CPE 631 Lecture 14:
Exploiting ILP with SW Approaches (2)

Aleksandar Milenković, milenka@ece.uah.edu
Electrical and Computer Engineering
University of Alabama in Huntsville

Outline

- Basic Pipeline Scheduling and Loop Unrolling
- Multiple Issue: Superscalar, VLIW
- Software Pipelining

ILP: Concepts and Challenges

- ILP (Instruction Level Parallelism) – overlap execution of unrelated instructions
- Techniques that increase amount of parallelism
  - reduce impact of data and control hazards
  - increase processor ability to exploit parallelism
- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
  - Reducing each of the terms of the right-hand side minimize CPI and thus increase instruction throughput

Basic Pipeline Scheduling: Example

- Simple loop: for(i=1; i<=1000; i++) x[i] = x[i] + s;
- Assumptions:
  Instruction producing result | Instruction using result | Latency in clock cycles
  FP ALU op | Another FP ALU op | 3
  FP ALU op | Store double | 2
  Load double | FP ALU op | 1
  Load double | Store double | 0
  Integer op | Integer op | 0

R1 points to the last element in the array for simplicity, we assume that x[0] is at the address 0
Loop:
L.D F0, 0(R1) ;F0 = array el.
ADD.D F4, F0, F2 ;add scalar in F2
S.D O(R1), F4 ;store result
SUBI R1, R1, #8 ;decrement pointer
BNZI R1, Loop ;branch
Executing FP Loop

1. Loop: LD F0, 0(R1)
2. Stall
3. ADDD F4, F0, F2
4. Stall
5. Stall
6. SD 0(R1), F4
7. SUBI R1, R1, #8
8. Stall
9. BNEZ R1, Loop
10. Stall

10 clocks per iteration (5 stalls)

=> Rewrite code to minimize stalls?

Revised FP loop to minimize stalls

Loop: SUBI R1, R1, #8
2. ADDD F4, F0, F2
3. Stall
4. BNEZ R1, Loop
5. SD 0(R1), F4

1 clocks per iteration (1 stall); but only 3 instructions do the actual work processing the array (LD, ADD, SD)

Unroll loop 4 times to improve potential for instr. scheduling

Swap BNEZ and SD by changing address of SD SUBI is moved up

Unrolled Loop

This loop will run 28 cc (14 stalls) per iteration; each LD has one stall, each ADDD 2, SUBI 3, BNEZ 1, plus 14 instruction issue cycles - or 28/4=7 for each element of the array (even slower than the scheduled version)!

=> Rewrite loop to minimize stalls

Where are the name dependencies?

1. Loop: LD F0, 0(R1)
2. ADDD F4, F0, F2
3. SD 0(R1), F4 : drop SUBI & BNEZ
4. ADDD F4, F0, F2
5. SD -8(R1), F4 : drop SUBI & BNEZ
6. ADDD F4, F0, F2
7. ADDD F4, F0, F2
8. ADDD F4, F0, F2
9. ADDD F4, F0, F2
10. ADDD F4, F0, F2
11. ADDD F4, F0, F2
12. ADDD F4, F0, F2
13. SUBI R1, R1, #32 : alter to 4*8
14. BNEZ R1, Loop
15. NOP

How can remove them?
Where are the name dependencies?

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

Unrolled Loop that Minimise Stalls

Loop: LD F0, 0(R1) 
LD F6, -8(R1) 
LD F10, -16(R1) 
LD F14, -24(R1) 
ADDD F4, F0, F2 
ADDD F8, F6, F2 
ADDD F12, F10, F2 
ADDD F16, F14, F2 
SD 0(R1), F4 
SD -8(R1), F8 
SUBI R1, R1, #32 
SD 16(R1), F12 
BNEZ R1, Loop 
SD 8(R1), F4 ;

This loop will run 14 cycles (no stalls) per iteration; or 14/4=3.5 for each element!

