NTNU

Det skapende universitet

TDT4255 Computer Design

Lecture 5: Branch Prediction

Magnus Jahre

Outline

- Appendix C.2
- Chapter 3.3

BRANCH PREDICTION

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Instruction fetch

- Goal is to fetch as many insns per cycle as possible

- Branch prediction is super important to fill the pipeline with correct-path instructions (useful work)

- Importance increases with deeper pipelines

Branch prediction

- Goal is to predict the branch direction and target address, and start fetching and executing insns along the predicted path
- Key observation:
  - Branches exhibit temporal locality
  - Predicting branch behavior
    - Keep track of past history
    - Predict the future based on the past
  - Branch behavior is predictable
    - typically over 90%, 95% or 99% of all dynamically executed branches are correctly predicted

Cost per mispredicted branch

- Cost per mispredicted branch is proportional to pipeline depth (no. pipeline stages)

- Early pipelines had 4-6 stages
- Modern pipelines have 12-15 stages
  - Example: Intel Pentium 4 had 20 stages (Extreme Edition had 31 stages)
  - Reason: Higher clock frequencies

Branch Direction Prediction

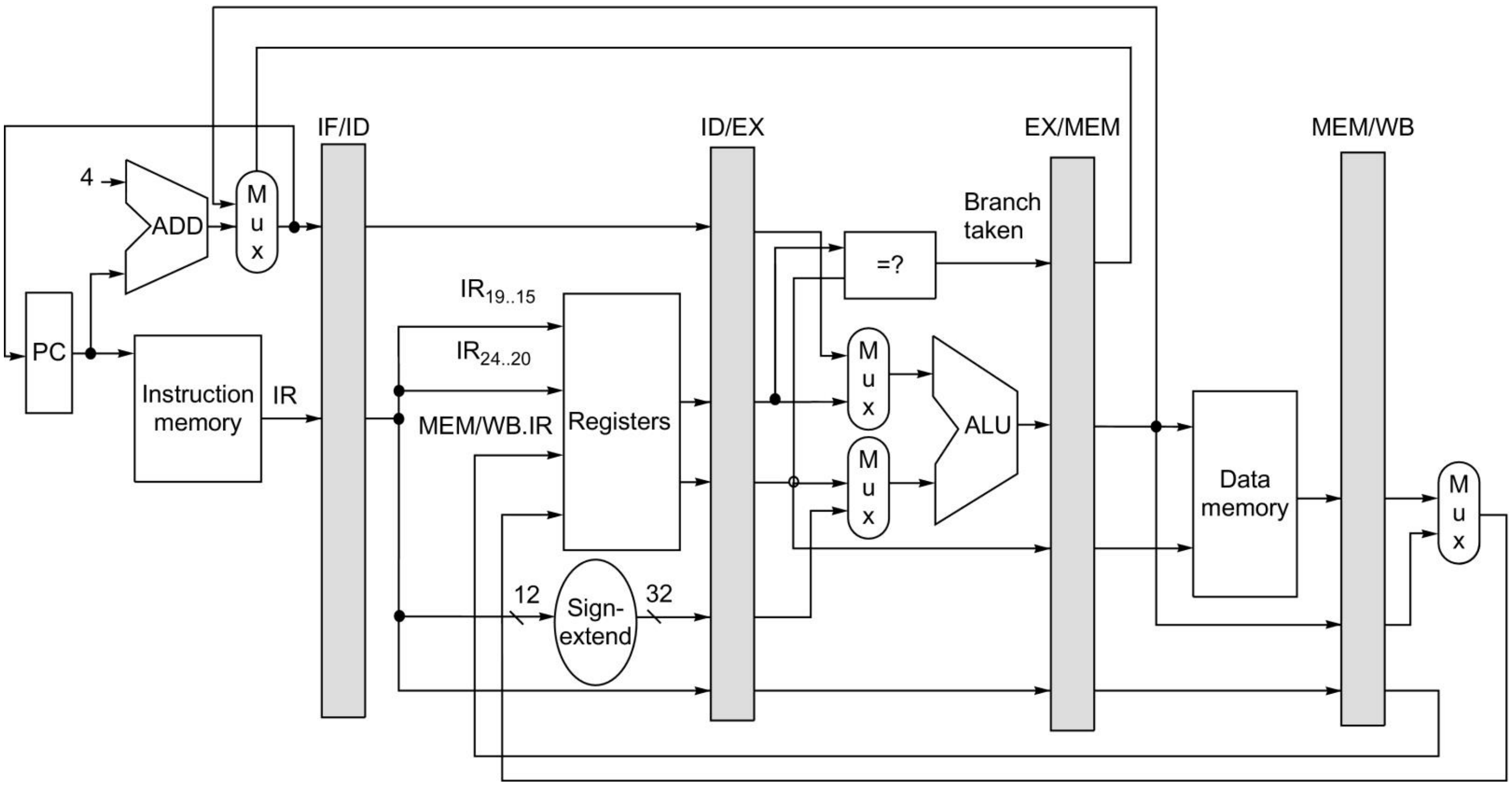
- Applies to conditional branches
- Static prediction
  - Static = before program execution
  - One prediction (taken/not-taken) per static branch in the program binary
  - Via software: compiler or programmer
- Dynamic prediction
  - Dynamic = during program execution
  - Multiple predictions per static branch, depending on history (= outcomes of prior branch executions) of that particular branch or even other branches
  - Done in hardware

Branch address vs. branch target address

```
000000012002e530 <__start>:
   12002e530:   f0 ff de 23     lda     sp,-16(sp)
   12002e534:   08 00 fe b7     stq     zero,8(sp)
   12002e538:   00 00 20 c0     br      t0,12002e53c <__start+0xc>
   12002e53c:   10 00 1e a2     ldl     a0,16(sp)
   12002e540:   18 00 3e 22     lda     a1,24(sp)
   12002e544:   00 20 a1 27     ldah    gp,8192(t0)
   12002e548:   52 06 11 42     s8addq  a0,a1,a2
   12002e54c:   12 14 41 42     addq    a2,0x8,a2
   12002e550:   24 28 bd 23     lda     gp,10276(gp)
   12002e554:   13 04 52 46     mov     a2,a3
   12002e558:   00 00 33 a4     ldq     t0,0(a3)
   12002e55c:   13 14 61 42     addq    a3,0x8,a3
   12002e560:   fd ff 3f f4     bne     t0,12002e558 <__start+0x28>
```

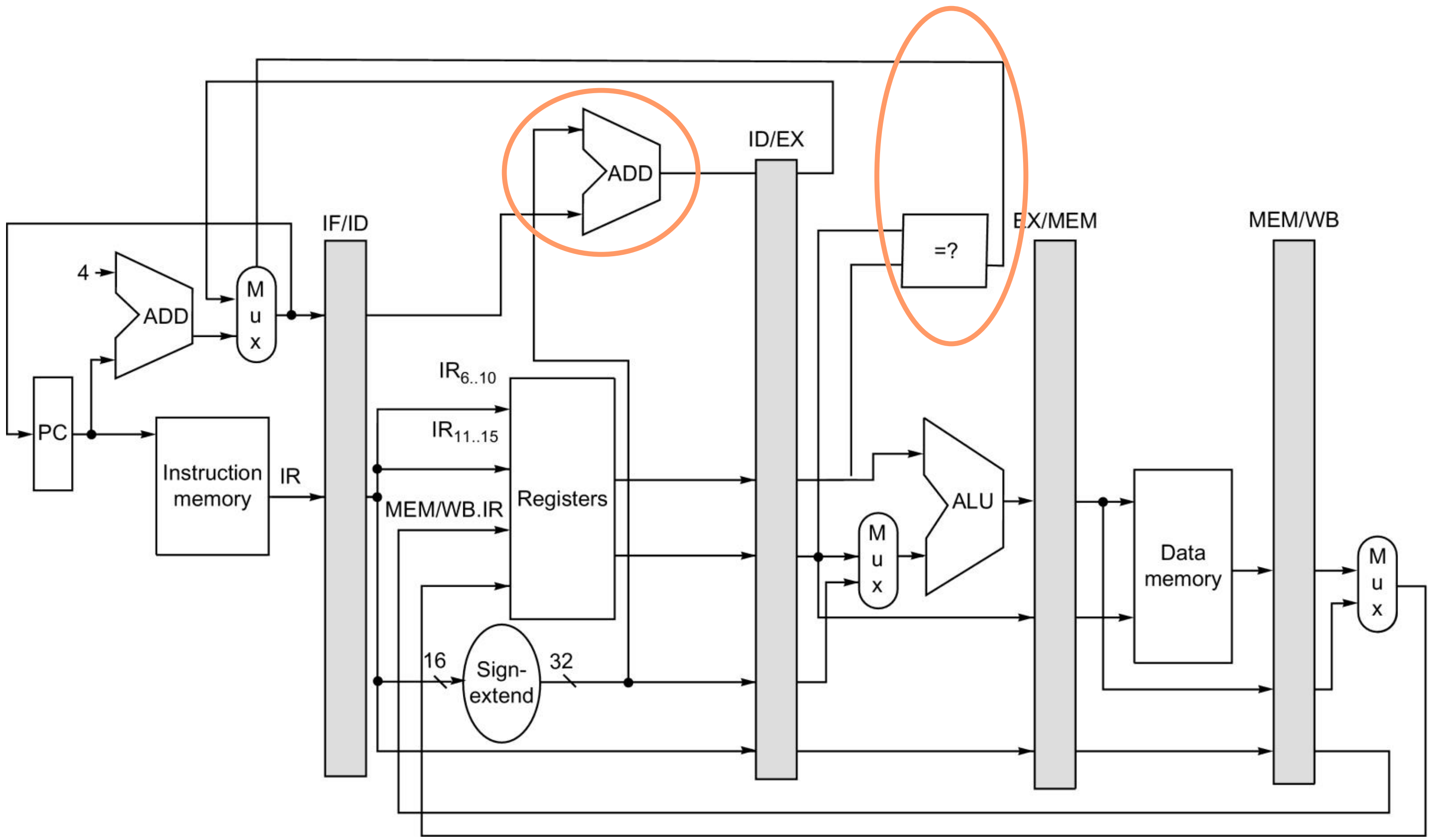
Command on Linux/UNIX/OS X: `objdump -d a.out`

Vanilla Pipeline



Branch penalty: Three cycles

Faster Conditional Branches



Branch penalty: Two cycles

STATIC BRANCH DIRECTION PREDICTION

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Static Branch Prediction

- Advantages
  - Easy to implement
  - Little HW is needed
- Disadvantages
  - Provides the same prediction regardless of input and/or dynamic execution behavior
- Three flavors
  - rule-based
  - program-based
  - profile-based

Rule-based

- Always not-taken
  - Simple HW: sequential fetch
- Always taken
  - HW is more complex because of unknown branch target
  - Branch target is known at the decode stage
  - May lead to a *bubble* ('lost cycle') in the pipeline
- BTFNT
  - *Backward taken - forward not-taken*
  - Branches to smaller addresses are typically backward loop branches and are typically taken

