



西安交通大学  
XIAN JIAOTONG UNIVERSITY

# Modern Computer Architecture

## Lecture5 Instruction Level Parallelism (II)

**Hongbin Sun**

**国家集成电路人才培养基地**

**Xi'an Jiaotong University**

# Branches must be resolved quickly

- In our loop-unrolling example, we relied on the fact that branches were under control of “fast” integer unit in order to get overlap!

- Loop:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

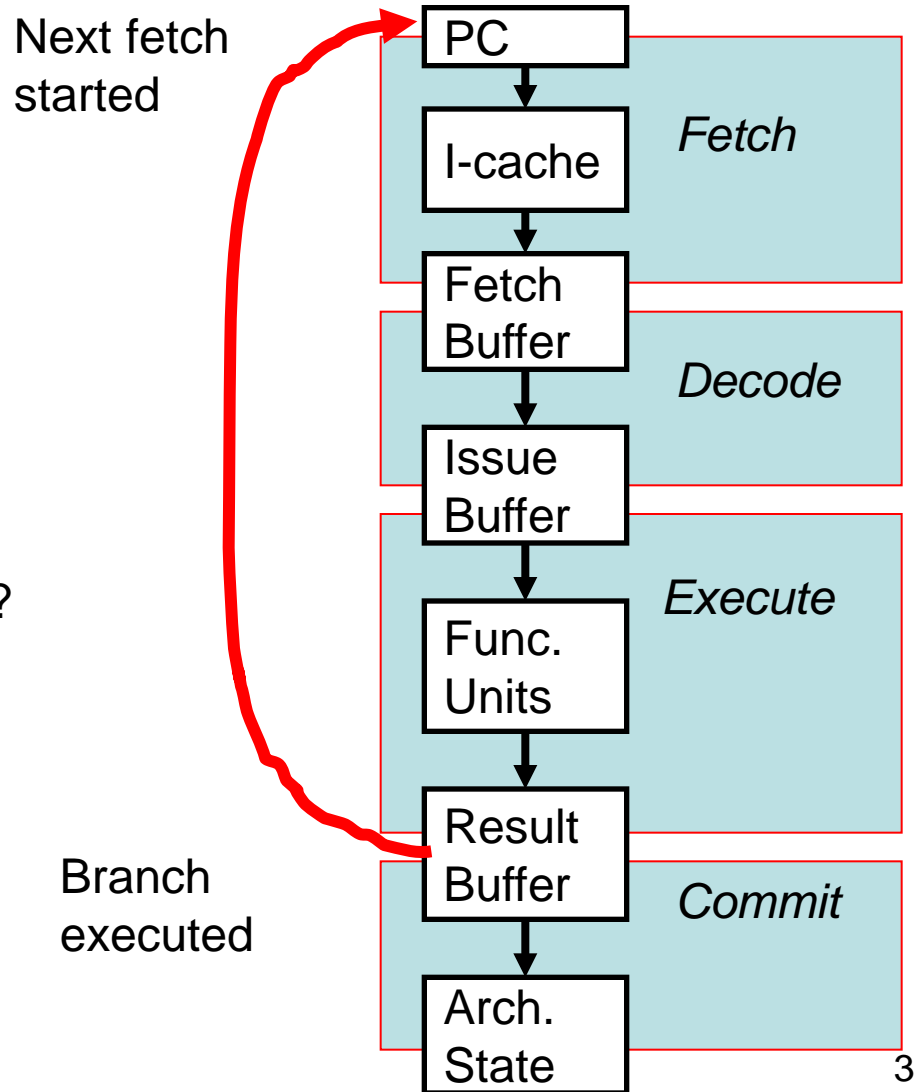
- What happens if branch depends on result of multd??
  - We completely lose all of our advantages!
  - Need to be able to “predict” branch outcome.
  - If we were to predict that branch was taken, this would be right most of the time.
- Problem **much** worse for superscalar machines!

# Control Flow Penalty

*Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !*

*How much work is lost if pipeline doesn't follow correct instruction flow?*

*~ Loop length x pipeline width*



# MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

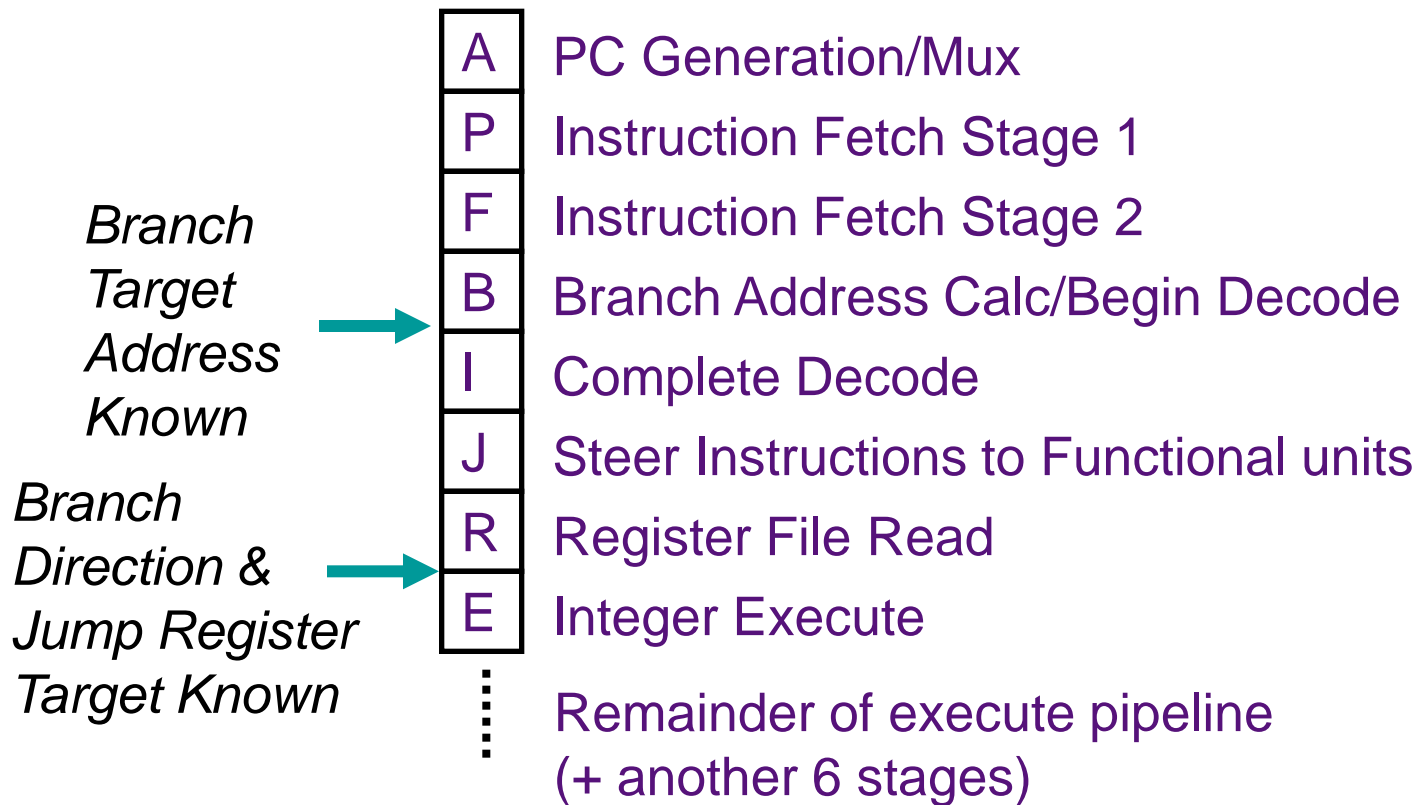
- 1) Is the preceding instruction a taken branch?
- 2) If so, what is the target address?

<i>Instruction</i>	<i>Taken known?</i>	<i>Target known?</i>
J	After Inst. Decode	After Inst. Decode
JR	After Inst. Decode	After Reg. Fetch
BEQZ/BNEZ	After Reg. Fetch*	After Inst. Decode

\*Assuming zero detect on register read

# Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages  
(in-order issue, 4-way superscalar, 750MHz, 2000)



# Reducing Control Flow Penalty

## Software solutions

- *Eliminate branches - loop unrolling*  
Increases the run length
- *Reduce resolution time - instruction scheduling*  
Compute the branch condition as early as possible (of limited value)

## Hardware solutions

- Find something else to do - *delay slots*  
Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*  
*Speculative execution* of instructions beyond the branch

# Branch Prediction

- ***Motivation:***
  - Branch penalties limit performance of deeply pipelined processors
  - 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.
  - ***Mispredict recovery mechanisms:***
    - *Keep result computation separate from commit*
    - Kill instructions following branch in pipeline
    - Restore state to state following branch

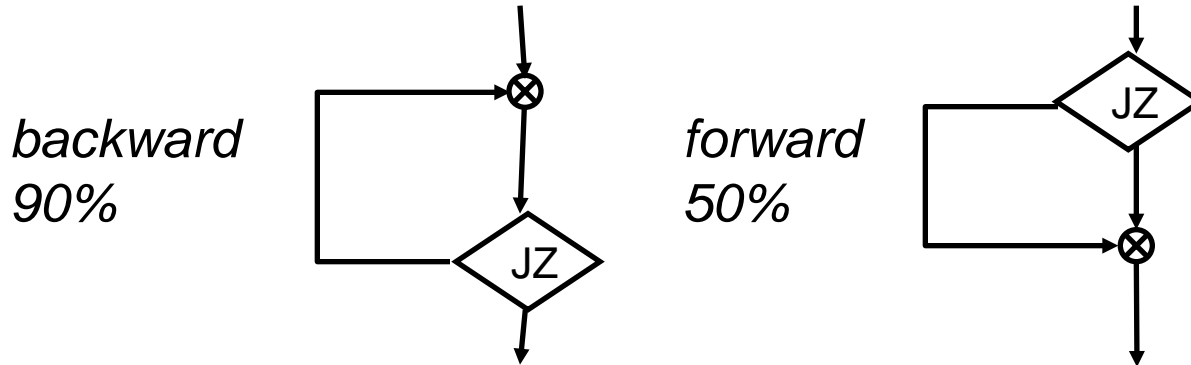
# Case for Branch Prediction when Issue $N$ instructions per clock cycle

- Branches will arrive up to  $n$  times faster in an  $n$ -issue processor
  - Amdahl's Law  $\Rightarrow$  relative impact of the control stalls will be larger with the lower potential CPI in an  $n$ -issue processor
  - conversely, need branch prediction to 'see' potential parallelism
- Performance =  $f(\text{accuracy, cost of misprediction})$ 
  - Misprediction  $\Rightarrow$  **Flush Reorder Buffer**
  - **Questions: How to increase accuracy or decrease cost of misprediction?**
- Decreasing cost of misprediction
  - Reduce number of pipeline stages before result known
  - Decrease number of instructions in pipeline
  - **Both contraindicated in high issue-rate processors!**



# Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



ISA can attach preferred direction semantics to branches, e.g.,  
Motorola MC88110

*bne0 (preferred taken)*

*beq0 (not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP  
PA-RISC, Intel IA-64

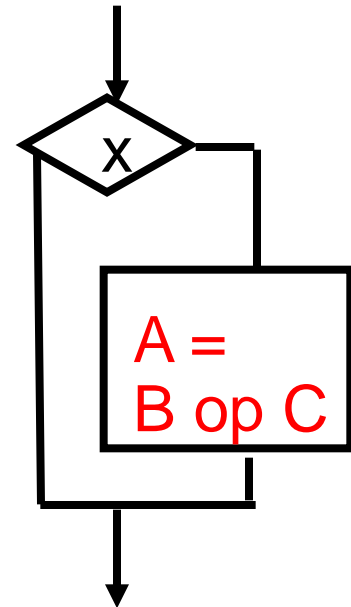
typically reported as ~80% accurate

# Predicated Execution

- Avoid branch prediction by turning branches into conditionally executed instructions:

**if (x) then A = B op C else NOP**

- If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction
  - This transformation is called **“if-conversion”**
- Drawbacks to conditional instructions
    - Still takes a clock even if “annulled”
    - Stall if condition evaluated late
    - Complex conditions reduce effectiveness; condition becomes known late in pipeline



