Lecture 6: Static ILP

• Topics: loop analysis, SW pipelining, predication, speculation (Section 2.2, Appendix G)
Loop Dependences

• If a loop only has dependences within an iteration, the loop is considered parallel → multiple iterations can be executed together so long as order within an iteration is preserved.

• If a loop has dependences across iterations, it is not parallel and these dependences are referred to as “loop-carried”.

• Not all loop-carried dependences imply lack of parallelism.
Examples

For \((i=1000; \ i>0; \ i=i-1)\) \n\[x[i] = x[i] + s;\]

For \((i=1; \ i<=100; \ i=i+1)\) \n\{ \n  A[i+1] = A[i] + C[i]; \hspace{1cm} S1 \\
  B[i+1] = B[i] + A[i+1]; \hspace{1cm} S2 \\
\}

For \((i=1; \ i<=100; \ i=i+1)\) \n\{ \n  A[i] = A[i] + B[i]; \hspace{1cm} S1 \\
  B[i+1] = C[i] + D[i]; \hspace{1cm} S2 \\
\}

For \((i=1000; \ i>0; \ i=i-1)\) \n\[x[i] = x[i-3] + s; \hspace{1cm} S1\]
Examples

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

For (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; \quad S1
    B[i+1] = B[i] + A[i+1]; \quad S2
}

For (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i]; \quad S1
    B[i+1] = C[i] + D[i]; \quad S2
}

For (i=1000; i>0; i=i-1)
\[ x[i] = x[i-3] + s; \quad S1 \]

No dependences

S2 depends on S1 in the same iteration
S1 depends on S1 from prev iteration
S2 depends on S2 from prev iteration

S1 depends on S2 from prev iteration

S1 depends on S1 from 3 prev iterations
Referred to as a recursion
Dependence distance 3; limited parallelism
Constructing Parallel Loops

If loop-carried dependences are not cyclic (S1 depending on S1 is cyclic), loops can be restructured to be parallel

\[
\begin{align*}
\text{For } & (i=1; i<=100; i=i+1) \{ \\
& \quad A[i] = A[i] + B[i]; \quad \text{S1} \\
& \quad B[i+1] = C[i] + D[i]; \quad \text{S2} \\
\} \\
S1 \text{ depends on } S2 \text{ from prev iteration}
\end{align*}
\]

\[
\begin{align*}
& \text{For } (i=1; i<=99; i=i+1) \{ \\
& \quad B[i+1] = C[i] + D[i]; \quad \text{S3} \\
& \quad A[i+1] = A[i+1] + B[i+1]; \quad \text{S4} \\
& \} \\
& B[101] = C[100] + D[100]; \\
S4 \text{ depends on } S3 \text{ of same iteration}
\end{align*}
\]
Finding Dependences – the GCD Test

• Do $A[ai + b]$ and $A[ci + d]$ refer to the same element?

• Restrict ourselves to affine array indices (expressible as $ai + b$, where $i$ is the loop index, $a$ and $b$ are constants) – example of non-affine index: $x[y[i]]$

• For a dependence to exist, must have two indices $j$ and $k$ that are within the loop bounds, such that
  
  $aj + b = ck + d$;
  $aj - ck = d - b$;
  $G = GCD(a,c)$;
  $(aj/G - ck/G) = (d-b)/G$;

• If $(d-b)/G$ is not an integer, the initial equality can not be true
Software Pipeline?! 

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

L.D ADD.D S.D
DADDUI BNE
L.D ADD.D S.D
DADDUI BNE
L.D ADD.D S.D
DADDUI BNE
L.D ADD.D S.D
DADDUI BNE

...
Software Pipelining

<table>
<thead>
<tr>
<th>Loop:</th>
<th>L.D</th>
<th>F0, 0(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td></td>
<td>S.D</td>
<td>F4, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI</td>
<td>R1, R1,#-8</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop:</th>
<th>S.D</th>
<th>F4, 16(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td></td>
<td>L.D</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI</td>
<td>R1, R1,#-8</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
</tbody>
</table>

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers
Predication

• A branch within a loop can be problematic to schedule

• Control dependences are a problem because of the need to re-fetch on a mispredict

• For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions
Predicated or Conditional Instructions

- The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op.

- Example: `lwc R1, 0(R2), R3` (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op.

- Replaces a control dependence with a data dependence (branches disappear); may need register copies for the condition or for values used by both directions.

```
if (R1 == 0)
  R2 = R2 + R4
else
  R6 = R3 + R5
R4 = R2 + R3
```

```
R7 = !R1 ; R8 = R2 ;
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
```
Complications

• Each instruction has one more input operand – more register ports/bypassing

• If the branch condition is not known, the instruction stalls (remember, these are in-order processors)

• Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work

• Increases register pressure, activity on functional units

• Does not help if the br-condition takes a while to evaluate
Title

• Bullet