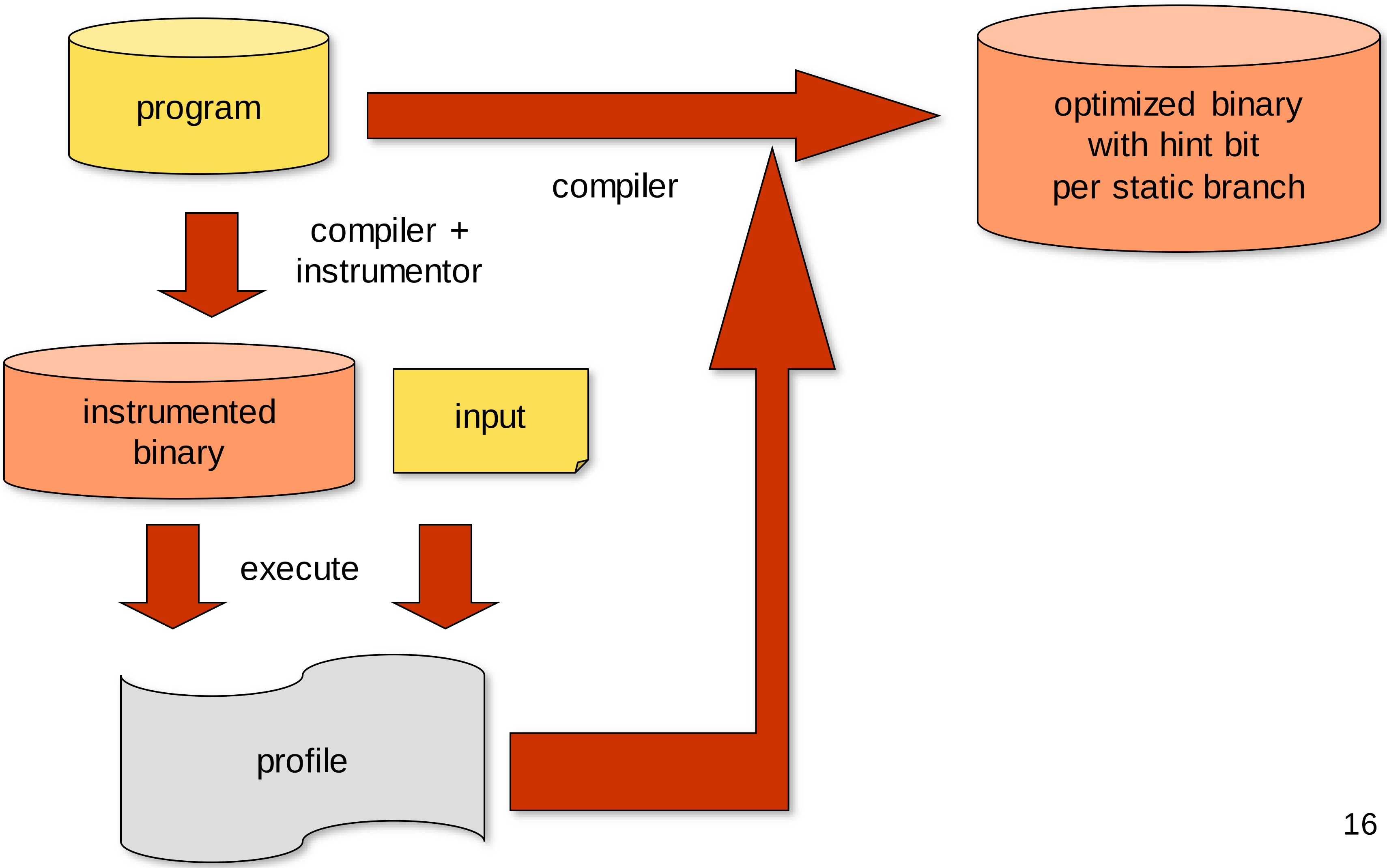
Program-based

- Ball and Larus heuristics (PLDI'93)
  - Requires a hint bit in instruction opcode
  - Branch direction is estimated based on program structure
  - Examples:
    - Predict loop branches to be taken
    - When comparing a pointer to NULL, predict branch direction to non-NULL path
    - When comparing two pointers, predict branch direction to path representing pointer inequality
  - Typically more accurate than rule-based

Profile-based

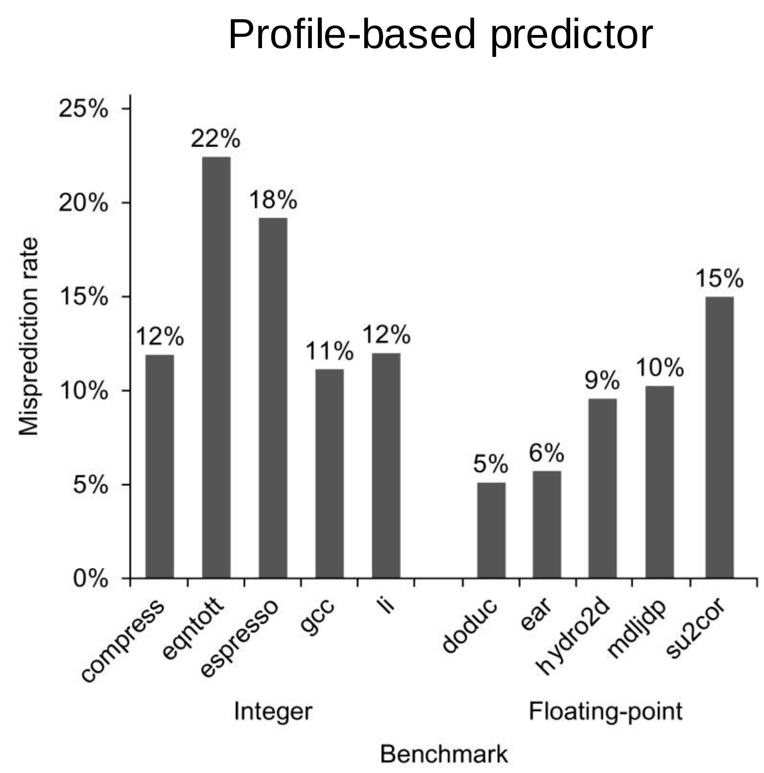
- Execute instrumented binary with a given training input to collect profile information
  - Count how often a static branch is taken/not-taken
- Use profile information during recompilation and add hint bits
  - Predict taken if branch has higher probability than 50% to be taken; predict not-taken otherwise
- Typically more accurate than rule- and program based

Profile-based



Static Branch Prediction Accuracy

- Some branches are biased towards being taken and some are biased towards being not taken
- A single, static policy for all branches is unlikely to work
- Even using profiling to determine the preferred direction of a branch is not sufficient (e.g., alternating between taken and not taken)



DYNAMIC BRANCH DIRECTION PREDICTION

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

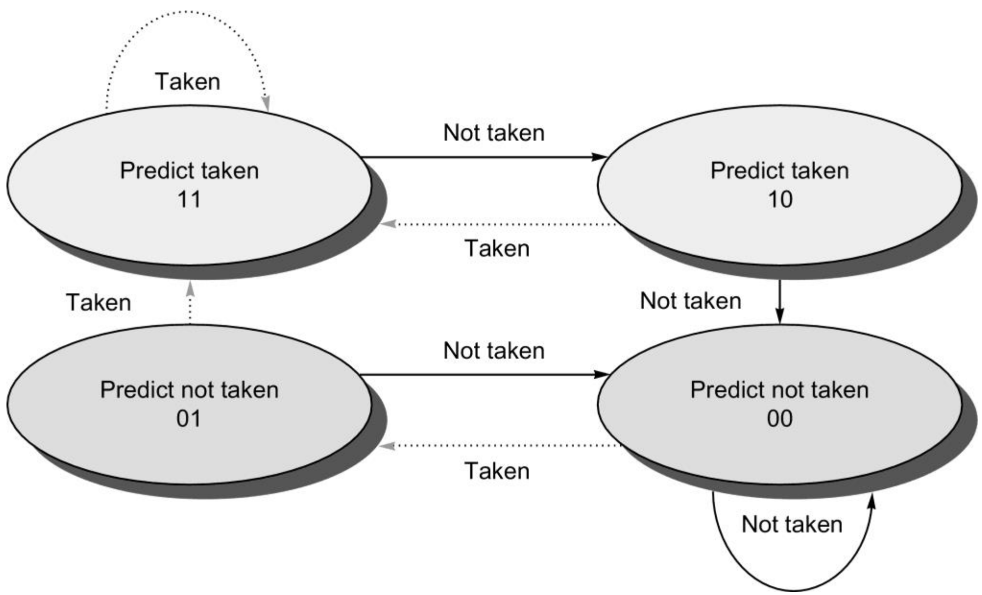
Dynamic branch prediction

- More accurate than static branch prediction
  - 80%-97% (dynamic) vs. 50%-80% (static)
- Some branches are hard to predict statically, but are easily predicted dynamically
  - Some examples
    - Not-taken during first half of execution, and taken during second half
    - Alternating taken/not-taken
- Takes into account branch context!
  - the branch’s own history (***local history***)
  - other branches’ histories (***global history***)

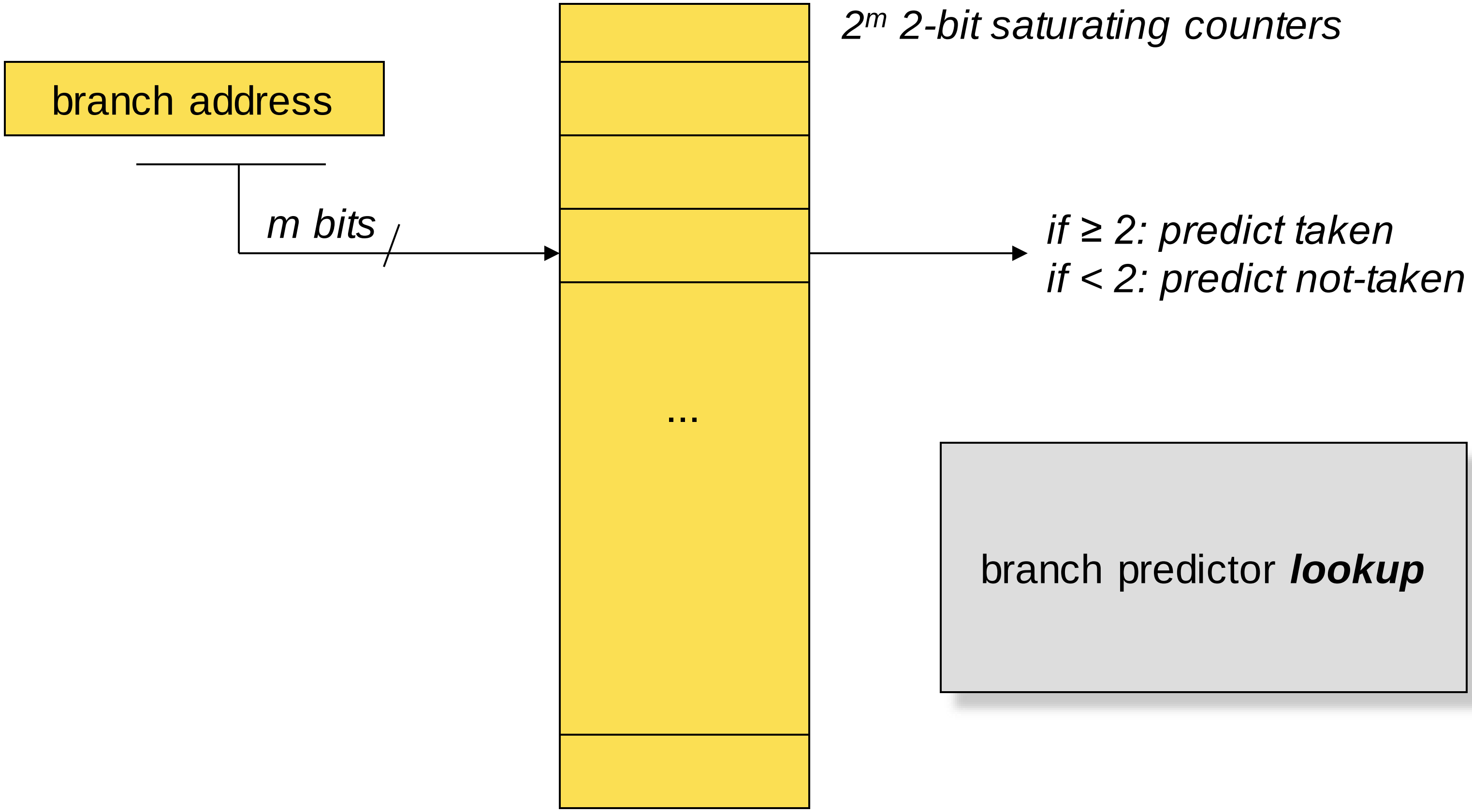
Bimodal predictor

- Hash the PC of a branch into a small buffer which stores the branch outcome and use this for making predictions

