332
Advanced Computer Architecture
Chapter 4

Part 1: Branch *Direction* Prediction

October 2023

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4-6th eds), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Branch Prediction

1. Control hazards are a problem in any pipelined processor
2. Branches occur a lot (ca. one in five?)
   - Branches will arrive up to *n* times faster in an *n*-issue processor
3. Amdahl’s Law:
   - relative impact of the control stalls will be larger with the lower potential CPI in an *n*-issue processor
4. Speculative dynamic instruction scheduling with register renaming enables us to speculate *many* instructions
   - Forwarding from one speculatively-executed instruction to the next

Branch prediction is *really* important….

Branch Prediction - alternatives

- We have seen how a dynamically-scheduled processor can handle speculative execution past conditional branches, virtual calls, page faults etc
- But branch mis-predictions are expensive
- This naturally leads us to consider branch prediction schemes

- But first: there are alternatives…
  – With enough threads per core…
  – By extending the instruction set with predication
  – By extending the instruction set with branch delays

With enough threads per core…

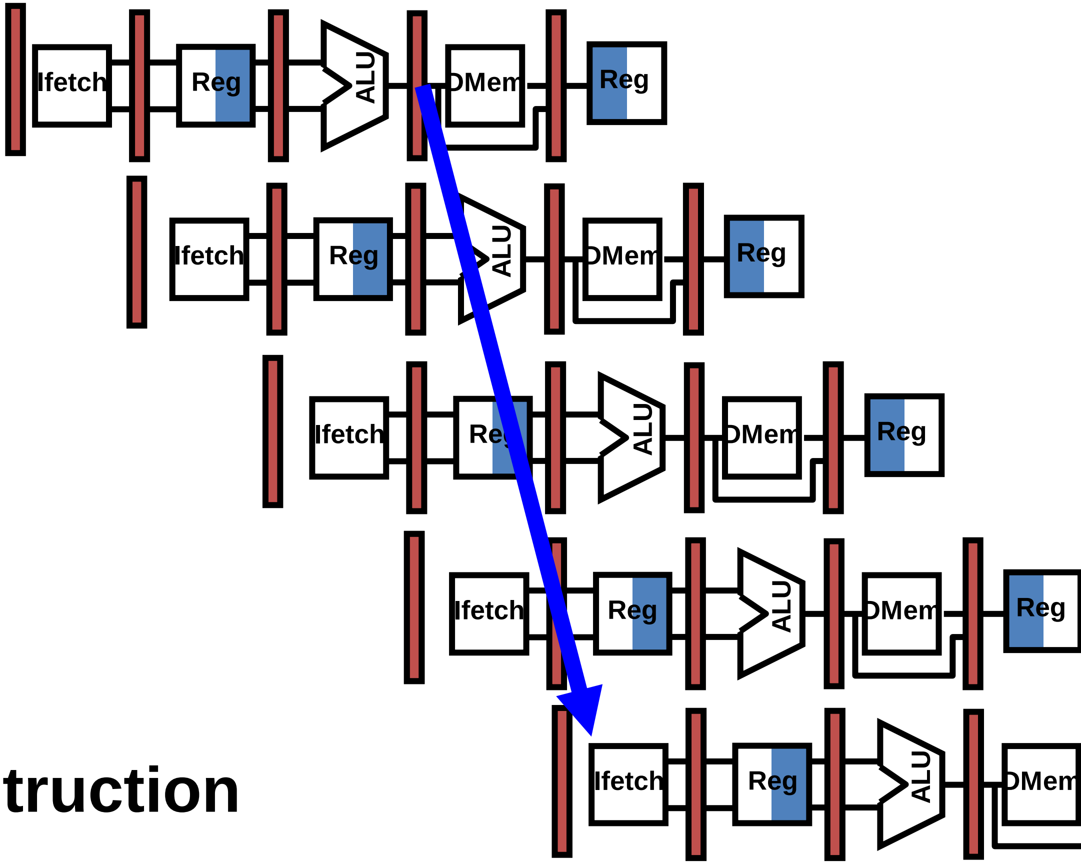
Thread0: beq…

Thread1: …

Thread2: …

Thread3: …

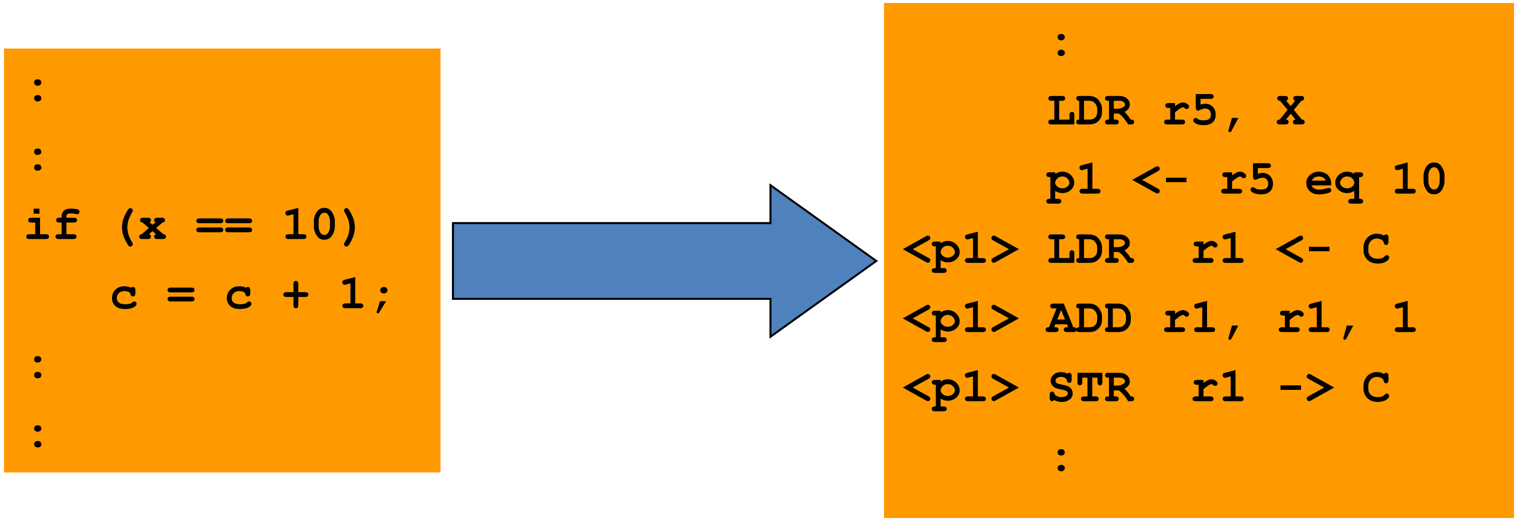
Thread0: next thread0 instruction



