UT 2. Pipelined Computers

Lecture 2.3 Dynamic Branch Prediction

J. Duato, J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

Bibliography

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Contents

Motivation

- Control dependencies $\rightarrow$ Control hazards $\rightarrow$ Penalty due to stalls.
- Is there enough ILP in a basic block? NO

  $\rightarrow$ Look for ILP across multiple basic blocks

$\Rightarrow$ It is necessary to obtain the execution result of a branch instruction *as soon as possible*: Branch prediction

Goal:

Predict the behavior of branches in order to have instructions fetched (and even under execution) by the time the final result of the branch becomes available.

- The behavior of the different branch instructions in a program is not usually the same
- The behavior of a given branch instruction uses to vary across the program execution

⇒ Dynamic branch prediction (supported by *hardware*).

Classification

- What is predicted? Condition vs. Condition + Address
- How is the prediction performed? Using the PC of the branch instruction. Complete address vs. only some bits
- How is the prediction stored? Direct vs. fully associative mapping
- When is the prediction performed? Before (IF) or after ($>=$ID) decoding the branch instruction

Contents

Also named as Branch History Table (BHT).

Goal:

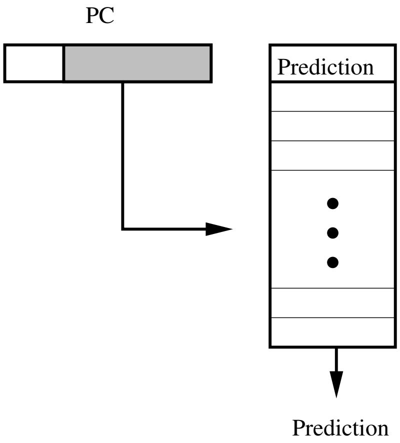
To predict branch conditions.

- Data:
  - Branch instruction address.
- Result:

  ⇒ Branch *taken* or *not taken*

Mechanism

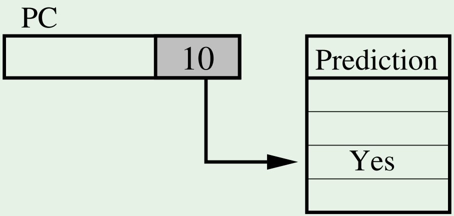
- A table indexed by the least significant bits of the branch instruction address.
- Each table entry contains the prediction for that branch instruction.



Example:

```
...010   BEQZ D1
......
    D1:  DADD R1,R2,R3
......
...110   BNEZ D2
```

After `BEQZ D1`:



Implementation

Table location: Several options:

- Small fast memory that is accessed in parallel with the instruction cache during IF.
- Adding prediction bits to each instruction cache block when using direct mapping.

Predictor: Prediction algorithm

- Finite state automaton.
- State changes according to the actual behavior of the branch instruction.
- The prediction (“taken”/“not taken”) depends on the state.

$\Rightarrow$ Simplest case: One-bit predictor

- Automaton state = Condition computed in the last execution of the branch.
- Prediction = Automaton state.

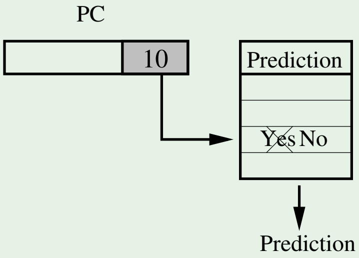
Problem: Two or more branches require the same table entry

- Two branch instructions may share their least significant bits → prediction may fail if their behavior is different.

Example:

```
...010   BEQZ D1
......
    D1:  DADD R1,R2,R3
......
...110   BNEZ D2
```

After `BNEZ D2`:



- Solution: Use more PC bits for the table index → increase the table size.

Branch prediction in loops

Branch instructions controlling loop iterations follow a predictable pattern.

- If the loop contains n iterations, the branch is taken $n - 1$ times, and only once it is not taken (for the last iteration).
- However, a one-bit predictor will fail twice: in the first iteration (it is not trained yet) and in the last one.