# Dynamic Branch Prediction

*learning based on past behavior*

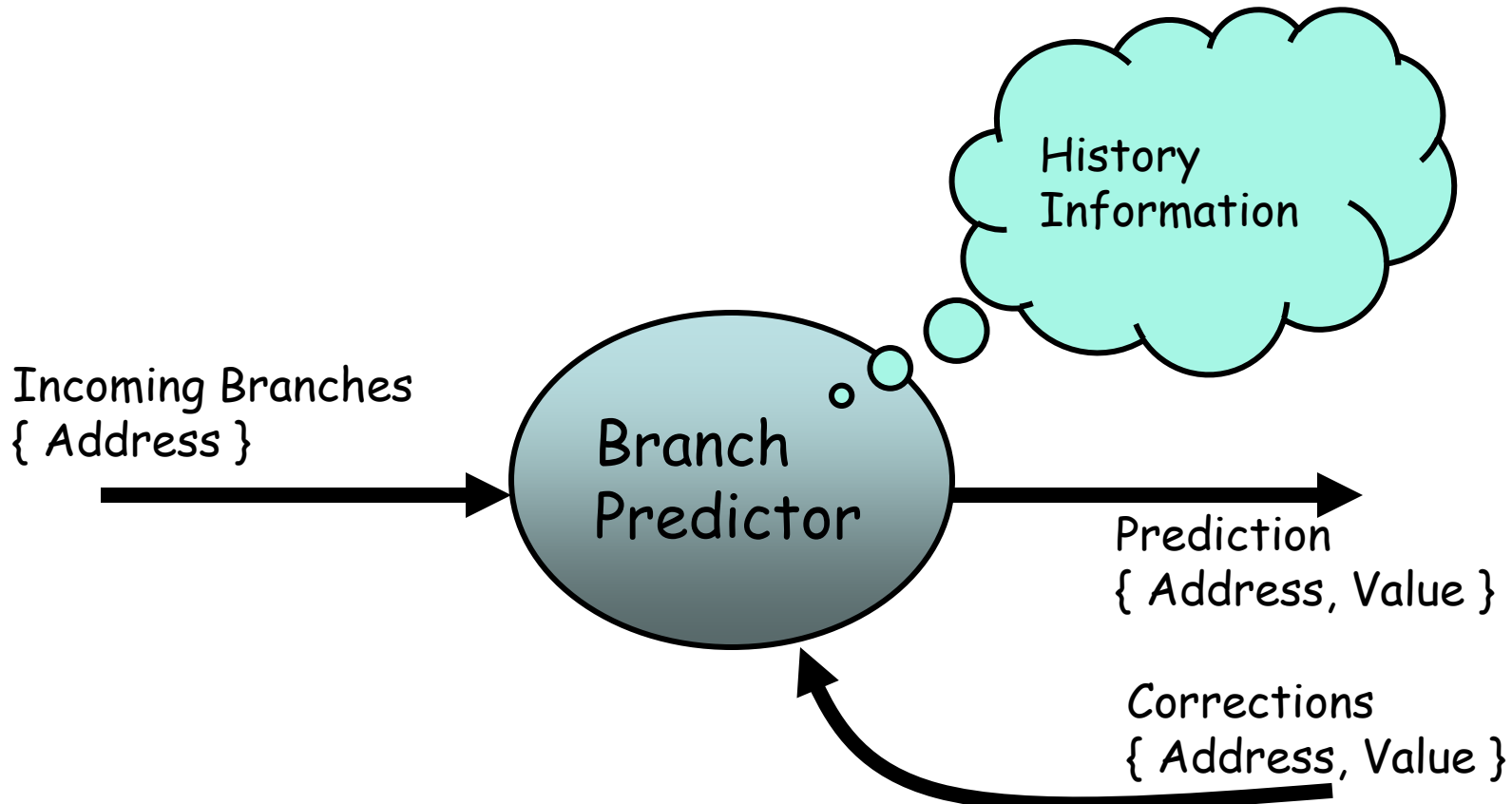
## *Temporal correlation*

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

## *Spatial correlation*

Several branches may resolve in a highly correlated manner (*a preferred path of execution*)

# Dynamic Branch Prediction Problem

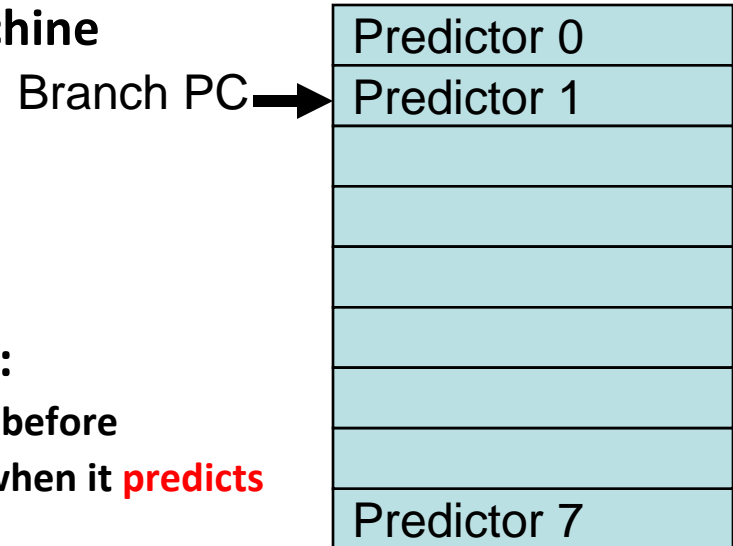


- Incoming stream of addresses
- Fast outgoing stream of predictions
- Correction information returned from pipeline

# What does history look like?

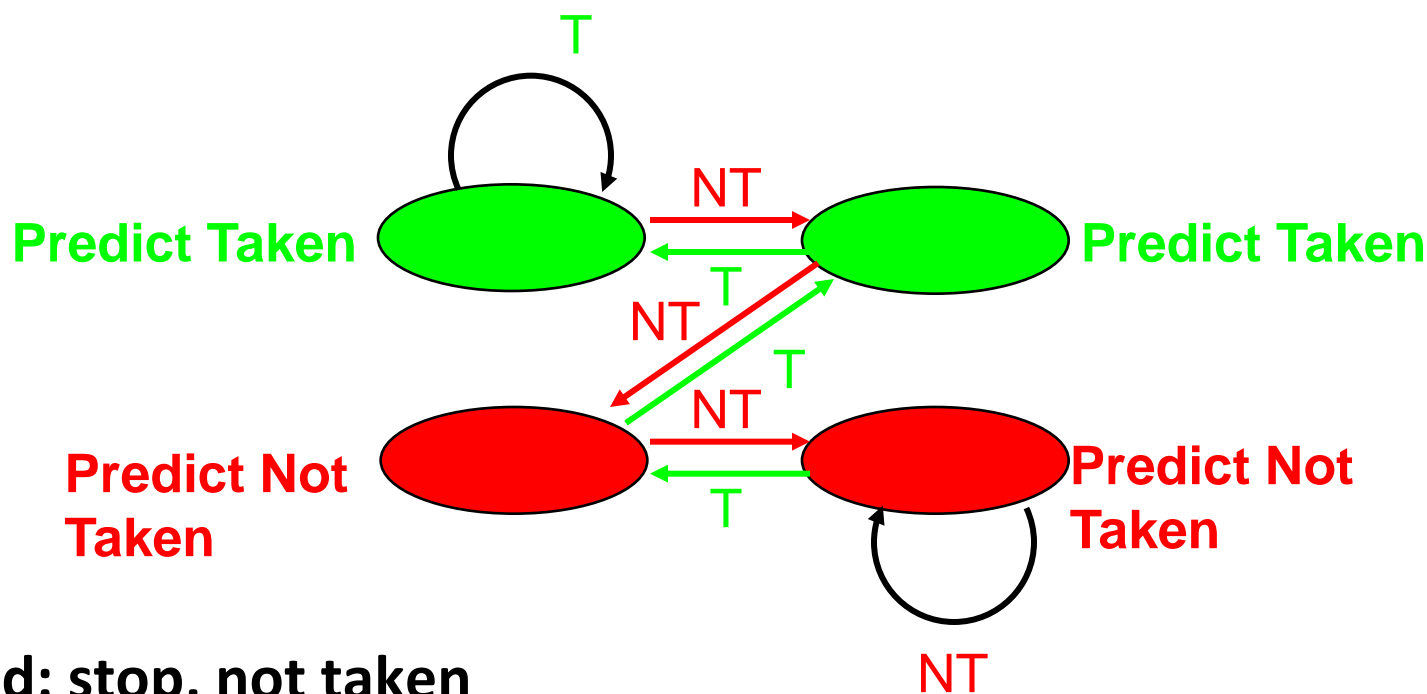
## E.g.: One-level Branch History Table (BHT)

- Each branch given its own predictor state machine
- BHT is table of “Predictors”
  - Could be 1-bit, could be complex state machine
  - Indexed by PC address of Branch – without tags
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case: when it **exits** instead of looping as before
  - First time through loop on *next* time through code, when it **predicts exit** instead of looping
- Thus, most schemes use at least 2 bit predictors
- Performance = f(accuracy, cost of misprediction)
  - Misprediction ⇒ **Flush Reorder Buffer**
- In Fetch state of branch:
  - Use Predictor to make prediction
- When branch completes
  - Update corresponding Predictor



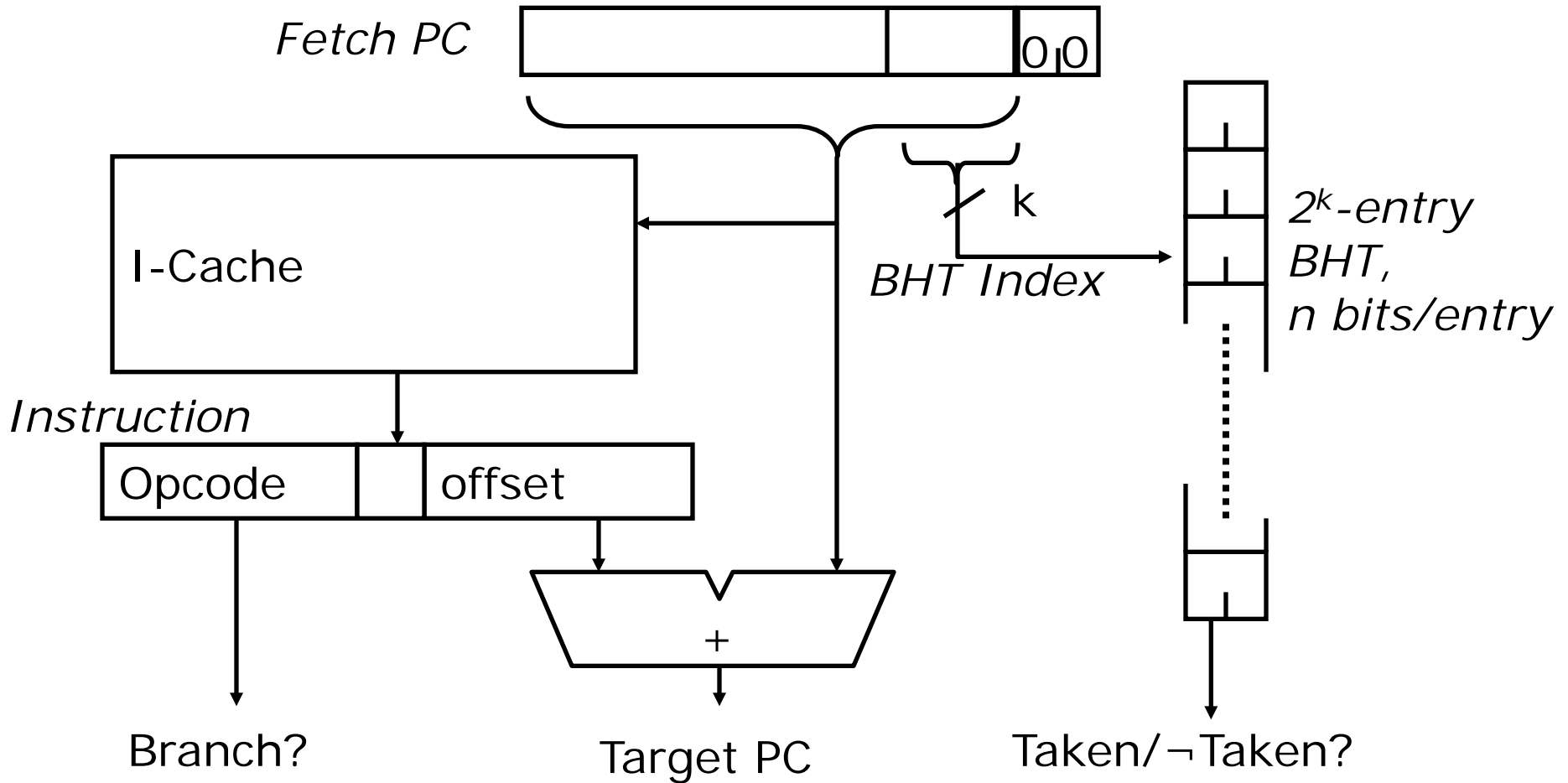
# 2-bit predictor

- Solution: 2-bit scheme where change prediction only if get misprediction *twice*:



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

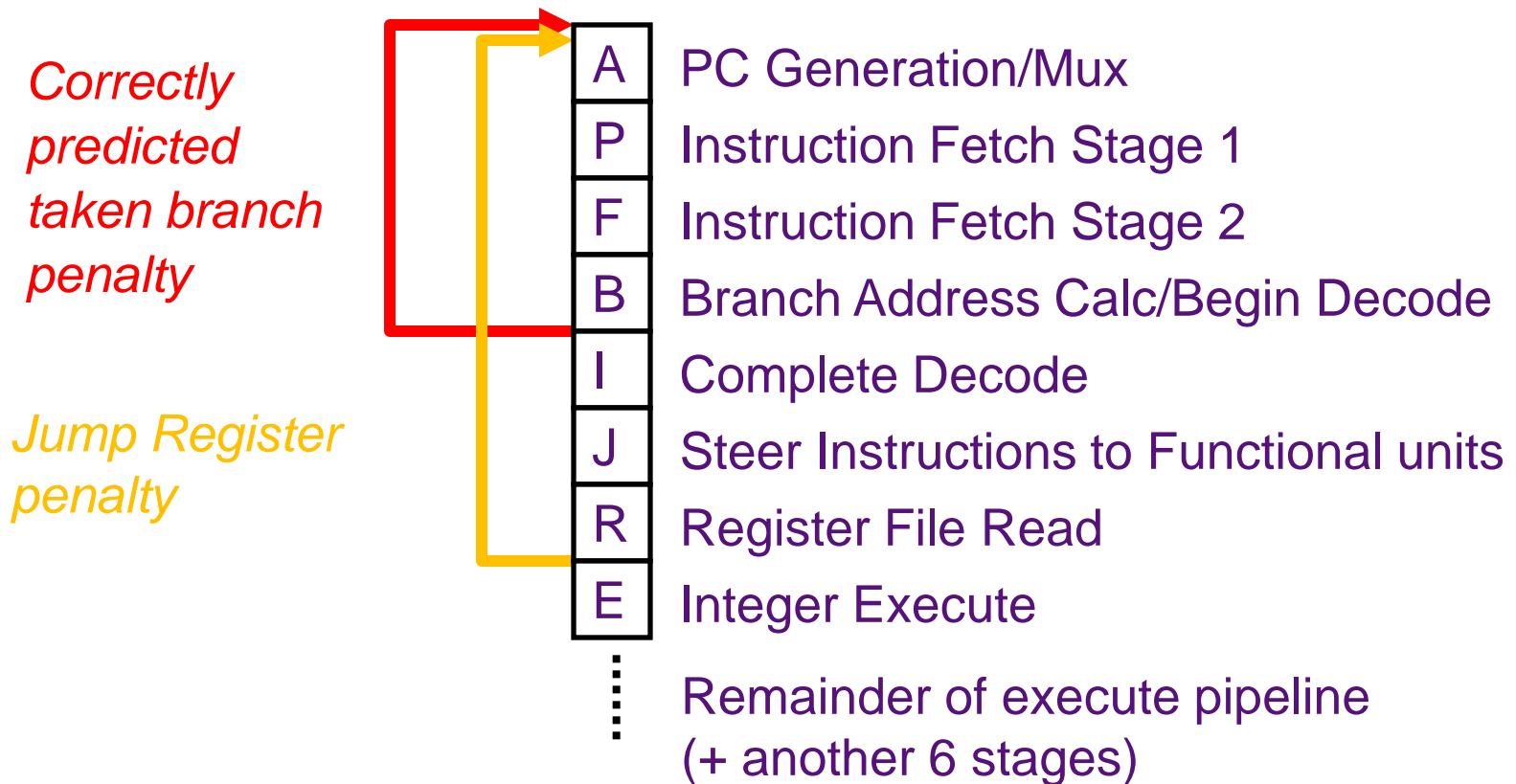
# Typical Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

