CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 7 – Branch Prediction and Advanced Out-of-Order Superscalars

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

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?

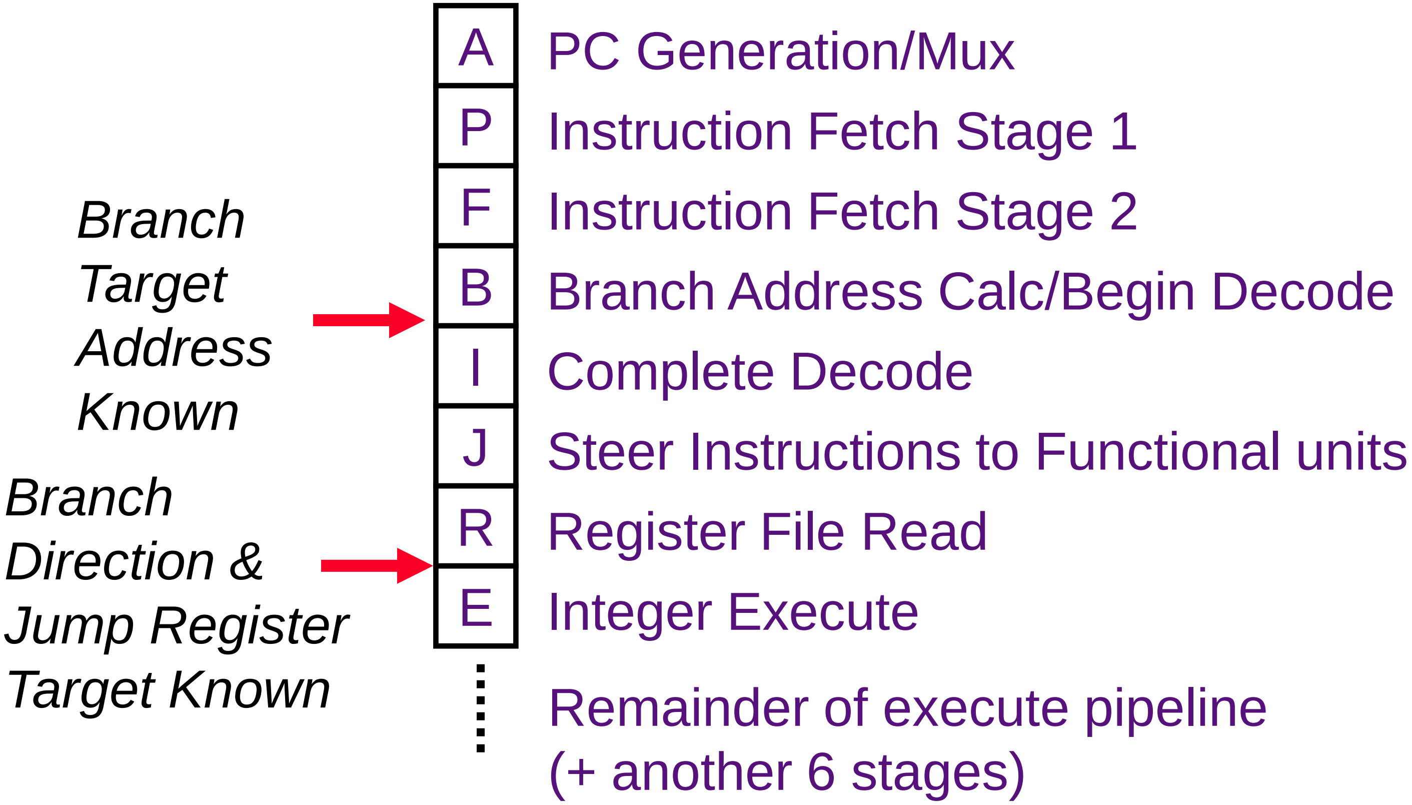
2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
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)



A PC Generation/Mux

P Instruction Fetch Stage 1

F Instruction Fetch Stage 2

B Branch Address Calc/Begin Decode

I Complete Decode

J Steer Instructions to Functional units

R Register File Read

E Integer Execute

Remainder of execute pipeline
(+ another 6 stages)

Branch Target Address Known

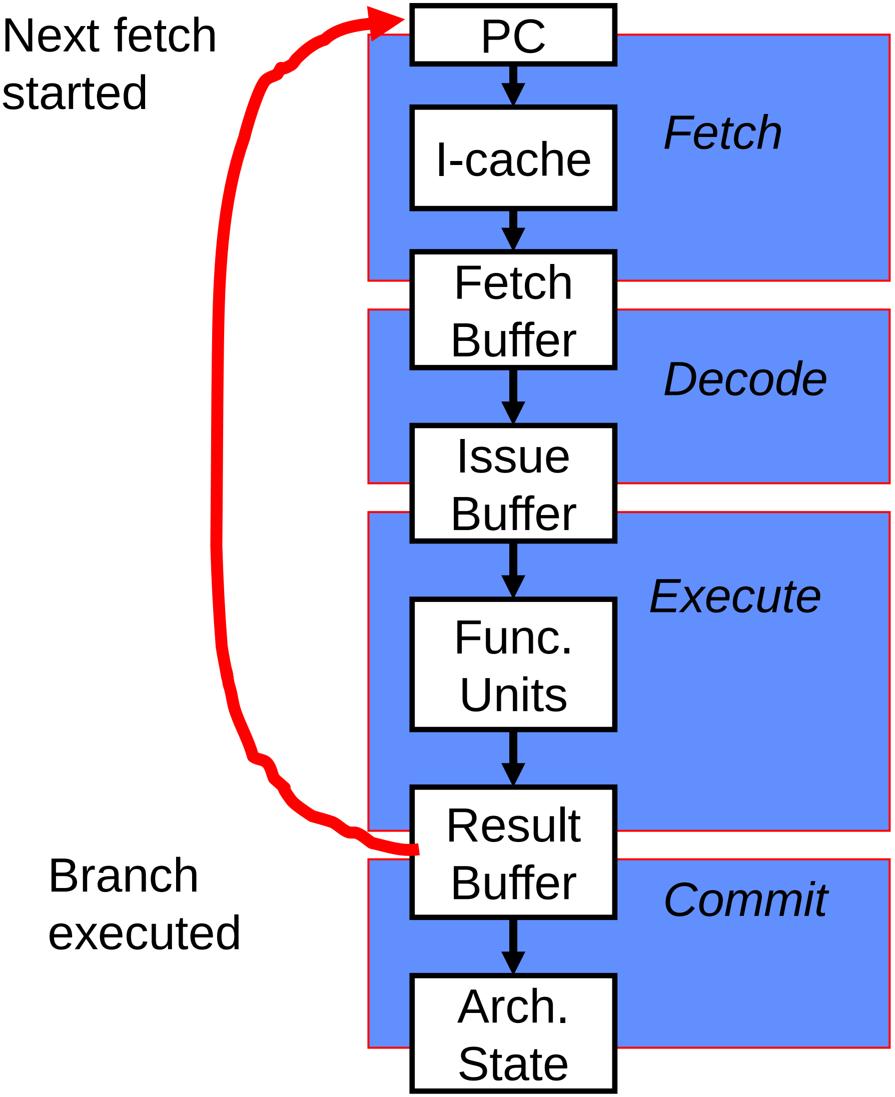
Branch Direction & Jump Register Target Known

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



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!

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 because branches often in critical path through code)

- Hardware solutions
  - Find something else to do (delay slots)
    - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
  - Speculate, i.e., branch prediction
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy, widely used

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:

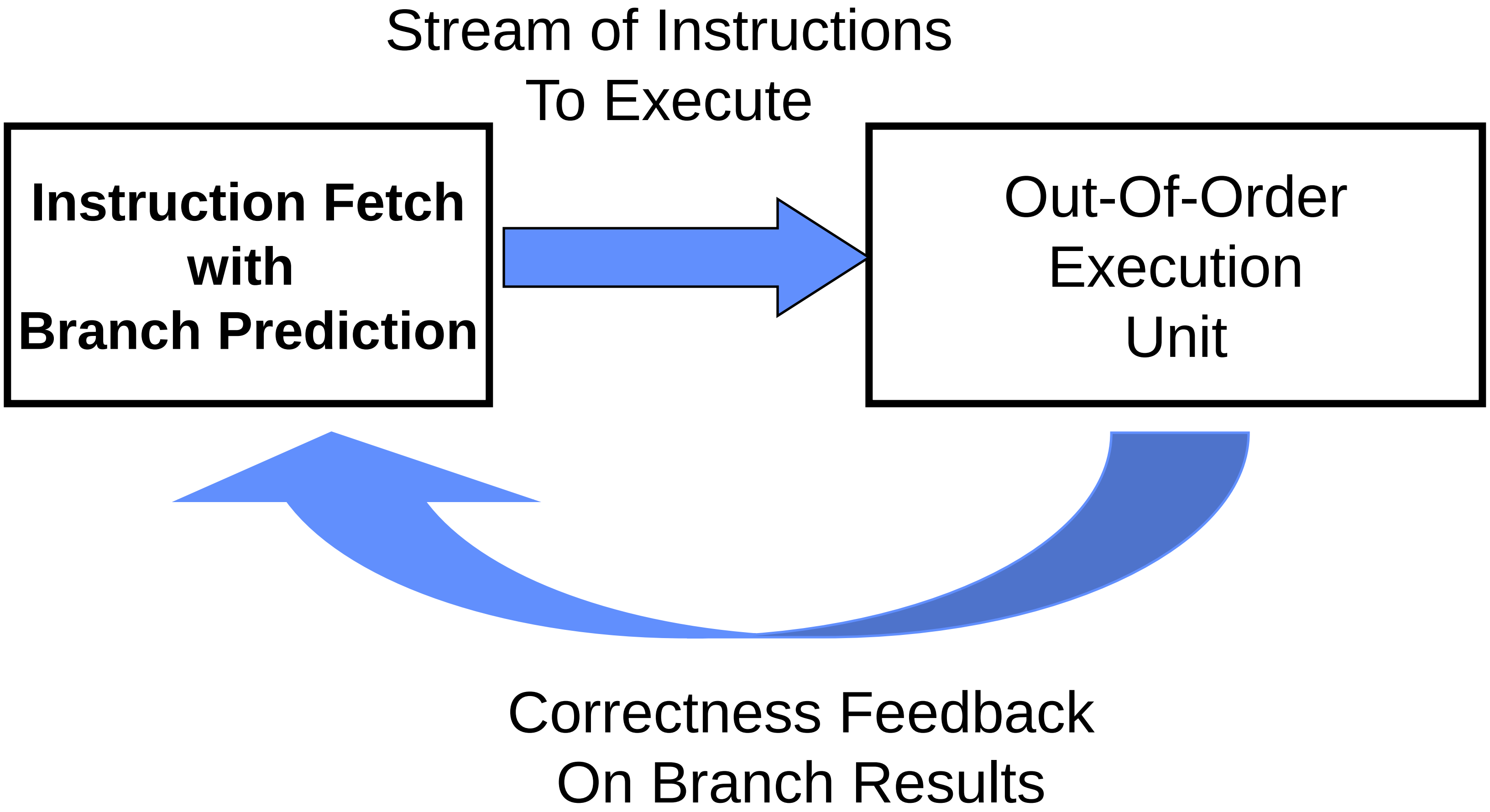
Prediction structures:

- 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 that following branch

Independent "Fetch" unit



Stream of Instructions
To Execute

Instruction Fetch with Branch Prediction

Out-Of-Order Execution Unit

Correctness Feedback
On Branch Results

- **Instruction fetch decoupled from execution**
- **Often issue logic (+ rename) included with Fetch**

Relationship between precise interrupts and speculation:

- Speculation is a form of guessing
  - Branch prediction, data prediction
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as precise exceptions!
- Branch prediction is a very important!
  - Need to "take our best shot" at predicting branch direction.
  - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    - » Consider 4 instructions per cycle
    - » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
  - This is why reorder buffers in all new processors

Case for Branch Prediction when Issue N instructions per clock cycle

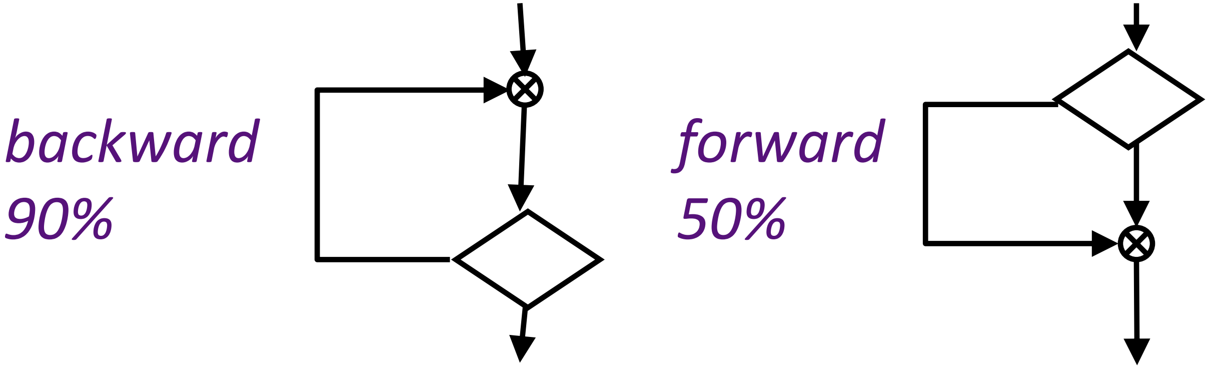
- **Branches will arrive up to *n* times faster in an *n*-issue processor**
  - **Amdahl's Law => relative impact of the control stalls will be larger with the lower potential CPI in an *n*-issue processor**
  - **conversely, need branch prediction to 'see' potential parallelism**
- **Performance = *ƒ*(accuracy, cost of misprediction)**
  - **Misprediction ⇒ 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!**

Importance of Branch Prediction

- Consider 4-way superscalar with 8 pipeline stages from fetch to dispatch, and 80-entry ROB, and 3 cycles from issue to branch resolution
- On a mispredict, could throw away 8*4+(80-1)=111 instructions
- Improving from 90% to 95% prediction accuracy, removes 50% of branch mispredicts
  - If 1/6 instructions are branches, then move from 60 instructions between mispredicts, to 120 instructions between mispredicts

Static Branch Prediction

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



backward
90%

forward
50%

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

bne0 *(preferred taken)* beq0 *(not taken)*

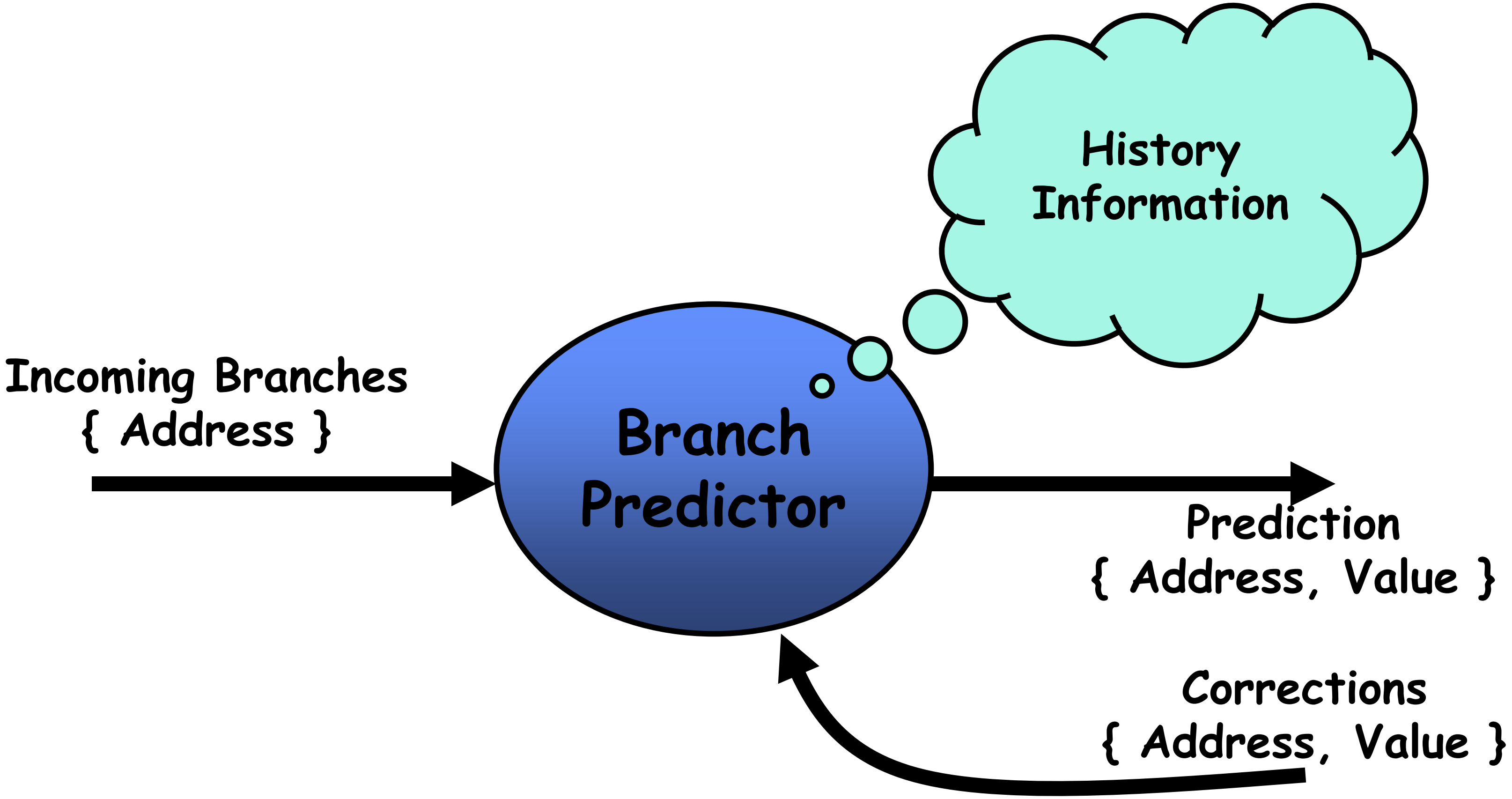
ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

