Instruction-Level Parallelism (1)

Lecture 4
September 18th, 2023

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Instructor's slides from Elsevier Inc.*

Instruction-Level Parallelism

- **Pipelining become universal technique in 1985**
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- **Beyond this, there are two main approaches:**
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - ~~Not used as extensively in mobile processors~~
    - Actually, commonplace in mobile processors as well!
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- **When exploiting instruction-level parallelism, goal is to minimize CPI**
  - Pipeline CPI =

    Ideal pipeline CPI +

    Structural stalls +

    Data hazard stalls +

    Control stalls

- **Parallelism with basic block is limited**
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Outline

Textbook: Chapter 3.1-3.3

- **Instruction-Level Parallelism and Dependences**
- **Compiler Techniques for Exposing ILP**
- **Advanced Branch Prediction**

How to reduce CPI

Instruction-Level Parallelism

- **ILP is limited by**
  - Resource conflicts (ALU, functional modules...)
  - Dependences

- **Three types of dependences**
  - (True) Data dependences
  - Name dependences
  - Control dependences

Data Dependence

RAW / WAW

- **Instruction *j* is data dependent on instruction *i* if** $I_i \rightarrow I_j$
  - Instruction *i* produces a result that may be used by instruction *j*
  - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i* $I_i \rightarrow I_k \rightarrow I_j$
- **Example*: which instruction pairs are data dependent?**

```
Loop:   fld     f0,0(x1)      # f0=array element
        fadd.d  f4,f0,f2      # add scalar in f2
        fsd     f4,0(x1)      # store result
        addi    x1,x1,-8      # decrement pointer 8 bytes
        bne     x1,x2,Loop    # branch x1!=x2
```

WAR

- **Dependent instructions cannot be executed simultaneously**

** Note: this example is based on MIPS ISA (5th Ed.). There is one-to-one correspondence to RISC-V ISA (6th Ed.).*

Data Dependence

- **Dependencies are a property of programs**
- **Pipeline organization determines if dependence is detected and if it causes a stall**
- **Data dependence conveys:**
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- **Dependencies that flow through memory locations are difficult to detect**
  - "memory disambiguation" problem
  - Does 100(R4) = 20(R6)? store load (it is possible that R4* = R6*)
  - From different loop iterations, does 20(R6) = 20(R6)?

Name Dependence

- **Two instructions use the <u>same name but no flow</u> of information**
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads (WAR)
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j <u>write the same register</u> or memory location
    - Ordering must be preserved

- **To resolve, <u>use renaming techniques</u>**

Data and Name Dependence: Examples

- **(True) Data dependence**

```
r3 ← (r1) op (r2)
r5 ← (r3) op (r4)
```

RAW

- **Anti-dependence**

```
r3 ← (r1) op (r2)
r1 ← (r4) op (r5)
```

WAR

- **Output dependence**

```
r3 ← (r1) op (r2)
r3 ← (r4) op (r5)
```

WAW

Data Hazards

- **A data hazard exists if**
  - There is a name or data dependence between instructions, and
  - They are close enough that overlap during execution would change the order of access to the operand involved in the dependence
- **Three types of data hazards (depending on the order of read and write accesses) corresponding to three types of dependences**
  - Read after write (RAW) hazard – true data dependence
  - Write after write (WAW) hazard – output dependence
  - Write after read (WAR) hazard – anti-dependence

Control Dependence

- **Ordering of instruction *i* with respect to a branch instruction**
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Control Dependence

- **Examples**

Example 1:

```
   add x1,x2,x3
   beq x4,x0,L   ← branch
   sub x1,x1,x6
L: … (potential)
   or  x7,x1,x8
```

- **or instruction data dependent on add and sub**

Example 2:

```
   add x1,x2,x3
   beq x12,x0,skip
   sub x4,x5,x6
   add x5,x4,x9
skip:
   or  x7,x8,x9
```

- **Assume x4 isn't used after skip**
  - Possible to move sub before the branch

Compiler Techniques for Exposing ILP

- **Technique 1: Pipeline scheduling**
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
  - Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Compiler Techniques for Exposing ILP

- **Technique 1: Pipeline scheduling**
  - Unoptimized code (with –O0)

```
Loop:   fld     f0,0(x1)
        stall
        fadd.d  f4,f0,f2
        stall
        stall
        fsd     f4,0(x1)
        addi    x1,x1,-8
        stall   (assume Int ALU to branch latency is 1)
        bne     x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Compiler Techniques for Exposing ILP

- **Technique 1: Pipeline scheduling**
  - Pipeline optimized code (with –O2)