# Pipeline considerations for BHT

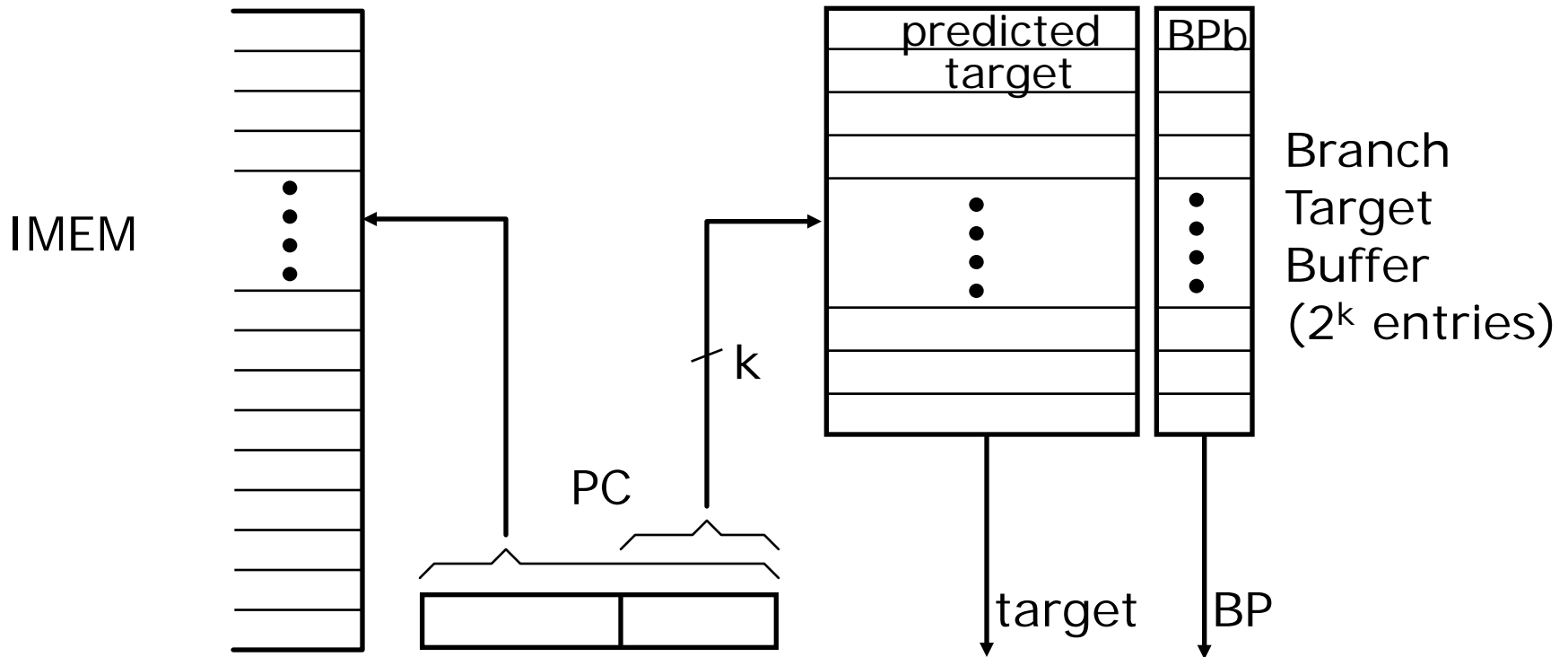
Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



*UltraSPARC-III fetch pipeline*



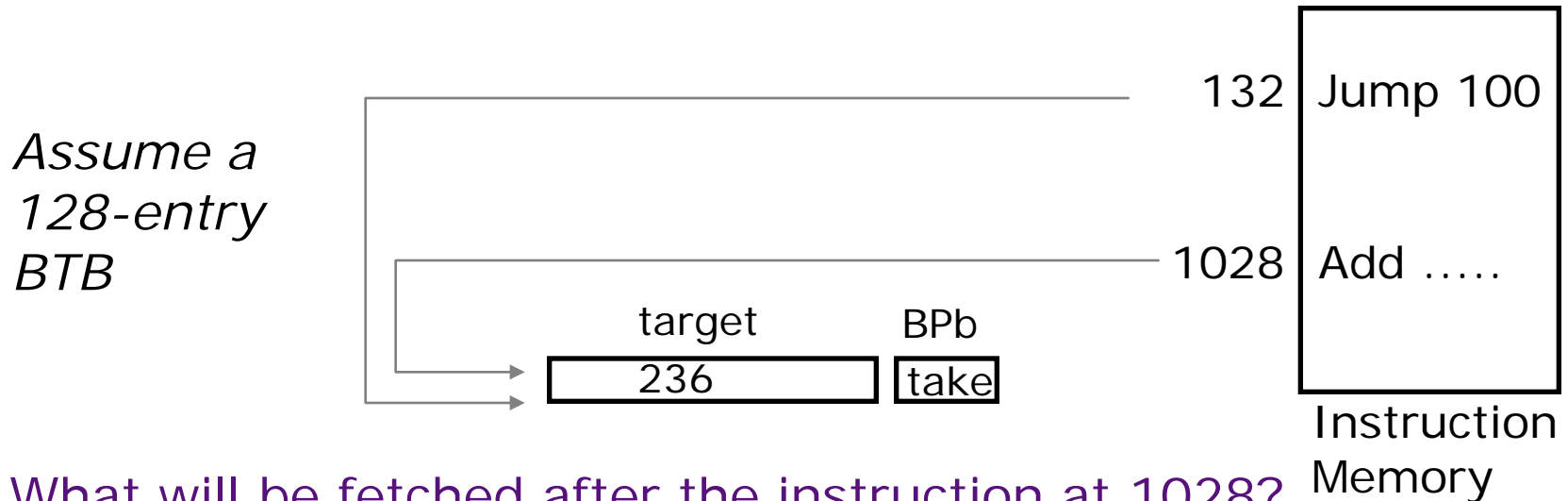
# Branch Target Buffer



BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*  
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

# Address Collisions in BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

*Is this a common occurrence?*

*Can we avoid these bubbles?*

## BTB is only for Control Instructions

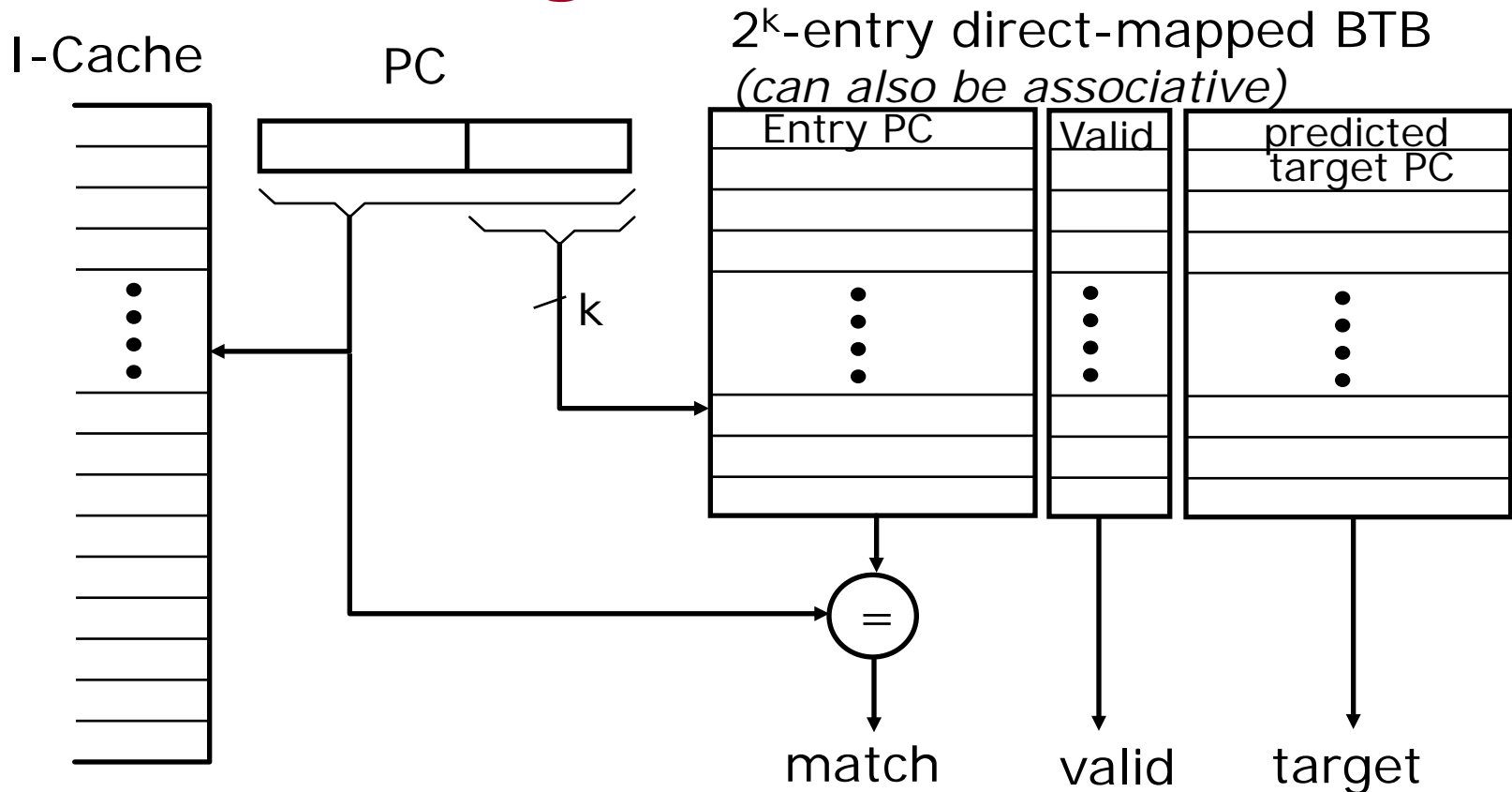
BTB contains useful information for branch and jump instructions only

⇒ Do not update it for other instructions

For all other instructions the next PC is  $PC+4$  !