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)

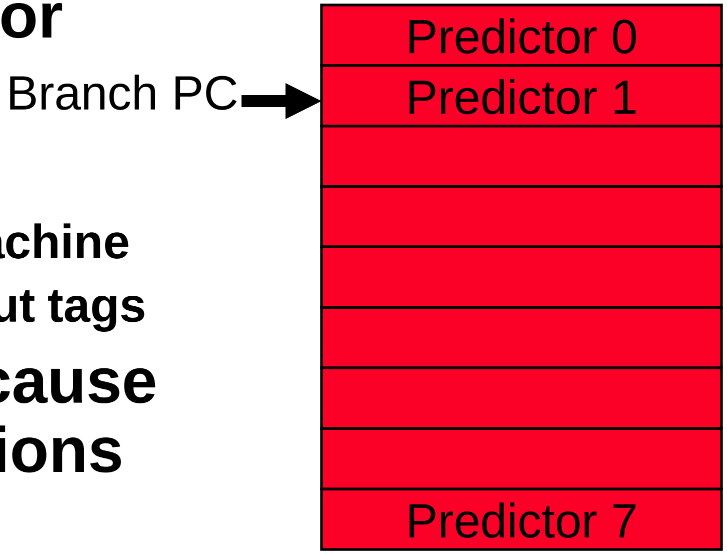
Dynamic Branch Prediction Problem



- 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**
- **In Fetch state of branch:**
  - **Use Predictor to make prediction**
- **When branch completes**
  - **Update corresponding Predictor**

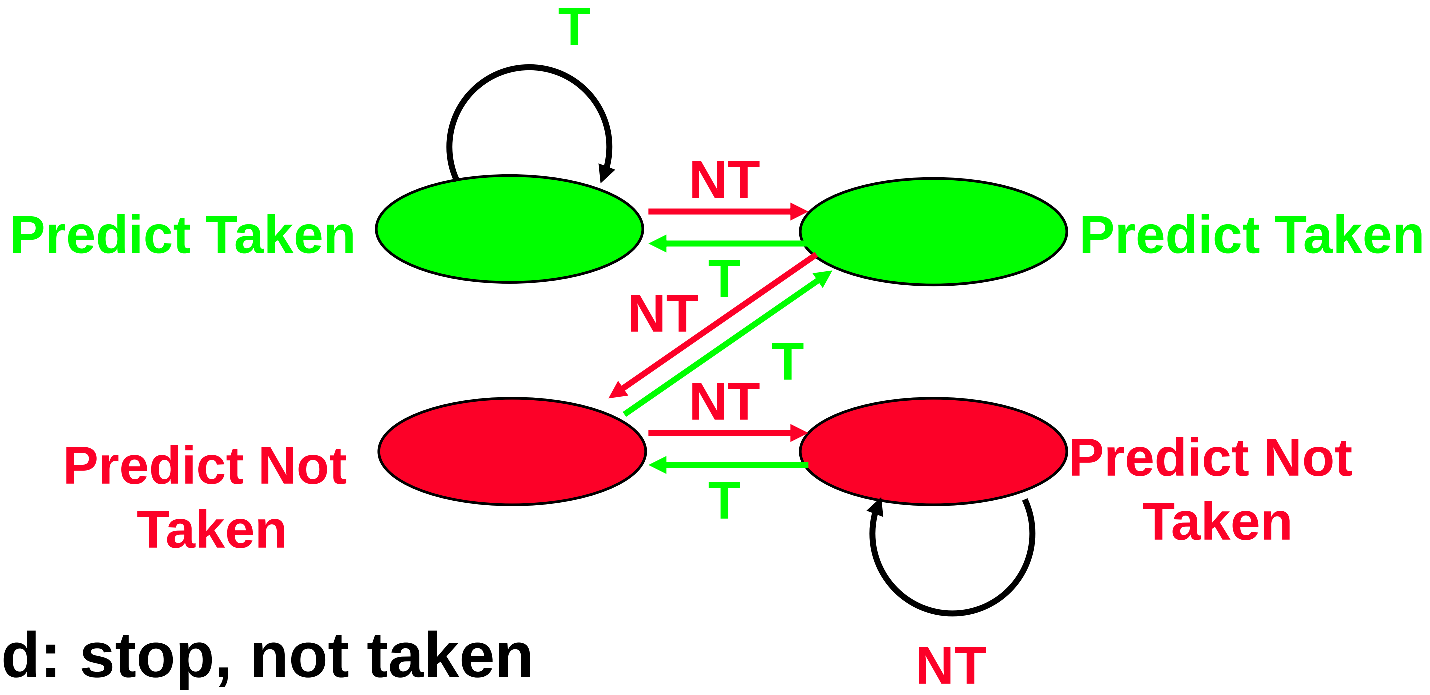


One-Bit Branch History Predictor

- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  1. first iteration predicts loop backwards branch not-taken (loop was exited last time)
  2. last iteration predicts loop backwards branch taken (loop continued last time)

2-bit predictor

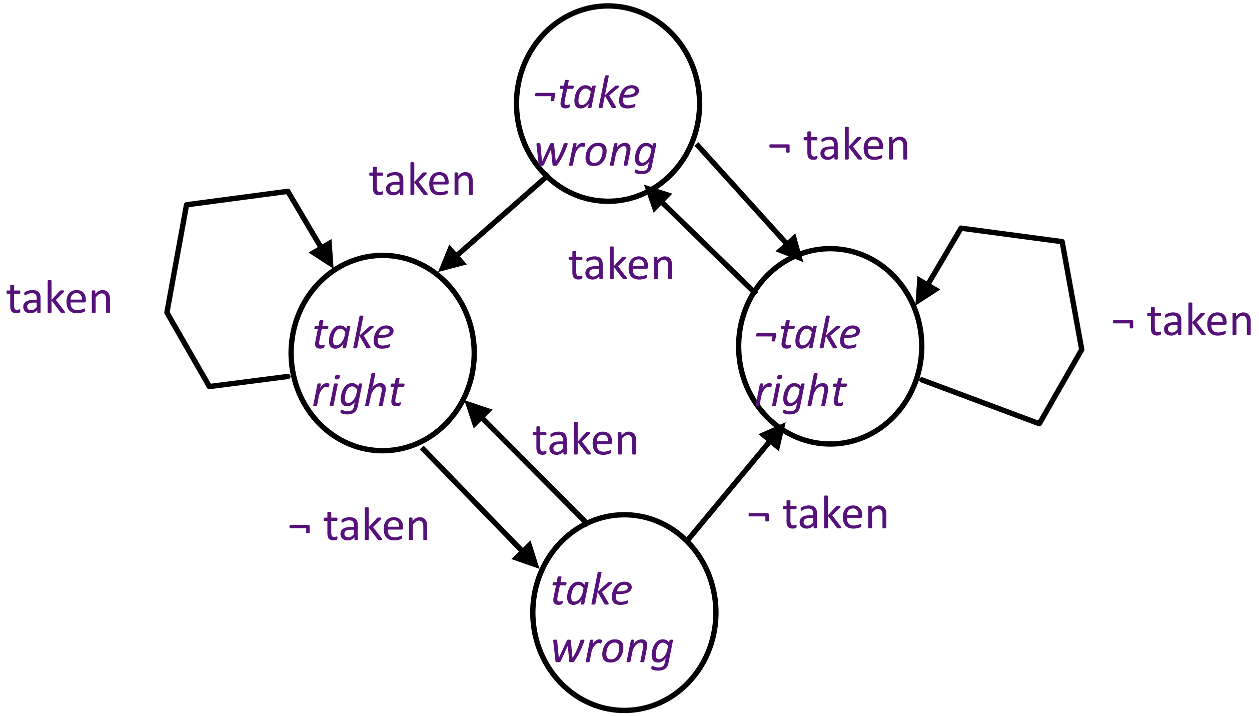
- **Solution: 2-bit scheme where change prediction only if get misprediction *twice:***



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

Branch Prediction Bits

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

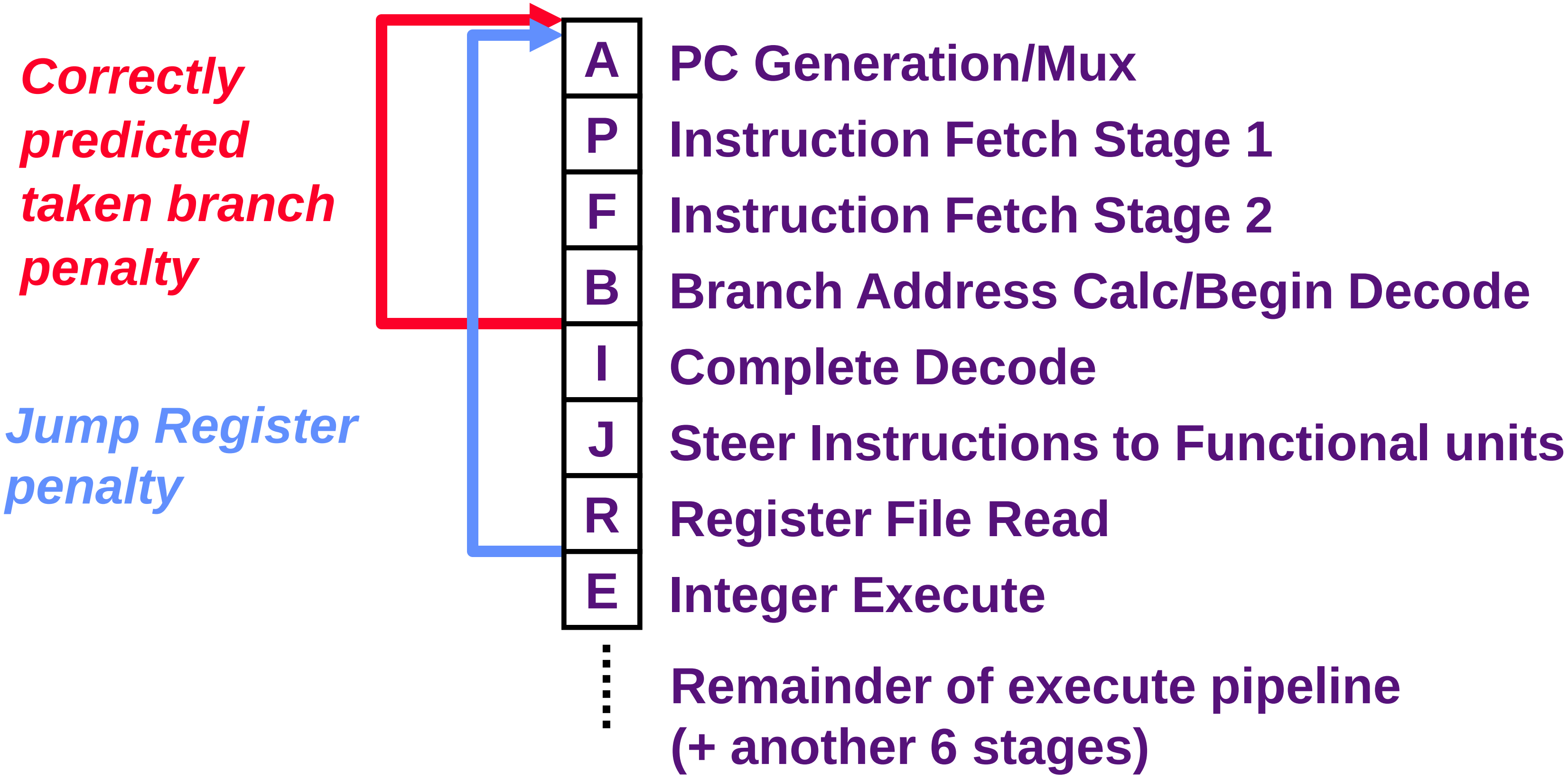


BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Pipeline considerations for BHT

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



Correctly predicted taken branch penalty

Jump Register penalty

A PC Generation/Mux

P Instruction Fetch Stage 1

F Instruction Fetch Stage 2

B Branch Address Calc/Begin Decode

I Complete Decode

J Steer Instructions to Functional units

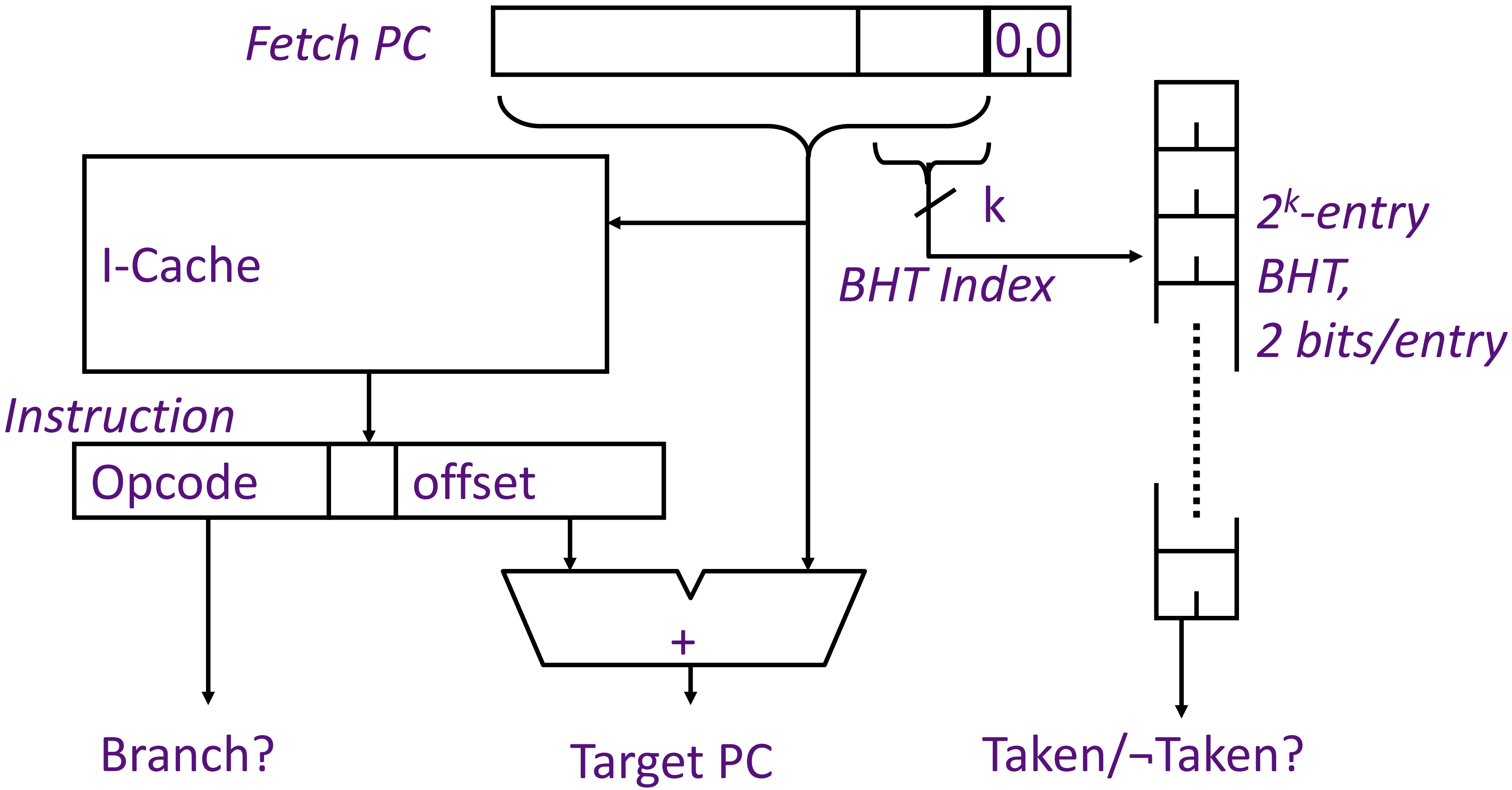
R Register File Read

E Integer Execute

Remainder of execute pipeline
(+ another 6 stages)

UltraSPARC-III fetch pipeline

Branch History Table (BHT)



