Lecture 25: October 18, 2021
Computer Architecture and Organization-II
Biplab K Sikdar

ILP -II

0.6 Compiler Support for Exploiting ILP

A language compiler can have direct contribution to exploit ILP within a program.

Simple techniques to exploit compiler support ILP: loop unrolling and speculation.

0.6.1 Loop unrolling

It is to reduce number of iterations by replicating the body of loop.

It effectively reduces control stalls in pipeline execution and, in effect, the CPI.

Consider the code fragment

```
for $i$ = 1 to 100 with increment 1 do
    $f()$;
```

Loop unrolling technique modifies it as

```
for $i$ = 1 to 25 with increment 1 do
    begin
      $f()$;
      $f()$;
      $f()$;
      $f()$;
    end
```

Loop body $f()$ is replicated four times and can be processed simultaneously.

Number of iterations is reduced to 25 now after loop unrolling.

When upper bound of loop not be known. Let the LOOP be

```
for $i$ = 1 to $n$ with increment 1 do
    $f()$;
```

and compiler unrolls it p times.

Original LOOP can be replaced by two loops say, LOOP1 and LOOP2.

```
(LOOP1)   for $i$ = 1 to $n \; mod \; p$ with increment 1 do
            $f()$;
(LOOP2)   for $i$ = 1 to $\frac{n}{p}$ with increment 1 do
            begin
              $f()$;
              $f()$;
              $f()$;
                $\vdots$
              $p$ times
```

LOOP1's body is as the original LOOP. It iterates $n \; mod \; p$ times.

LOOP2 iterates $\frac{n}{p}$ times. Its body is unrolled -that is, p copies of LOOP's body.

Example 0.2 Consider the following loop.

```
for $i$ = 1 to 103 with increment 1 do
    $f()$;
```

Here n=103. It is unrolled p=4 times ($n \; mod \; p$=3, $\frac{n}{p}$=25). After unrolling we get

```
(a)   for $i$ = 1 to 3 with increment 1 do
          $f()$;
(b)   for $i$ = 1 to 25 with increment 1 do
          begin
            $f()$;
            $f()$;
            $f()$;
            $f()$;
          end
```

Loop-carried dependence

Creates hindrance to loop unrolling.

For the loop,

```
for $i$ = 1 to 1000 with increment 1 do
    begin
      A[$i$] $\leftarrow$ A[$i$-1] + B[$i$-1]
    end
```

i^{th} iteration depends on value produced in $(i-1)^{th}$ iteration.

It has, therefore, loop carried dependence.

0.6.2 Speculation

Speculation: Partial execution of instruction before visit (actual execution) to it.

Results of partial execution, however, are not committed[1].

Effective for superscalar/VLIW processors when branch prediction affects speedup.

Consider the program segment.

$\vdots$

L: load R_x

I_j (reads R_x)

$\vdots$

Load takes time to prepare data from memory to CPU register R_x.

This register may be used in instruction I_j immediately after load.
R_x can be an operand (read) in I_j. Now I_j may have to be stopped until R_x is loaded.

Speculative execution of load

$\vdots$

L-K: spec load

$\vdots$

L: check load

I_j

$\vdots$

may produce R_x for I_j at right time. Loading is rescheduled at L-K well before I_j.

That is, *spec load* and *check load* instructions hide memory latency for load.

The checking is done where load instruction originally located (at L).

Care should be taken if load follows a store in original program.

[1]Instruction commits means write results generated out of the instruction execution -that is, updates register with results (or store to memory).

Load follows a store in original program

Consider

```
⋮
L-J: store
⋮
L: load
⋮
```

Then *spec load* is placed (at location L-J-I) prior to store, as shown below,

```
⋮
L-J-I: spec load
⋮
L-J: store
⋮
L: check load
⋮
```

check load, placed at location L of load, resolves whether load depends on store.

In case of a match, check load rejects result of spec load.

Speculative techniques are implemented in hardware as well as in software.

Software speculation is done by compiler with the help of hardware.

Example 0.3 Let take the program segment.

```
Load   r1, addr1
mult   r1, r2      ⇒  r1 = r1 * r2
sub    r3, r4      ⇒  r3 = r3 - r4
and    r5, r1      ⇒  r5 = r5 . r1
```

Execution of it can follow

$$\begin{tikzcd}
\underline{\text{at processor 1}} & \underline{\text{at processor 2}} \\
\text{load} \arrow[d] & \text{sub} \\
\text{mult} \arrow[d] & \\
\text{and} &
\end{tikzcd}$$

⇒ Execute ‘load’, ‘mult’, ‘and’ serially (they have dependencies) - ‘sub’ in parallel.

Data speculation can improve execution of instruction sequence at processor 1.

If value to be loaded at r1 is predicted, ‘load’ and ‘mult’ can be executed in parallel.

That is, execution order can be

$$\begin{tikzcd}
\underline{\text{at processor 1}} & \underline{\text{at processor 2}} & \underline{\text{at processor 3}} \\
 & \text{predicted (r1)} \arrow[d] & \\
\text{load} & \text{mult} \arrow[d] & \text{sub} \\
 & \text{and} &
\end{tikzcd}$$

When r1 is loaded from memory at processor 1, prediction at processor 2 is verified.

Execution proceed normally for correct prediction.

Otherwise, speculatively executed instructions are quashed and re-executed.

Execution order

$$\begin{tikzcd}
\underline{\text{at processor 1}} & \underline{\text{at processor 2}} & \underline{\text{at processor 3}} \\
 & \text{predicted (r1)} \arrow[d] & \\
\text{load} & \text{mult} \arrow[d] & \text{sub} \\
 & \text{and} &
\end{tikzcd}$$

Execution proceed normally for correct prediction.

Otherwise, speculatively executed instructions are squashed (Figure 23).

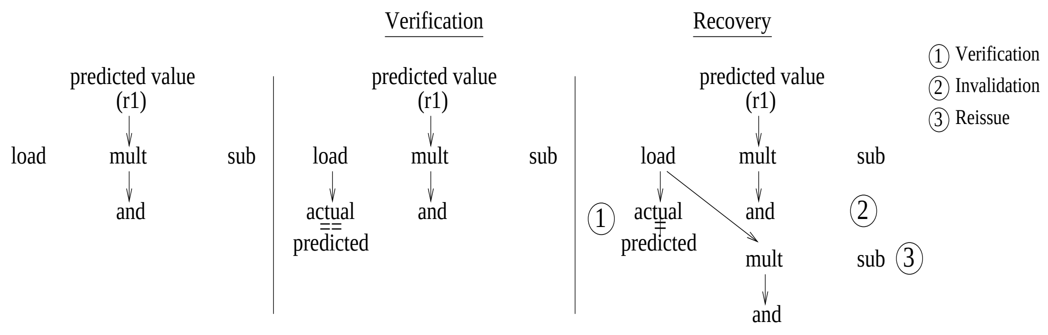

Figure 23: Data value speculation

Squashing of all instructions that follow mispredicted instruction can result in loss.

It is due to squashing of some instructions that do not depend on prediction (*sub*).

A selective recovery is desirable.

Conventionally multi-bit predictor is required for data speculation.

For effective decision of a prediction scheme, confidence estimation is necessary.