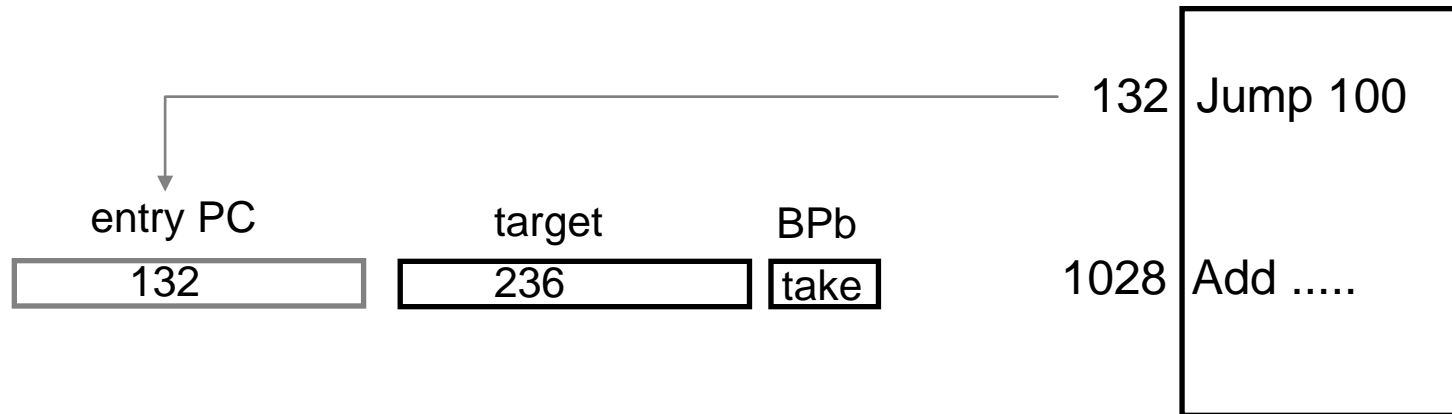
*How to achieve this effect without decoding the instruction?*

# Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC + 4 is fetched if match fails
- Only *predicted taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

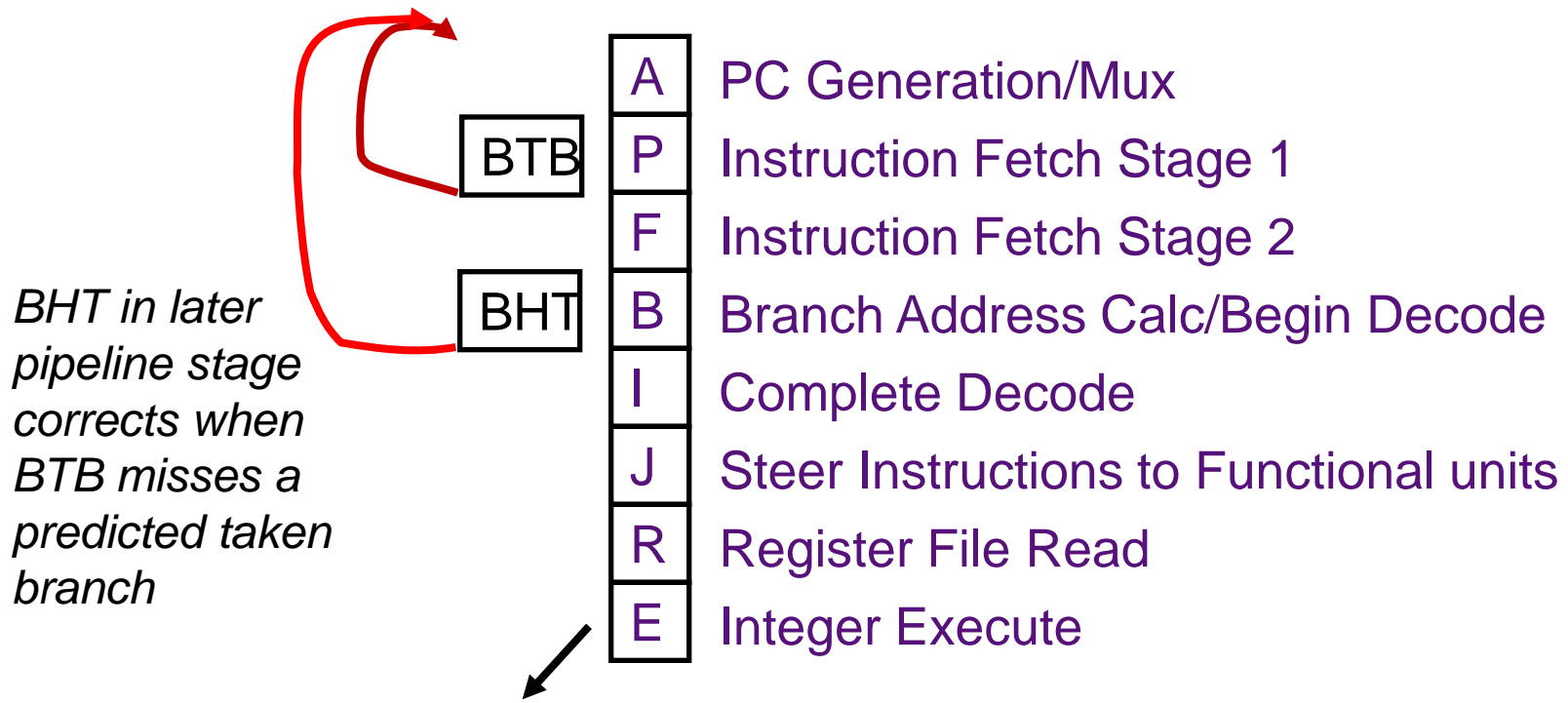
# Consulting BTB Before Decoding



- The match for PC=1028 fails and 1028+4 is fetched
  - eliminates false predictions after ALU instructions*
- BTB contains entries only for control transfer instructions
  - more room to store branch targets*

# Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



*BTB/BHT only updated after branch resolves in E stage*

# Uses of Jump Register (JR)

- **Switch statements (jump to address of matching case)**

BTB works well if same case used repeatedly

- **Dynamic function call (jump to run-time function address)**

BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- **Subroutine returns (jump to return address)**

BTB works well if usually return to the same place

⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

# Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); nexta: }  
fb() { fc(); nextb: }  
fc() { fd(); nextc: }
```

*Push return address when  
function call executed*

*Pop return address when  
subroutine return  
decoded*



*k entries  
(typically k=8-16)*



# Mispredict Recovery

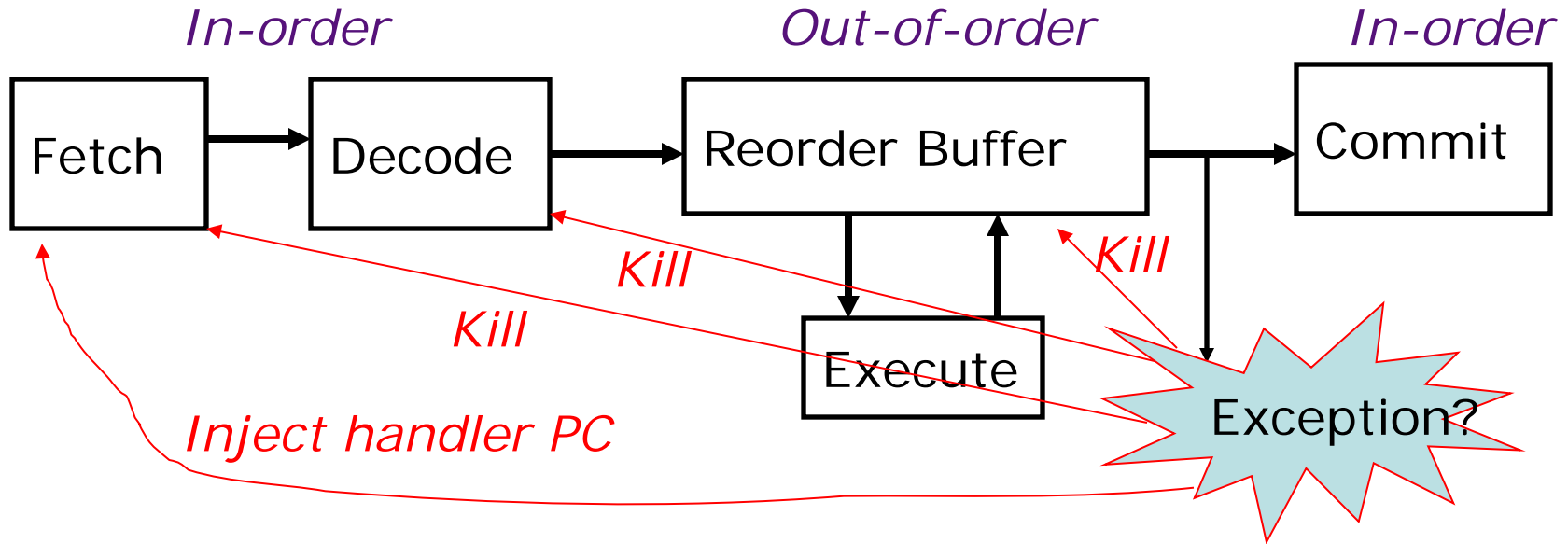
## In-order execution machines:

- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

## Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves

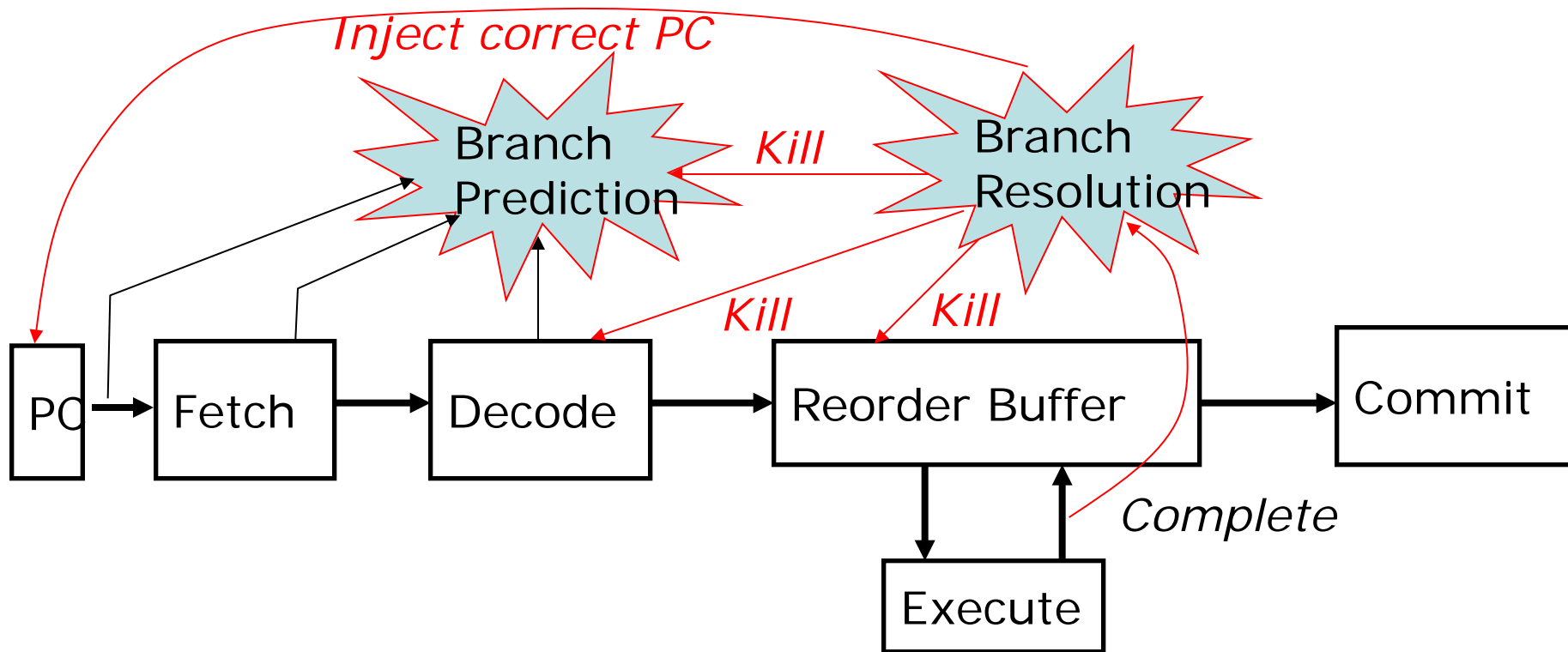
# In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (  $\Rightarrow$  out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order)

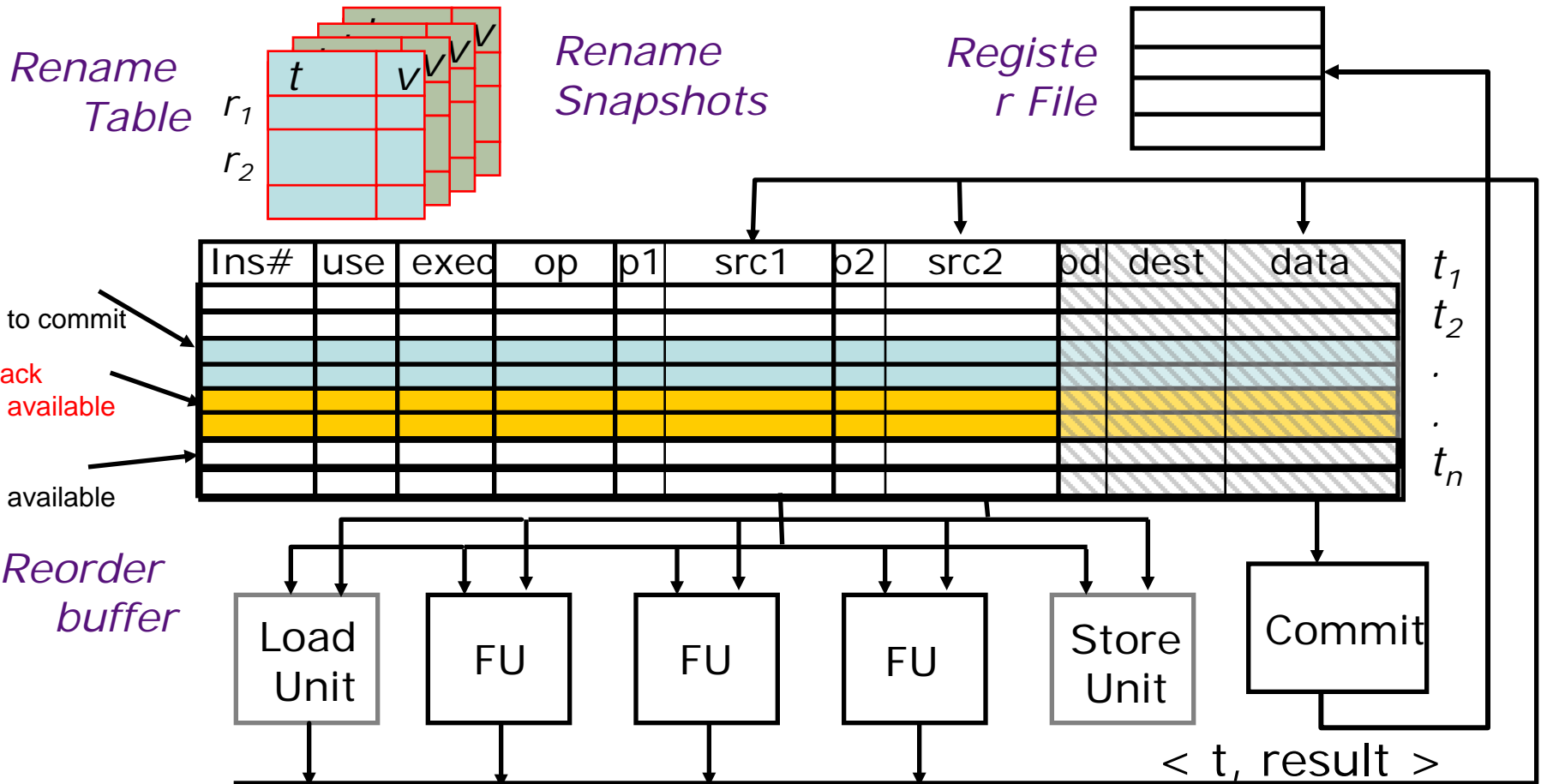
*Temporary storage needed in ROB to hold results before commit*

# Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

# Recovering ROB/Renaming Table



Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

# Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

*With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction*

# Correlating Branches

- **Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch**
- **Two possibilities; Current branch depends on:**
  - Last m most recently executed branches anywhere in program  
Produces a “GA” (for “global adaptive”) in the Yeh and Patt classification (e.g. GAg)
  - Last m most recent outcomes of same branch.  
Produces a “PA” (for “per-address adaptive”) in same classification (e.g. PAg)
- **Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table entry**
  - A single history table shared by all branches (appends a “g” at end), indexed by history value.
  - Address is used along with history to select table entry (appends a “p” at end of classification)
  - If only portion of address used, often appends an “s” to indicate “set-indexed” tables (i.e. GAs)

# Exploiting Spatial Correlation

*Yeh and Patt, 1992*