- A 2-bit predictor is does not change its prediction when it is wrong once (which typical occur on branches that check loop conditions):



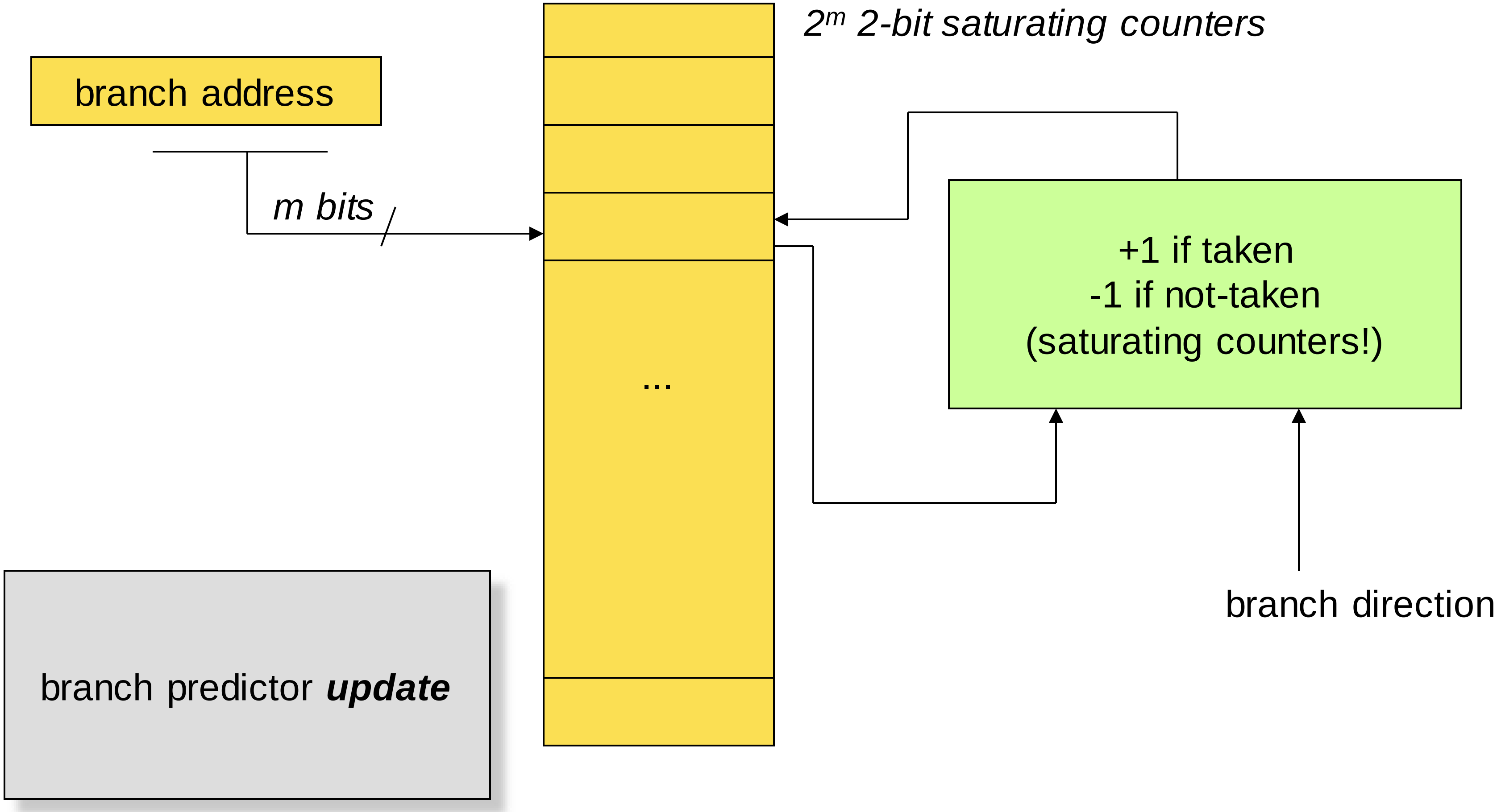
Bimodal predictor



branch predictor ***lookup***

James E. Smith (1981)

Bimodal predictor

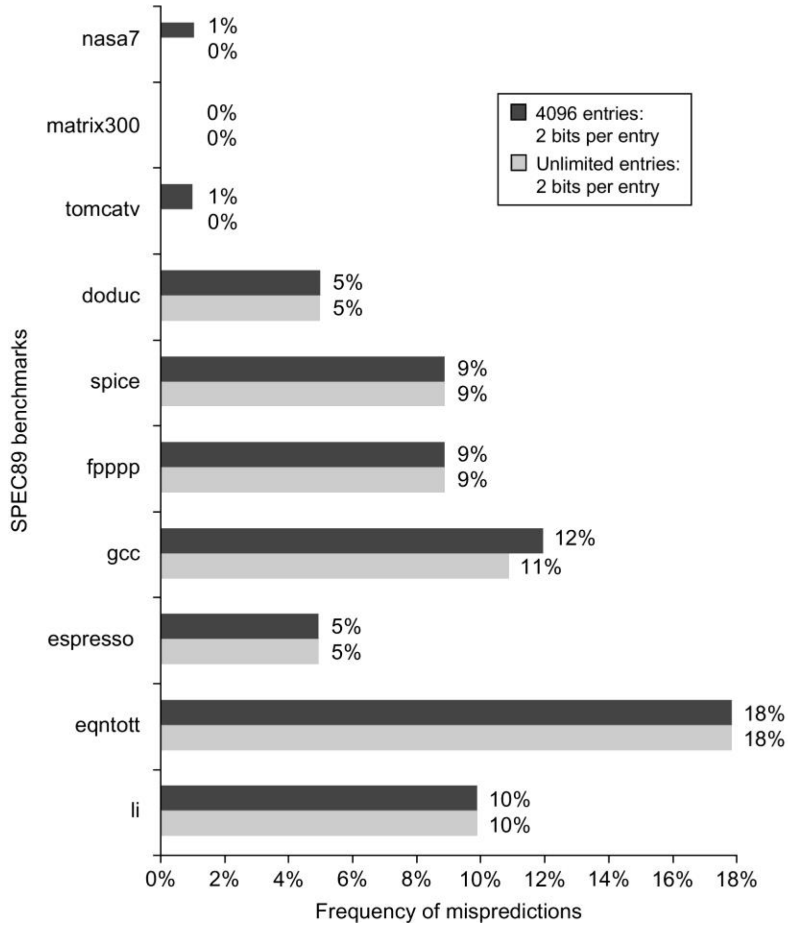


Example bimodal predictor

old state	*prediction*	*branch direction*	*new state*
00	**0**	0	00
00	**0**	0	00
00	**0**	1	01
01	**0**	1	10
10	**1**	1	11
11	**1**	1	11
11	**1**	0	10
10	**1**	1	11
11	**1**	1	11
11	**1**	0	10
10	**1**	0	01

Predictions and predictor state for a single PC (i.e., single branch) over time

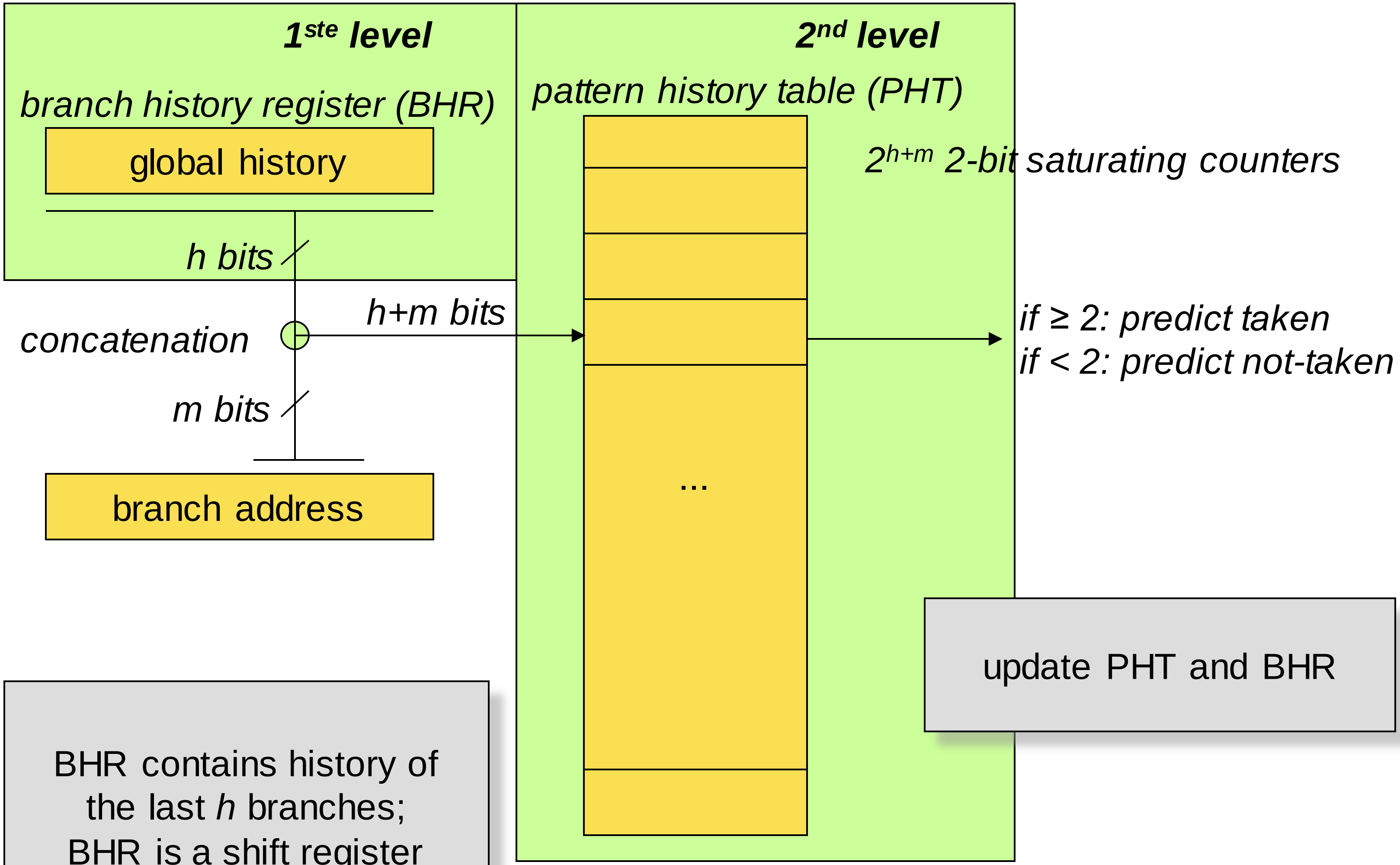
How good is the 2-bit predictor?