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

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

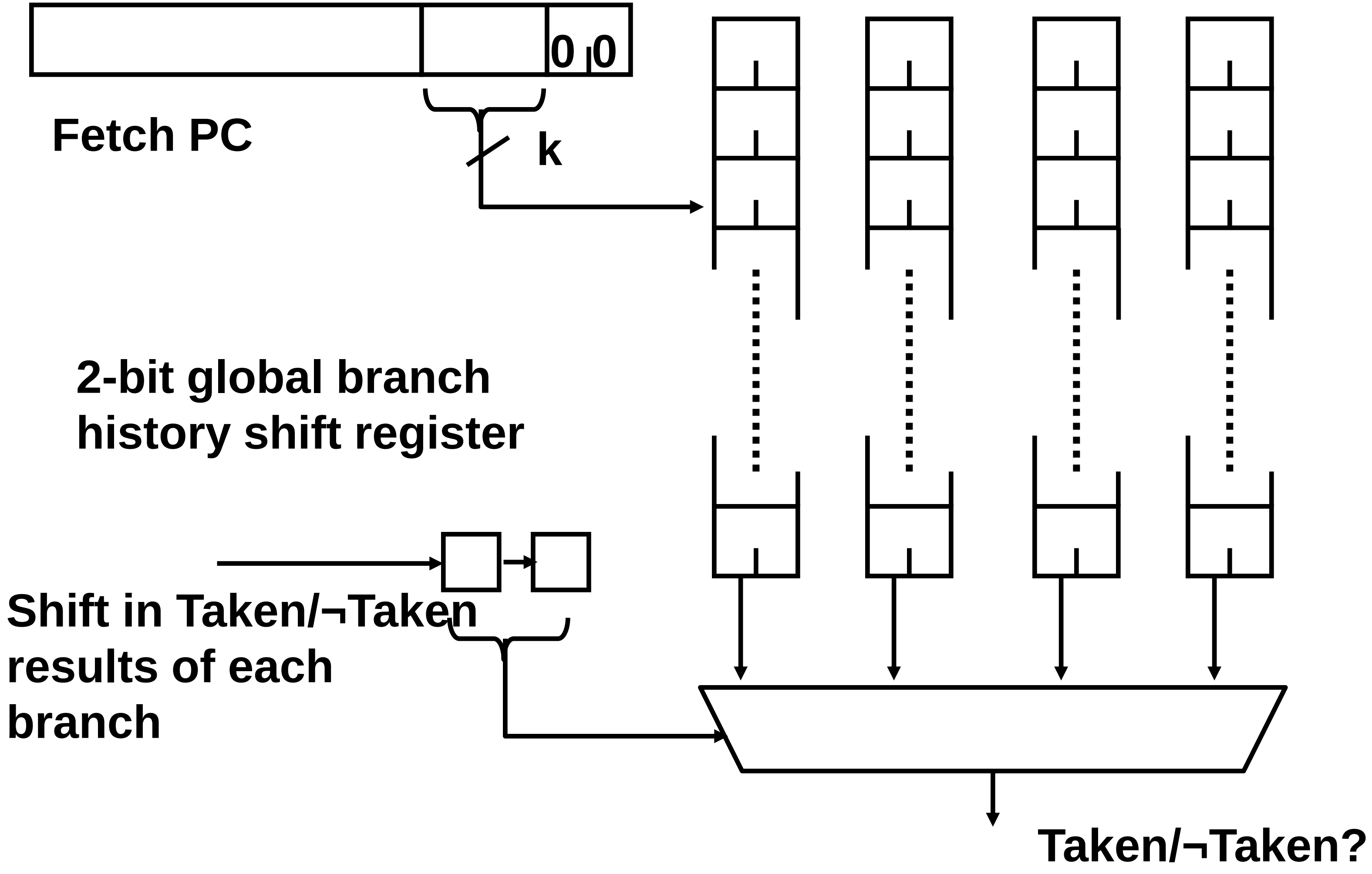
History register, H, records the direction of the last N branches executed by the processor

Correlating Branches

- 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)

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)

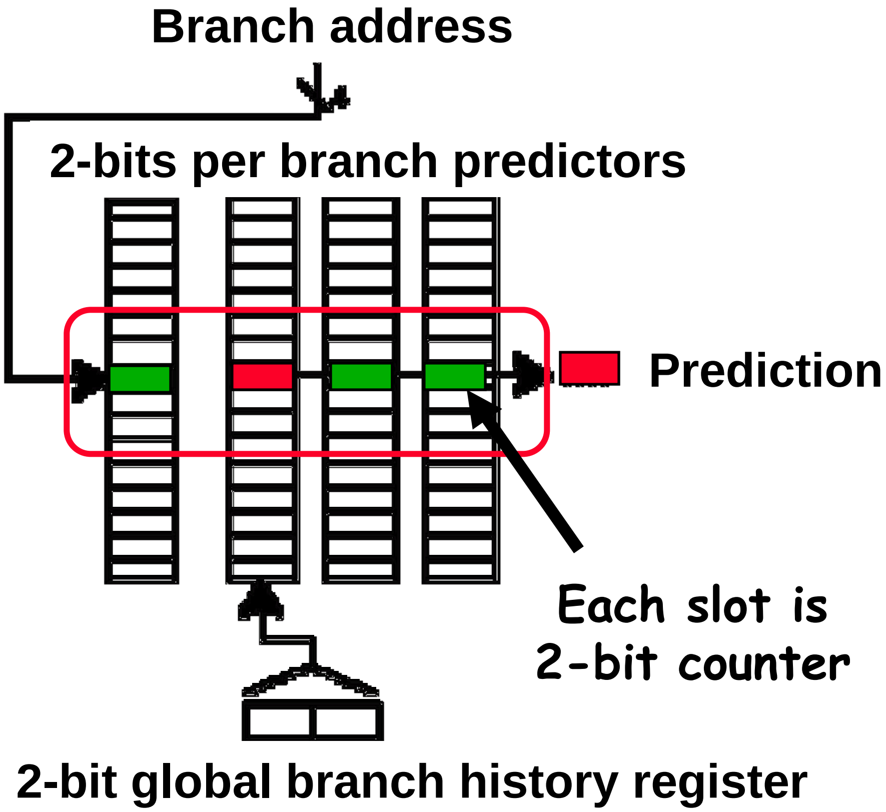

0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

Correlating Branches

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

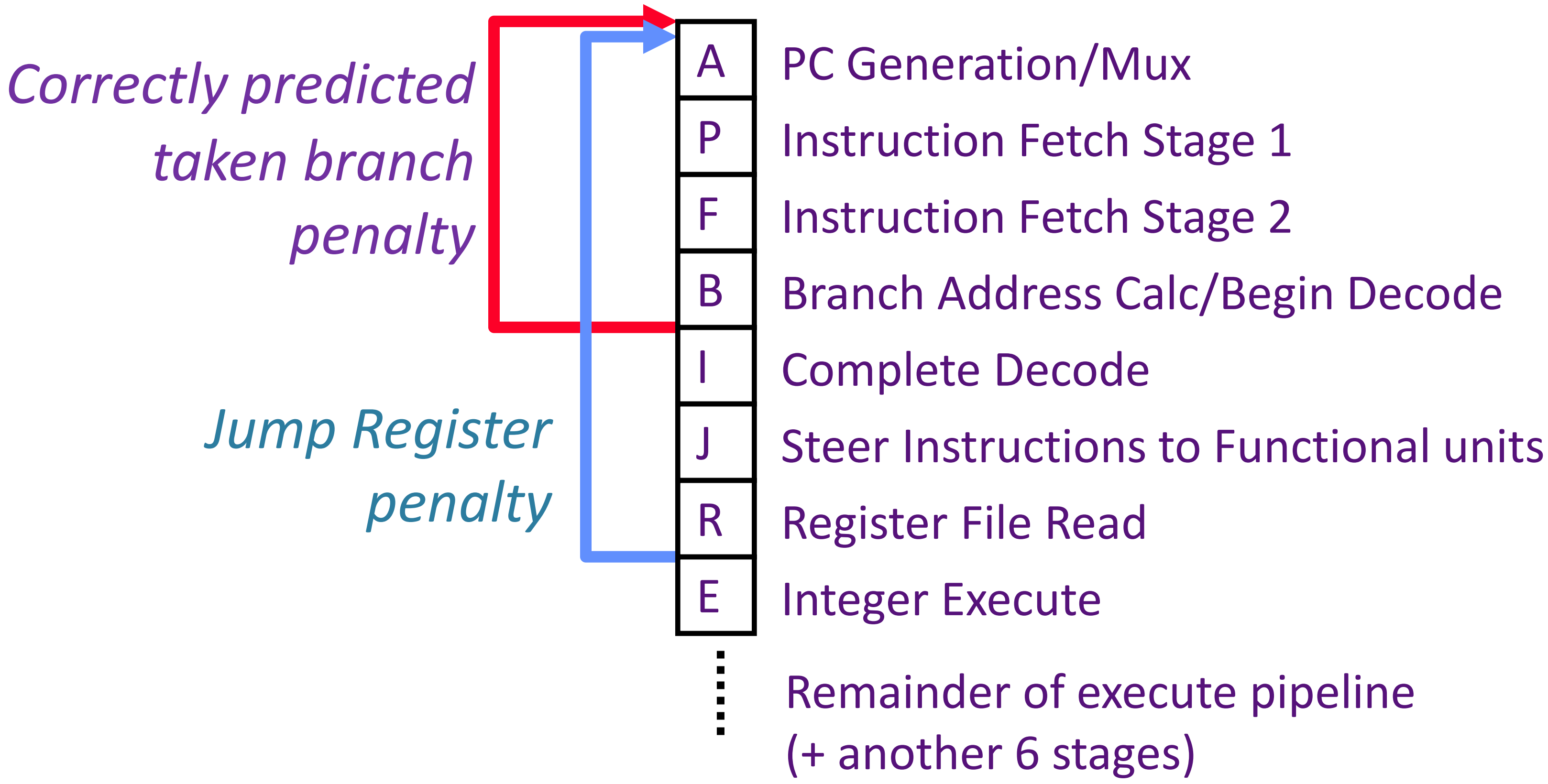
(2,2) GAs predictor

- 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...


Branch address
2-bits per branch predictors
Prediction
Each slot is
2-bit counter
2-bit global branch history register

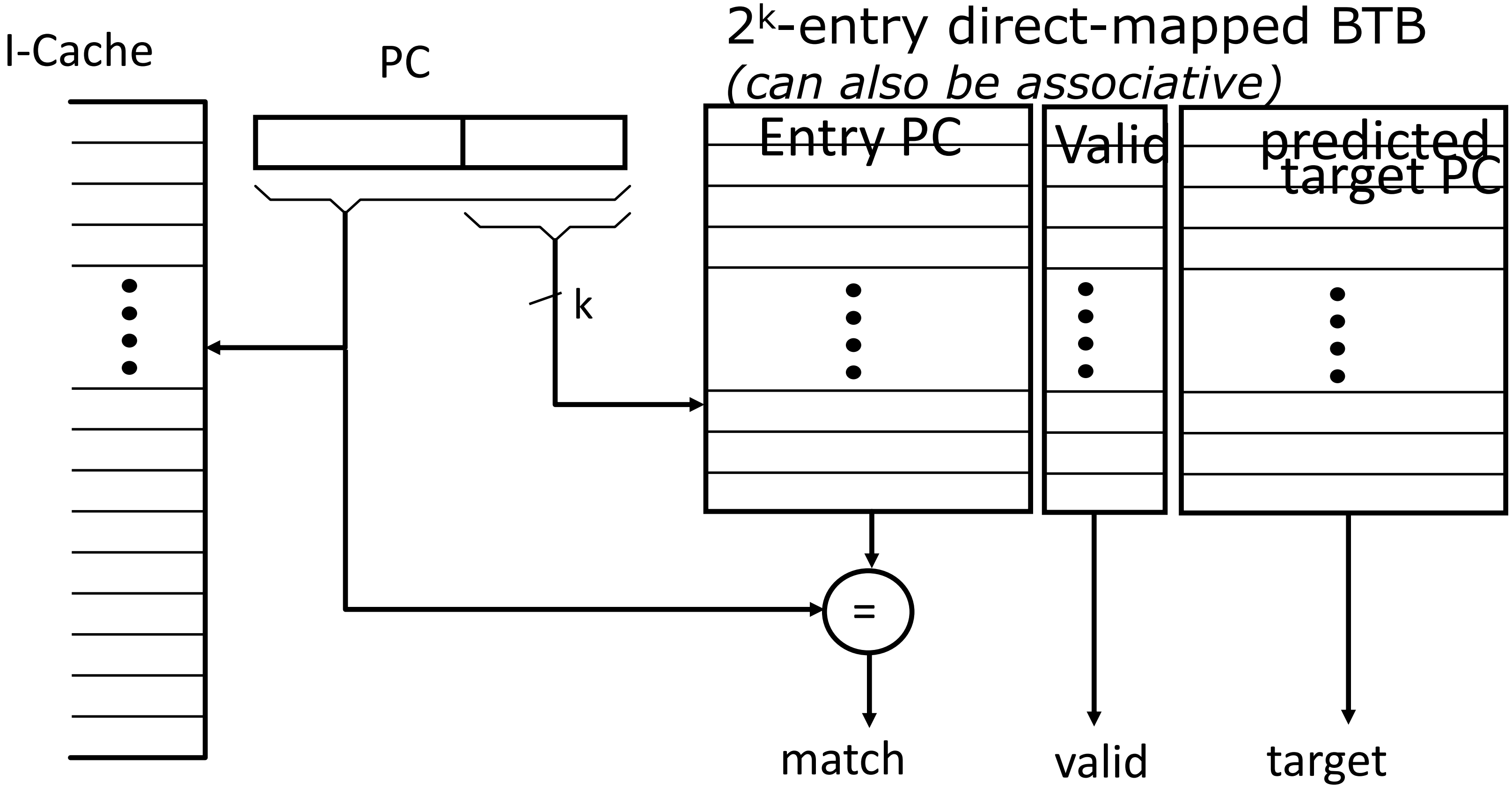
Limitations of BHTs

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


Correctly predicted taken branch penalty
Jump Register penalty
A
P
F
B
I
J
R
E
PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute
Remainder of execute pipeline
(+ another 6 stages)

UltraSPARC-III fetch pipeline

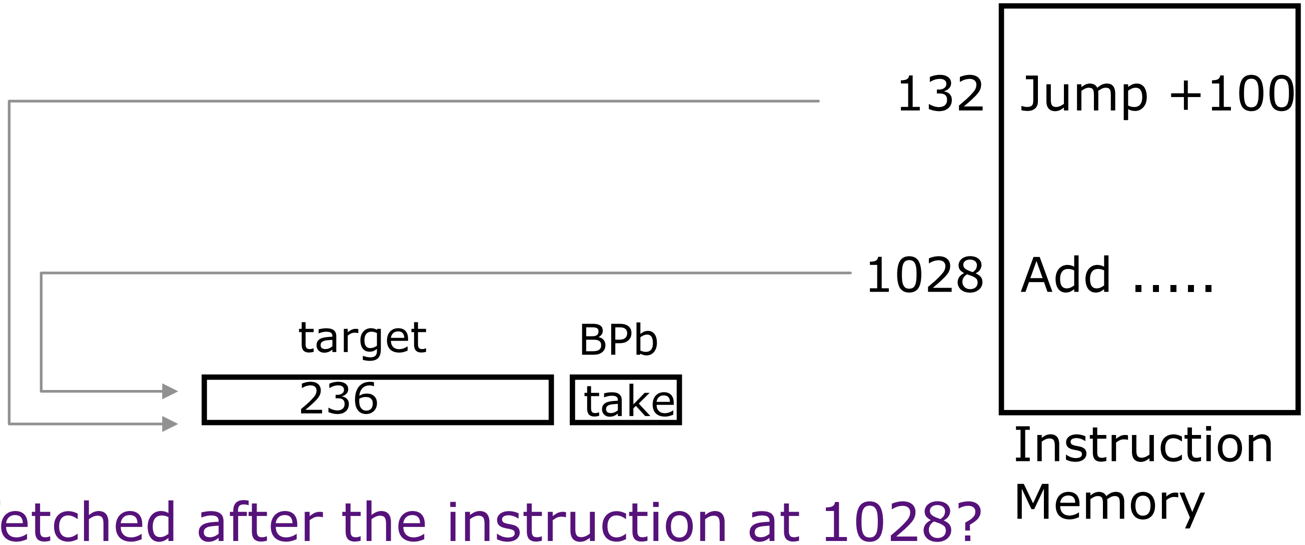
Branch Target Buffer (BTB)



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

Address Collisions in BTB

Assume a 128-entry 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

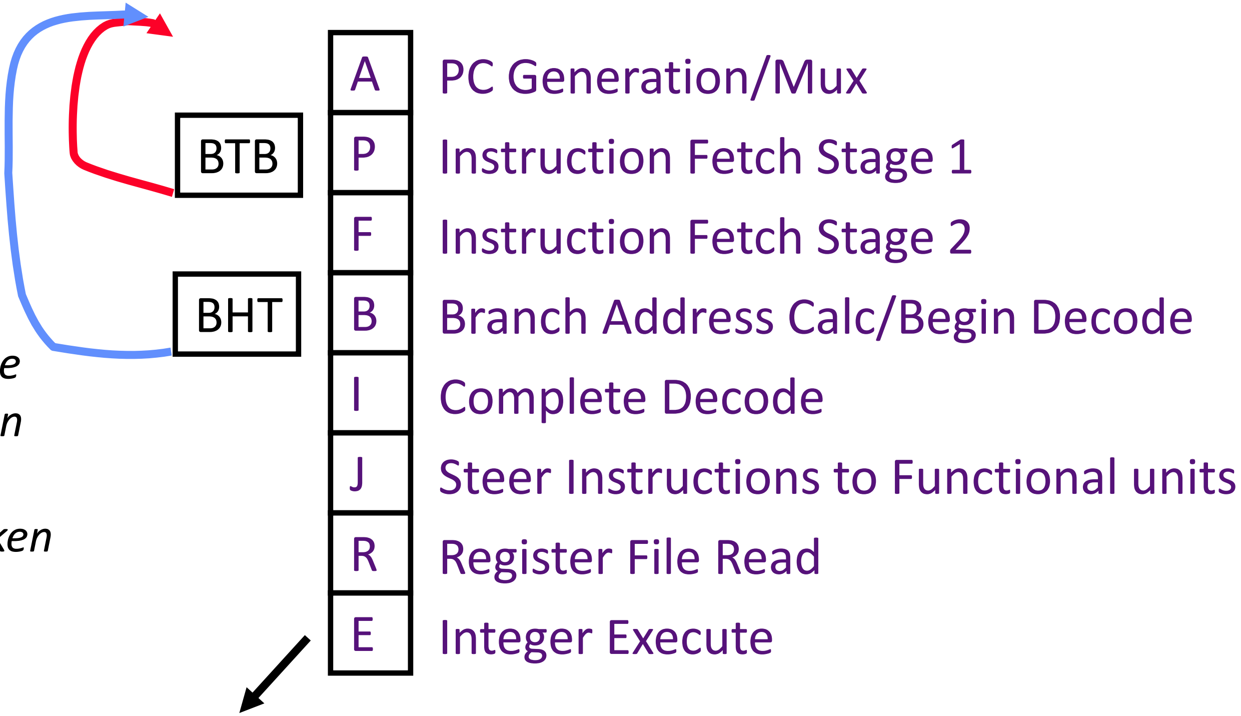
⇒ Do not update it for other instructions

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

How to achieve this effect without decoding the instruction?