```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

If first condition false, second condition also false

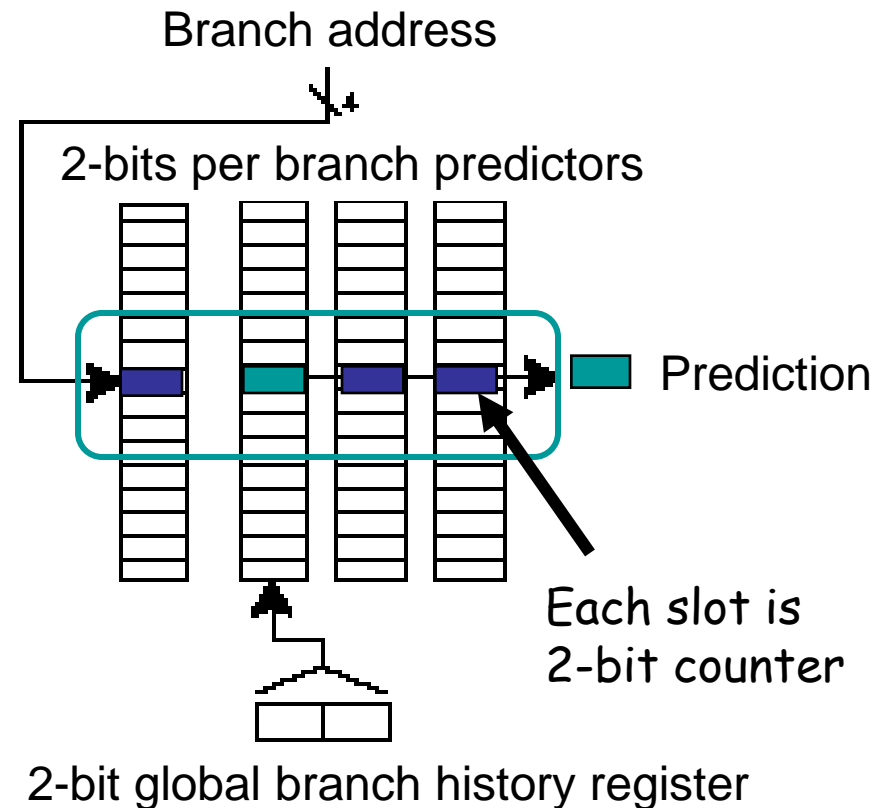
*History register*, H, records the direction of the last N branches executed by the processor

# Correlating Branches

- For instance, consider global history, set-indexed BHT. That gives us a GAs history table.

## (2,2) GAs predictor

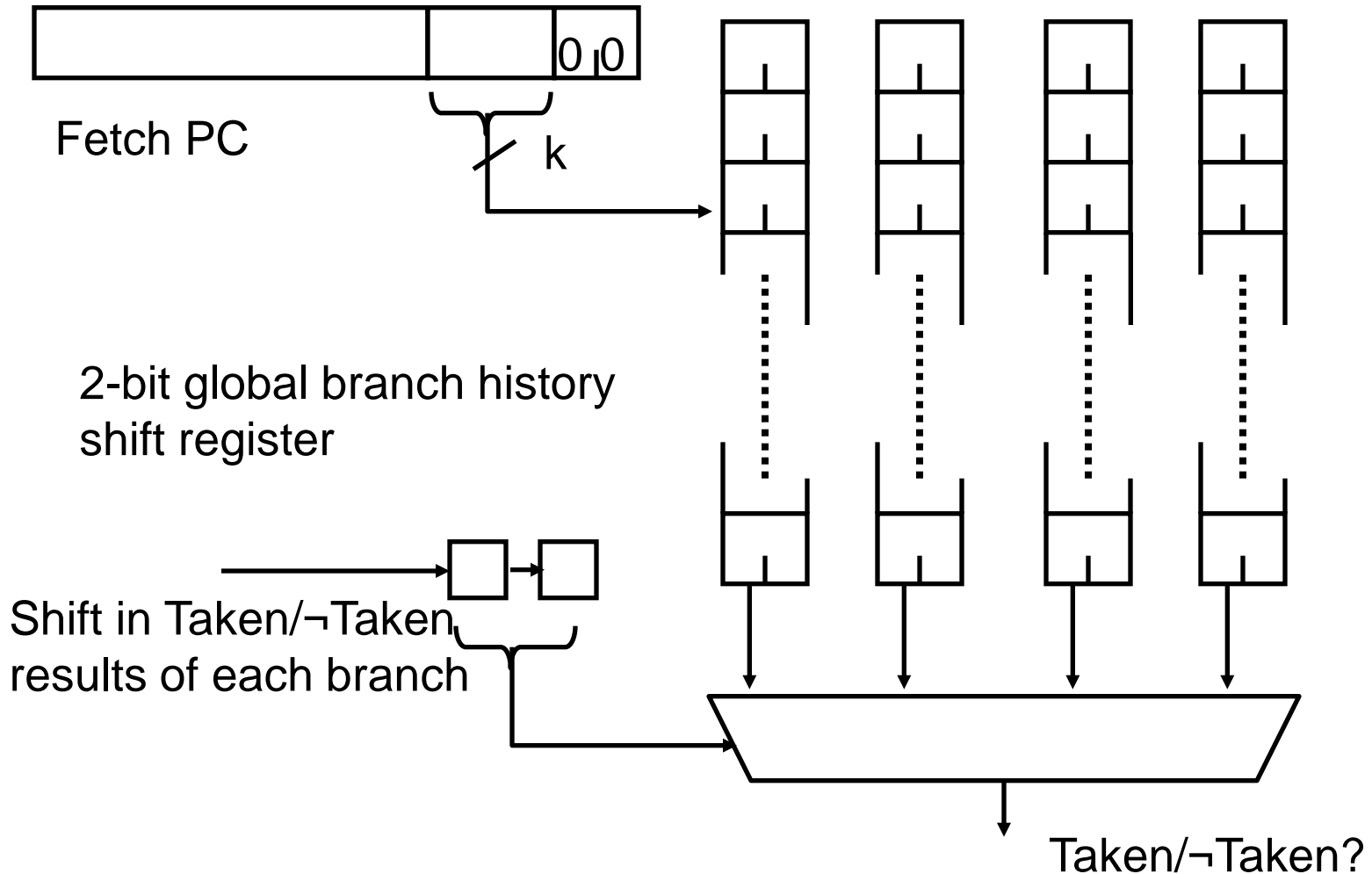
- First 2 means that we keep two bits of history
- Second means that we have 2 bit counters in each slot.
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction
- Note that the original two-bit counter solution would be a (0,2) GAs predictor
- Note also that aliasing is possible here...





# Two-Level Branch Predictor (e.g. GAs)

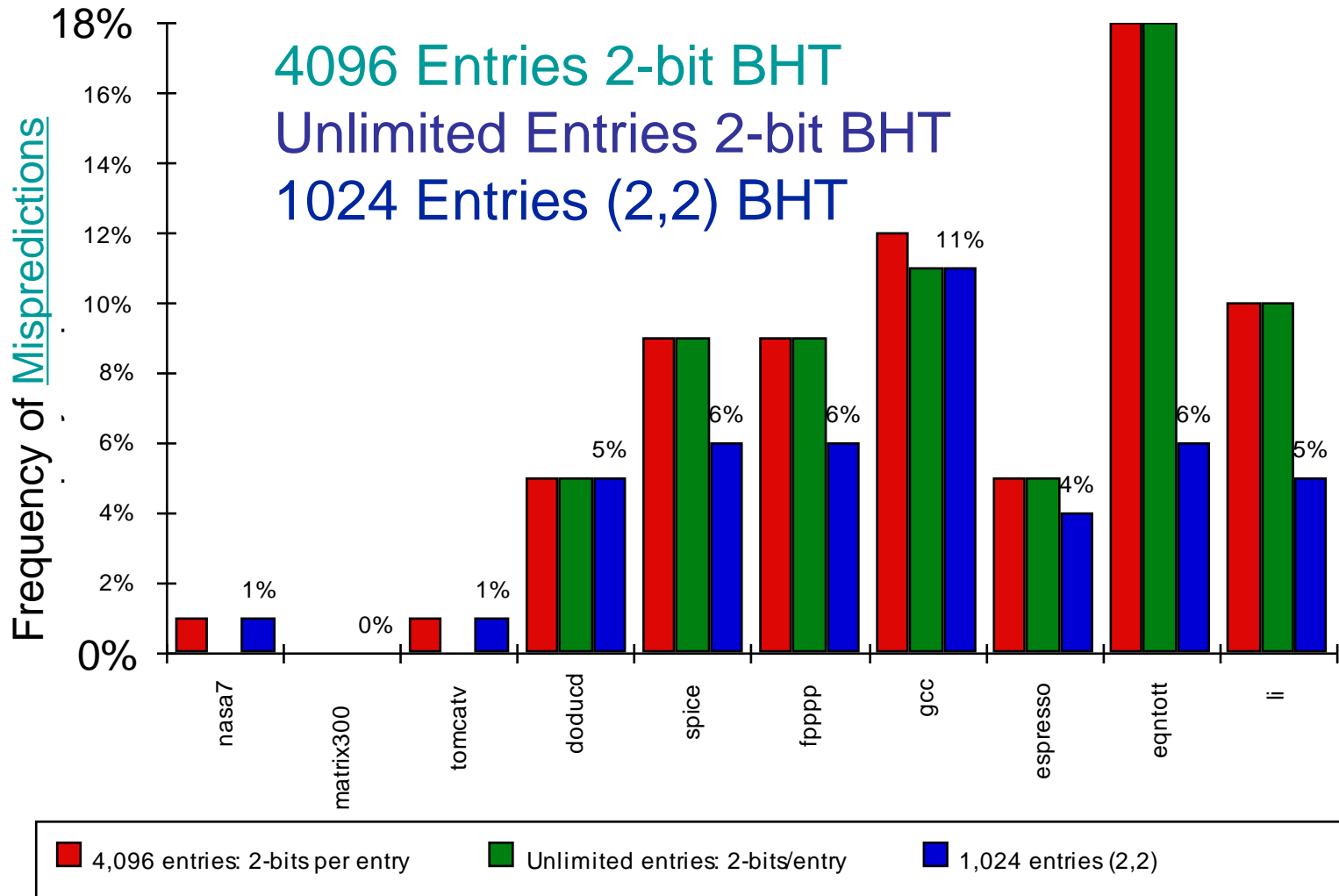
*Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*



# What are Important Metrics?

- **Clearly, Hit Rate matters**
  - Even 1% can be important when above 90% hit rate
- **Speed: Does this affect cycle time?**
- **Space: Clearly Total Space matters!**
  - Papers which do not try to normalize across different options are playing fast and lose with data
  - Try to get best performance for the cost