Two-level predictors

- *Correlating branch predictor*
- Uses (two levels of) branch history to make a prediction
  - Bimodal predictor uses the branch address only
  - Two-level predictors also use
    - *Local history* of that particular branch, or
    - *Global history* of all prior branches
    - history = outcome of prior branch executions
    - history acts as a shift register: newest branch outcome is inserted; oldest branch outcome is discarded
- Proposed by Tse-Yu Yeh and Yale N. Patt in 1991

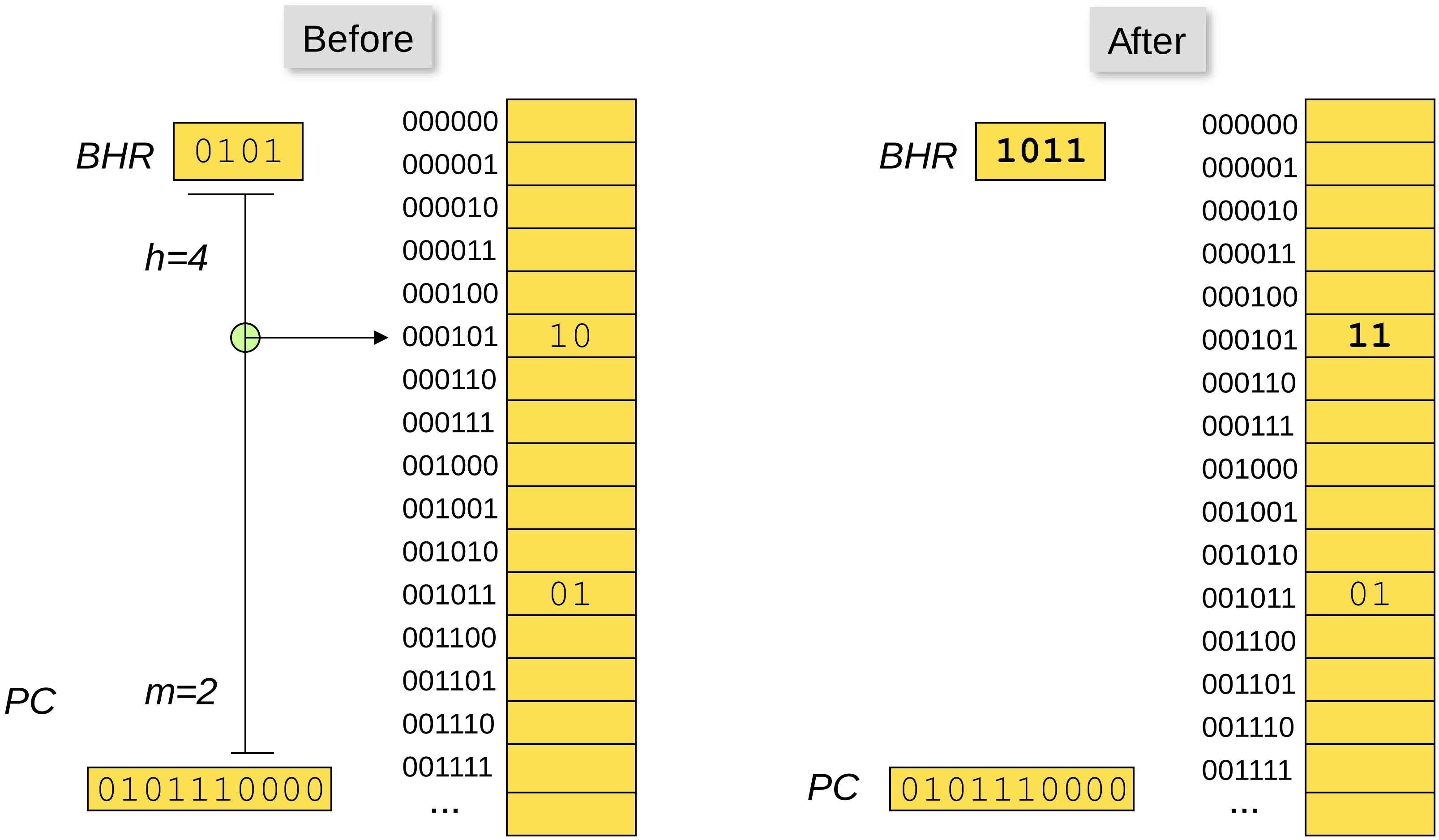
… using global history



update PHT and BHR

BHR contains history of the last *h* branches; BHR is a shift register

Example



Assume: branch direction is taken

Why does this work?

- Correlation among branches
  - For example, if a particular branch is taken (not-taken), probability may be high that next branches are also taken (not-taken), or vice versa
- Concrete examples
  - Branch conditions that depend on the same variable
  - A variable is control dependent on branch; a subsequent branch is data dependent on the variable; both branches will be correlated

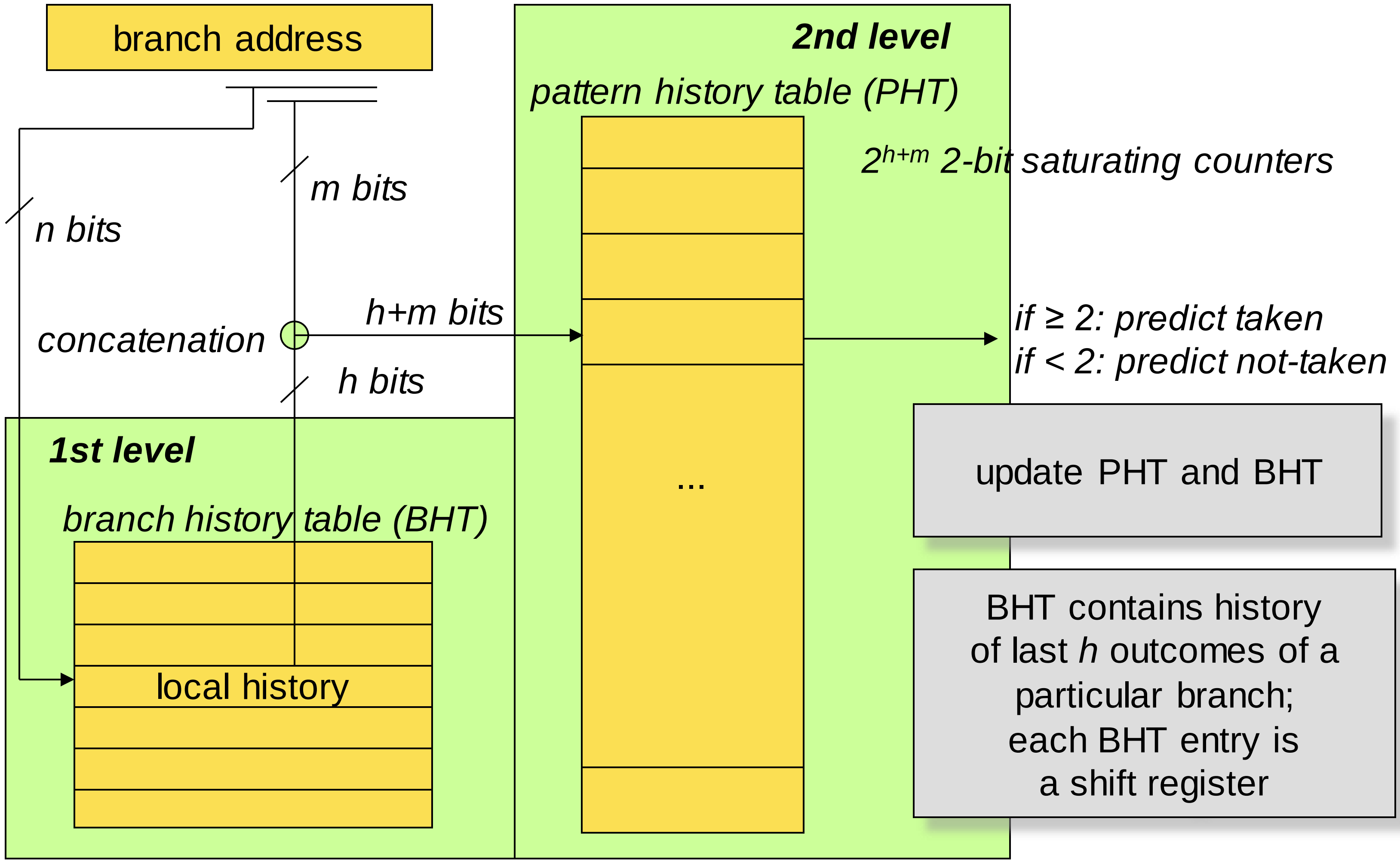
Example

```
x = 0;
if (some_condition) {
  x = 3;
}
if (another_condition) {
  y += 19;
}
if (x <= 0) {
  do_something ();
}
```

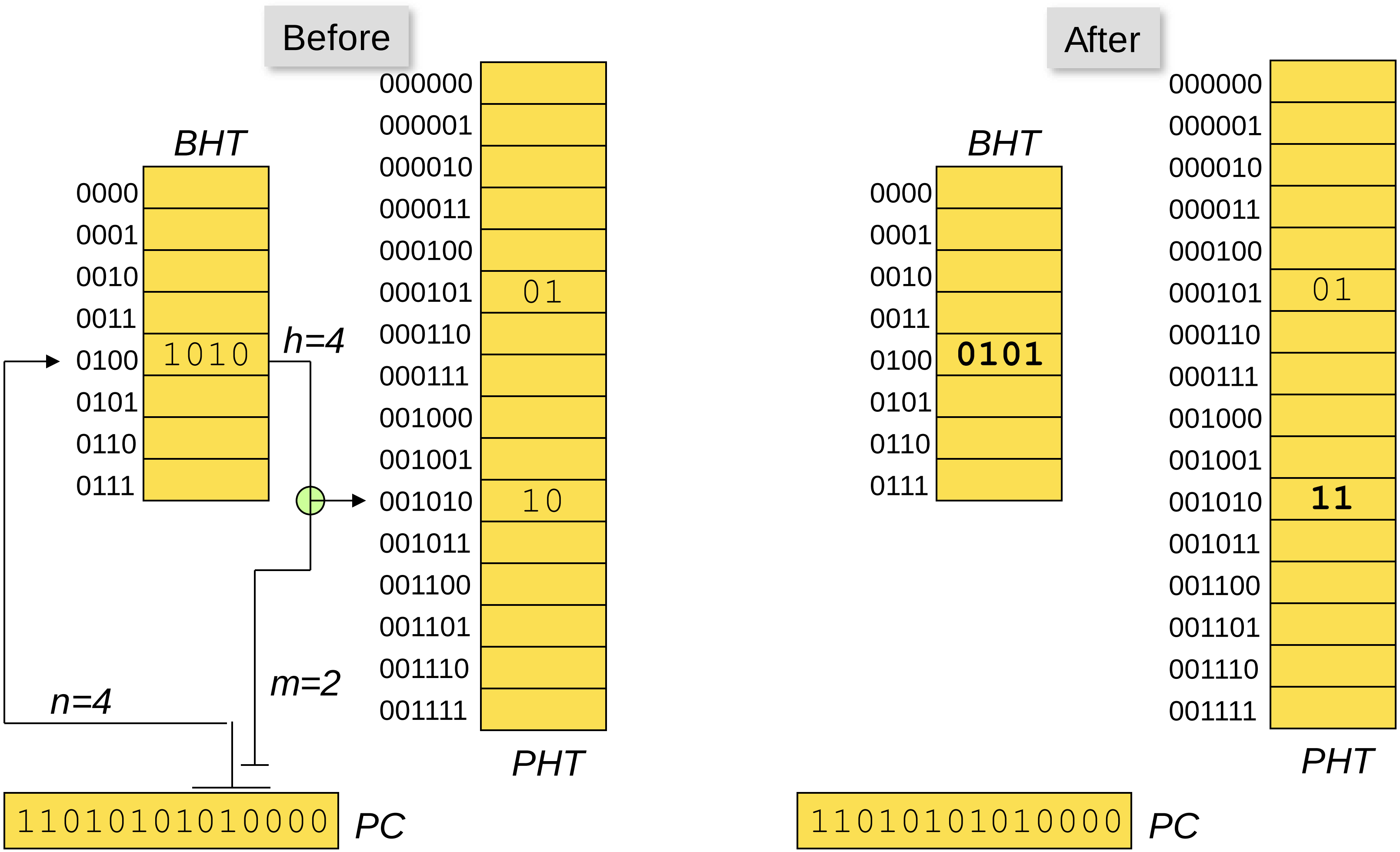
strong correlation
between 1st and 3rd branch

Note: there is no correlation between the 2nd and 3rd branch → branch prediction needs to learn!

