Computer Architecture, Lecture 10: Static scheduling and VLIW

Hossam A. H. Fahmy

Very Long Instruction Word

We have seen some problems with multiple issue superscalars:

- $N^2$ dependence checks (large stall and bypass logic),
- $N^2$ bypass buses (partially fixed with clustering), as well as wider fetch and problems with branch prediction.

In VLIW,

- a single issue pipeline that has $N$ parallel units is used,
- the compilers only puts independent “instructions” in the same group,
- VLIW travels down the pipeline as one unit, and
- in pure VLIW machines the processor does not need to do any dependence checks.

VLIW purity

In a pure (classical/ideal) VLIW design the compiler schedules the pipeline including the stall cycles.

⇒ The compiler must know the exact latencies and organization of the pipeline.

Problem 1: These details vary in different implementations. We must recompile the code. (TransMeta recompiles on the fly.)

Problem 2: Even for a specific implementation, the latencies are not fixed. What shall the hardware do for a cache miss?

Real implementations are not ideal.

\[
\begin{align*}
  &Ax + y \\
  \text{for } (i=0; i<n; i++) \ z[i] = A*x[i] + y[i]; \\
\end{align*}
\]

Assume that R1 starts at 0, that R2 holds the value of N, that A is in F2, and that the label loop: is at the first Ldf instruction. The FP Mul takes five cycles and the FP add takes two cycles.

- Latency of a single iteration (7 instruction) = 15 cycles.
- Performance = 7/15 i.e. $IPC \approx 0.47$.
- $IPC_{peak} = 1$, ⇒ Utilization = $0.47/1 = 47\%$
Two pipelines

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>STALL</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ldf F1, X(R1)</td>
<td>IF</td>
<td>blank</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>Mulf F3, F1, F2</td>
<td>IF</td>
<td>stall</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>Ldf F4, Y(R1)</td>
<td>IF</td>
<td>stall</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>Add R1, R1, #4</td>
<td>IF</td>
<td>stall</td>
<td>EX</td>
<td>M</td>
</tr>
<tr>
<td>Ble R1, R2, loop</td>
<td>IF</td>
<td>stall</td>
<td>EX</td>
<td>M</td>
</tr>
</tbody>
</table>

- Latency of a single iteration (7 instructions) = 14 cycles.
- Performance = 7/14 i.e. $IPC = 0.50$.
- $IPC_{peak} = 2 \Rightarrow$ Utilization = $0.50 / 2 = 25\%$

We have more hazards leading to many stalls.
- We have too many dependent instructions.

### Scheduling and issuing

**Schedule**: Decide the order of the instructions.
- Put independent instructions between the slow operations and the instructions that need their results.

**Issue**: Decide the time a specific instruction starts.
- Once all the dependencies are clear we can start.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure VLIW</td>
<td>SW</td>
</tr>
<tr>
<td>In-order superscalar</td>
<td>SW</td>
</tr>
<tr>
<td>Out-of-order (dynamic)</td>
<td>HW</td>
</tr>
</tbody>
</table>

### Compiler techniques

**Compiler**:
- Large scope (may be the whole program).
- Leads to a simpler hardware.
  - Low branch prediction accuracy.
  - No information about memory delays (cache misses).
  - Difficult to speculate and recover.

**Hardware**:
- Better branch prediction accuracy.
- Dynamic information about memory delays.
- Easier to speculate and recover.
- Finite resources to buffer instructions.
- Complicated hardware (harder to verify, may lead to slower clock).

We want to increase the number of independent instructions.

**Loop unrolling**: Put more than one iteration in sequence in a wider loop.

**Software pipelining**: Similar to what happens in hardware, a part of the first iteration is done with a part of the second iteration.

**Trace scheduling**: Programs include other things beyond loops.
Loop unrolling
Combine two (in general n) iterations by fusing the loop control and adjusting implicit uses of the loop counter.

Now we have too many WAW and WAR hazards. How do you solve that?

Scheduling

Unrolled on a single pipeline

We get less stalls ⇒ Two iterations (12 instructions) in 17 cycles.
Problems with loop unrolling

- Code growth.
- Does not handle inter-iteration dependences.

for (i=0; i<n; i++) x[i] = A*x[i-1];

Software pipelines

Ldf F1, X(R1)
Mul F3, F1, F2
Ldf F4, Y(R1)
Add F5, F3, F4
Stf F5, Z(R1) ⇒
Ldf F1, X(R1)
Mul F3, F1, F2
Ldf F4, Y(R1)
Add F5, F3, F4
Stf F5, Z(R1)
Ldf F4, Y+4(R1)
Mul F3, F1, F2
Ldf F4, Y+4(R1)
Add R1, R1, #4
Ble R1, R2, loop
Add F5, F3, F4
Stf F5, Z(R1)

Trace scheduling

For the other parts that are not loops

- Find the most common path across branches.
- Provide repair blocks in case this guess is incorrect.