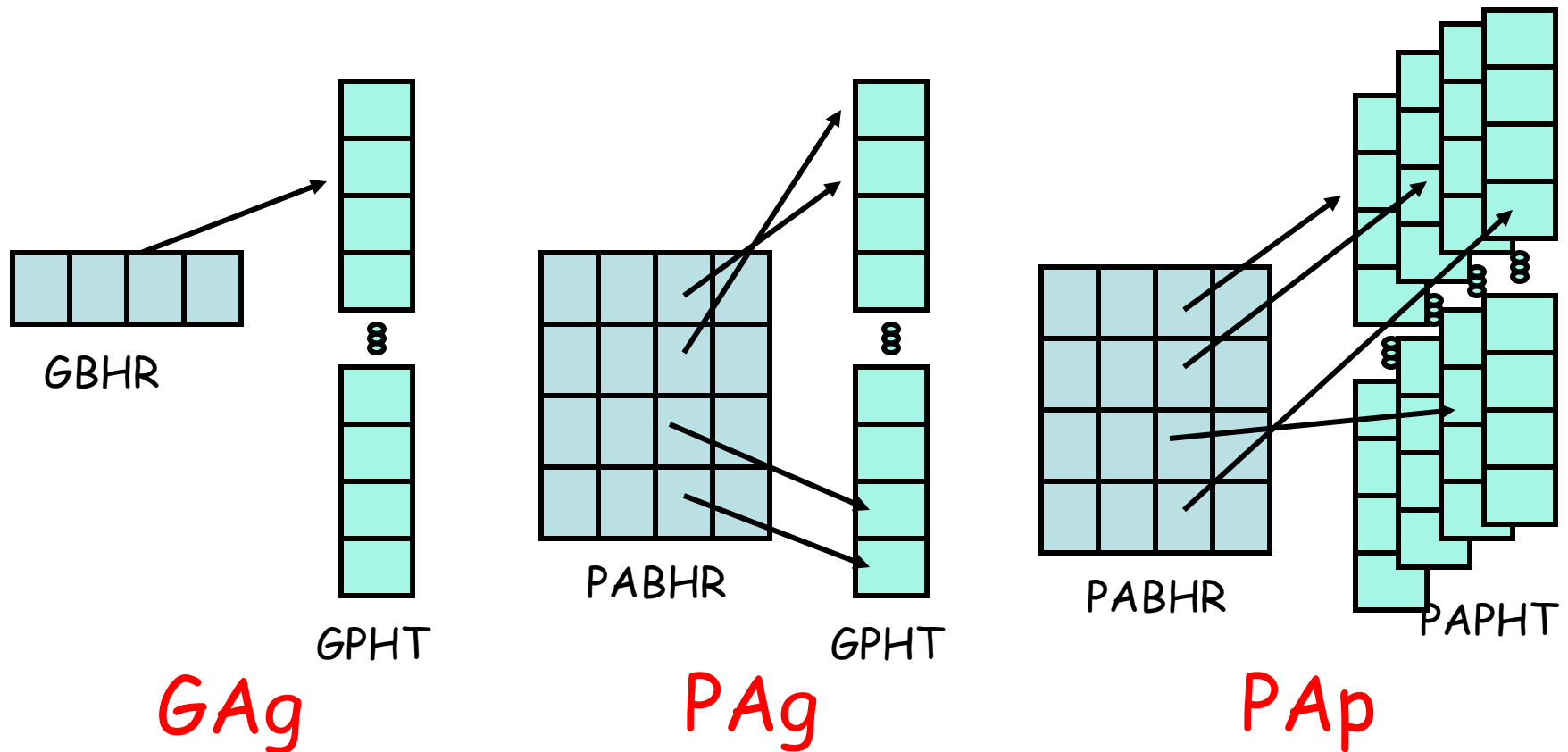
# Accuracy of Different Schemes



# BHT Accuracy

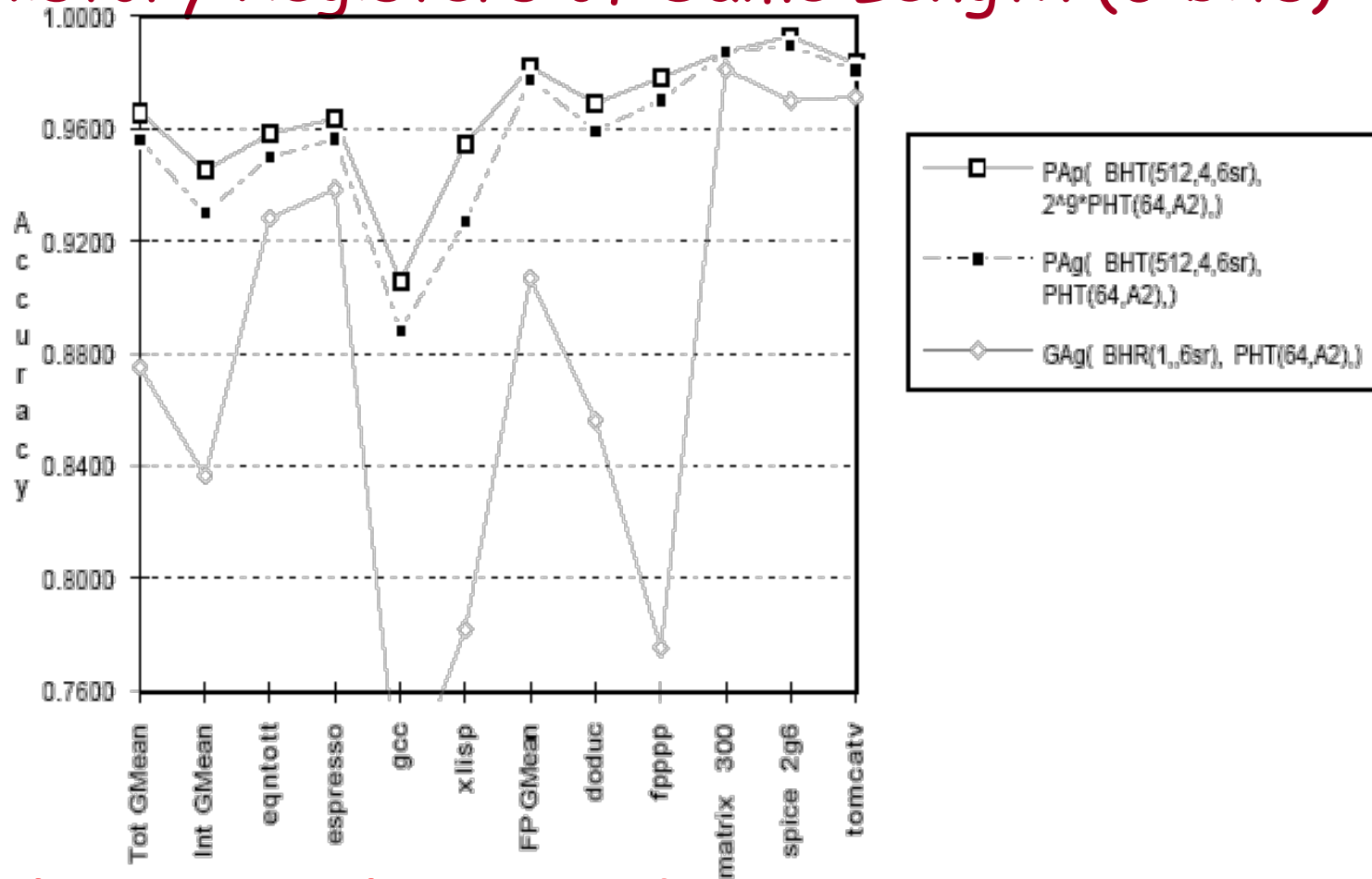
- **Mispredict because either:**
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- **4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%**
  - For SPEC92, 4096 about as good as infinite table
- **How could HW predict “this loop will execute 3 times” using a simple mechanism?**
  - Need to track history of just that branch
  - For given pattern, track most likely following branch direction
- **Leads to two separate types of recent history tracking:**
  - GBHR (Global Branch History Register)
  - PABHR (Per Address Branch History Table)
- **Two separate types of Pattern tracking**
  - GPHT (Global Pattern History Table)
  - PAPHT (Per Address Pattern History Table)

# Yeh and Patt classification



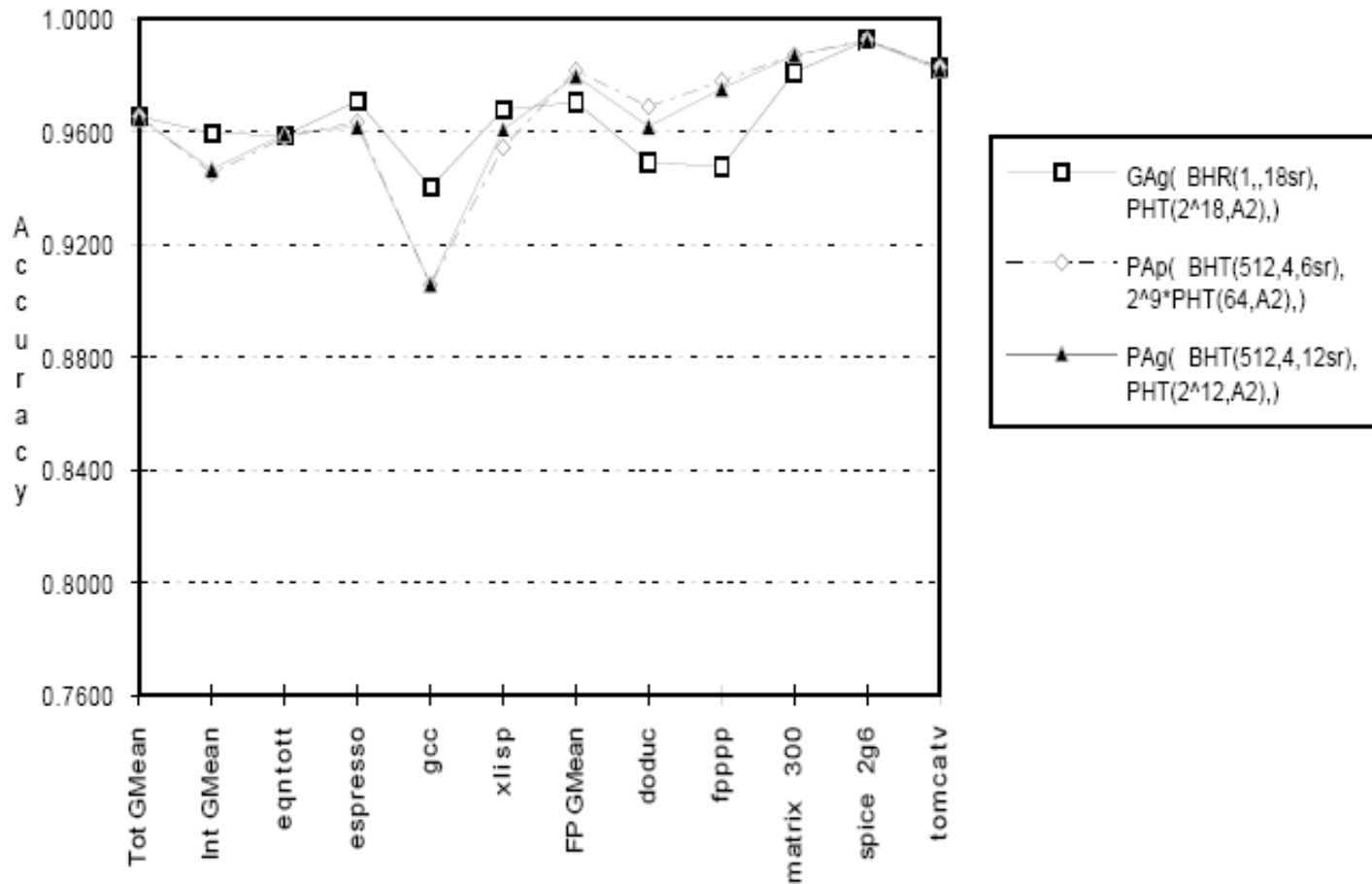
- **GAg**: Global History Register, Global History Table
- **PAg**: Per-Address History Register, Global History Table
- **PAp**: Per-Address History Register, Per-Address History Table

# Two-Level Adaptive Schemes: History Registers of Same Length (6 bits)



- PAp best: But uses a lot more state!
- GAg not effective with 6-bit history registers
  - Every branch updates the same history register  $\Rightarrow$  interference
- PAg performs better because it has a branch history table

# Versions with Roughly same accuracy (97%)



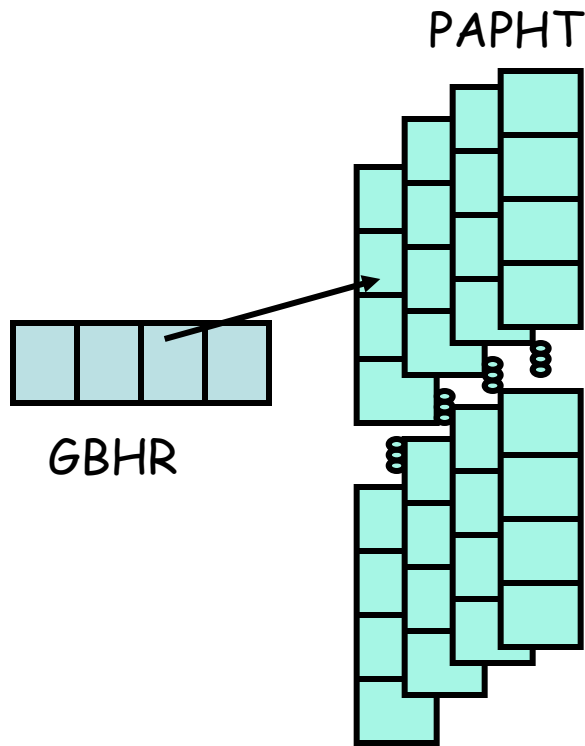
- **Cost:**
  - GAg requires 18-bit history register
  - PAg requires 12-bit history register
  - PAp requires 6-bit history register
- PAg is the cheapest among these

# Why doesn't GAg do better?

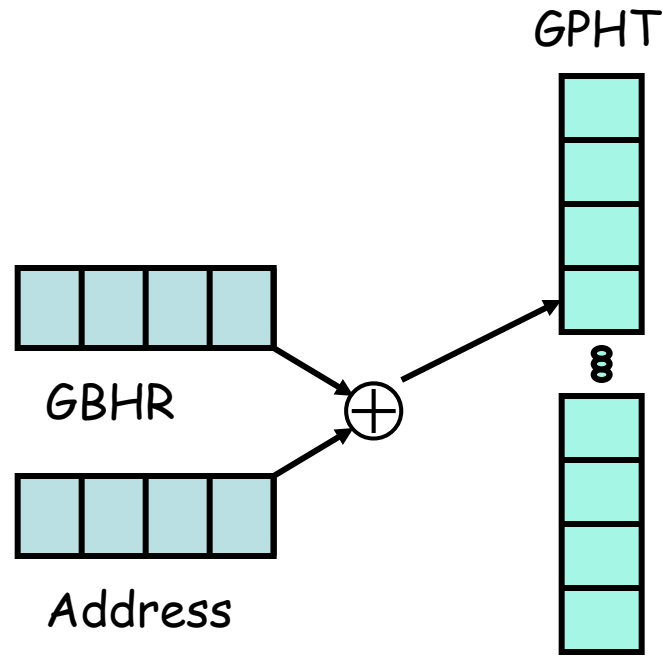
- **Difference between GAg and both PA variants:**
  - GAg tracks correlations between different branches
  - PAg/PAP track correlations between different instances of the same branch
- **These are two different types of pattern tracking**
  - Among other things, GAg good for branches in straight-line code, while PA variants good for loops
- **Problem with GAg? It aliases results from different branches into same table**
  - Issue is that different branches may take same global pattern and resolve it differently
  - GAg doesn't leave flexibility to do this