Assumptions that make this possible:
- move LDs before SDs 
- move SD after SUBI and BNEZ 
- use different registers

When is it safe for compiler to do such changes?

Steps Compiler Performed to Unroll

- Determine that is OK to move the S.D after SUBUI and BNEZ, and find amount to adjust SD offset
- Determine that unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependencies needed to yield same result as the original code

Multiple Issue

- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
- Decrease Ideal pipeline CPI
- Multiple issue
  - Superscalar
    - Statically scheduled (compiler techniques)
    - Dynamically scheduled (Tomasulo’s alg.)
  - VLIW (Very Long Instruction Word)
    - parallelism is explicitly indicated by instruction
    - EPIC (explicitly parallel instruction computers)
Superscalar MIPS:
- Superscalar MIPS: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

<table>
<thead>
<tr>
<th>Time [clocks]</th>
<th>I</th>
<th>IF</th>
<th>ID</th>
<th>Ex</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: FP operations extend EX cycle

Loop Unrolling in Superscalar
- Unrolled 5 times to avoid delays
- This loop will run 12 cycles (no stalls) per iteration - or 12/5=2.4 for each element of the array

<table>
<thead>
<tr>
<th>Integer Instr.</th>
<th>FP Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: bge R1,0,S1</td>
<td></td>
</tr>
<tr>
<td>2: LD F0,-24(S1)</td>
<td>ADD F0,F1,F2</td>
</tr>
<tr>
<td>3: LD F1,-16(S1)</td>
<td>ADD F2,F0,F2</td>
</tr>
<tr>
<td>4: ADD F0,F1,F2</td>
<td></td>
</tr>
<tr>
<td>5: ADD F2,F1,F0</td>
<td></td>
</tr>
<tr>
<td>6: ADD F0,F1,F2</td>
<td></td>
</tr>
<tr>
<td>7: ADD F2,F1,F0</td>
<td></td>
</tr>
<tr>
<td>8: SUBR R1,R1,R1</td>
<td></td>
</tr>
<tr>
<td>9: SD 16(R1),F16</td>
<td></td>
</tr>
<tr>
<td>10: BNEZ R1,Loop</td>
<td></td>
</tr>
<tr>
<td>11: SD -16(R1),F12</td>
<td></td>
</tr>
<tr>
<td>12: ADD F12,F10,F2</td>
<td></td>
</tr>
</tbody>
</table>

The VLIW Approach
- VLIWs use multiple independent functional units
- VLIWs package the multiple operations into one very long instruction
- Compiler is responsible to choose instructions to be issued simultaneously

<table>
<thead>
<tr>
<th>Time [clocks]</th>
<th>I</th>
<th>IF</th>
<th>ID</th>
<th>E</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiple Issue Processors
- Two variations
  - Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
    - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
    - Crusoe VLIW processor [www.transmeta.com]
    - Intel Architecture-64 (IA-64) 64-bit address
    - Style: “Explicitly Parallel Instruction Computer (EPIC)”
- Anticipated success lead to use of Instructions Per Clock cycle (IPC) vs. CPI
### Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Mem. Ref1</th>
<th>Mem. Ref. 2</th>
<th>FP1</th>
<th>FP2</th>
<th>Int/Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 R1 (i)</td>
<td>R1 (i+1)</td>
<td>ADD F24, F0, F22</td>
<td>ADD F16, F0, F14</td>
<td>ADD F8, F0, F6</td>
</tr>
<tr>
<td>2 R1 (i)</td>
<td>R1 (i+1)</td>
<td>ADD F28, F0, F26</td>
<td>ADD F20, F0, F18</td>
<td>ADD F12, F0, F10</td>
</tr>
<tr>
<td>3 R1 (i)</td>
<td>R1 (i+1)</td>
<td>ADD F4, F0, F2</td>
<td>ADD F14, F0, F12</td>
<td>ADD F10, F0, F4</td>
</tr>
<tr>
<td>4 R1 (i)</td>
<td>R1 (i+1)</td>
<td>SD 8(R1), F28</td>
<td>SD 16(R1), F24</td>
<td>SD -24(R1), F16</td>
</tr>
<tr>
<td>5 R1 (i)</td>
<td>R1 (i+1)</td>
<td>LD F26, -48(R1)</td>
<td>LD F22, -40(R1)</td>
<td>LD F18, -32(R1)</td>
</tr>
<tr>
<td>6 Loop</td>
<td>Mem. Ref.1</td>
<td>Mem. Ref.2</td>
<td>Mem. Ref.1</td>
<td>Mem. Ref.1</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays.