```
Loop:   fld     f0,0(x1)
        addi    x1,x1,-8      ← independent
        fadd.d  f4,f0,f2
        stall                 ↕ dependency
        stall
        fsd     f4,8(x1)
        bne     x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Compiler Techniques for Exposing ILP

- **Technique 2: Loop unrolling**
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   fld      f0,0(x1)          // 1-cycle stall
        fadd.d   f4,f0,f2          // 2-cycle stall
        fsd      f4,0(x1)  ) no dep // drop addi & bne
        fld      f6,-8(x1)
        fadd.d   f8,f6,f2
        fsd      f8,-8(x1)         // drop addi & bne
        fld      f0,-16(x1)
        fadd.d   f12,f0,f2
        fsd      f12,-16(x1)       // drop addi & bne
        fld      f14,-24(x1)
        fadd.d   f16,f14,f2
        fsd      f16,-24(x1)
        addi     x1,x1,-32
        bne      x1,x2,Loop
```

Note: number of live registers vs. original loop

How may cycles it would take??

program longer (code memory↑), less branch + register pressure, complex code

Compiler Techniques for Exposing ILP

- **Technique 2: Loop unrolling**
  - What if we combine loop unrolling with pipeline scheduling?

```
Loop:   fld     f0,0(x1)
        fld     f6,-8(x1)
        fld     f8,-16(x1)
        fld     f14,-24(x1)
        fadd.d  f4,f0,f2
        fadd.d  f8,f6,f2
        fadd.d  f12,f0,f2
        fadd.d  f16,f14,f2
        fsd     f4,0(x1)
        fsd     f8,-8(x1)
        fsd     f12,-16(x1)
        fsd     f16,-24(x1)
        addi    x1,x1,-32
        bne     x1,x2,Loop
```

No stall!

How may cycles it would take??

Compiler Techniques for Exposing ILP

- **Technique 2: Loop unrolling**
  - What if number of loop iterations is unknown at compile time?
    - Number of iterations = *n*
    - Goal: <u>make *k* copies of the loop body</u>
  - Solution: Strip mining!
    - Generate pair of loops:
      - First executes *n* mod *k* times
      - Second executes *n* / *k* times

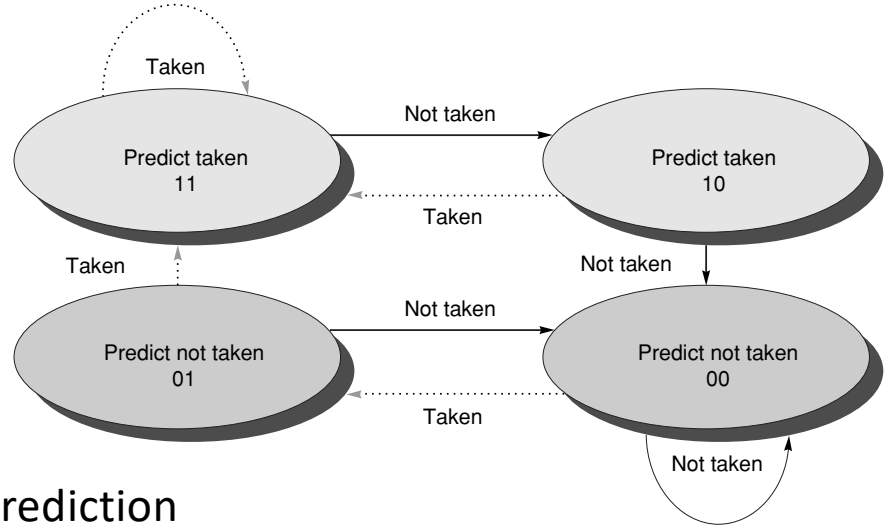
Advanced Branch Prediction

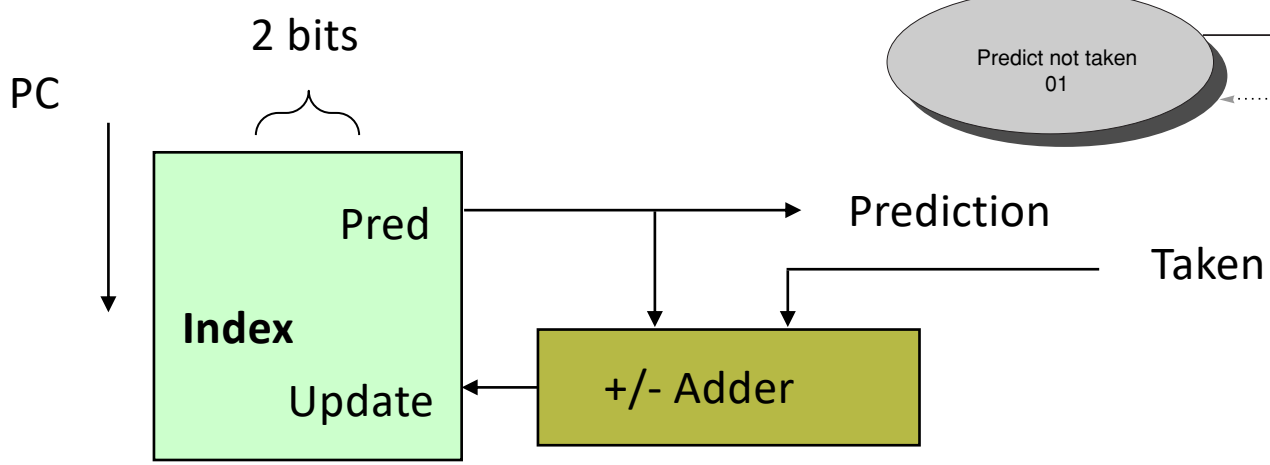
- **Motivation**
  - Branch penalties limit performance of deeply pipelined processors
    - Accounting for 16% of total instructions in SPECint92