# Other Global Variants: Try to Avoid Aliasing



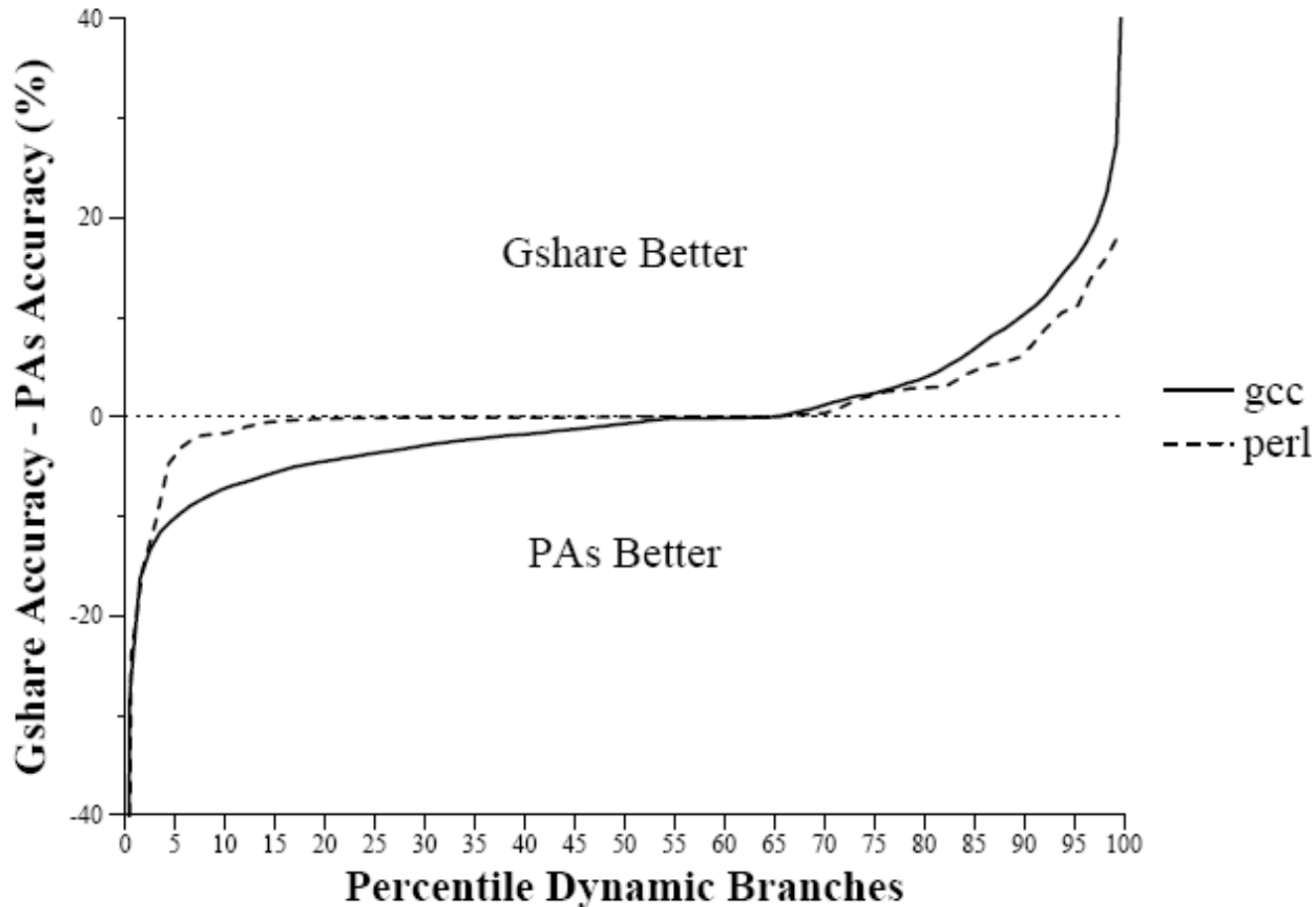
*GAs*



*GShare*

- **GAs: Global History Register, Per-Address (Set Associative) History Table**
- **Gshare: Global History Register, Global History Table with Simple attempt at anti-aliasing**

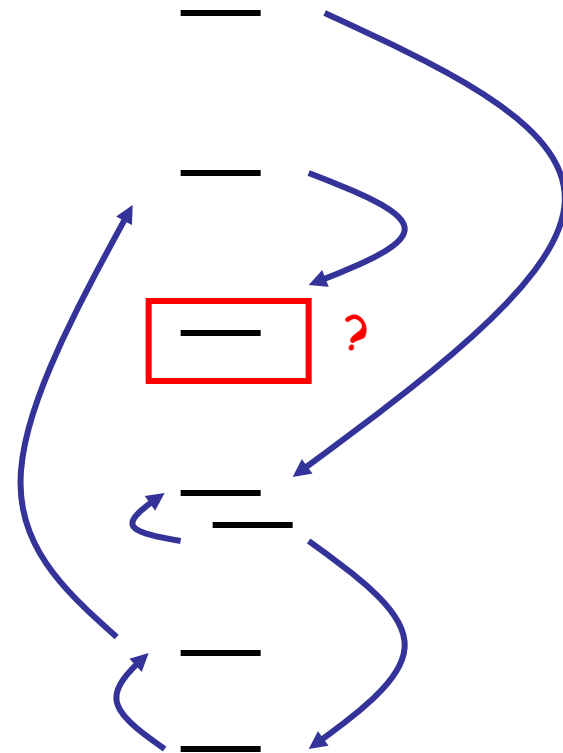
# Is Global or Local better?



- **Neither: Some branches local, some global**
  - From: “An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work,” Evers, Patel, Chappell, Patt
  - Difference in predictability quite significant for some branches!

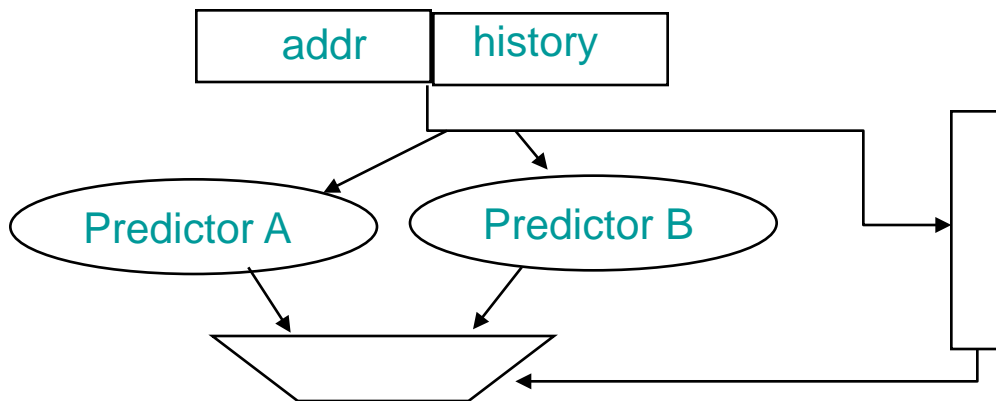
# Dynamically finding structure in Spaghetti

- Consider complex “spaghetti code”
- Are all branches likely to need the same type of branch prediction?
  - No.
- What to do about it?
  - How about predicting which predictor will be best?
  - Called a “Tournament predictor”



# Tournament Predictors

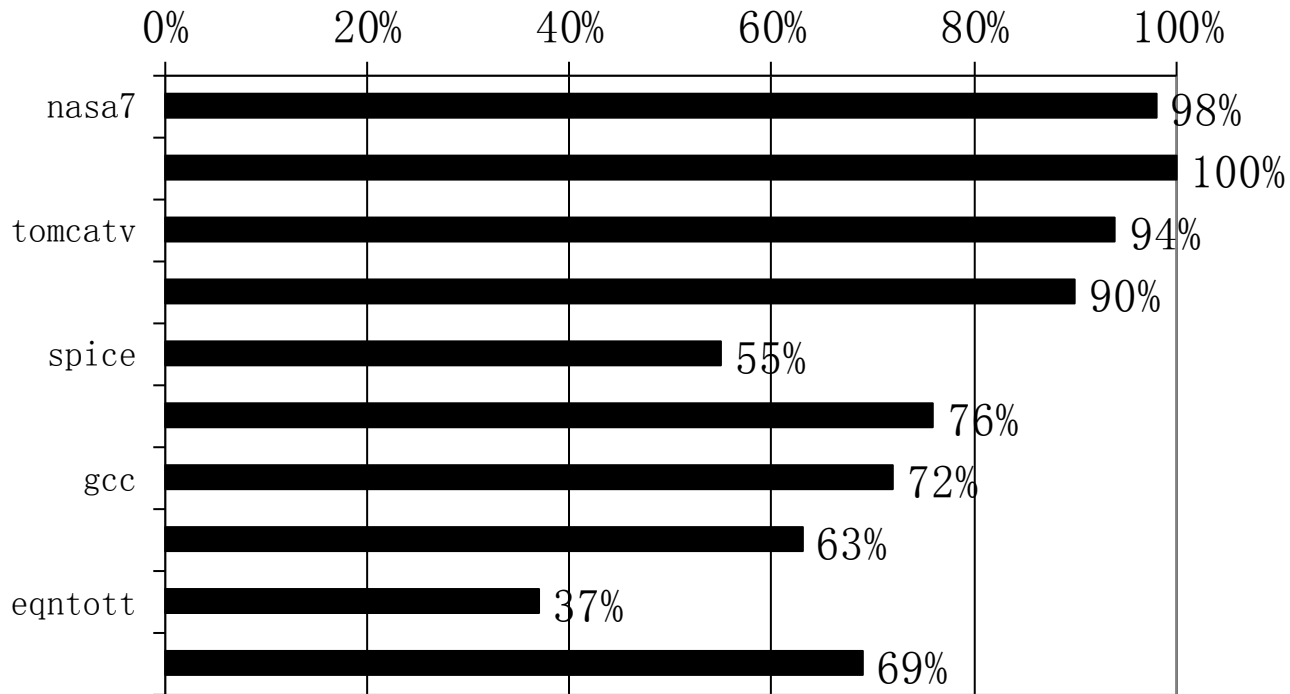
- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Use the predictor that tends to guess correctly



# Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- **Global predictor** also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken;  
ith bit 1 => ith prior branch taken;
- **Local predictor** consists of a 2-level predictor:
  - **Top level** a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - **Next level** Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size:  $4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K \text{ bits!}$   
(~180,000 transistors)

