulo Algorithm
Recall from Pipelining

• Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  – Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  – Structural hazards: HW cannot support this combination of instructions
  – Data hazards: Instruction depends on result of prior instruction still in the pipeline
  – Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Instruction-Level Parallelism

• Instruction-Level Parallelism (ILP)
  – Overlap the execution of instructions to improve performance
• 2 approaches to exploit ILP
  1. Rely on hardware to help discover and exploit the parallelism dynamically
     – Pentium 4, AMD Opteron, IBM Power
  2. Rely on software technology to find parallelism, statically at compile-time
     – Itanium 2 / IA-64
Instruction-Level Parallelism (cont.)

- Determining dependences is critical
- If 2 instructions are
  - Parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - Dependent, they are not parallel and must be executed in order, although they may often be partially overlapped
- Types of dependences
  - Data
    » RAW (true/flow dependences)
    » WAR (anti-dependence)
    » WAW (output dependence)
  - Control

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - Branches make flow dynamic, determine which instruction is supplier of data
- Example:

  DADDU R1, R2, R3
  BEQZ R4, L
  DSUBU R1, R5, R6

L: ...
OR R7, R1, R8

- OR depends on DADDU or DSUBU?
  Must preserve data flow on execution
Instruction-Level Parallelism

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - Average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- Need ILP across multiple basic blocks

Loop-Level Parallelism

- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop.
  - Example
    
    for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
  
  - Exploit loop-level parallelism by “unrolling loop” either by
    - Dynamic via branch prediction or
    - Static via loop unrolling by compiler
- Another way is vectors, to be covered later
Compiler Techniques - Example

• This code, add a scalar to a vector:
  
  \[ \text{for (i=1000; i>0; i=i-1)} \]
  \[ \text{x[i]} = \text{x[i]} + s; \]

• Assume following latencies for all examples
  – Ignore delayed branch in these examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in cycles</th>
<th>stalls between in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

FP Loop: Where are the Hazards?

• First translate into MIPS code:
  - To simplify, assume 8 is lowest address

  ```
  Loop: L.D F0,0(R1) ;F0=vector element
       ADD.D F4,F0,F2 ;add scalar from F2
       S.D 0(R1),F4 ;store result
       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
       BNEZ R1,Loop ;branch R1!=zero
  ```

  ```
  for (i=1000; i>0; i=i−1)
  \[ \text{x[i]} = \text{x[i]} + s; \]
  ```
FP Loop Showing Stalls

1 Loop: L.D  F0,0(R1) ;F0=vector element
2 stall
3 ADD.D  F4,F0,F2 ;add scalar in F2  plus branch delay!
4 stall
5 stall
6 S.D  0(R1),F4 ;store result
7 DADDUI  R1,R1,-8 ;decrement pointer 8B (DW)
8 stall ;assumes can’t forward to branch
9 BNEZ  R1,Loop ;branch R1!=zero

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

Revised FP Loop Minimizing Stalls

1 Loop:  L.D  F0,0(R1)
2 DADDUI  R1,R1,-8
3 ADD.D  F4,F0,F2
4 stall
5 stall
6 S.D  8(R1),F4; altered offset when move DADDUI
7 BNEZ  R1,Loop

Swap DADDUI and S.D by changing address of S.D

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?
Unroll Loop Four Times (straightforward way)

1 Loop:
1. L.D  F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D  0(R1),F4 ; drop DADDUI & BNEZ
4. L.D  F6,-9(R1)
5. ADD.D F8,F6,F2
6. S.D  -8(R1),F8 ; drop DADDUI & BNEZ
7. L.D  F10,-16(R1)
8. ADD.D F12,F10,F2
9. S.D  -16(R1),F12 ; drop DADDUI & BNEZ
10. L.D F14,-24(R1)
11. ADD.D F16,F14,F2
12. S.D  -24(R1),F16
13. DADDUI R1,R1,#-32 ; alter to 4*8
14. BNEZ R1,LOOP

27 clock cycles, or 6.75 per iteration
(Assumes R1 is multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is \(n\), and we would like to unroll the loop to make \(k\) copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes \((n \mod k)\) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates \((n/k)\) times
- For large values of \(n\), most of the execution time will be spent in the unrolled loop
## Unrolled Loop That Minimizes Stalls

1. Loop: 
   - L.D F0, 0(R1)
   - L.D F6, -8(R1)
   - L.D F10, -16(R1)
   - L.D F14, -24(R1)
   - ADD.D F4, F0, F2
   - ADD.D F8, F6, F2
   - ADD.D F12, F10, F2
   - ADD.D F16, F14, F2
   - S.D 0(R1), F4
   - S.D -8(R1), F8
   - S.D -16(R1), F12
   - DADDUI R1, R1, # -32
   - S.D #8(R1), F16; #8-32 = -24
   - BNEZ R1, LOOP

14 clock cycles, or 3.5 per iteration

## 5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
  1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
  2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
5 Loop Unrolling Decisions (cont.)

3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code

4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   » Transformation requires analyzing memory addresses and finding that they do not refer to the same address

5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   – Amdahl’s Law

2. Growth in code size
   – For larger loops, concern it increases the instruction cache miss rate

3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   – If not be possible to allocate all live values to registers, may lose some or all of its advantage

   • Loop unrolling reduces impact of branches on pipeline; another way is branch prediction