Example: Nested loops

```
for j := 1 to m do           loop_j: ...
  for i := 1 to n do         loop_i: ...
    ...                              ...
  end;                               bnez Ri, loop_i
end;                                 bnez Rj, loop_j
```

Behaviour of branch `bnez Ri, loop_i:`

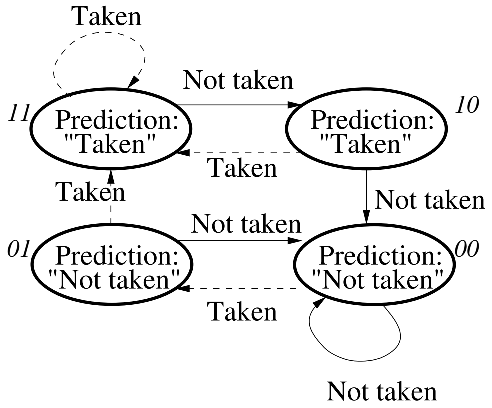
j	1				2				3
i	1	2	3	n	1	2	3	n	...
Taken?	Yes	Yes	Yes	No	Yes	Yes	Yes	No	
Prediction	-	Yes	Yes	Yes	No	Yes	Yes	Yes	
Is it correct?	-	OK	OK	NO	**NO**	OK	OK	**NO**	

- The prediction for the inner loop fails twice for each iteration of the outer loop.
- Percentage of times the branch is taken = $\frac{n-1}{n} * 100$ %
- Prediction accuracy = $\frac{n-2}{n} * 100$ %
- Solution: Two-bit predictor

Two-bit predictors

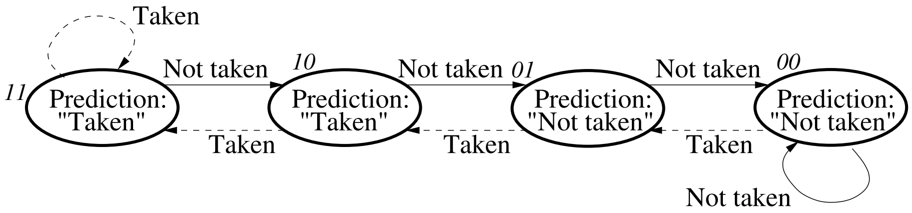
- With two-bit states, the automaton has four states:
  Strongly not taken (state 00)
  Weakly not taken (state 01)
  Weakly taken (state 10)
  Strongly taken (state 11)
- Basic idea: A prediction must fail twice before being modified.
- The two most usual designs:
  - Two-bit predictor with hysteresis
  - Two-bit predictor with saturation

Two-bit predictor with hysteresis:



n-bit predictor with saturation:

Each entry has a value of n bits (0 to 2^n -1):



- If the value is $\geq 2^{n-1}$, prediction is “taken”
- If the value is $< 2^{n-1}$, prediction is “not taken”

$\rightarrow$ Prediction must fail 2^{n-1} times before being modified.

Two-level or correlating predictors (1/3)

→ In addition to the behavior of the current branch instruction (local history) they consider the behavior of other branches (global history)

Example (eqntott):

Source Code:

```
if (aa==2)
   aa=0;
if (bb==2)
   bb=0;
if (aa!=bb)
```

MIPS assembler code:

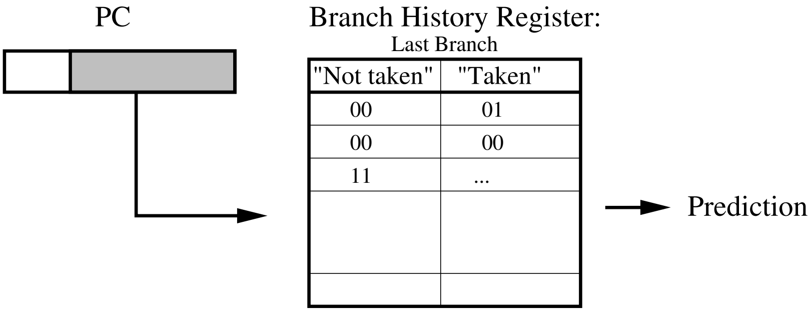
```
    DSUB R3,R1,#2
    BNEZ R3,L1      ; aa!=2 (b1)
    DADD R1,R0,R0   ; aa=0
L1: DSUB R3,R2,#2
    BNEZ R3,L2      ; bb!=2 (b2)
    DADD R2,R0,R0   ; bb=0
L2: DSUB R3,R1,R2   ; aa-bb
    BEQZ R3,L3      ; aa==bb (b3)
```

→ If branches *b*1 and *b*2 are “not taken” then `aa` and `bb` will be 0, and branch *b*3 will be “taken”.

