EE 4683/5683: COMPUTER ARCHITECTURE

Lecture 4A: Instruction Level Parallelism - Static Scheduling

Avinash Kodi, kodi@ohio.edu

Agenda

- Dependences
  - RAW, WAR, WAW
- Static Scheduling
- Loop-carried Dependence
Recall from Pipelining Review

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 (e.g., Pentium 4, AMD Opteron, IBM Power), and
  2. Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2)
### Instruction-Level Parallelism (ILP)

- 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
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: **loop-level parallelism** to exploit parallelism among iterations of a loop. E.g.,
  ```
  for (i=1; i<=1000; i=i+1)
      x[i] = x[i] + y[i];
  ```

### Loop-Level Parallelism

- Exploit loop-level parallelism to parallelism by “unrolling loop” either by
  1. dynamic via branch prediction or
  2. static via loop unrolling by compiler
- Determining instruction dependence is critical to Loop Level Parallelism
- 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
Data Dependence and Hazards

1. Instr_j is **data dependent** (aka **true dependence**) on Instr_i:
   1. Instr_j tries to read operand before Instr_i writes it.
      - I: add r1, r2, r3
      - J: sub r4, r1, r3
   2. or Instr_j is data dependent on Instr_k which is dependent on Instr_i

2. If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

3. Data dependence in instruction sequence
   ⇒ data dependence in source code ⇒ effect of original data dependence must be preserved.

4. If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**.

ILP and Data Dependencies, Hazards

1. HW/SW must preserve **program order**:
   - order instructions would execute in if executed sequentially as determined by original source program
   - Dependences are a property of programs

2. Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is property of the pipeline.

3. Importance of the data dependencies
   - 1) indicates the possibility of a hazard
   - 2) determines order in which results must be calculated
   - 3) sets an upper bound on how much parallelism can possibly be exploited

4. HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program.
Name Dependence #1: Anti-dependence

- **Name dependence**: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; **2 versions of name dependence**
- Instr_\text{j} writes operand *before* Instr_\text{i} reads it
  
  ```
  I: sub r4, r1, r3  
  J: add r1, r2, r3  
  K: mul r6, r1, r7
  ```

  Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”
- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- Instr_\text{j} writes operand *before* Instr_\text{i} writes it.
  
  ```
  I: sub r1, r4, r3  
  J: add r1, r2, r3  
  K: mul r6, r1, r7
  ```

  Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline, called a **Write After Write (WAW) hazard**
- Instructions involved in a name dependence can execute simultaneously if **name used in instructions is changed** so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW
Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```c
if p1 {
    S1;
};
if p2 {
    S2;
}
```

- \( S_1 \) is control dependent on \( p_1 \), and \( S_2 \) is control dependent on \( p_2 \) but not on \( p_1 \).

Software Techniques - Example

- This code, add a scalar to a vector:

```c
for (i=1000; i>0; i=i-1)
    x[i] = 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>stalls between in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</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

```mips
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
```

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

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
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

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

- 9 clock cycles: Rewrite code to minimize stalls?
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 DSUBUI
7. BNEZ  R1,Loop

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Latency in producing result</th>
<th>Using result</th>
<th>clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How can we make the loop faster?

Unroll Loop Four Times
(straightforward way)

1. Loop: L.D  F0,0(R1)
3. ADD.D  F4,F0,F2
6. S.D  0(R1),F4  ; drop DSUBUI & BNEZ
7. L.D  F6,-8(R1)
9. ADD.D  F8,F6,F2
12. S.D  -8(R1),F8  ; drop DSUBUI & BNEZ
13. L.D  F10,-16(R1)
15. ADD.D  F12,F10,F2
18. S.D  -16(R1),F12  ; drop DSUBUI & BNEZ
19. L.D  F14,-24(R1)
21. ADD.D  F16,F14,F2
24. S.D  -24(R1),F16
25. DADDUI R1,R1,#-32  ; alter to 4*8
26. BNEZ  R1,LOOP

27 clock cycles, or 6.75 per iteration (Assumes R1 is multiple of 4)
# Unrolled Loop That Minimizes Stalls

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loop: L.D F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L.D F6, -8(R1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L.D F10, -16(R1)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L.D F14, -24(R1)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ADD.D F4, F0, F2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ADD.D F8, F6, F2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ADD.D F12, F10, F2</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ADD.D F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>S.D 0(R1), F4</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>S.D -8(R1), F8</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>S.D -16(R1), F12</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>DSUBUI R1, R1, #32</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>S.D 8(R1), F16; 8-32 = -24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>BNEZ R1, LOOP</td>
<td></td>
</tr>
</tbody>
</table>

14 clock cycles, or 3.5 per iteration

---

# 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
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
  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

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```c
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 */
}
```
When Safe to Unroll Loop?

Example: Where are data dependencies?
(A,B,C distinct & nonoverlapping)

```c
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 */
}
```

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:
- Implies that iterations are dependent, and can't be executed in parallel

Sometimes it is possible to have a loop-carried dependence but still parallelize

Example:

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

- Where are the dependences here? Is the loop parallelizable and how?
Sometimes it is possible to have a loop-carried dependence but still parallelize

- S1 uses the value assigned in the previous iteration by statement S2, therefore there is a loop-carried dependence

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

- Unlike the earlier loop-carried dependence this is not a circular dependence: while statement S1 depends on S2, S2 does not depend on S1.
- When there is no circular dependences, the loop can be parallelized.
- Let's write the code for the parallel version.

```c
for (i = 1; i<=99; i = i+1) {
    B[i+1] = C[i] + D[i];     /* old S2 */
    A[i+1] = A[i+1] + B[i+1]; /* old S1 */
}
B[101] = C[100] + D[100];
```

- The dependence between the two statements is no longer loop-carried, so different iterations may be overlapped provided the statements in each iteration are kept in order.