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

A PC Generation/Mux

P Instruction Fetch Stage 1

F Instruction Fetch Stage 2

B Branch Address Calc/Begin Decode

I Complete Decode

J Steer Instructions to Functional units

R Register File Read

E Integer Execute

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

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!*

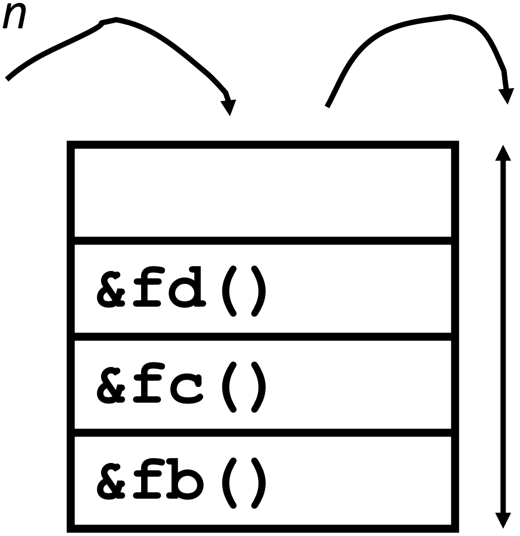
  How well does BTB work for each of these cases?

Subroutine Return Stack

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

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Push call address when function call executed

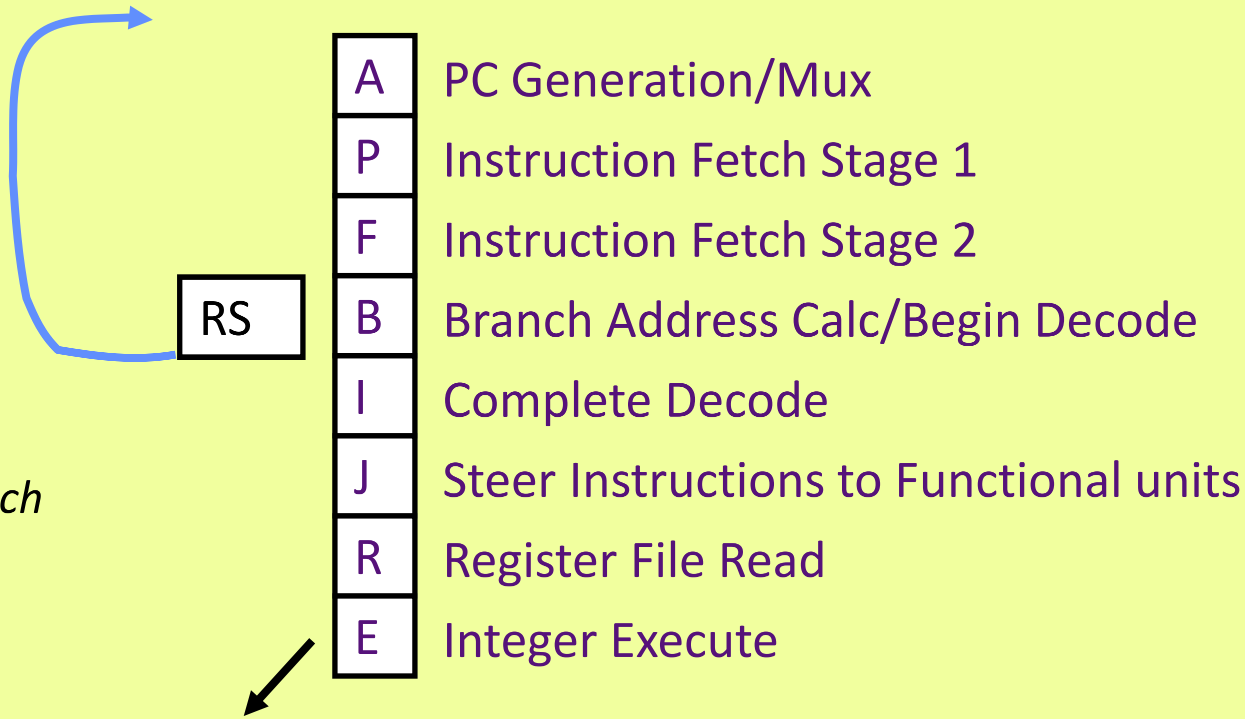


Pop return address when subroutine return decoded

k entries (typically k=8-16)

Return Stack in Pipeline

- How to use return stack (RS) in deep fetch pipeline?
- Only know if subroutine call/return at decode

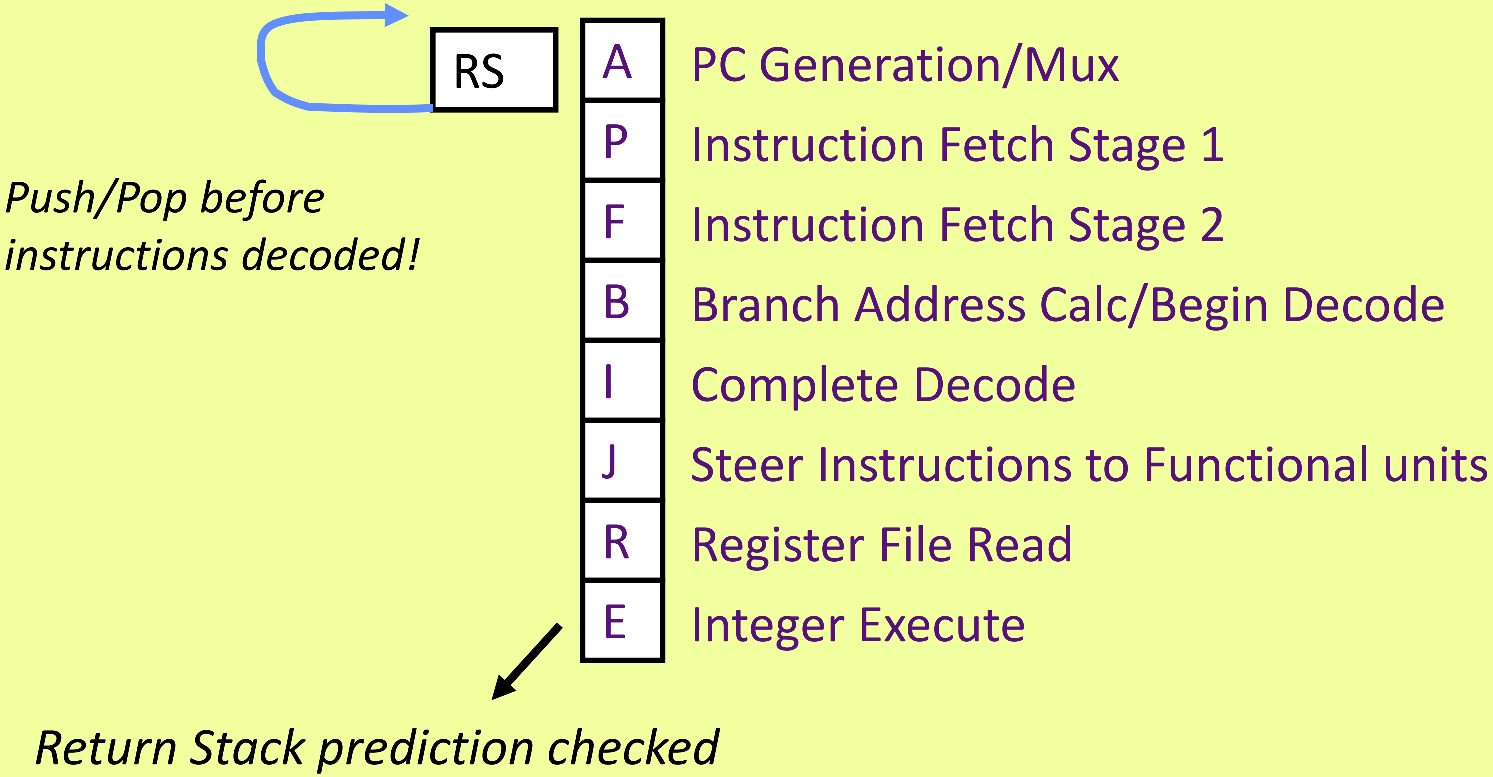

RS
A
P
F
B
I
J
R
E
PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute

RS Push/Pop after decode gives large bubble in fetch stream.

Return Stack prediction checked

Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



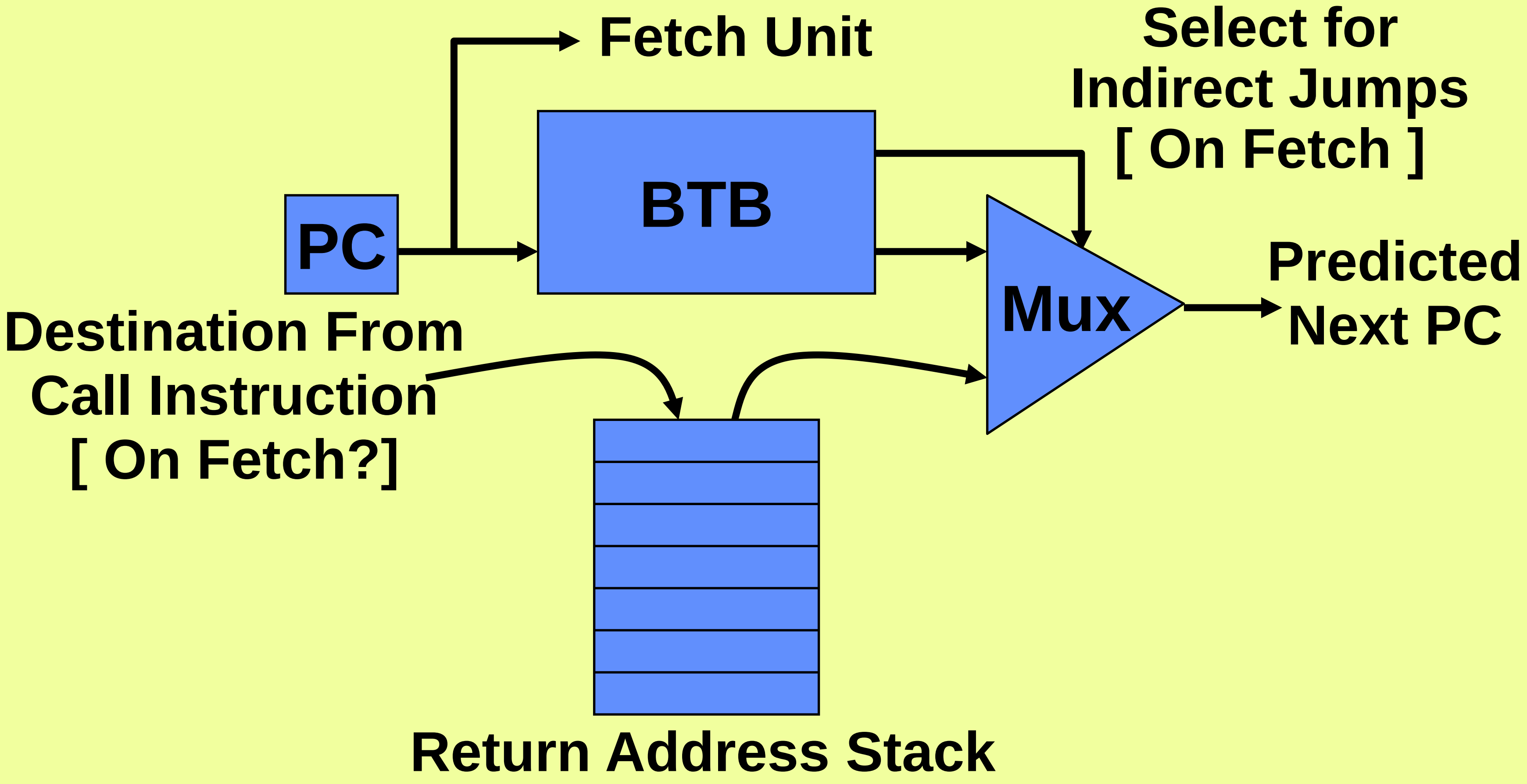
Push/Pop before instructions decoded!

A	PC Generation/Mux
P	Instruction Fetch Stage 1
F	Instruction Fetch Stage 2
B	Branch Address Calc/Begin Decode
I	Complete Decode
J	Steer Instructions to Functional units
R	Register File Read
E	Integer Execute

Return Stack prediction checked

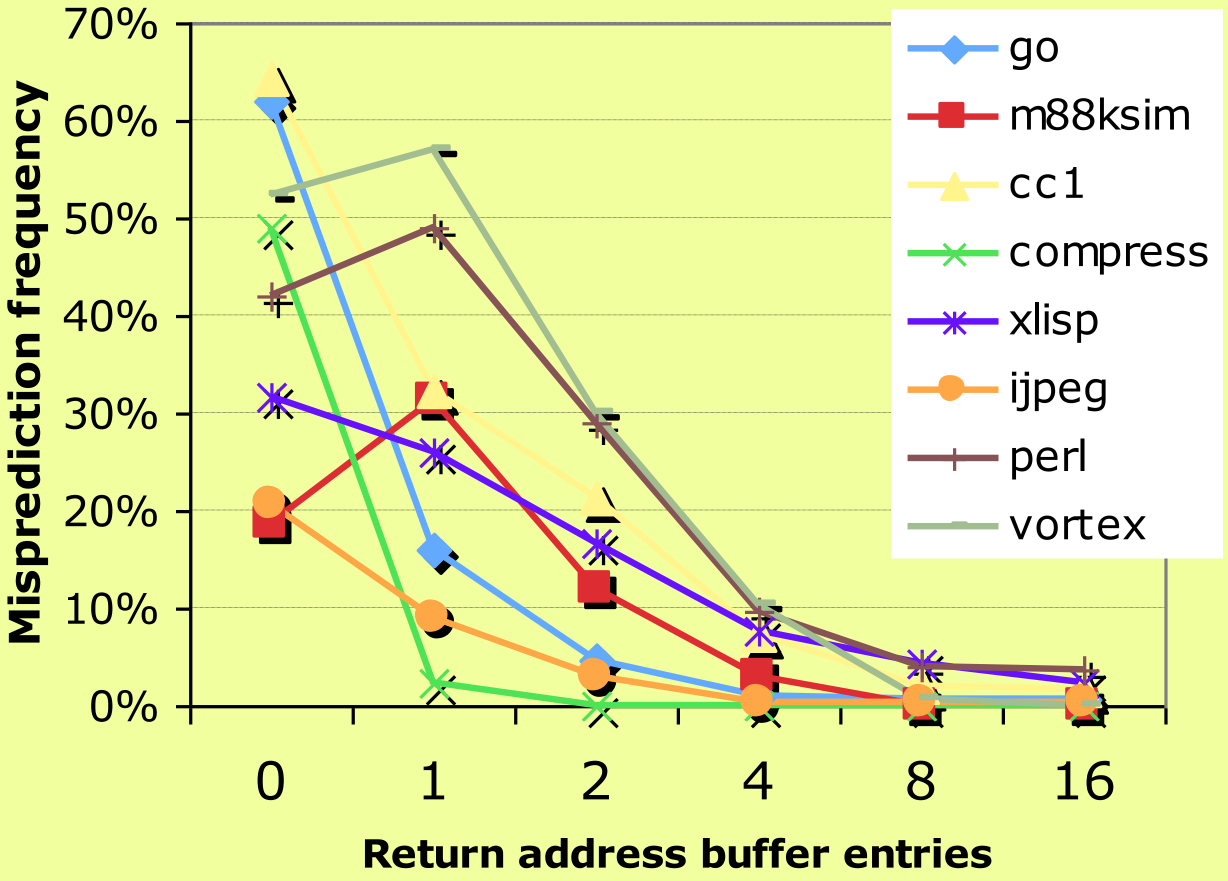
Special Case Return Addresses

- 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

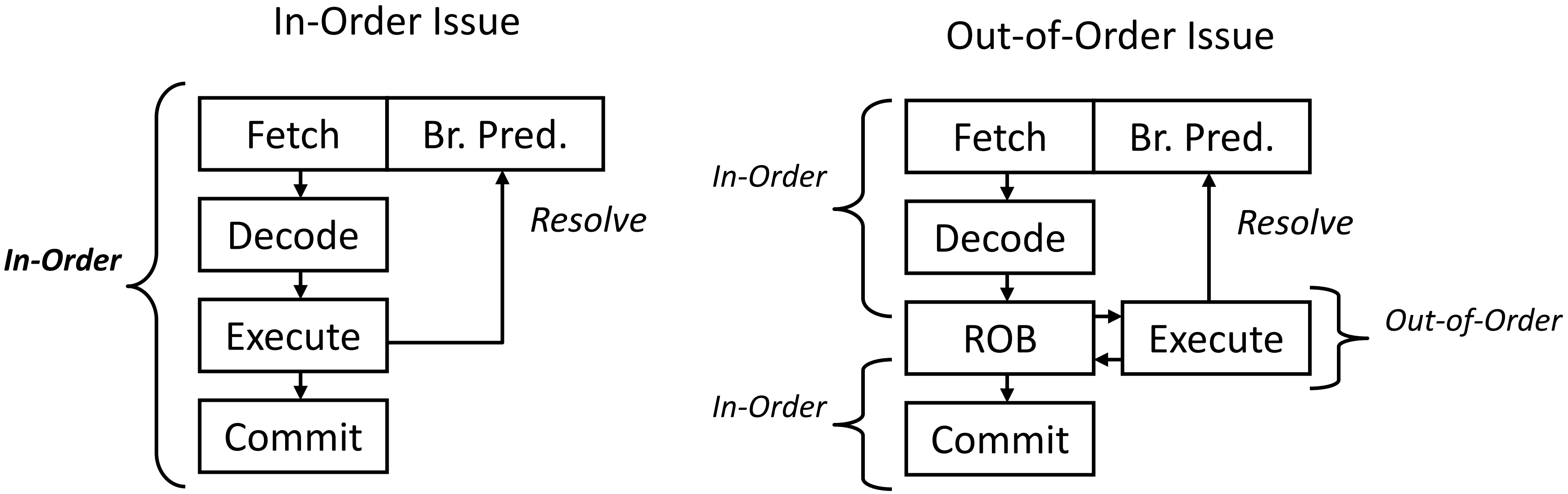
- Cache most recent return addresses:
  - Call ⇒ Push a return address on stack
  - Return ⇒ Pop an address off stack & predict as new PC



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!