Two-level or correlating predictors (2/3)

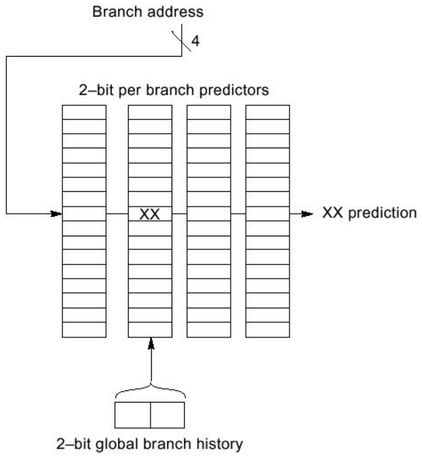
A predictor (g : *global*, l : *local*) uses the behavior of the last g branches in order to choose among 2^g l-bit predictors

- $(0, 2)$: 2-bit predictor.
- $(1, 2)$: Predictor that considers the behavior of the last branch:



Two-level or correlating predictors (3/3)

- $(2, 2)$: Predictor that considers the behavior of the last two branches:



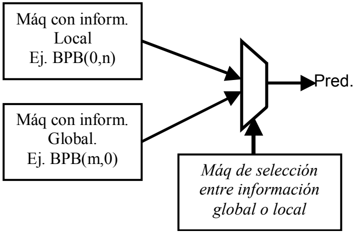
Hybrid predictors (1/4)

Basic idea

- Each predictor is suitable for different patterns
- By combining the use of different predictors and applying the most suitable one according to the needs, it is possible to increase the prediction accuracy.

Hybrid predictor. Several predictors plus a selection mechanism.

Example: *Tournament Predictor:*



Hybrid predictors (2/4)

Selection mechanism. Choose the predictor that provided the best results up to now

Implementation of the selection mechanism: Table of saturating counters indexed by the branch instruction address.

- For two predictors:

P1	P2	Actions
Misspred	Misspred	Counter =
Misspred	Hit	Counter++
Hit	Misspred	Counter– –
Hit	Hit	Counter =

  - If P2 hits more than P1 $\rightarrow$ Counter++
  - If P1 hits more than P2 $\rightarrow$ Counter– –

Hybrid predictors (3/4)

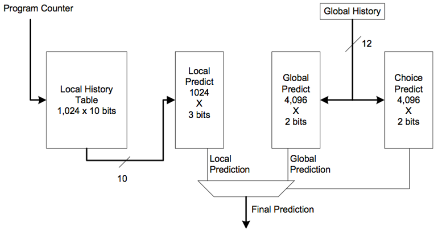
- The most significant counter bit selects the predictor to use:

MSB(Counter)	Selected predictor
0	P1
1	P2

Hybrid predictors (4/4)

Example: Alpha 21264 *tournament predictor*
2-level local predictor:

- PC gains access to a local history table for each branch...
- ...that stores the behavior of the last 10 executions. For example ”NT-T-T-T-T-T-T-T-T-NT”
- The resulting pattern is used to access the (3-bit) saturating counters providing the local prediction


Program Counter
Global History
12
Local History Table 1,024 x 10 bits
Local Predict 1024 X 3 bits
Global Predict 4,096 X 2 bits
Choice Predict 4,096 X 2 bits
10
Local Prediction
Global Prediction
Final Prediction

Contents

Goal:

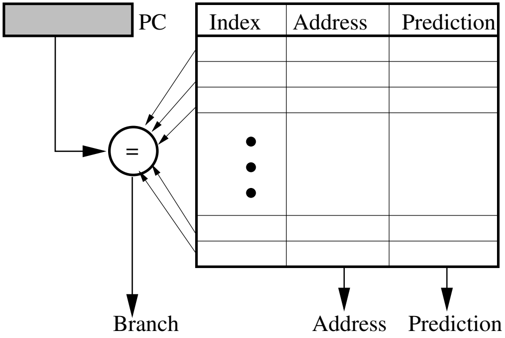
To predict the branch condition and the branch target address