7 results in 9 clocks, or 1.3 clocks per each element (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Need more registers in VLIW (15 vs. 6 in SS)

### Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7/16 or 112 bits to 7/24 or 168 bits wide
    - Need compiling technique that schedules across several branches

### When Safe to Unroll Loop?

- Example: Where are data dependencies? (A, B, C distinct & nonoverlapping)
  
  | for (i=0; i<100; i=i+1) { 
  | A[i+1] = A[i] + C[i]; /* S1 */ 
  | B[i+1] = B[i] + A[i+1]; /* S2 */ 
  | } 

  - 1. S2 uses the value, A[i+1], computed by S1 in the same iteration
  - 2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1]
  - This is a “loop-carried dependence”: between iterations
  - For our prior example, each iteration was distinct

### Does a loop-carried dependence mean there is no parallelism???

- Consider:
  
  | for (i=0; i<8; i=i+1) { 
  | A = A + C[i]; /* S1 */ 
  | } 

  - Could compute:
    - “Cycle 2”: temp4 = temp0 + temp1; temp5 = temp2 + temp3;
    - “Cycle 3”: A = temp4 + temp5;

  - Relies on associative nature of “+”.

---

Aleksandar Milenkovich
Another Example

Loop carried dependences?

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

To overlap iteration execution:

```c
for (i=1; i<100; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Another possibility: Software Pipelining

Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations

Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)

Software Pipelining Example

Before: Unrolled 3 times

```
1  LD  F6,8(R1)
2  ADDD F4,F0,F2
3  SD  0(R1),F4
4  LD  F6,-8(R1)
5  ADDD F8,F6,F2
6  SD  -8(R1),F8
7  LD  F10,-16(R1)
8  ADDD F12,F10,F2
9  SD  -16(R1),F12
10 SUBUI R1,R1,#24
11 BNEZ R1,LOOP
```

After: Software Pipelined

```
1  SD  0(R1),F4 ; Stores M[i]
2  ADDD F4,F0,F2 ; Adds to M[i-1]
3  LD  F0,-16(R1); Loads M[i-2]
4  SUBUI R1,R1,#8
5  BNEZ R1,LOOP
```

Things to Remember

- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
- Loop unrolling to minimise stalls
- Multiple issue to minimise CPI
  - Superscalar processors
  - VLIW architectures

Symbolic Loop Unrolling
- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop
- Less, once per each unrolled iteration in loop unrolling

Another possibility: Software Pipelining
Statically Scheduled Superscalar

- E.g., four-issue static superscalar
- 4 instructions make one issue packet
- Fetch examines each instruction in the packet in the program order
  - instruction cannot be issued
  - will cause a structural or data hazard
  - either due to an instruction earlier in the issue packet or due to an instruction already in execution
  - can issue from 0 to 4 instruction per clock cycle

Multiple Issue with Dynamic Scheduling

Assumptions:
- One FP and one integer operation can be issued;
- Resources: ALU (int + effective address), a separate pipelined FP for each operation type, branch prediction hardware, 1 CDB
- 2 cc for loads, 3 cc for FP Add
- Branches single issue, branch prediction is perfect
Multiple Issue with Dynamic Scheduling:

- DADDIU waits for ALU used by S.D
  - Add one ALU dedicated to effective address calculation
  - Use 2 CDBs
- Draw table for the dual-issue version of Tomasulo’s pipeline