In-Order vs. Out-of-Order Branch Prediction



- Speculative fetch but not speculative execution - branch resolves before later instructions complete
- Completed values held in bypass network until commit

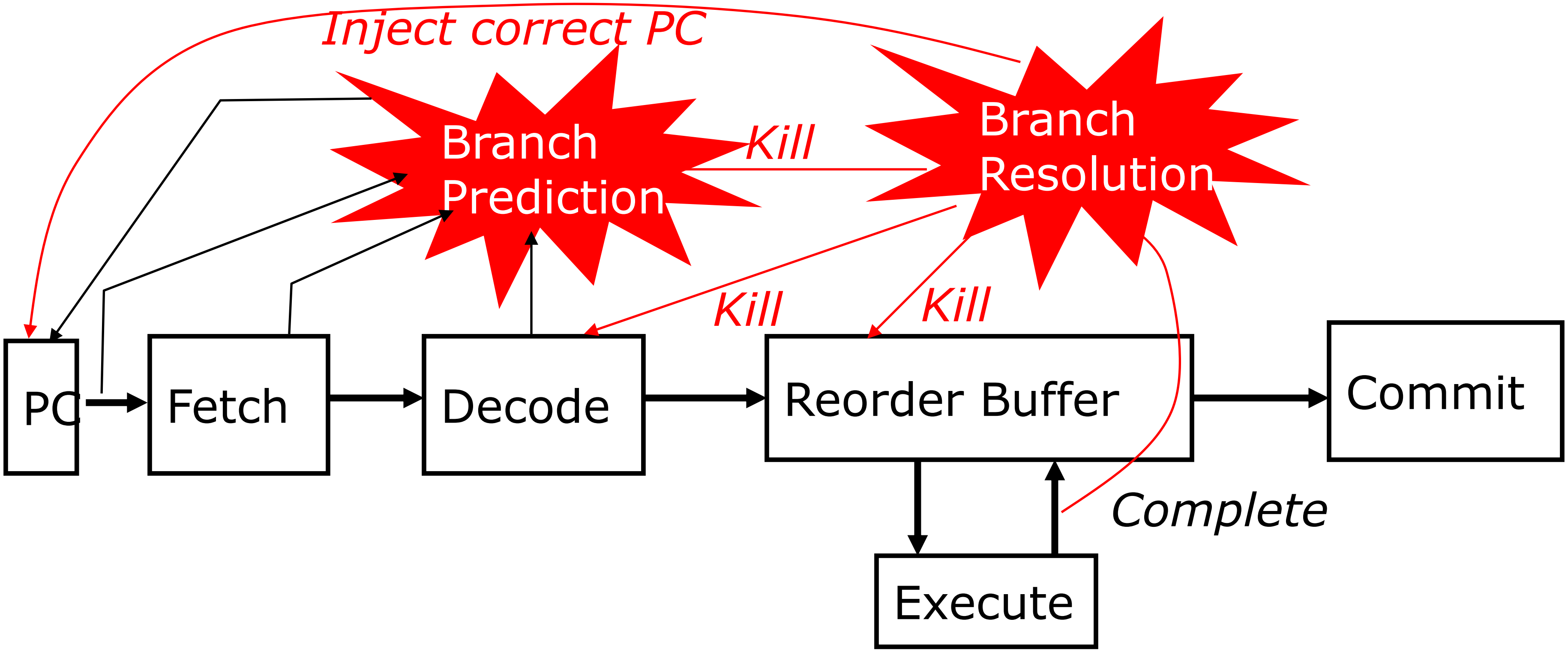
- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

- Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle
- Common to have 10-30 pipeline stages in either style of design

InO vs. OoO Mispredict Recovery

- In-order execution?
  - Design so 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
  - A simple solution would be to handle like precise traps
    - Problem?

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
- MIPS R10K uses four mask bits to tag instructions that are dependent on up to four speculative branches
- Mask bits cleared as branch resolves, and reused for next branch

Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

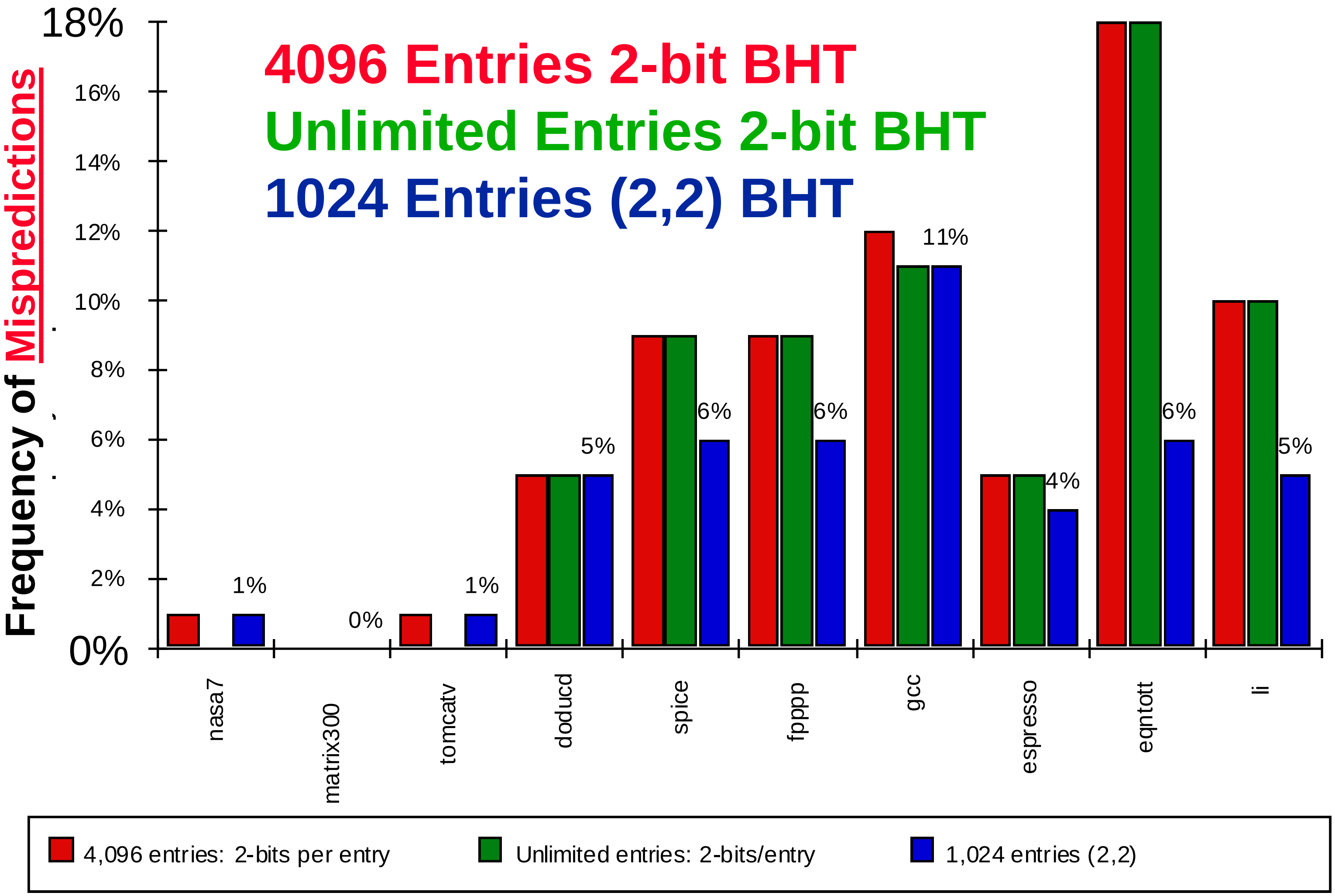
Improving Instruction Fetch

- Performance of speculative out-of-order machines often limited by instruction fetch bandwidth
  - speculative execution can fetch 2-3x more instructions than are committed
  - mispredict penalties dominated by time to refill instruction window
  - taken branches are particularly troublesome

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



4096 Entries 2-bit BHT

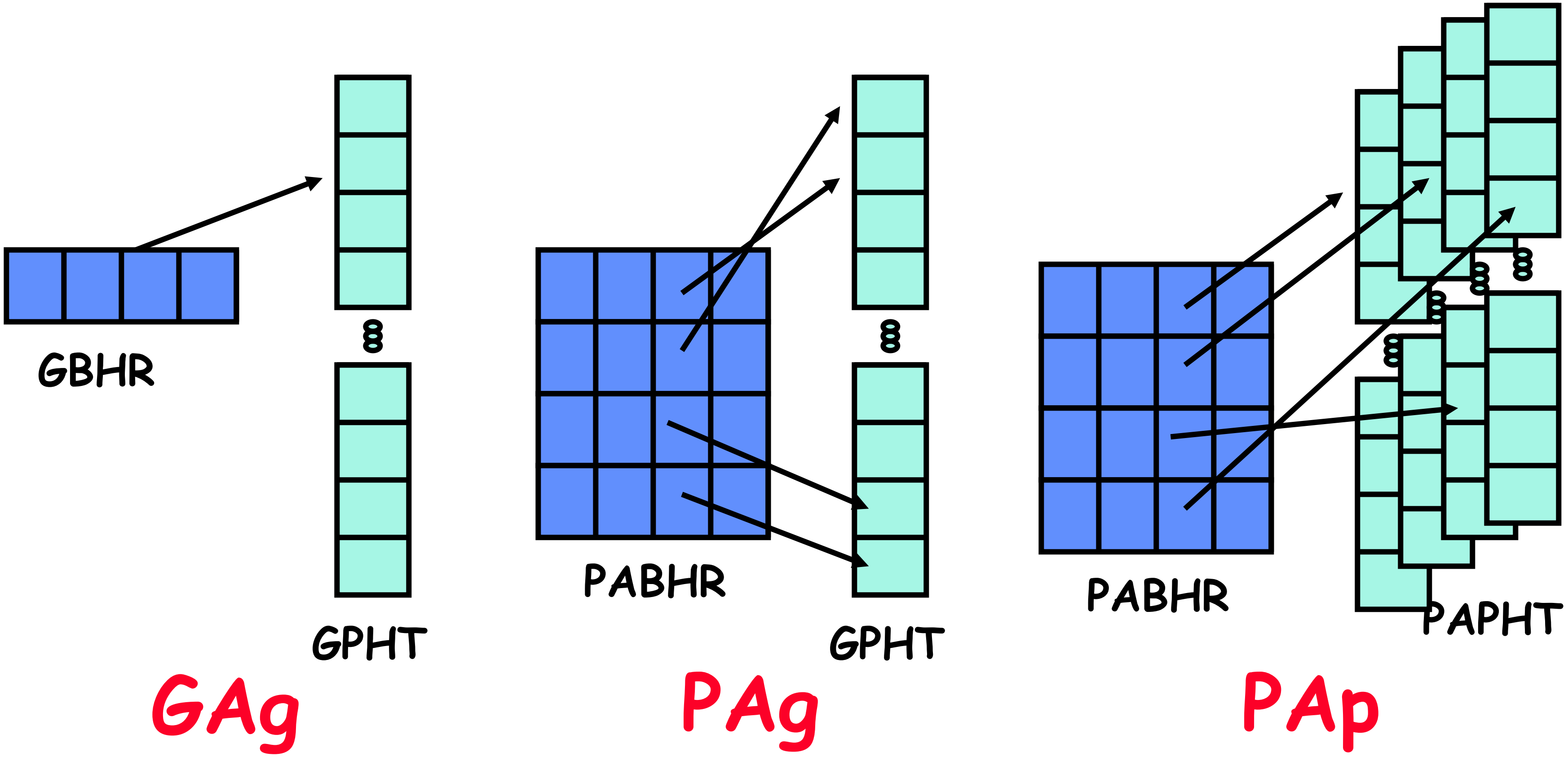
Unlimited Entries 2-bit BHT

1024 Entries (2,2) BHT

BHT Accuracy

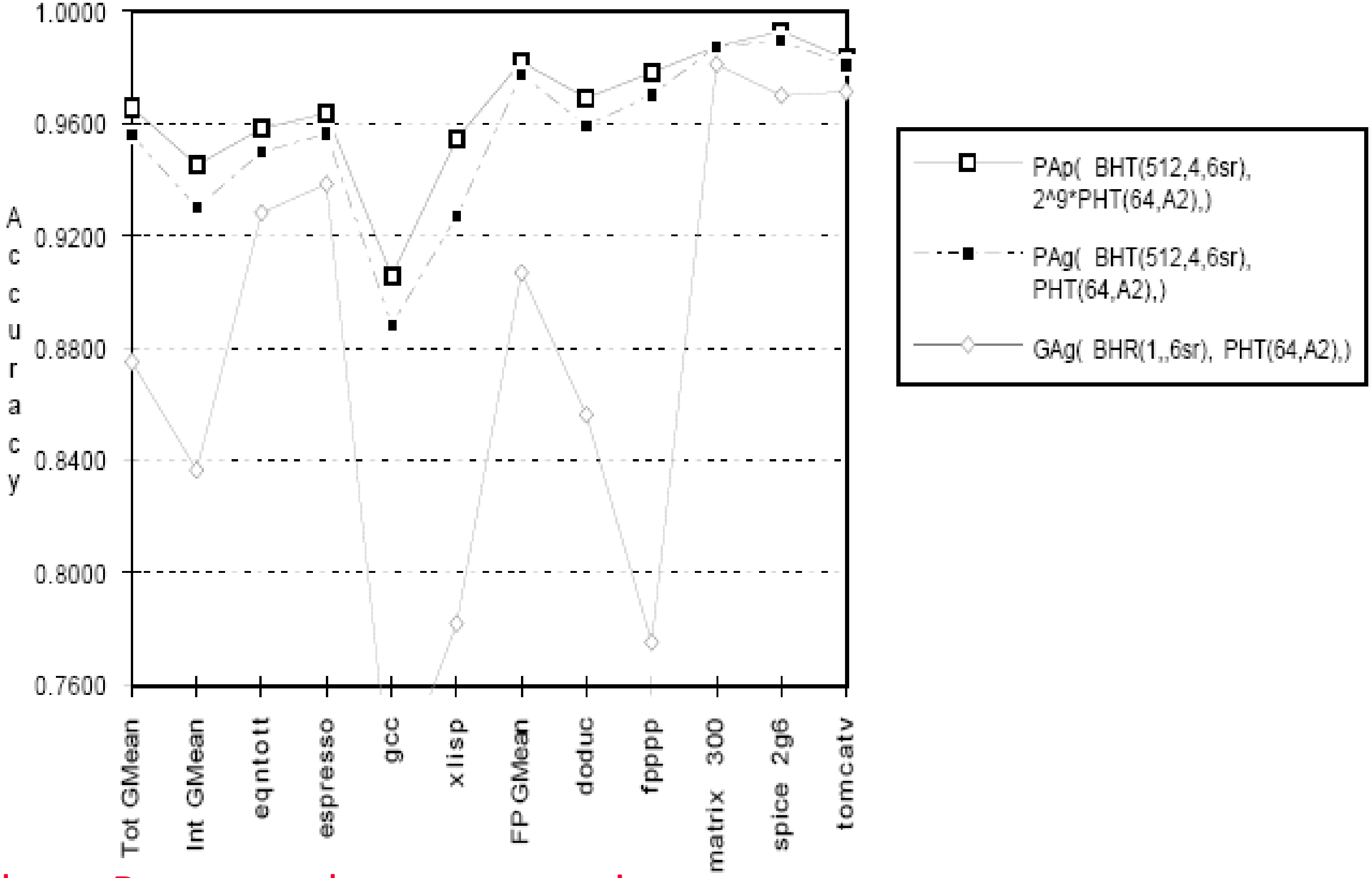
- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
  - For SPEC92, 4096 about as good as infinite table
- How 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