… using local history



Example



Assume: branch direction is taken

Why does this work?

- Local behavior of a branch
- Examples
  - Alternating: 01010101…
  - Loops with limited number of iterations: 111011101110…

Two-level predictor implementations

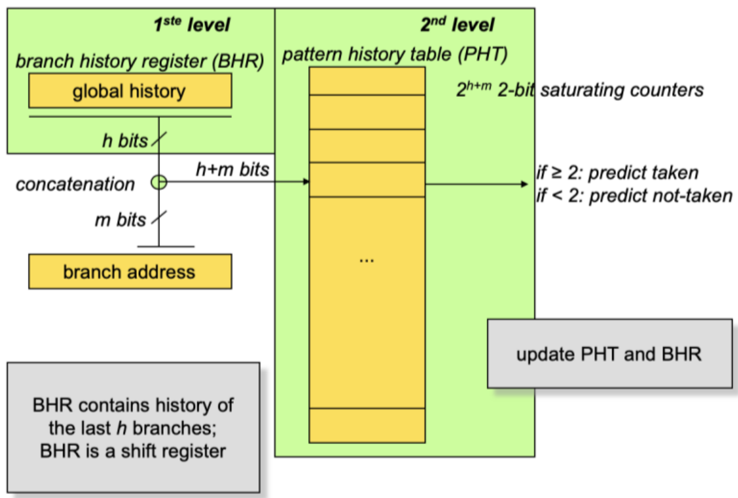
- If $m=0$
  - *Global pattern history table (gPHT)*
  - All branches share the same PHT
- If $m \neq 0$
  - *Per-address pattern history table (pPHT)*
  - PHT is partitioned based on branch address bits
- Four variations: GAg, GAp, PAg, PAp
  - 'G' global history, 'P' per-address (local)
  - 'A' adaptive
  - 'g' $\rightarrow$ gPHT and 'p' $\rightarrow$ pPHT

Good configurations

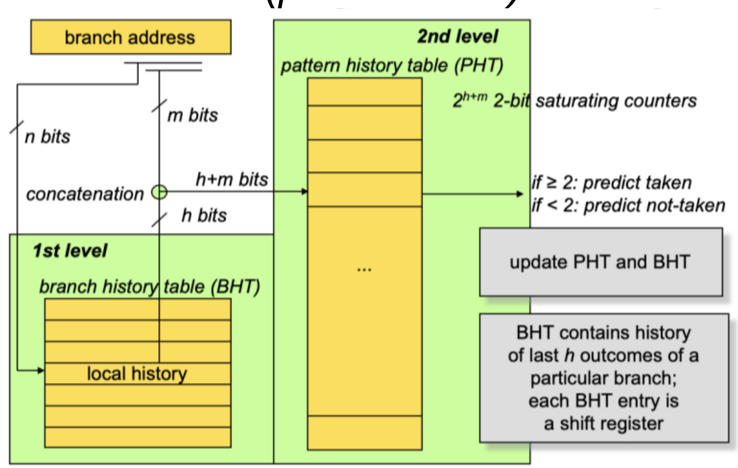
- Obtained through experimental evaluation by Yeh/Patt (1991)
- GAg
  - BHR: 18 bits
  - PHT: 2^{18} x 2 bits
- PAg
  - BHT: 2^{11} x 12 bits
  - PHT: 2^{12} x 2 bits
- PAp
  - BHT: 2^{11} x 6 bits
  - PHT: 2^9 x 2^6 x 2 bits
- Prediction accuracy of 97%

Remember: gPHT: m=0

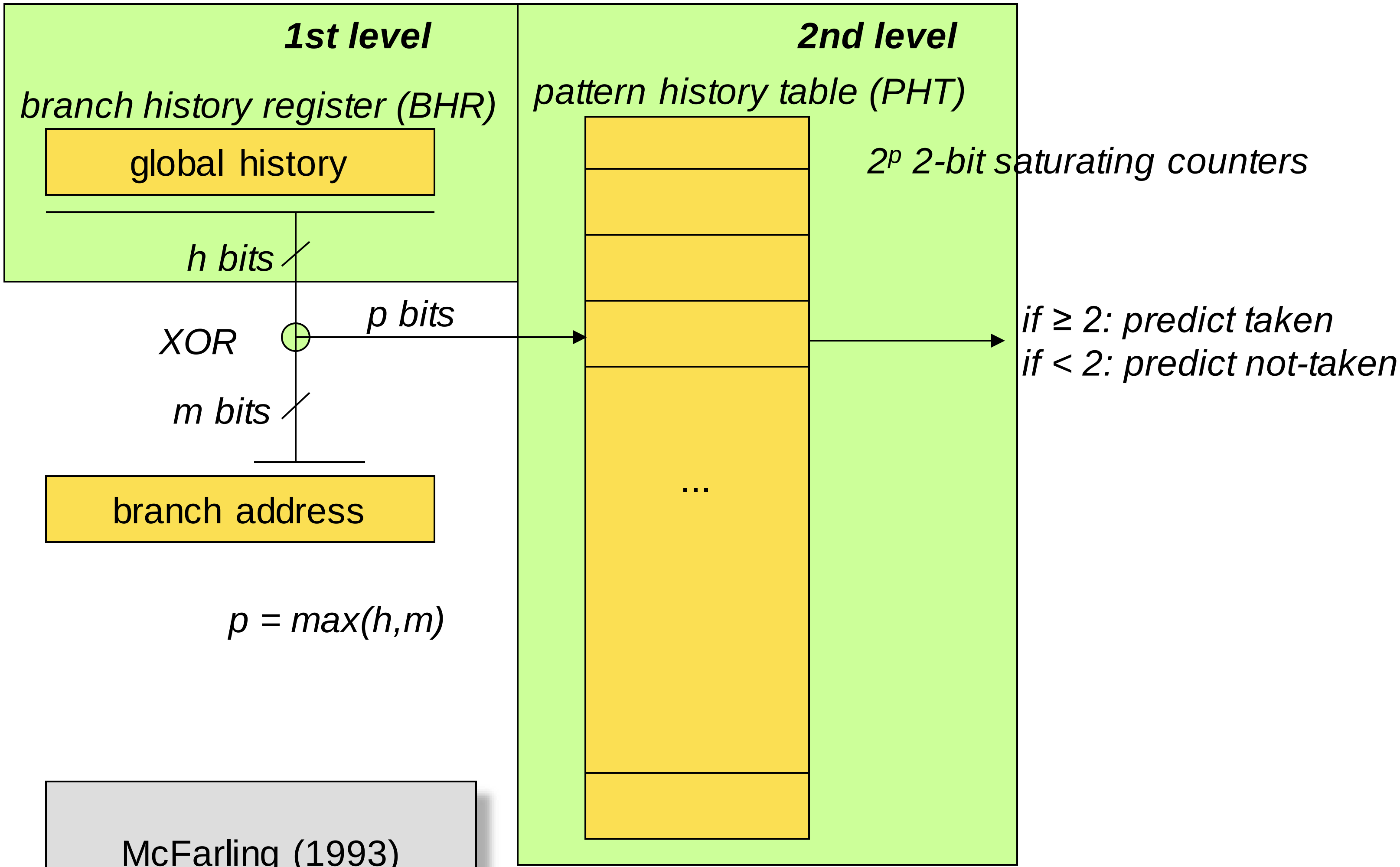
G (global)



P (per-address)



gshare



McFarling (1993)

GAp vs. gshare

- GAp needs to make a choice
  - How many bits to concatenate from BHR vs. branch address?
  - e.g., 5 bits from BHR and 5 bits from branch address
- gshare doesn't need to make this choice
  - e.g., 10 bits from BHR and 10 bits from branch address
  - More information is used to index the PHT

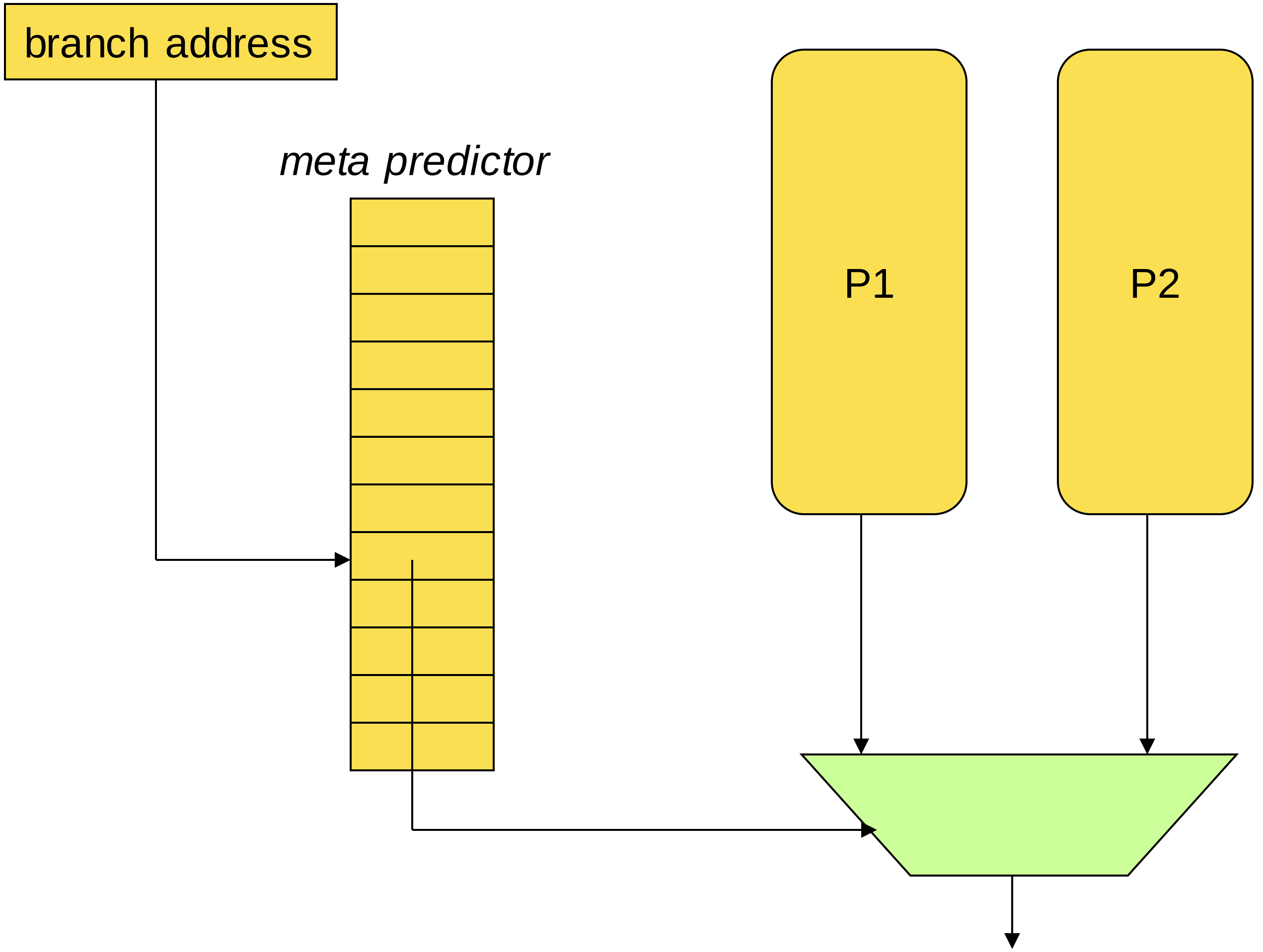
Towards hybrid branch prediction

- Reasons for mispredictions?
- Branches may be hard to predict
  - During training phase of the branch predictor, e.g., after a context switch
- Interference or aliasing
  - Mostly negative interference
    - Branch predictor is too small for the number of branches
    - Hence, branches with different behavior may be mapped onto the same PHT entry
- Branch behavior doesn't match branch predictor type

Hybrid branch prediction

- Idea: Combine different types of branch predictors and learn which one is most accurate for which branch

Tournament predictor



McFarling in 1993

Tournament predictor

- Meta predictor determines which predictor will be followed
  - If $< 2 \rightarrow$ P1; If $\geq 2 \rightarrow$ P2
- Update meta predictor
  - If both correct or incorrect: do nothing
  - If P1 is correct and P2 incorrect: decrement
  - If P1 is incorrect and P2 is correct: increment
- Both predictors are updated
  - Irrespective of whether P1 or P2 was followed

Tournament predictor (cont.)