    - Accounting for 8% in SPECfp92
  - Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

- **Required hardware support**
  - Prediction structures
    - Branch history tables, branch target buffers, etc.
  - Misprediction recovery mechanisms
    - <u>Keep result computation separate from commit</u>
    - Kill instructions following branch in pipeline
    - Restore state to state following branch

Advanced Branch Prediction

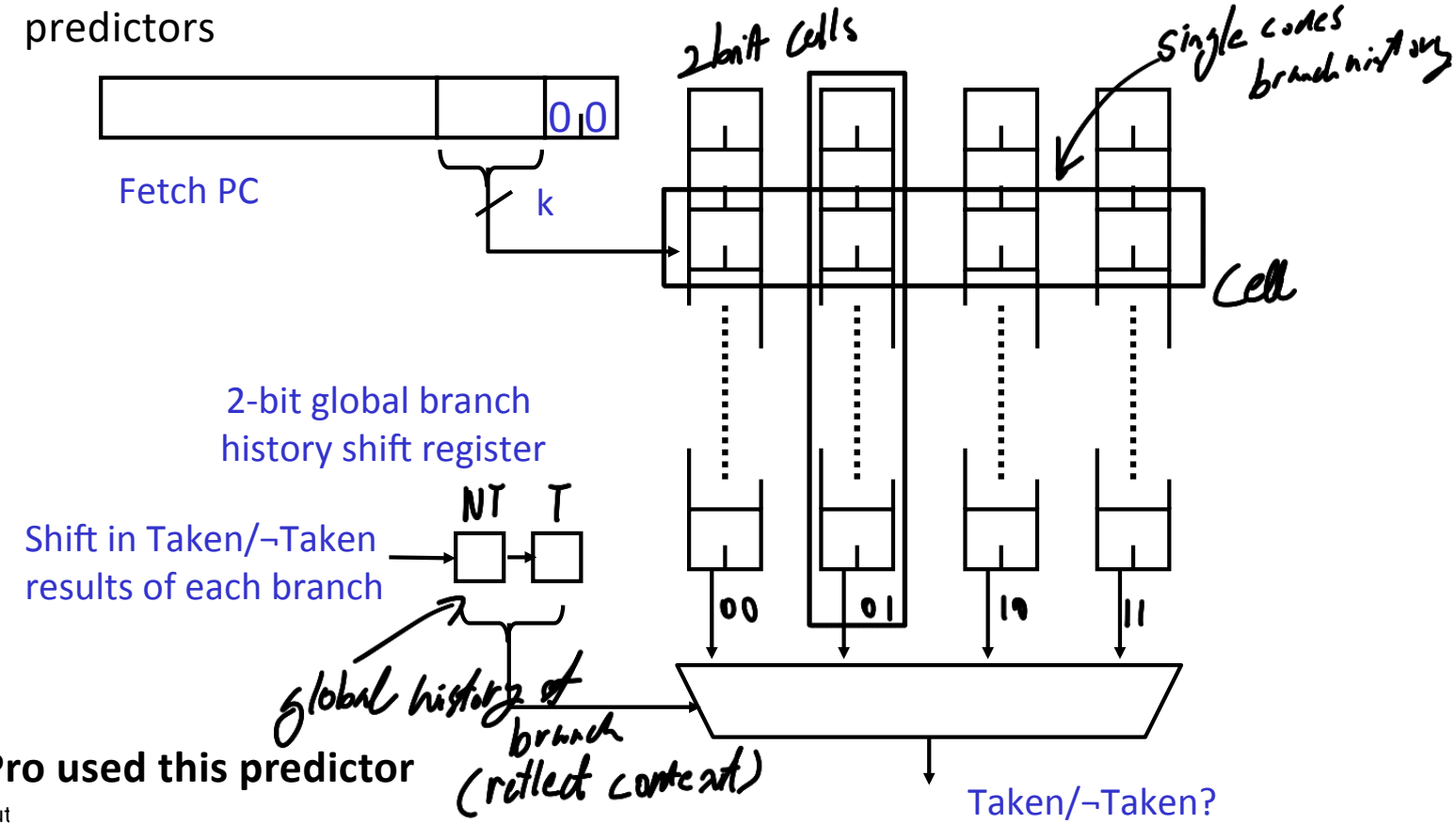
- **Basic 2-bit predictor**
  - For each branch
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction





Advanced Branch Prediction

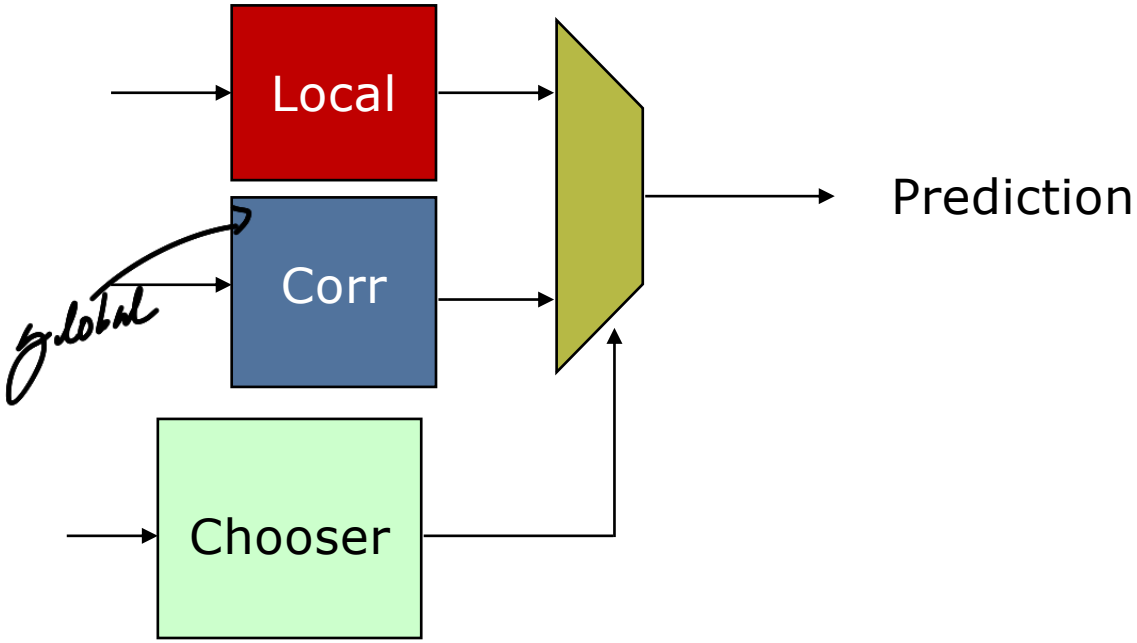
- **Correlating predictor**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
    - (*m*,*n*) predictor: behavior from last *m* branches to choose from 2^m *n*-bit predictors



*** Pentium Pro used this predictor**

Advanced Branch Prediction

- **Local predictor**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- **Tournament predictor:**
  - Combine correlating predictor with local predictor



Advanced Branch Prediction

- **Branch Prediction Performance**