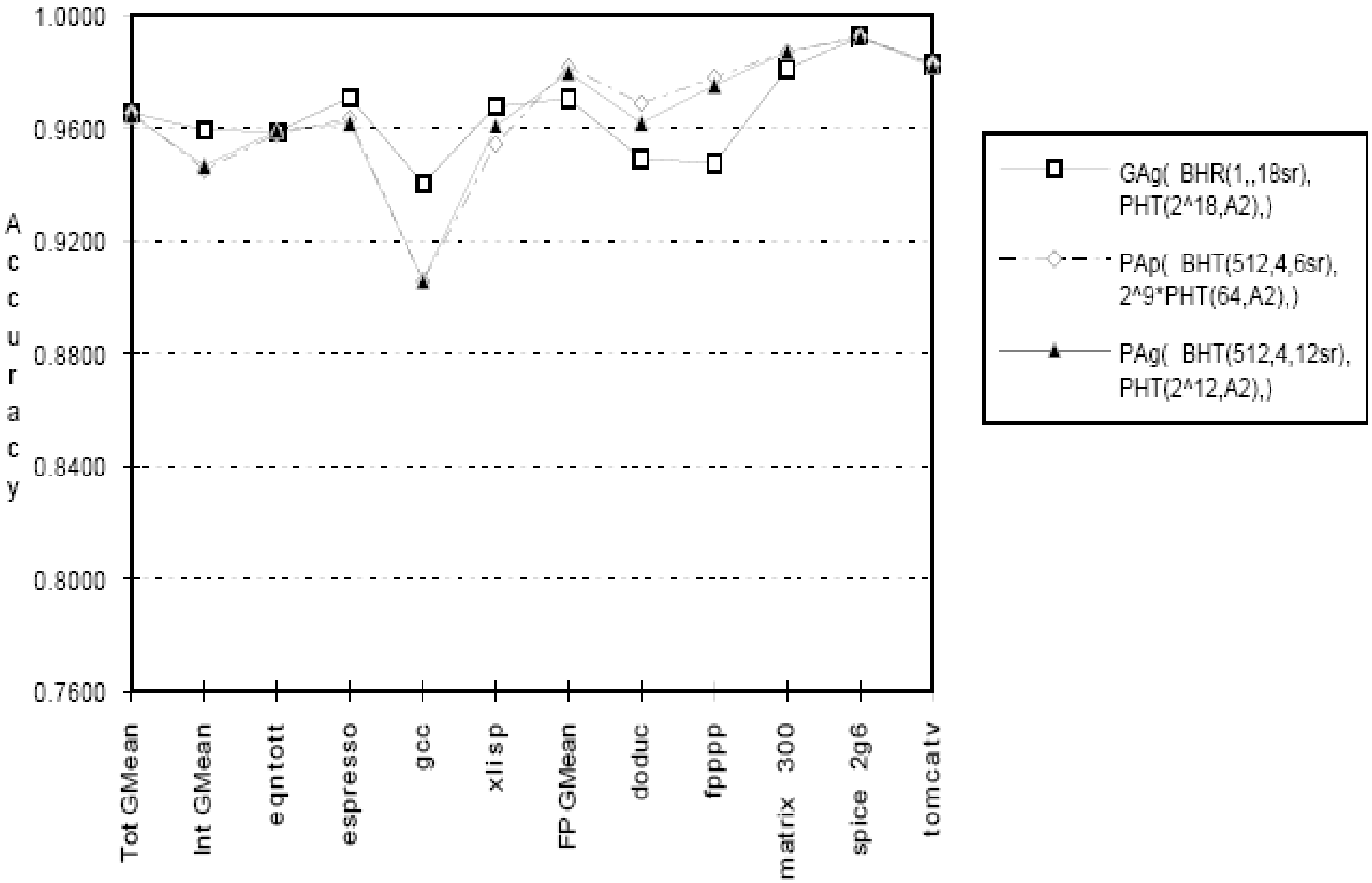
- GAg: Global History Register, Global History Table
- PAg: Per-Address History Register, Global History Table
- PAp: Per-Address History Register, Per-Address History Table

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⇒interference
- PAg performs better because it has a branch history table

Versions with Roughly same accuracy (97%)

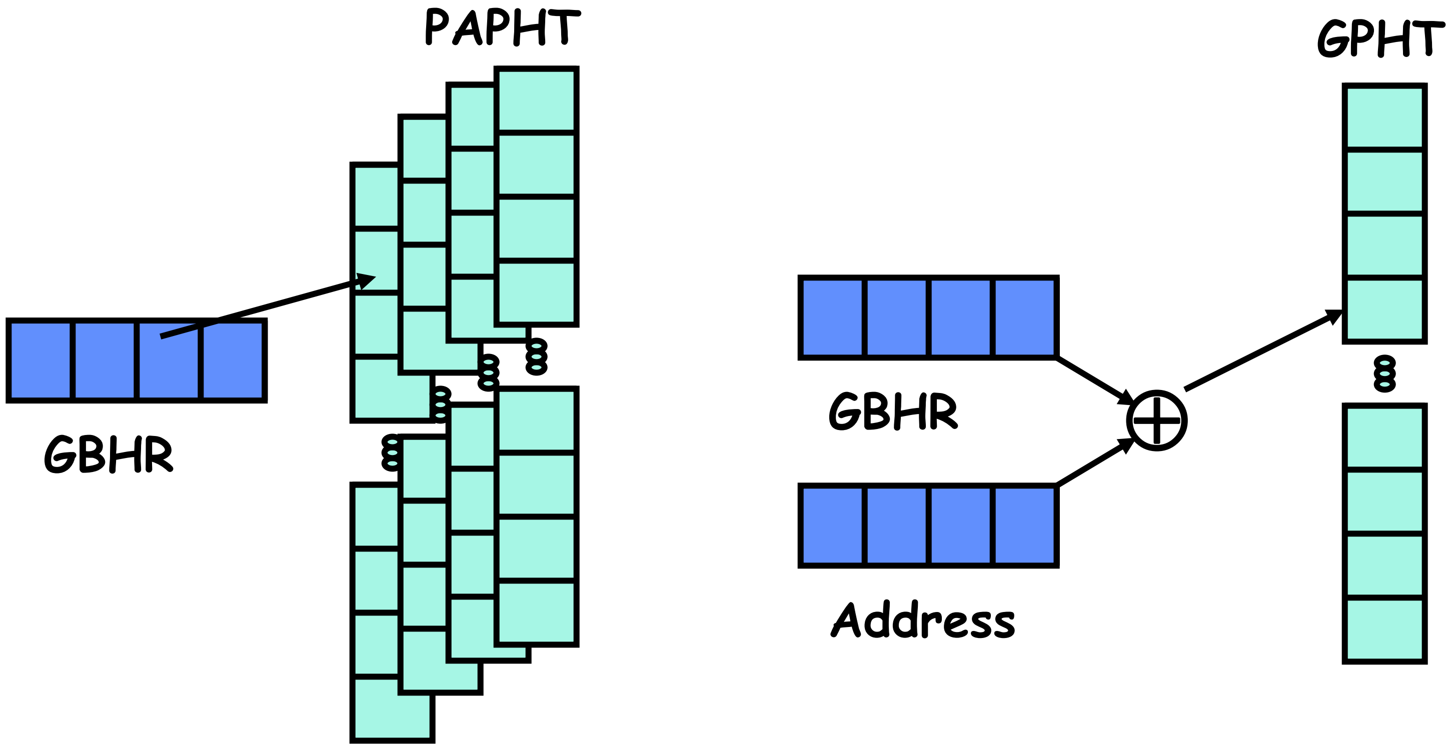


- Cost:
  - GAg requires 18-bit history register
  - PAg requires 12-bit history register
  - PAp requires 6-bit history register
- PAg is the cheapest among these

Why doesn’t GAg do better?

- Difference between GAg and both PA variants:
  - GAg tracks correllations between different branches
  - PAg/PAp track corellations 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

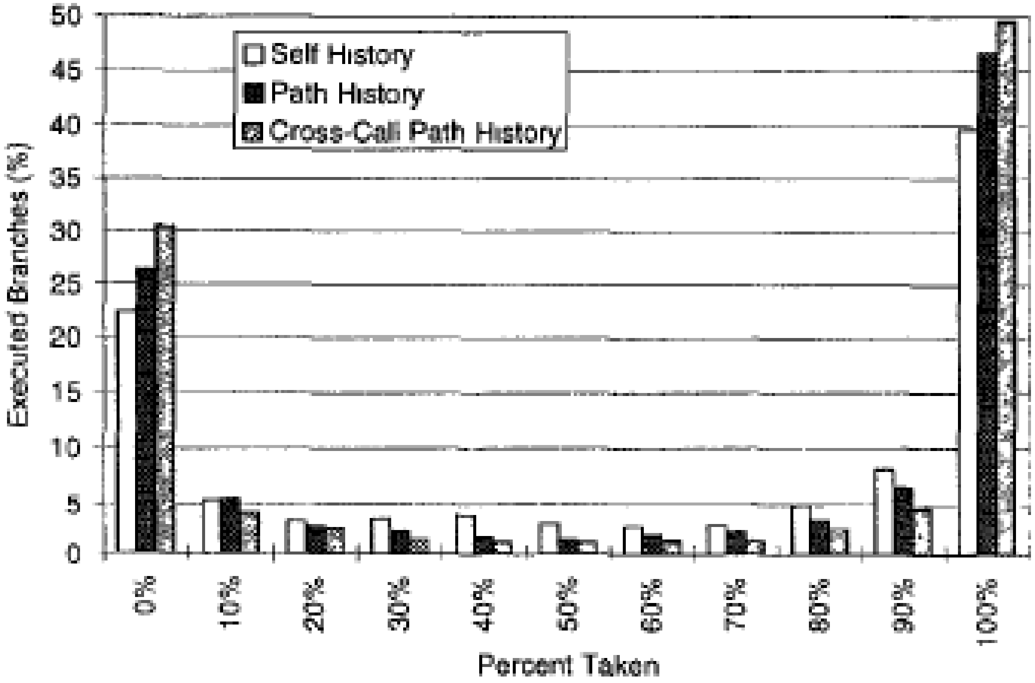


GAs

GShare

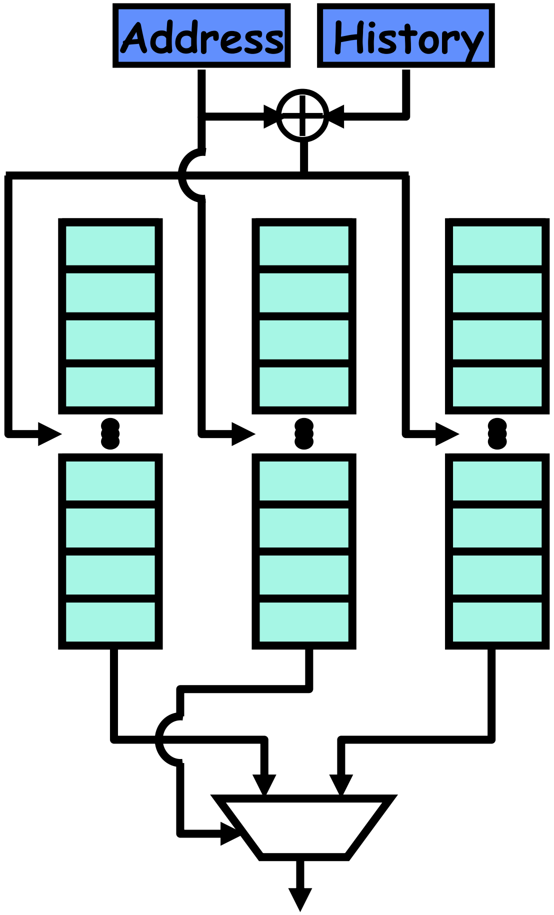
- GAs: Global History Register,
  Per-Address (Set Associative) History Table
- Gshare: Global History Register, Global History Table with Simple attempt at anti-aliasing

Branches are Highly Biased

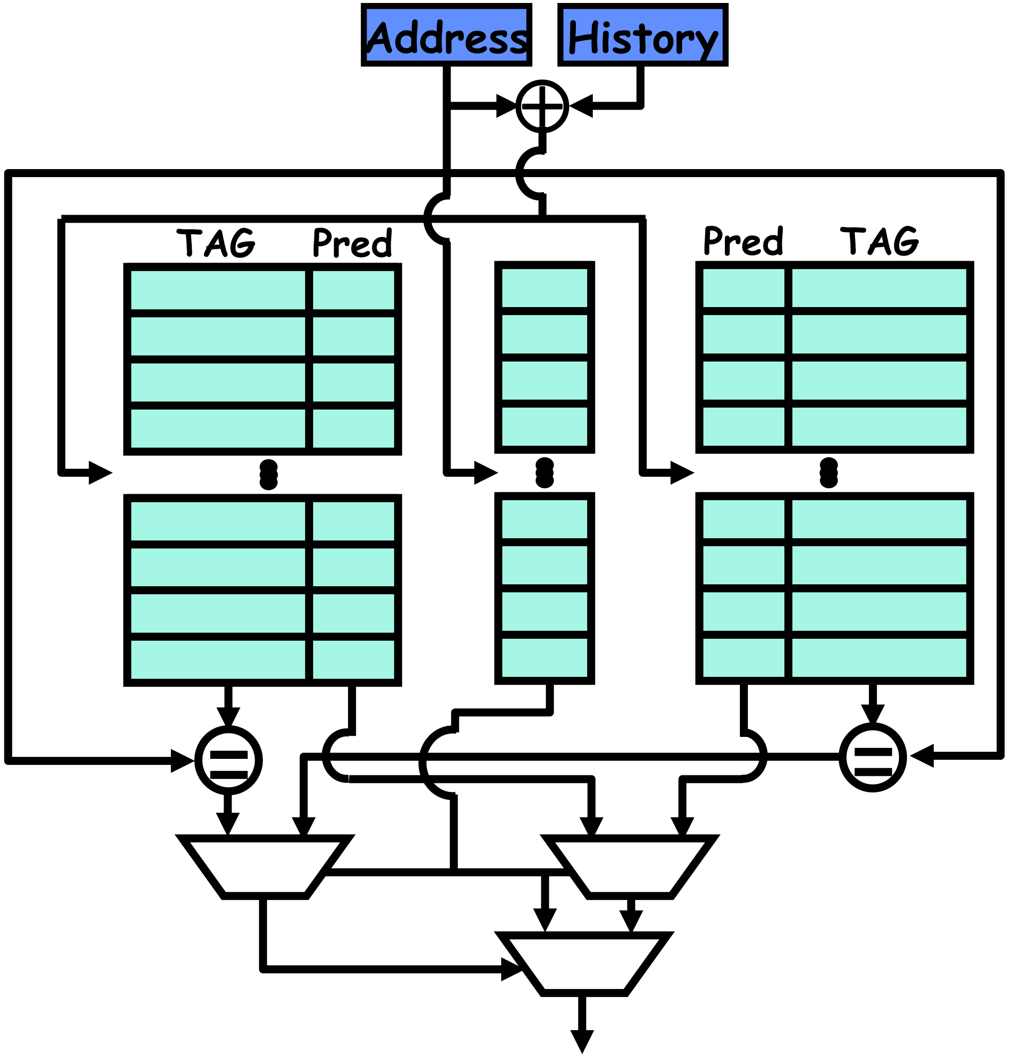


- From: “A Comparative Analysis of Schemes for Correlated Branch Prediction,” by Cliff Young, Nicolas Gloy, and Michael D. Smith
- Many branches are highly biased to be taken or not taken
  - Use of path history can be used to further bias branch behavior
- Can we exploit bias to better predict the unbiased branches?
  - Yes: filter out biased branches to save prediction resources for the unbiased ones