- Various flavors exist
- Any predictor can be part of a hybrid predictor
- Indexing of meta predictor is to be chosen
- Hybrid predictor typically combines local and global history based predictors

Alpha 21264

- Hybrid predictor
  - PAg
    - 1st level: 1K 10-bit entries
    - 2nd level: 1K 3-bit saturating counters
  - GAg
    - 4K 2-bit saturating counters
    - 12-bit global history
  - Meta predictor
    - 4K 2-bit saturating counters
    - Indexed using 12-bit global history

IBM POWER4

- Hybrid predictor
  - Bimodal predictor
    - 16K 1-bit saturating counters
  - Gshare
    - 16K 1-bit saturating counters
    - 11-bit global history
  - Meta predictor
    - 16K 1-bit saturating counters
    - gshare indexing

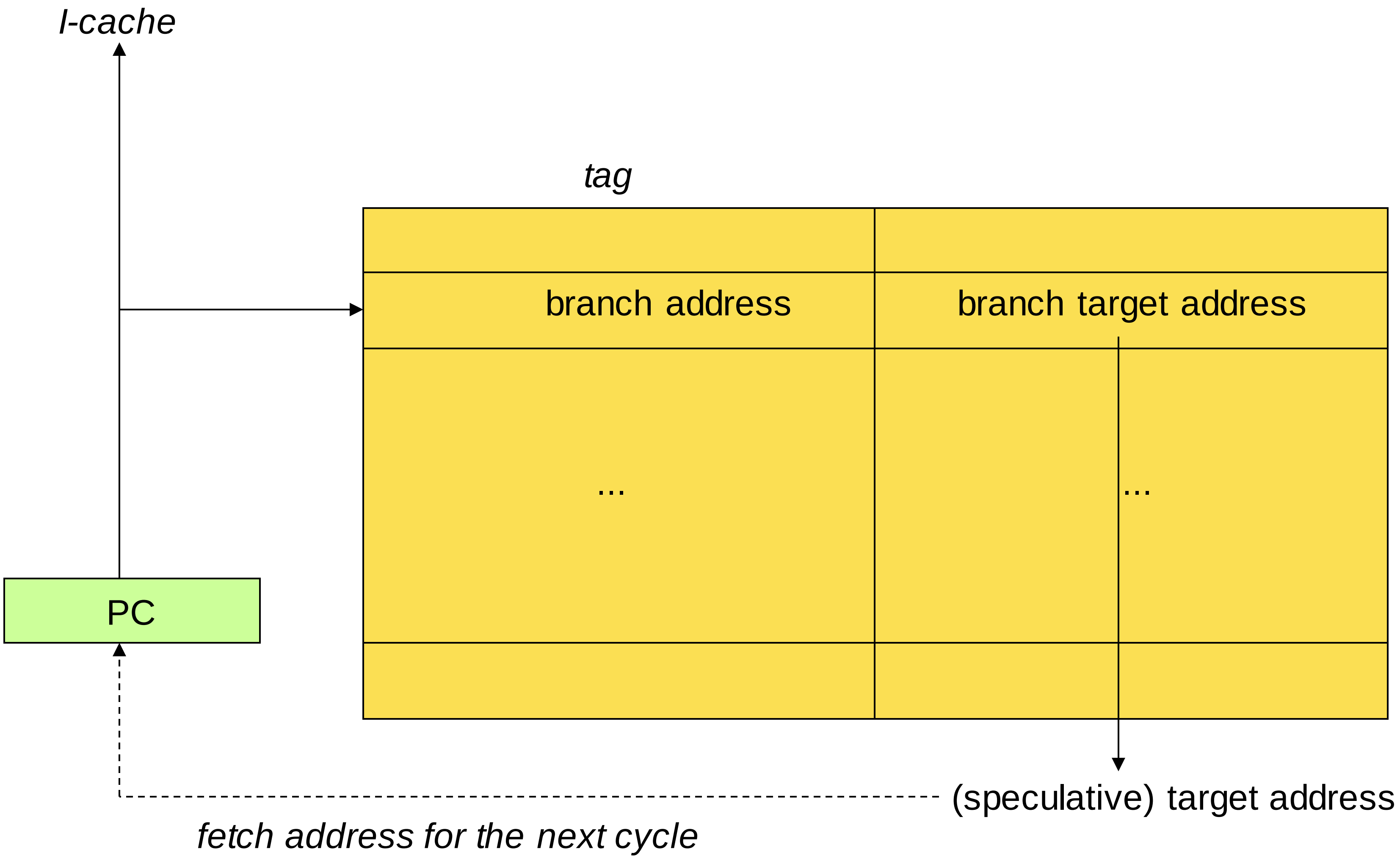
BRANCH TARGET PREDICTION

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Branch target prediction

- Branch target buffer (BTB) or Branch target address cache (BTAC)
- Small (set-associative) cache
- Idea:
  - Input = branch address
  - Output = branch target address
  - Accessed in the same cycle as the I-cache access
  - Branch target address is the fetch address in the next cycle
- Done for both conditional as well as non-conditional branches

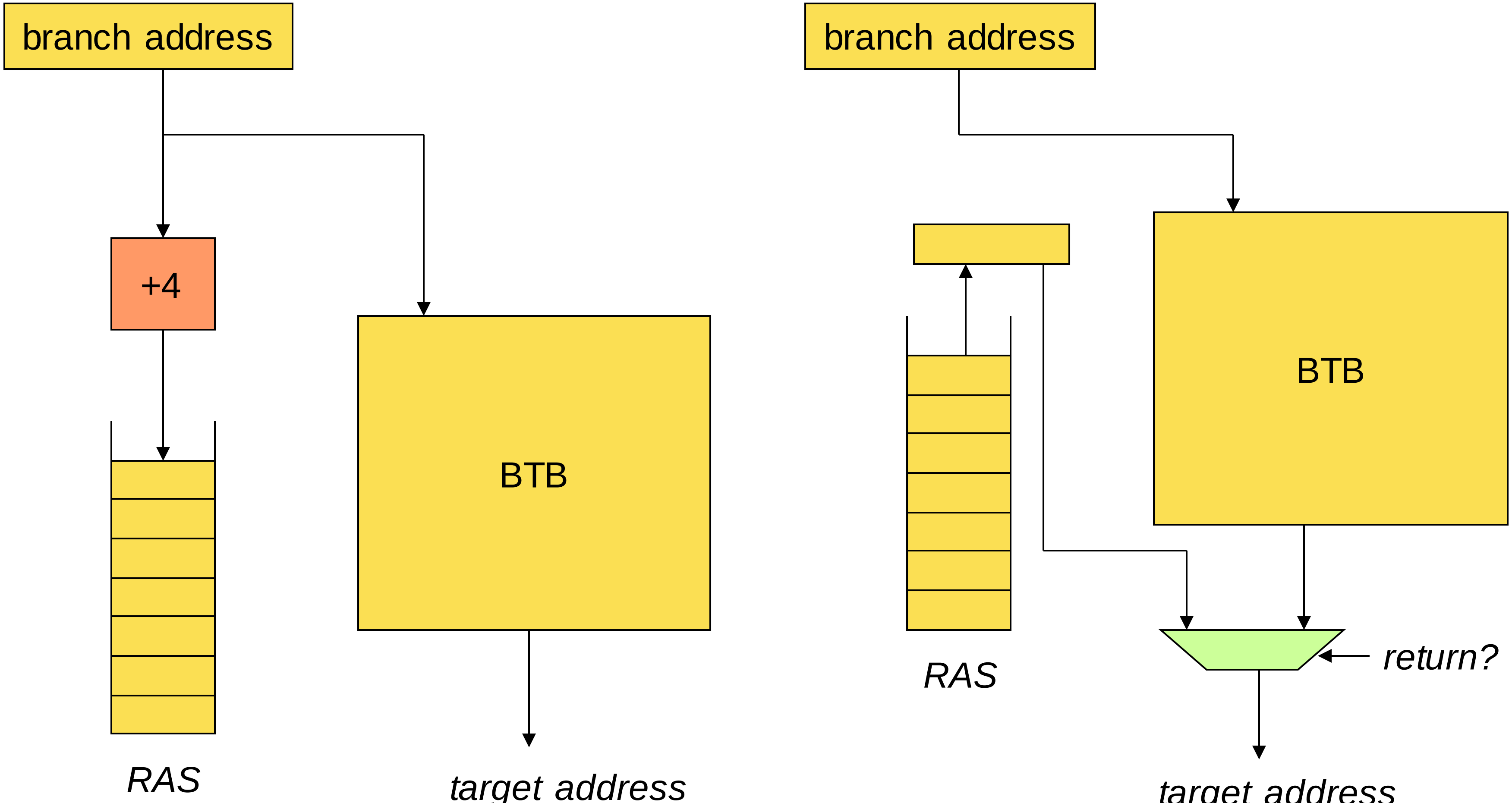
Branch Target Buffer



Return Address Stack

- Function calls are easy to predict
- Function returns are much more complicated to predict
- Solution: Return Address Stack (RAS)
  - by Kaeli and Emma in 1991
  - In Pentium 4: 16 entries
  - Circular buffer
    - Push function return address on RAS; pop the address upon a return
    - If depth of function call stack > RAS size: incorrect predictions