- Data:
  - Branch instruction address
- Result:
  - Branch *taken* or *not-taken*
  - Branch target address (if taken)
  - Whether or not the instruction is a branch

How it works

Completely associative table, with three fields:

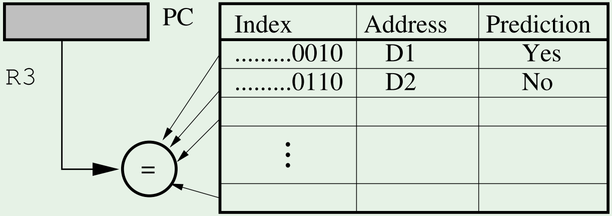
- Index: Branch instruction address
- Prediction: “Taken” or “Not taken”
- Address: Branch target address



Example:

After `BNEZ D2`:

```
...010   BEQZ D1
......
    D1:  DADD R1,R2,R3
......
...110   BNEZ D2
```



Index	Address	Prediction
.........0010	D1	Yes
.........0110	D2	No
⋮		

Events at stages of the instruction unit:

Example:

Consider a pipelined instruction unit that computes:

- branch target address, branch condition, and PC update at ID
- prediction at stage IF

Behavior of *predict-not-taken*:

- Branch is not taken:

Branch	IF	**ID**	EX	ME	WB	
PC+1		IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 0 cycles.

- Branch is taken:

Branch	IF	**ID**	EX	ME	WB		
PC+1		IF	ID	EX	ME	WB	
Branch target			IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 1 cycle.

Events at stages of the instruction unit:

IF If (∃ instruction in BTB table)
// It is a branch
- If (prediction == “taken”)
  Start fetching instructions during next cycle from the address provided by the table

// Zero penalty cycles

ID Instruction decoding.
If (instruction == branch)
- Compute branch target address and condition
- If (∃ instruction in BTB table)
  - If (prediction != condition)
    - ★ Abort incorrectly fetched instructions
    - ★ Update the prediction bit in the table entry
    - ★ Start fetching instruction from the correct address
- else // There was no entry in the table for the instruction
  - Add entry to the table for the instruction

Behavior:

- The branch has no entry in the table and it is finally **not taken**:

Branch	**IF**	ID	EX	ME	WB	
PC+1		IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 0 cycles.

- The branch has no entry in the table and it is finally **taken**:

Branch	**IF**	ID	EX	ME	WB		
PC+1		IF	ID	EX	ME	WB	
Branch target			IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 1 cycle.

- Prediction is **not taken** and it is finally **not taken** :

Branch	**IF**	ID	EX	ME	WB	
PC+1		IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 0 cycles.

- Prediction is **not taken** and it is finally **taken** :

Branch	**IF**	ID	EX	ME	WB		
PC+1		IF	ID	EX	ME	WB	
Branch target			IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 1 cycle.

- Prediction is **taken** and it is finally **not taken** :

Branch	**IF**	ID	EX	ME	WB		
Branch target		IF	ID	EX	ME	WB	
PC+1			IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 1 cycle.

- Prediction is **taken** and it is finally **taken** :

Branch	**IF**	ID	EX	ME	WB	
Branch target		IF	ID	EX	ME	WB

$\Rightarrow$ Penalty: 0 cycles.

Comparison with predict-not-taken:

Prediction	Condition	BTB	*pnt*	BTB is ... than/to *pnt*
no	no	0	0	equal
no	yes	1	1	equal
yes	no	1	0	worse
yes	yes	0	1	better

$\Rightarrow$ The effective behavior depends on the prediction accuracy: high predictor hit rate.

Observation:

In practice, the branch condition may depend on a previous long-latency instruction, and thus, it may take long to be computed. In this case, the improvement achieved by BTB would be much higher.

Implementation:

- The table stores the complete address of the branch instruction, since this address is required during IF, *before* decoding the instruction. A hit in the table only occurs if the instruction is a branch *for sure*.
- It is not mandatory to implement a fully associative table. In practice, it is a set-associative table (the least significant bits of the PC are used to index sets and they are not stored in the table).
- The BTB and the predictor can be decoupled:
  - BHT stores prediction bits
  - BTB stores the target address of the last “taken” branches. The size of the BTB can be small.

  If the prediction of the BHT is “taken” and there is a hit in the BTB, it starts fetching instructions from the target address.