Exploiting Bias to avoid Aliasing: Bimode and YAGS

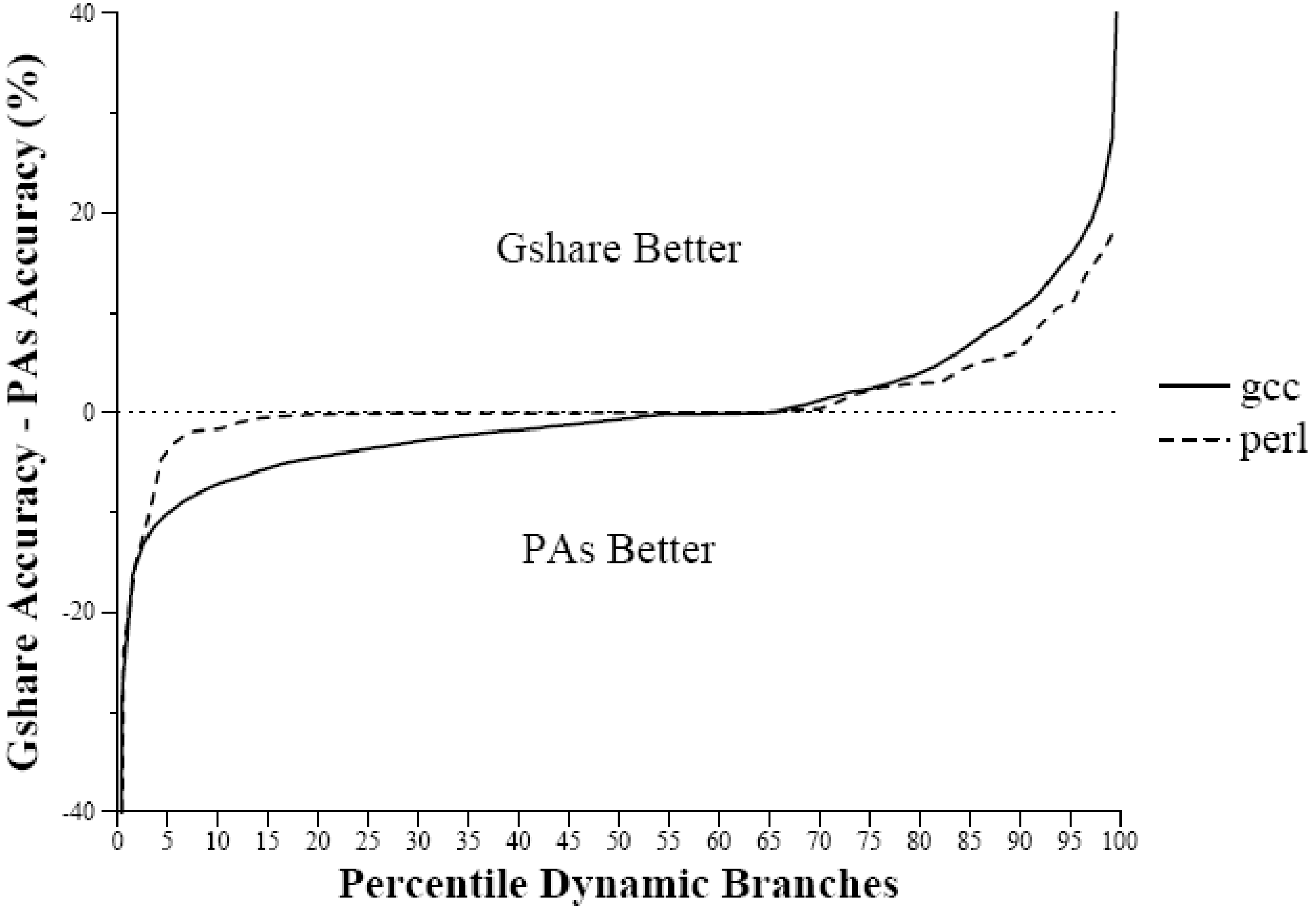


BiMode



YAGS

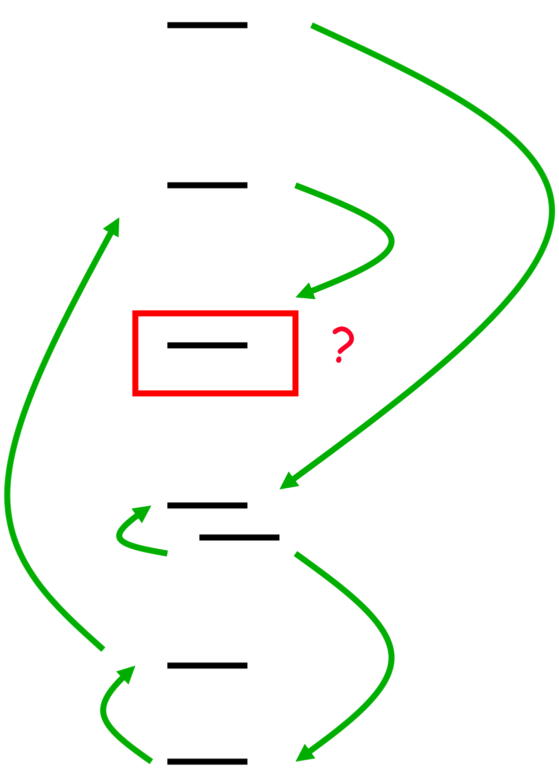
Is Global or Local better?



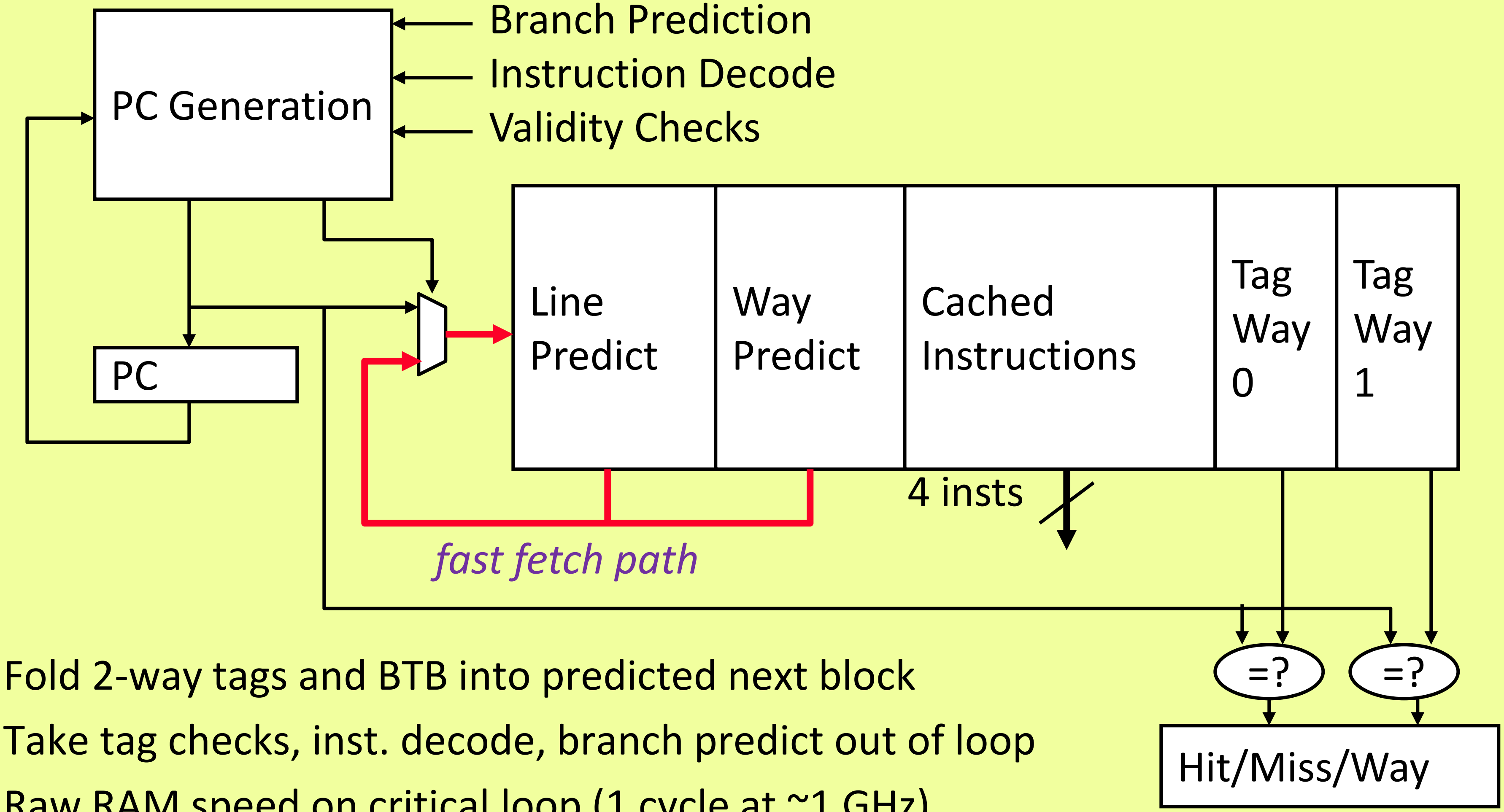
- Neither: Some branches local, some global
  - From: "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," Evers, Patel, Chappell, Patt
  - Difference in predictability quite significant for some branches!

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”



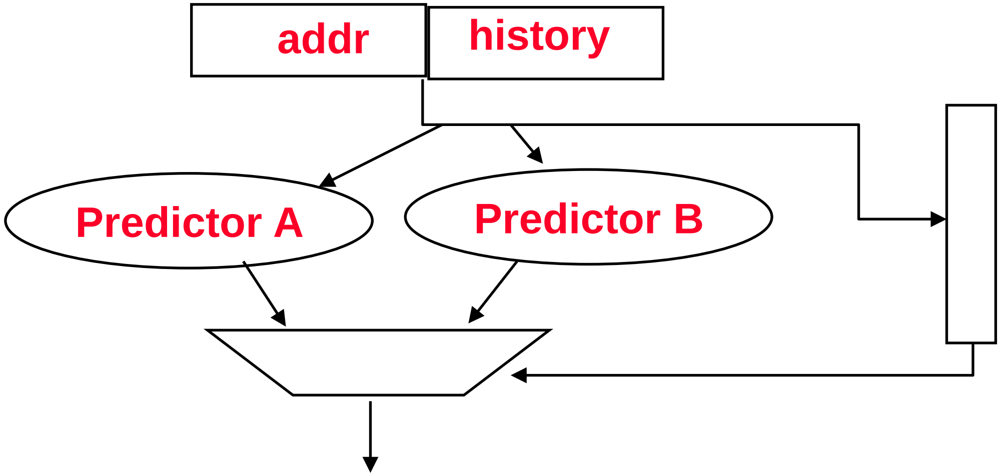
Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

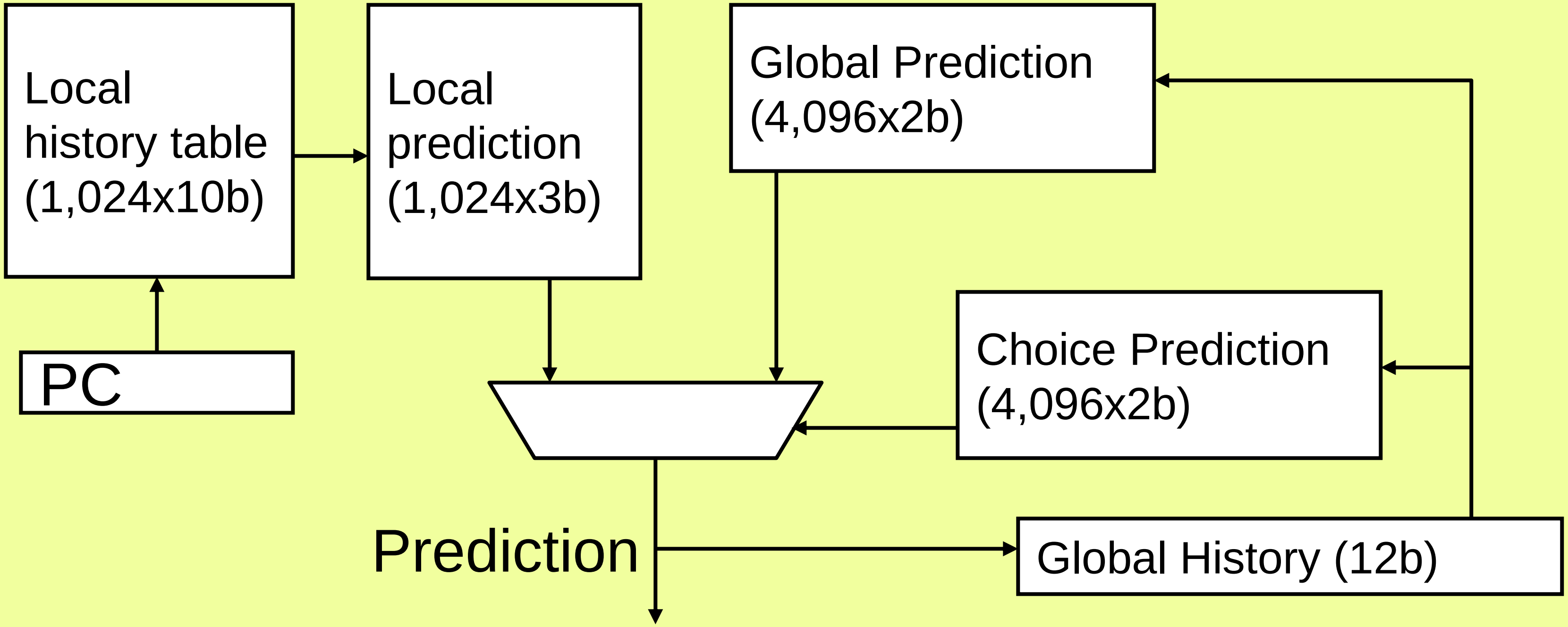
Tournament Predictors

- 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 Branch Predictor (Alpha 21264)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

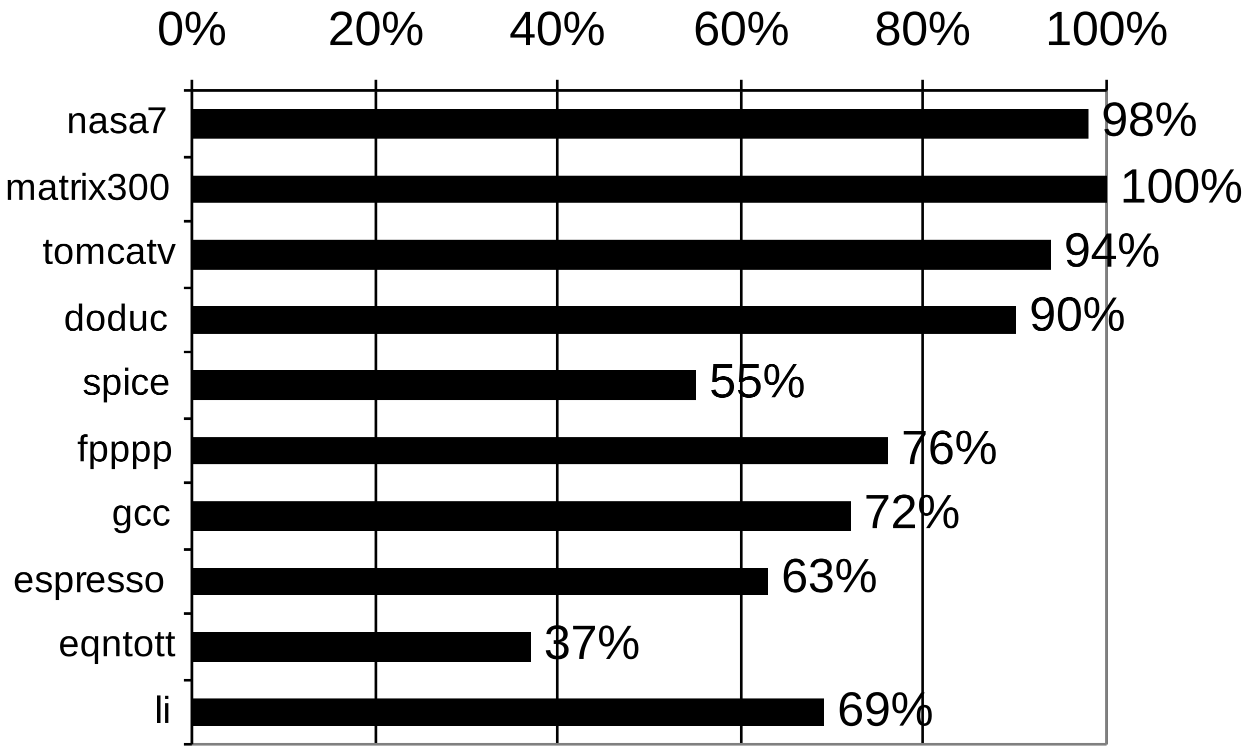

Local history table (1,024x10b)
Local prediction (1,024x3b)
Global Prediction (4,096x2b)
Choice Prediction (4,096x2b)
PC
Prediction
Global History (12b)

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 bits!

  (~180,000 transistors)