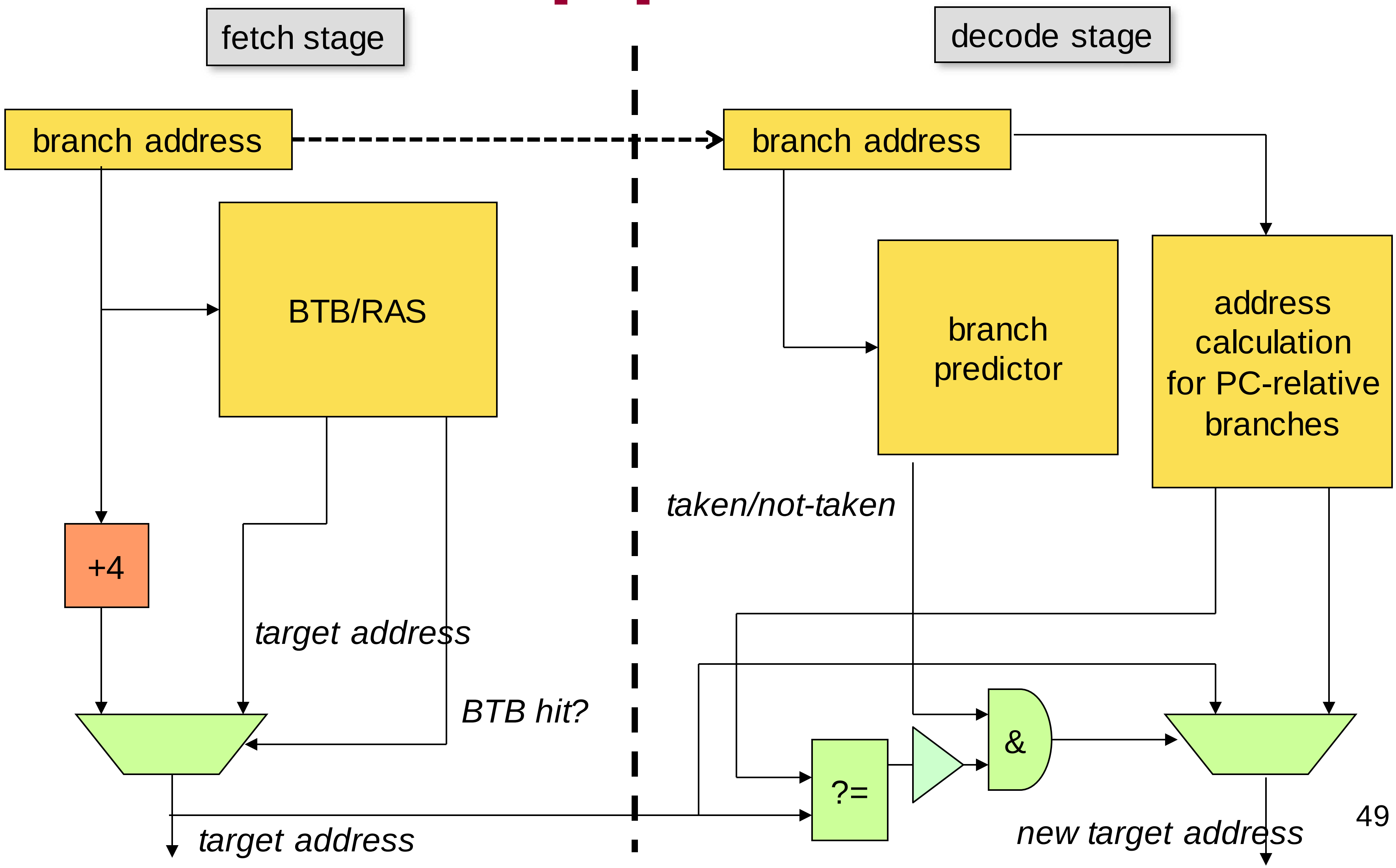
Return Address Stack



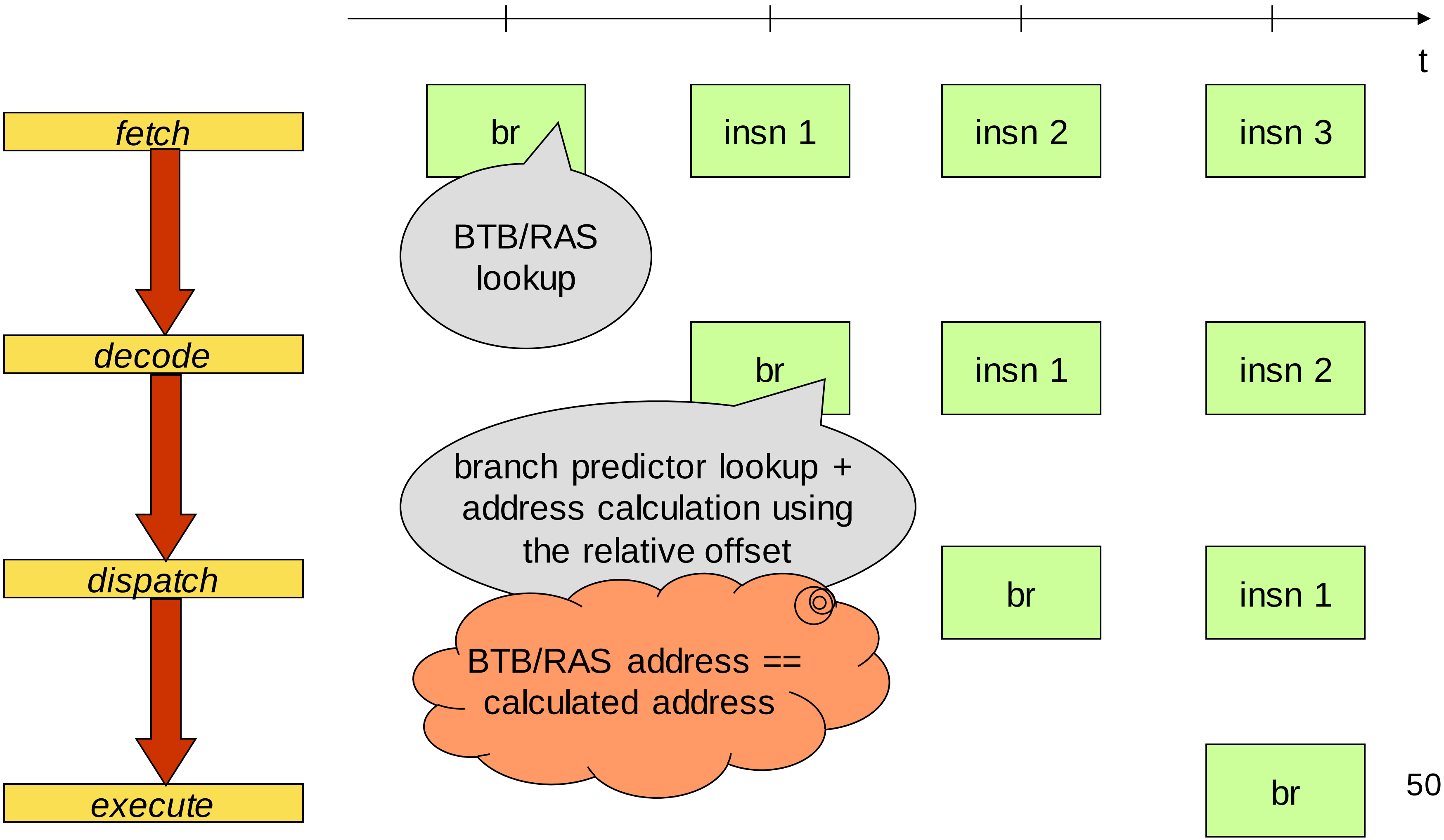
In case of a function call

In case of a return

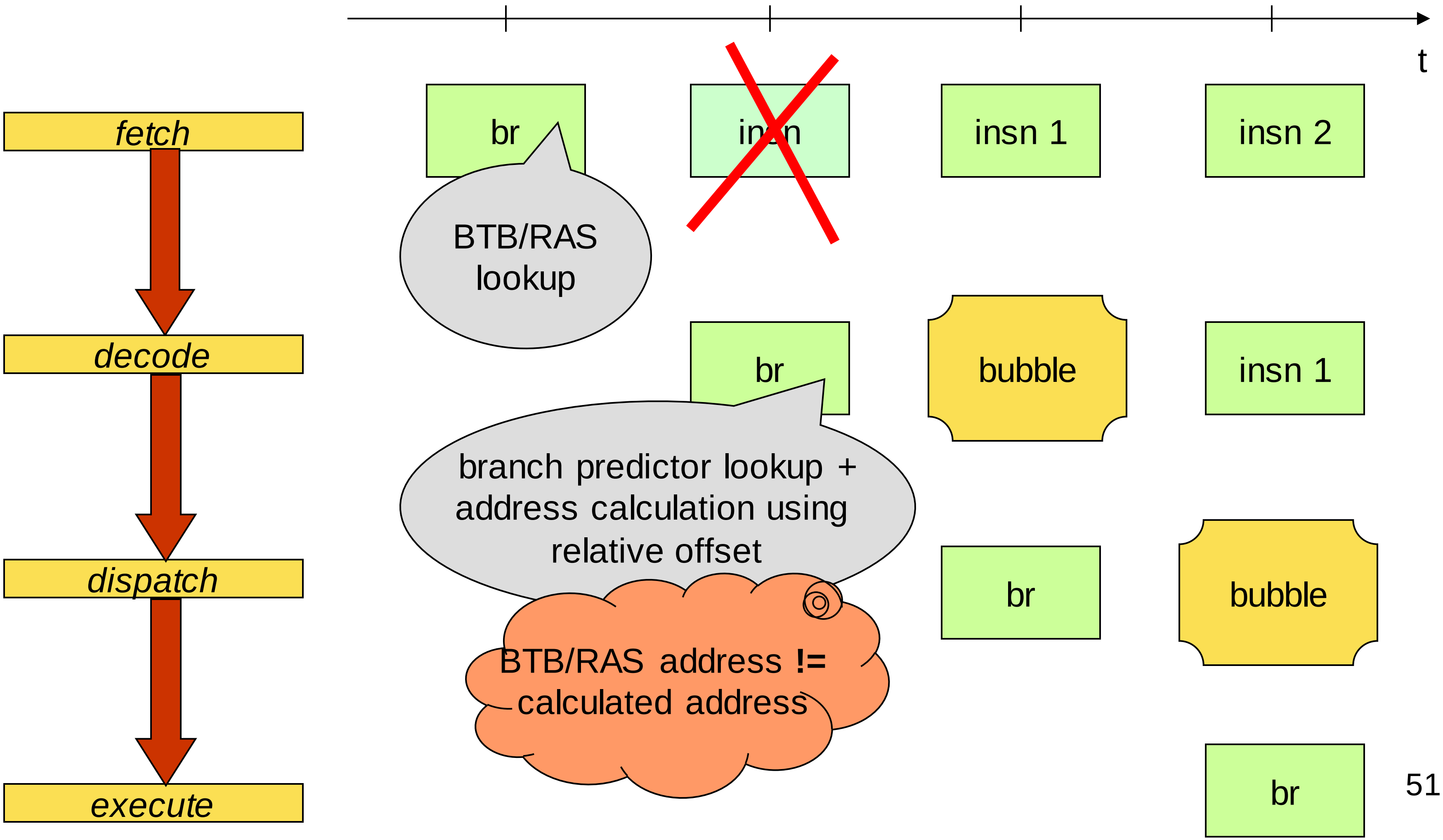
Branch prediction in the pipeline



Branch prediction predicts taken and target address is correct



Branch predictor predicts taken but target address is incorrect



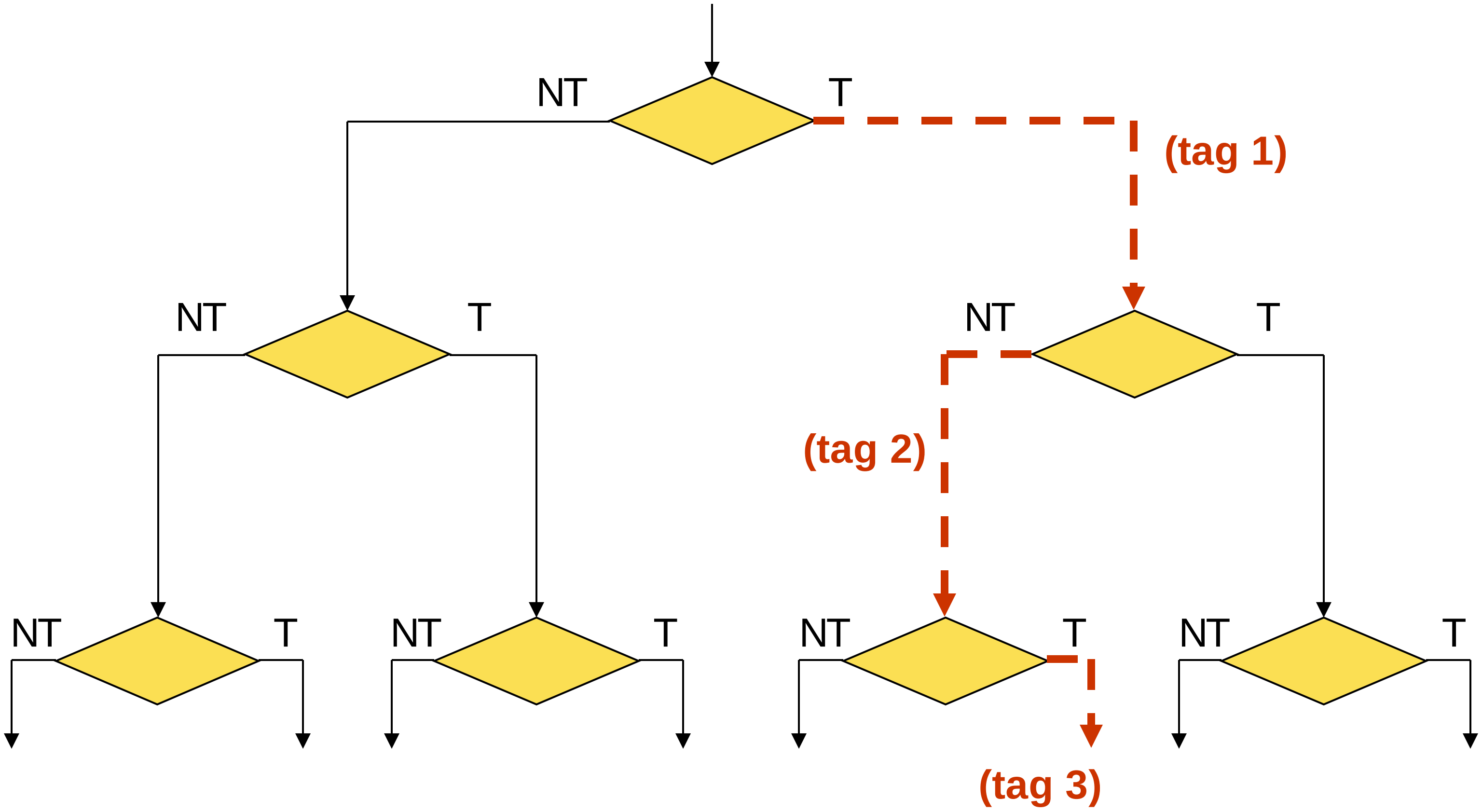
BRANCH MISPREDICTION RECOVERY

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Speculative execution

- Predict branch target address and start fetching and executing instructions along the predicted path
  - no completion of speculative insns!
- There might be multiple branches in flight along the predicted path
  - tags are added

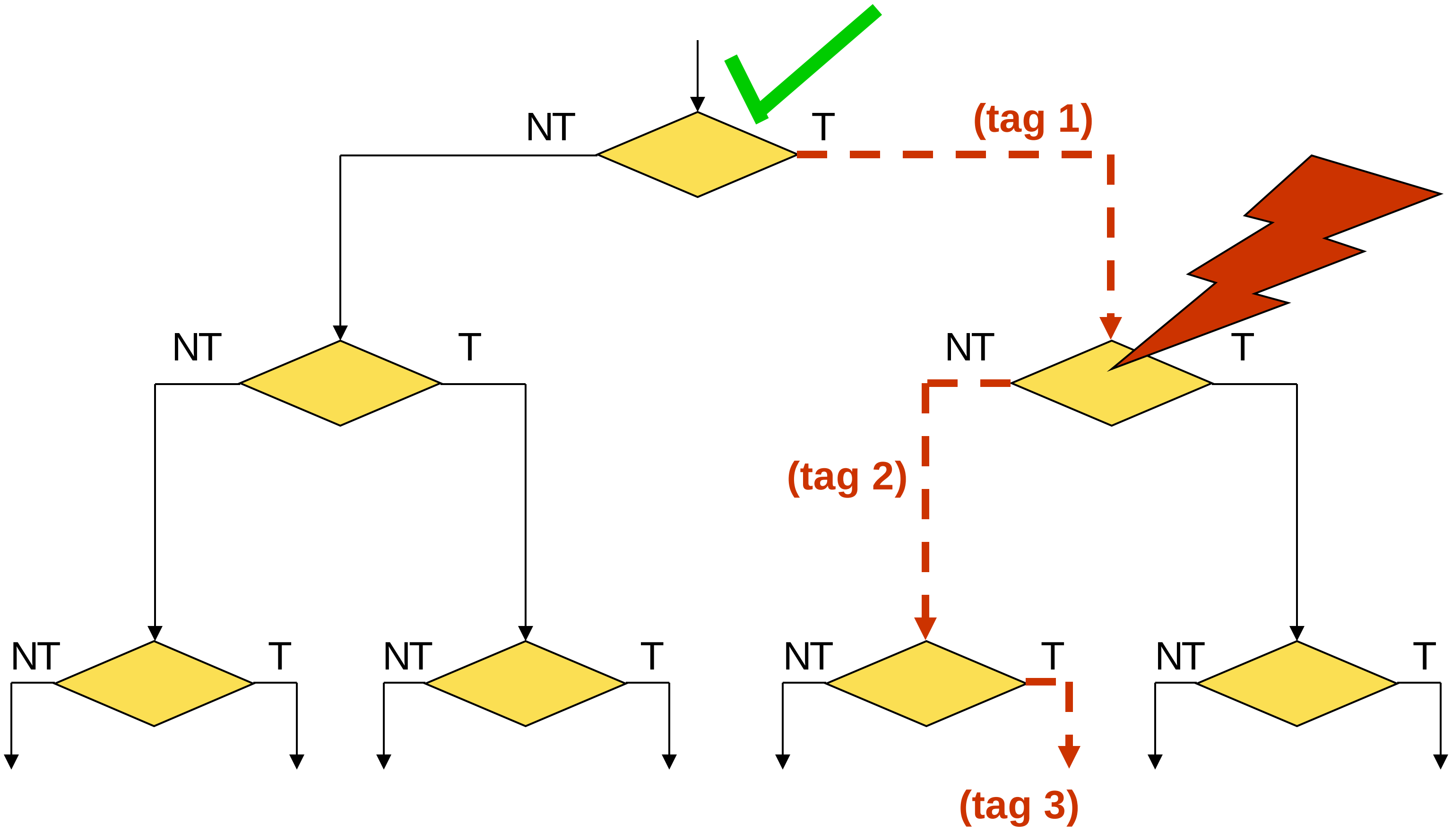
Speculative execution



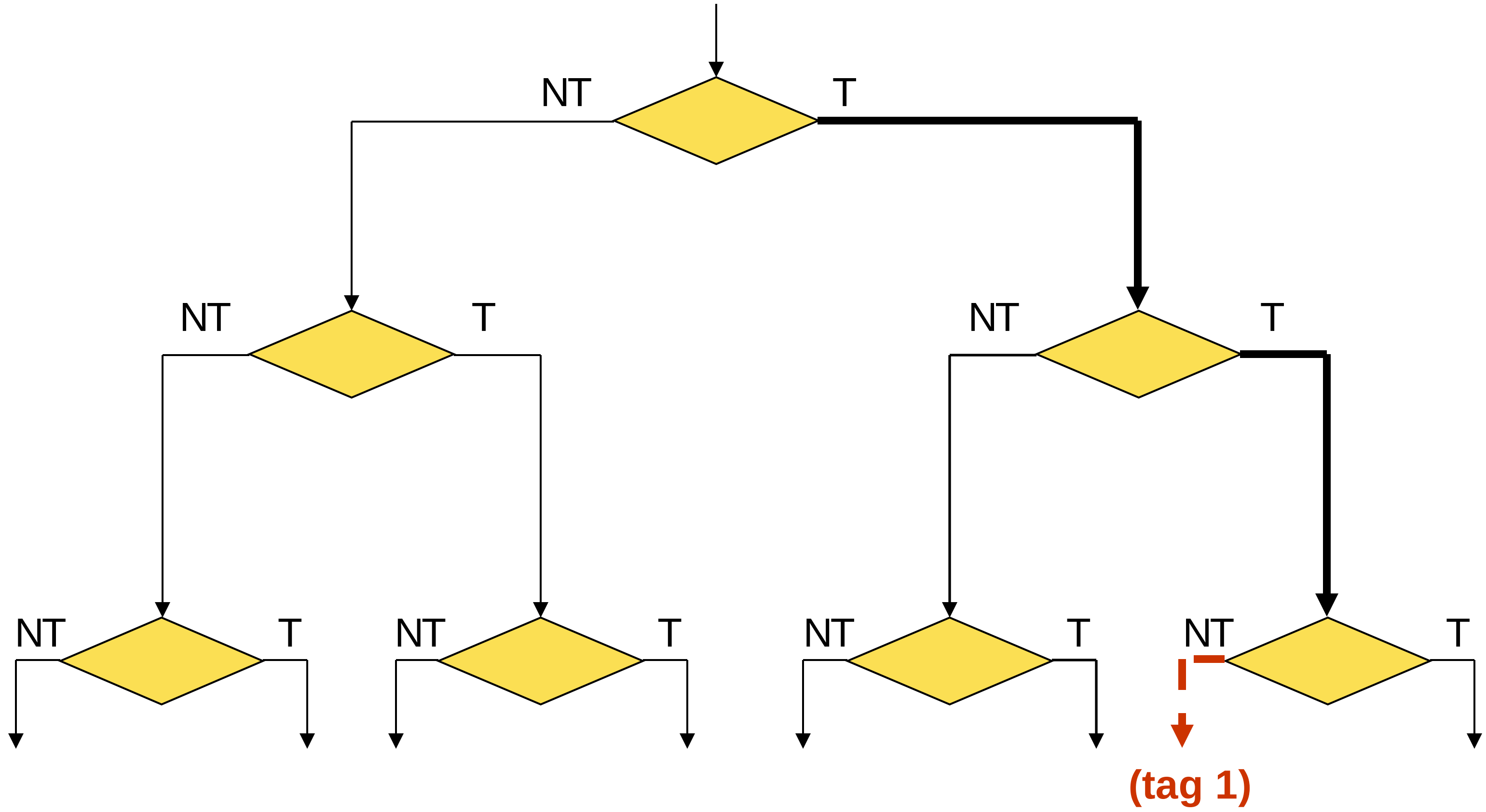
Pipeline squashing

- Branch outcome is known at execution stage
- If correctly predicted
  - Corresponding tags are deallocated
  - Instructions become non-speculative
- If mispredicted
  - Wrong-path instructions are nullified
  - Start fetching instructions along correct path

Speculative execution



Squash wrong-path insns and fetch insns along correct path



SUMMARY

Summary

- Branch prediction is critical to achieving high performance
  - Must be able to fill the pipeline with (mostly) useful instructions
  - The deeper the pipeline, the more important branch prediction becomes
  - Luckily, branches are quite predictable (90-99% prediction accuracy is common)
- Branch prediction schemes
  - Statically predict branch direction: Rule-based, program-based, and profile-based
  - Dynamically predict branch direction: Bimodal predictors, two-level predictors, and tournament predictors
  - Predict the branch target: Branch Target Buffer (BTB) and Return Address Stack (RAS)