Lecture 7: Static ILP (Continued)
Branch prediction

Anton Burtsev
January, 2019
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

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)
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
Support for Speculation

• In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences

• However, we need hardware support
  ➢ to ensure that an exception is raised at the correct point
  ➢ to ensure that we do not violate memory dependences
Detecting Exceptions

- Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults).

- For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss.

- In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative.

- Note that a speculative instruction needs a special opcode to indicate that it is speculative.
Program-Terminate Exceptions

• When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register

• If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the segfault happens two procedures later)

• Alternatively, an instruction (the sentinel) in the speculative instruction’s original location checks the register value and initiates recovery
Memory Dependence Detection
(Advanced Load Address Table)

In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences.

However, we need hardware support:
- to ensure that an exception is raised at the correct point
- to ensure that we do not violate memory dependences
Memory Dependence Detection

- If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

- When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

- If a store finds its address in the ALAT, it indicates that a violation occurred for that address

- A special instruction (the sentinel) in the load’s original location checks to see if the address had a violation and re-executes the load if necessary
Dynamic ILP techniques
Static vs Dynamic Scheduling

• Arguments against dynamic scheduling:
  ➢ requires complex structures to identify independent instructions (scoreboards, issue queue)
    ▪ high power consumption
    ▪ low clock speed
    ▪ high design and verification effort
  ➢ the compiler can “easily” compute instruction latencies and dependences – complex software is always preferred to complex hardware (?)
ILP

• Instruction-level parallelism: overlap among instructions: pipelining or multiple instruction execution

• What determines the degree of ILP?
  ▶ dependences: property of the program
  ▶ hazards: property of the pipeline
Branch prediction
Pipeline without Branch Predictor

In the 5-stage pipeline, a branch completes in two cycles → If the branch went the wrong way, one incorrect instr is fetched → One stall cycle per incorrect branch
Pipeline with Branch Predictor

In the 5-stage pipeline, a branch completes in two cycles \( \rightarrow \)
If the branch went the wrong way, one incorrect instr is fetched \( \rightarrow \)
One stall cycle per incorrect branch
1-Bit Bimodal Prediction

• For each branch, keep track of what happened last time and use that outcome as the prediction

• What are prediction accuracies for branches 1 and 2 below:

```c
while (1) {
    for (i=0;i<10;i++) { branch-1
        ...
    }
    for (j=0;j<20;j++) { branch-2
        ...
    }
}
```
2-Bit Bimodal Prediction

• For each branch, maintain a 2-bit saturating counter: if the branch is taken: counter = min(3,counter+1) if the branch is not taken: counter = max(0,counter-1)

• If (counter >= 2), predict taken, else predict not taken

• Advantage: a few atypical branches will not influence the prediction (a better measure of “the common case”)

• Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)

• Can be easily extended to N-bits (in most processors, N=2)
Bimodal 1-Bit Predictor

The table keeps track of what the branch did last time.
Correlating Predictors

• Basic branch prediction: maintain a 2-bit saturating counter for each entry (or use 10 branch PC bits to index into one of 1024 counters) – captures the recent “common case” for each branch

• Can we take advantage of additional information?
  - If a branch recently went 01111, expect 0; if it recently went 11101, expect 1; can we have a separate counter for each case?
  - If the previous branches went 01, expect 0; if the previous branches went 11, expect 1; can we have a separate counter for each case?

Hence, build correlating predictors
Global Predictor

Branch PC

10 bits

CAT

Global history

Table of 16K entries

Each entry is a 2-bit sat. counter

The table keeps track of the common-case outcome for the branch/history combo
Local Predictor

Branch PC

Use 6 bits of branch PC to index into local history table

| 10110111011001 |

Table of 64 entries of 14-bit histories for a single branch

Table of 16K entries of 2-bit saturating counters

14-bit history indexes into next level

Also a two-level predictor that only uses local histories at the first level
Local Predictor

Branch PC

Local history
10 bit entries

Table of 1K entries

The table keeps track of the common-case outcome for the branch/local-history combo
Local/Global Predictors

• Instead of maintaining a counter for each branch to capture the common case,
  ➔ Maintain a counter for each branch and surrounding pattern
  ➔ If the surrounding pattern belongs to the branch being predicted, the predictor is referred to as a local predictor
  ➔ If the surrounding pattern includes neighboring branches, the predictor is referred to as a global predictor
Tournament Predictors

- A local predictor might work well for some branches or programs, while a global predictor might work well for others.

- Provide one of each and maintain another predictor to identify which predictor is best for each branch.

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>Local Predictor</th>
<th>Global Predictor</th>
<th>Tournament Predictor</th>
<th>Table of 2-bit saturating counters</th>
<th>MUX</th>
</tr>
</thead>
</table>

- Alpha 21264:
  - 1K entries in level-1
  - 1K entries in level-2
  - 4K entries
  - 12-bit global history
  - 4K entries
  - Total capacity: ?
Thank you!