# % of predictions from local predictor in Tournament Scheme



# Accuracy of Branch Prediction

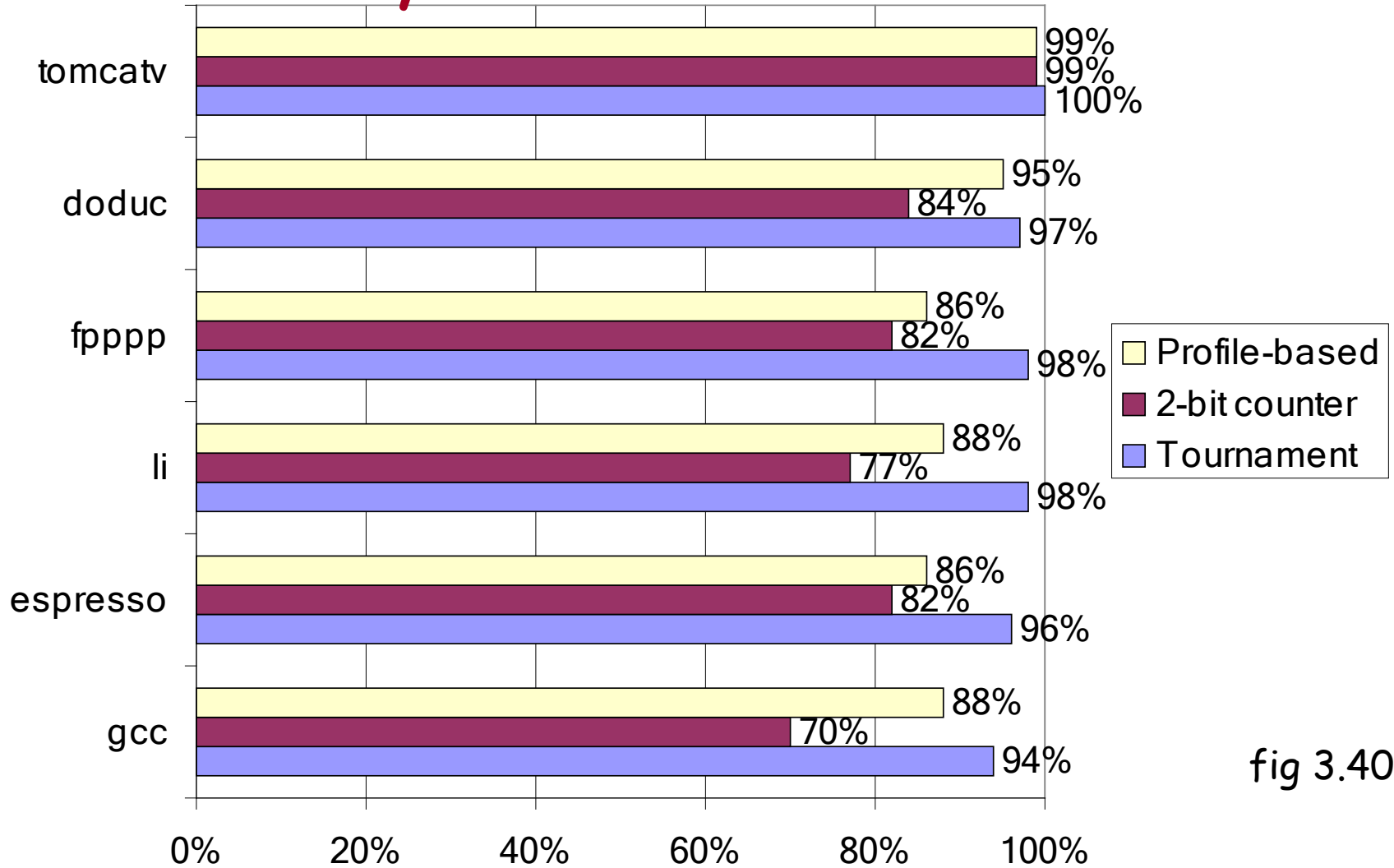
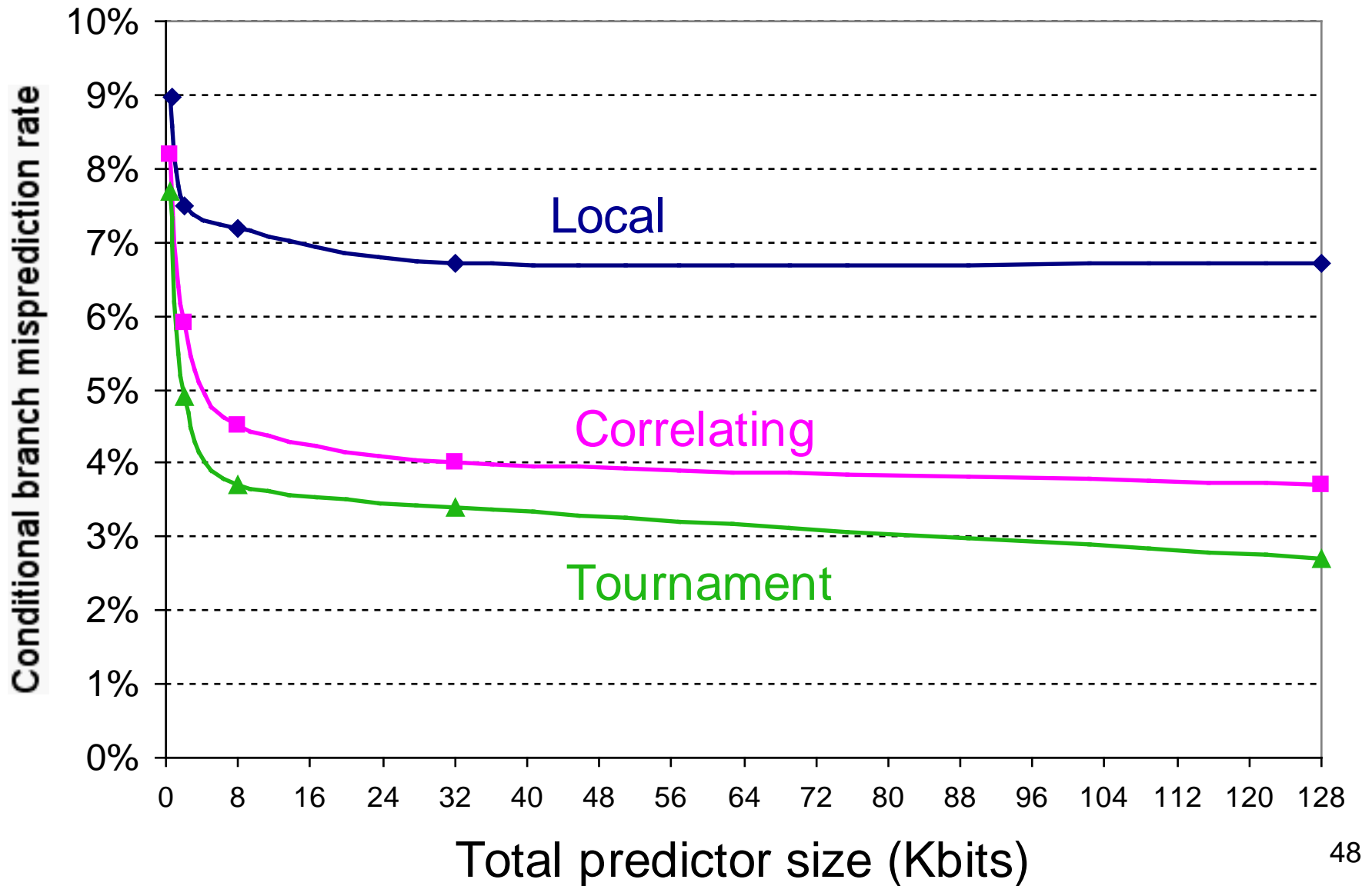


fig 3.40

- **Profile: branch prediction accuracy**  
(static in that it is encoded in instruction, but profile)

# Accuracy v. Size (SPEC89)





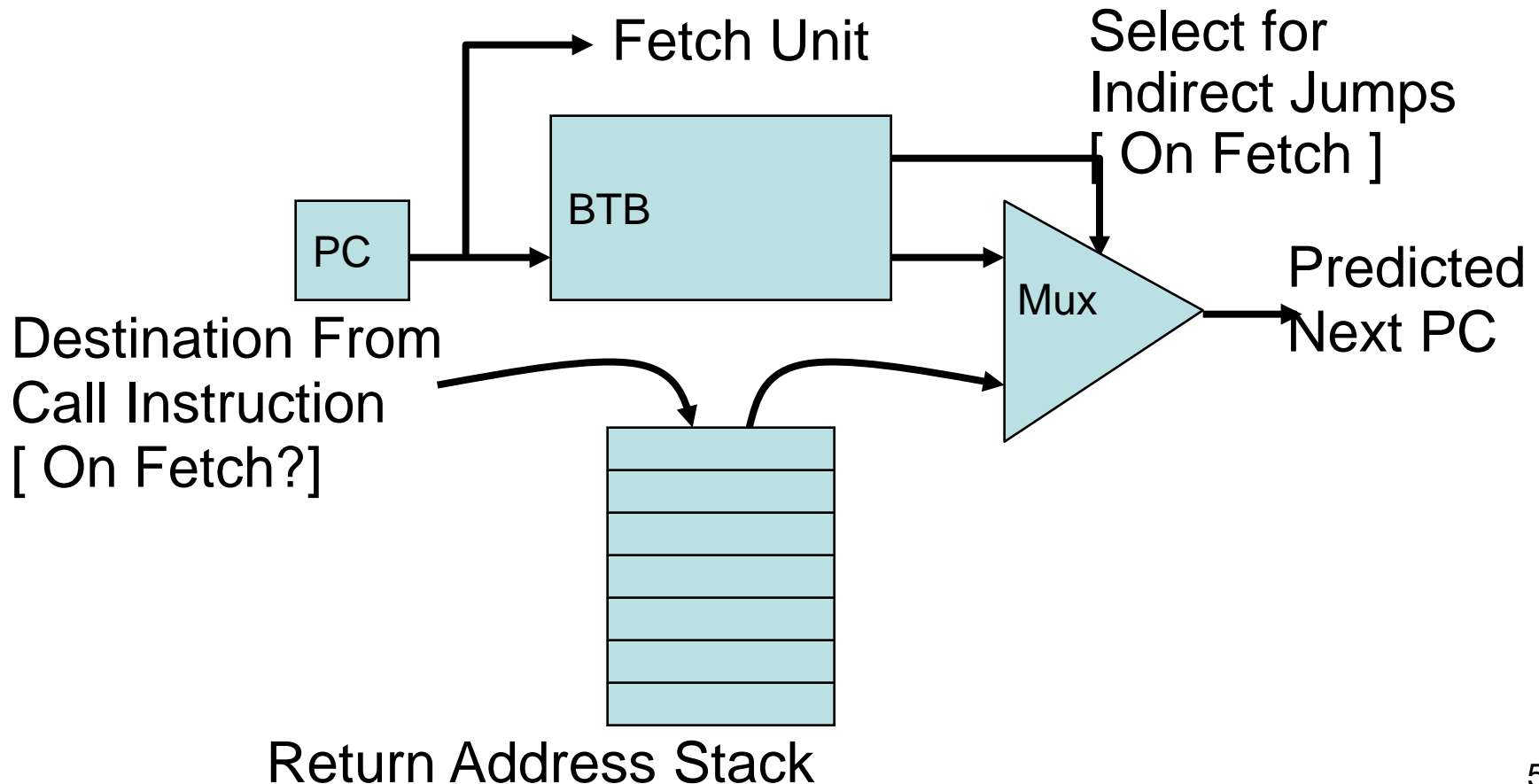
# Pitfall:

## Sometimes bigger and dumber is better

- **21264** uses tournament predictor (29 Kbits)
- Earlier **21164** uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- SPEC95 benchmarks, **21264** outperforms
  - **21264** avg. 11.5 mispredictions per 1000 instructions
  - **21164** avg. 16.5 mispredictions per 1000 instructions
- Reversed for transaction processing (TP) !
  - **21264** avg. 17 mispredictions per 1000 instructions
  - **21164** avg. 15 mispredictions per 1000 instructions
- TP code much larger & **21164** hold 2X branch predictions based on local behavior (2K vs. 1K local predictor in the **21264**)

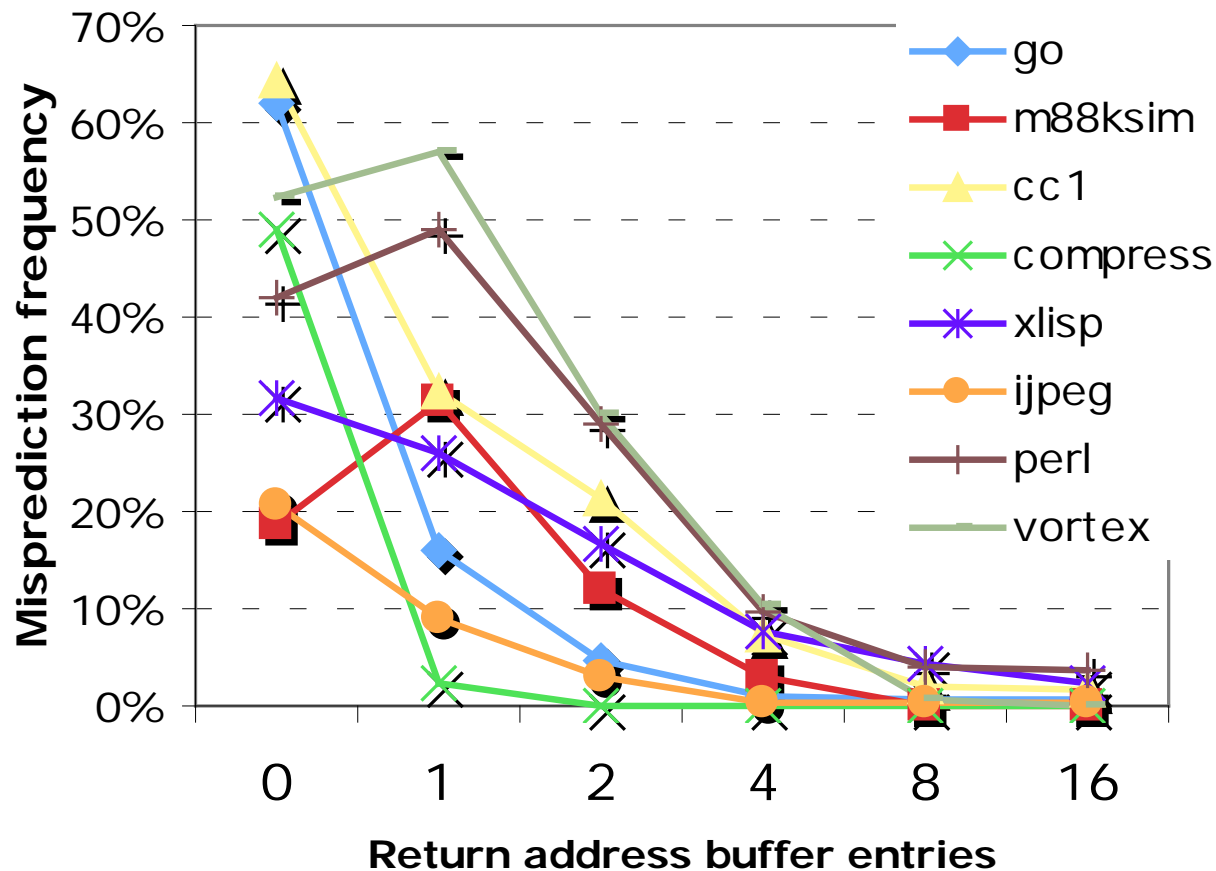
# Special Case Return Addresses

- Register Indirect branch hard to predict address
  - SPEC89 85% such branches for procedure return
  - Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate



# Performance: Return Address Predictor

- Cache most recent return addresses:
  - Call  $\Rightarrow$  Push a return address on stack
  - Return  $\Rightarrow$  Pop an address off stack & predict as new PC



# Conclusion

- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly
- **Prediction works because....**
  - Programs have patterns
  - Just have to figure out what they are
  - Basic Assumption: Future can be predicted from past!
- **Correlation: Recently executed branches correlated with next branch.**
  - Either different branches (GA)
  - Or different executions of same branches (PA).
- **Two-Level Branch Prediction**
  - Uses complex history (either global or local) to predict next branch
  - Two tables: a history table and a pattern table
  - Global Predictors: GAg, GAs, GShare
  - Local Predictors: PAg, Pap