% of predictions from local predictor in Tournament Scheme



Accuracy of Branch Prediction

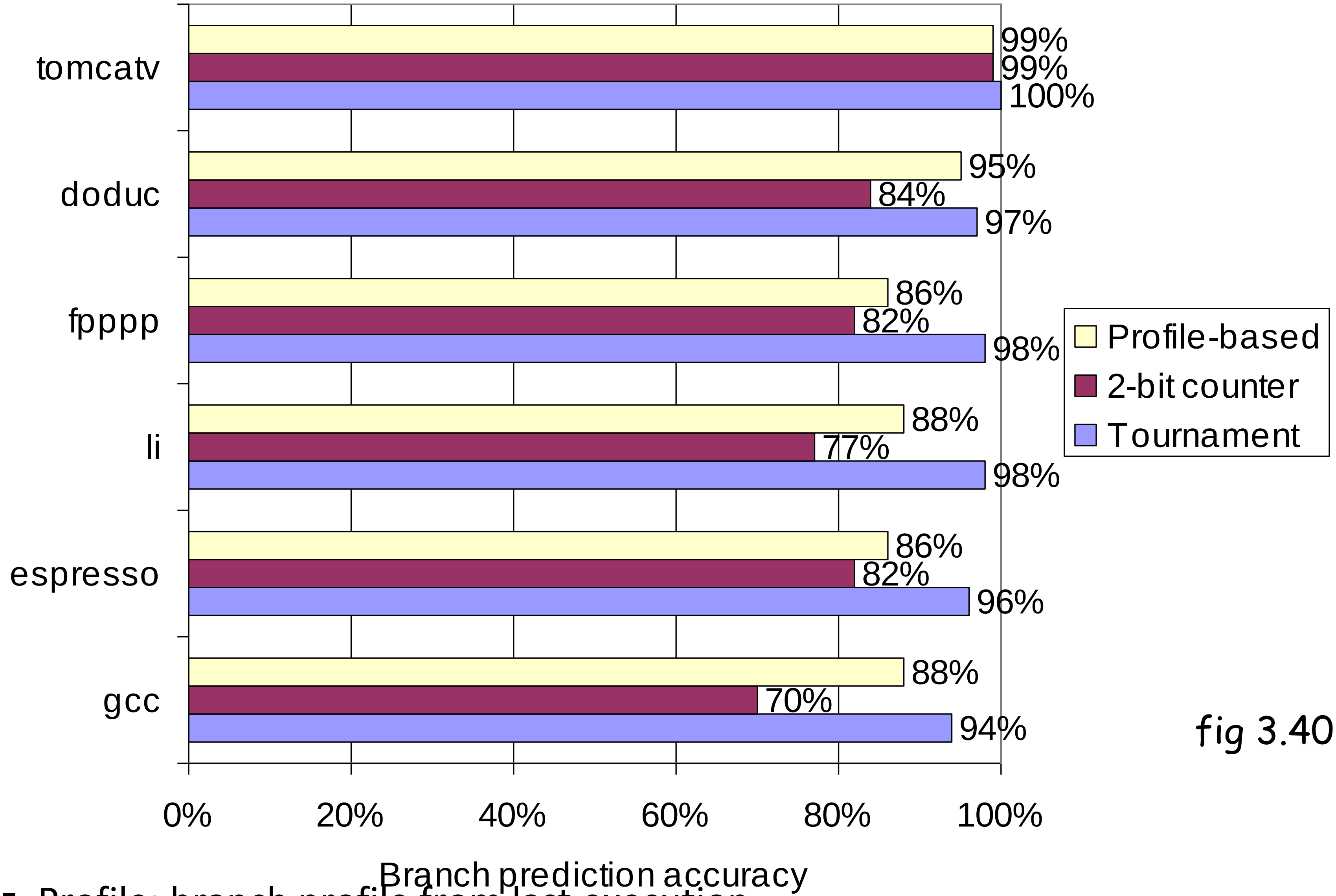
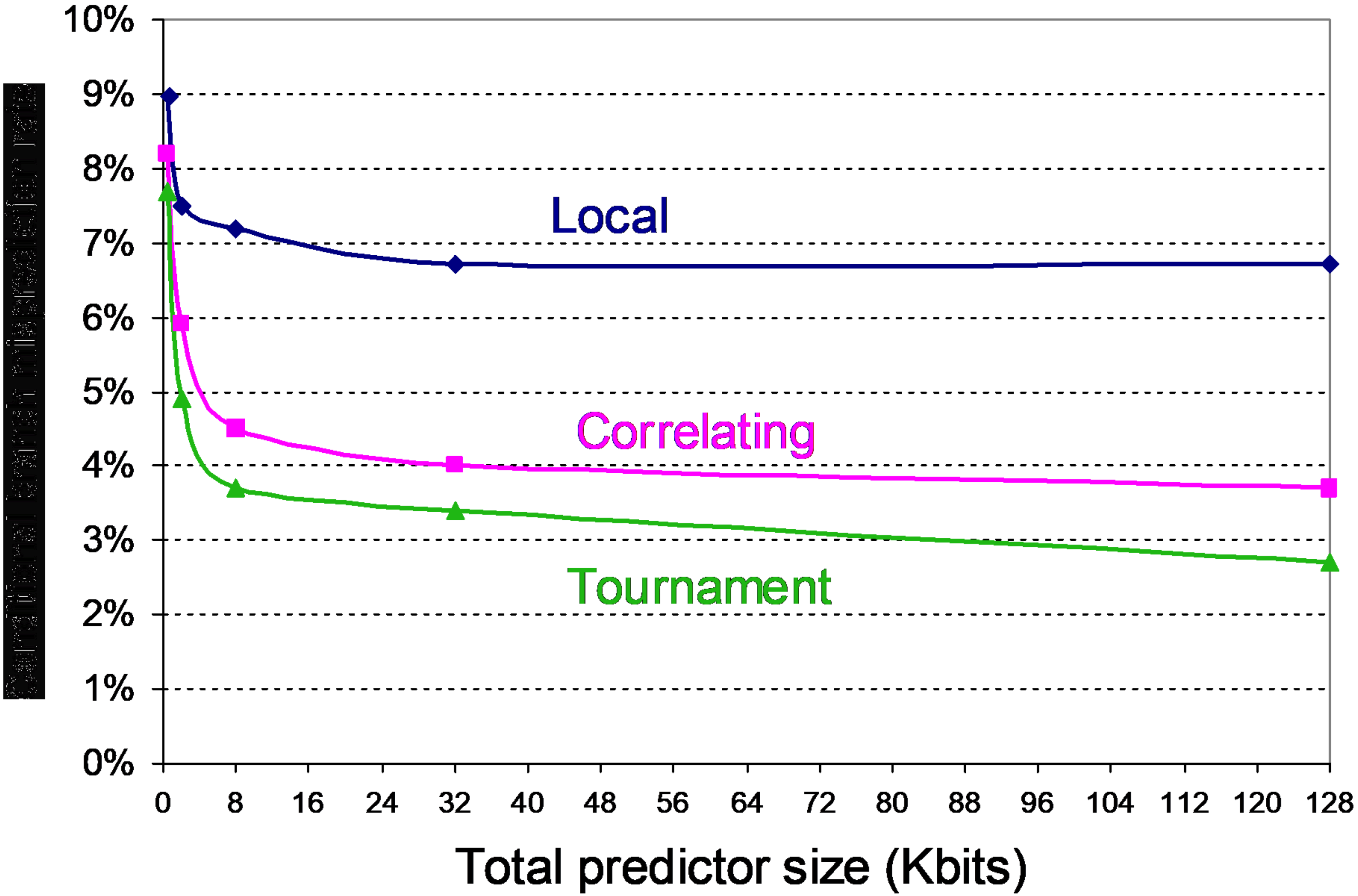


fig 3.40

- Profile: branch profile from last execution (static in that in encoded in instruction, but profile)

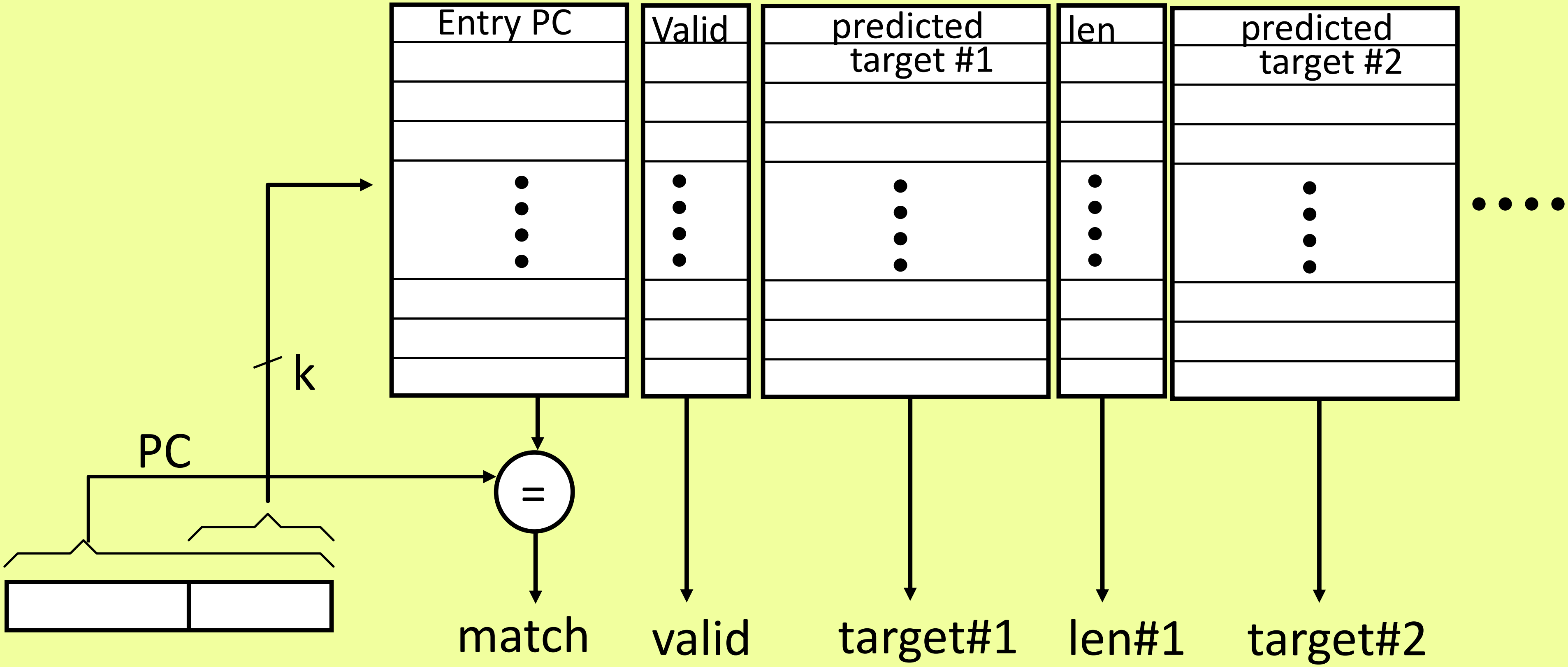
Accuracy v. Size (SPEC89)



Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions
- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance
- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

Branch Address Cache (Yeh, Marr, Patt)



Extend BTB to return multiple branch predictions per cycle

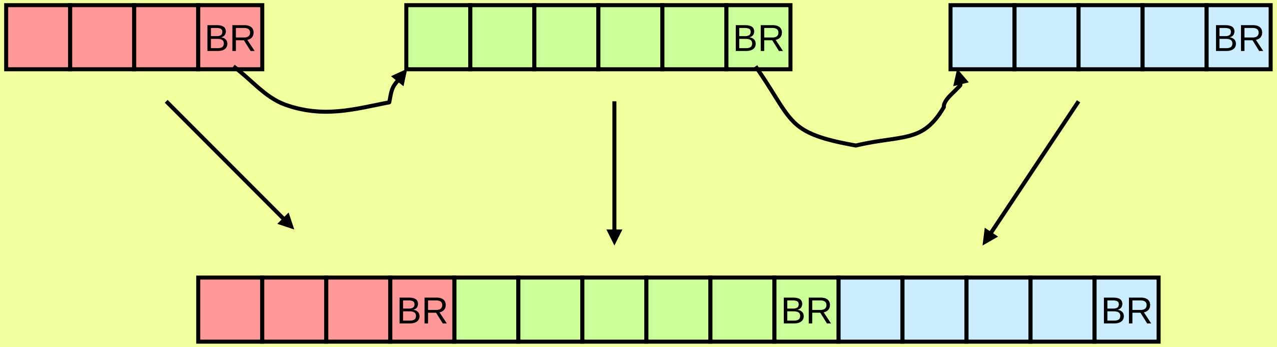
Fetching Multiple Basic Blocks

- Requires either
  - multiported cache: expensive
  - interleaving: bank conflicts will occur

- Merging multiple blocks to feed to decoders adds latency, increasing mispredict penalty and reducing branch throughput

Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line



- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next *n* branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

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)

Load-Store Queue Design

- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

Conservative O-o-O Load Execution

```
st r1, (r2)
ld r3, (r4)
```

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and r4 != r2
- Each load address compared with addresses of all previous uncommitted stores *(can use partial conservative check i.e., bottom 12 bits of address)*
- Don’t execute load if any previous store address not known

(MIPS R10K, 16 entry address queue)

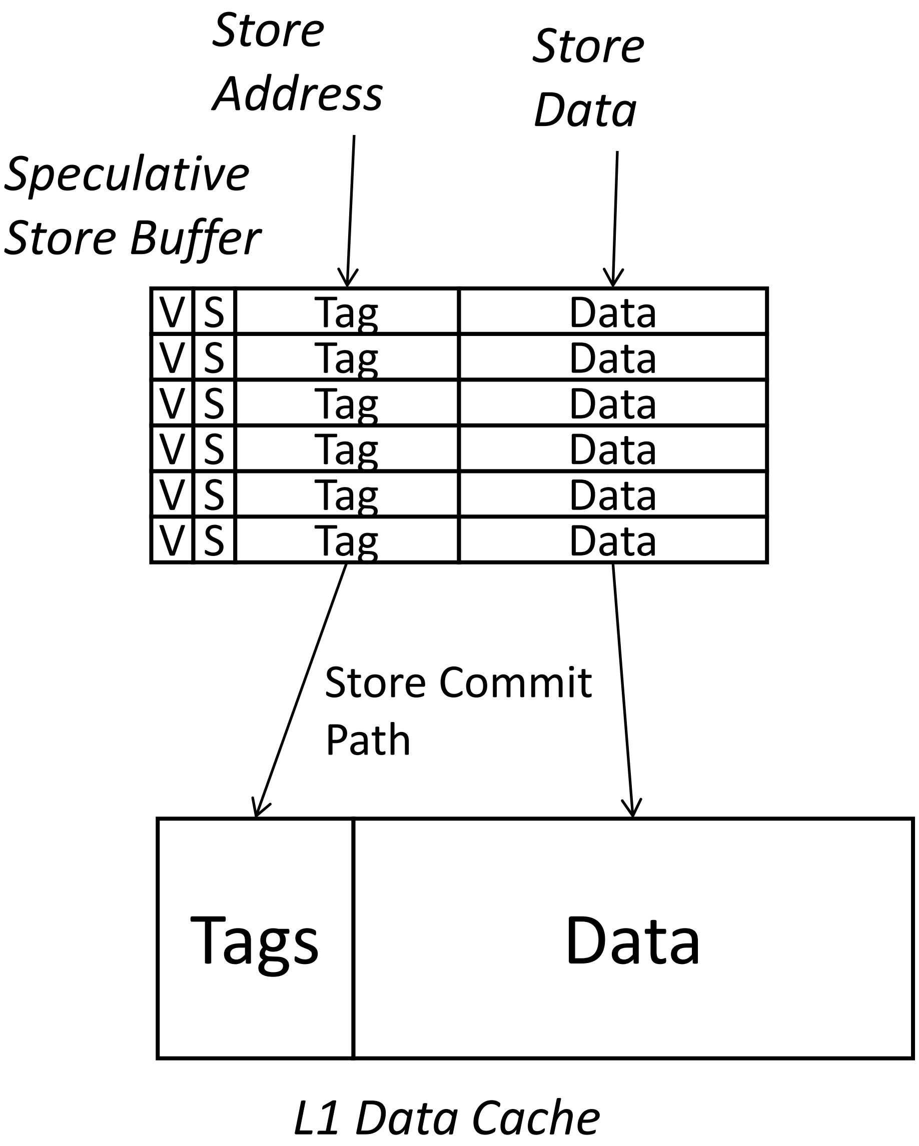
Premise: Past indicates Future

- Basic Premise is that past dependencies indicate future dependencies
  - Not always true! Hopefully true most of time
- Store Set: Set of store insts that affect given load
  - Example:

Addr	Inst
0	Store C
4	Store A
8	Store B
12	Store C
28	Load B ⇒ Store set { PC 8 }
32	Load D ⇒ Store set { (null) }
36	Load C ⇒ Store set { PC 0, PC 12 }
40	Load B ⇒ Store set { PC 8 }

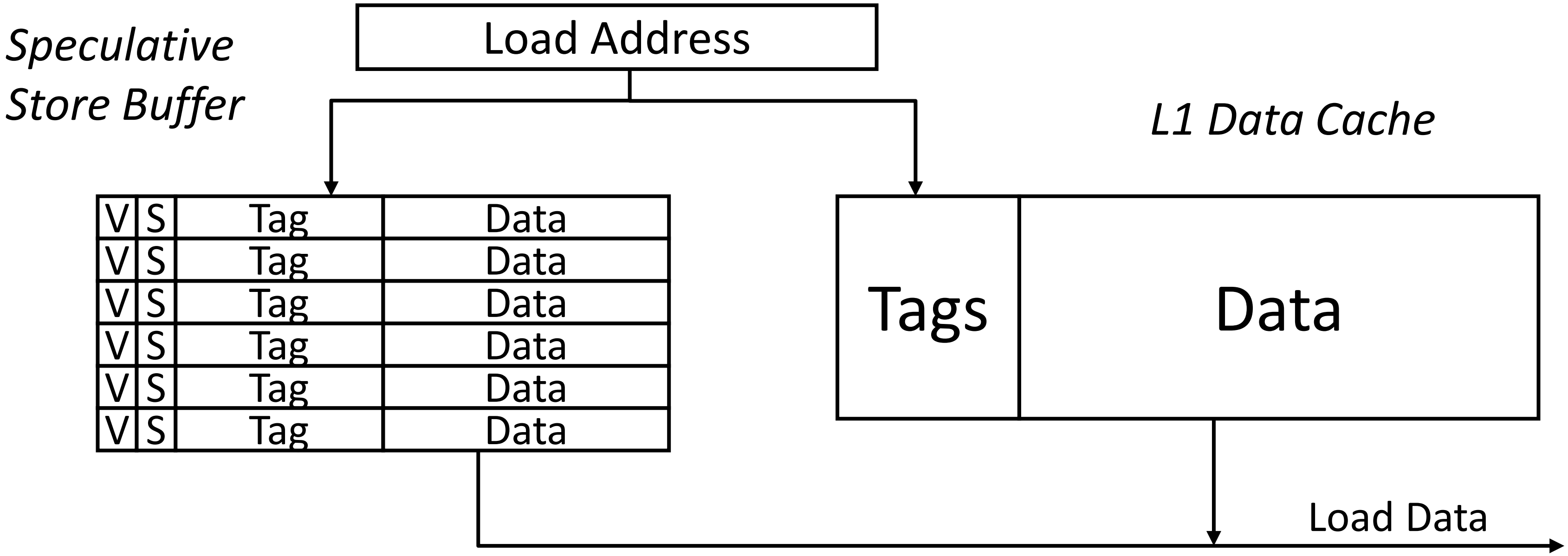
  - Idea: Store set for load starts empty. If ever load go forward and this causes a violation, add offending store to load's store set
- Approach: For each indeterminate load:
  - If Store from Store set is in pipeline, stall
    Else let go forward
- Does this work?

Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execution writes tag
- “Store data” execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



Speculative Store Buffer

Load Address

L1 Data Cache

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Tags

Data

Load Data

- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
 ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
 ld x3, (x4)
```

- Can execute load before store, if addresses known and **`x4`** != **`x2`**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **x4==x2**, squash load and all following instructions
- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
 ld x3, (x4)
```

- Guess that **`x4`** != **`x2`** and execute load before store

- If later find **`x4==x2`**, squash load and all following instructions, but mark load instruction as store-wait

- Subsequent executions of the same load instruction will wait for all previous stores to complete

- Periodically clear store-wait bits

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)