- In this example we have four threads per core
- Four PCs
- Four sets of registers
- And plenty of time to determine branch outcome without prediction

Predicated Execution (predic*a*ted…)

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



```
:
:
if (x == 10)
    c = c + 1;
:
:
```

```
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
```

Some instruction sets allow predication of almost any instruction

- Load condition value into a predicate register
- Each instruction specifies which predicate register it depends on
- If predicate is false, no exception or effect occurs
- Compiler can schedule instructions from different conditional branches to fill stalls

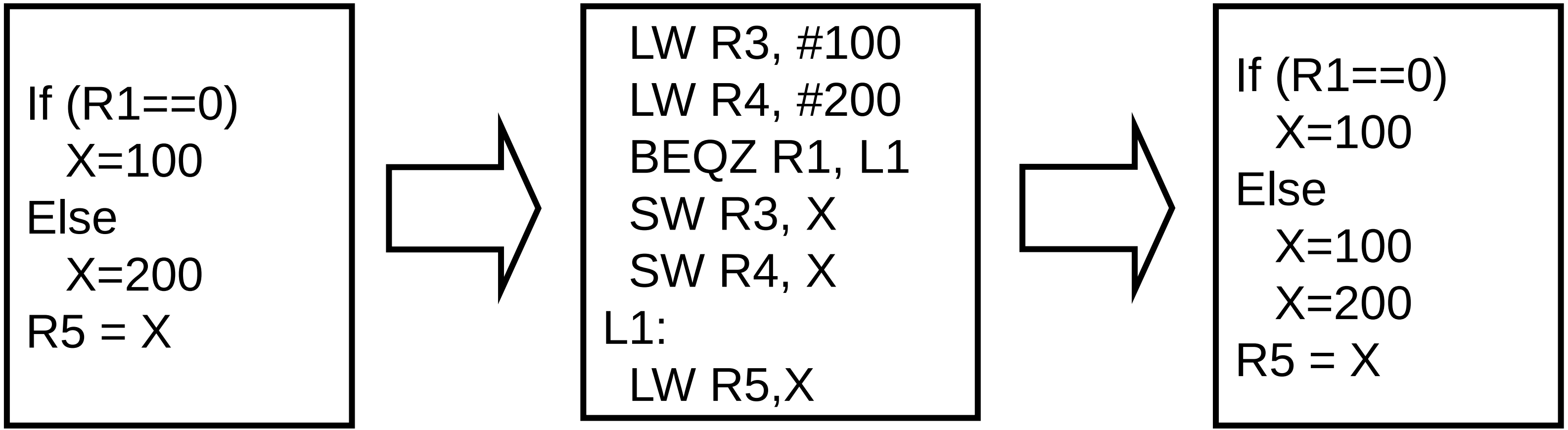
(Some instruction sets offer only partial support, eg predicated moves/stores, eg Alpha, MIPS, PowerPC, SPARC) (we will revisit this with Itanium & in GPUs)

When is this better than a conditional branch instruction?

Delayed Branch

- ***Define*** branch to take place AFTER a following instruction
- After all we have already fetched the next instruction

- A delay of just one instruction allows proper decision and branch target address in 5 stage pipeline
  - MIPS uses this; eg in



```
If (R1==0)
   X=100
Else
   X=200
R5 = X
```

Source code

```
   LW R3, #100
   LW R4, #200
   BEQZ R1, L1
   SW R3, X
   SW R4, X
L1:
   LW R5,X
```

Assembly code

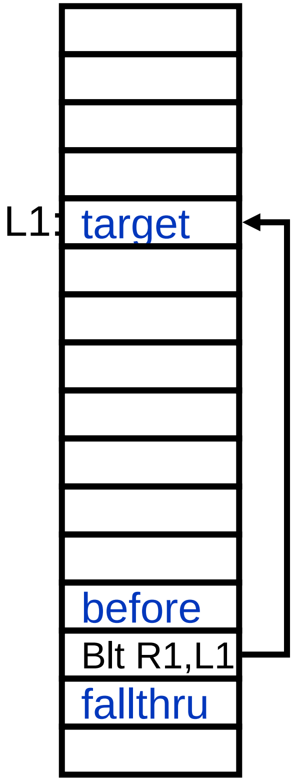
```
If (R1==0)
   X=100
Else
   X=100
   X=200
R5 = X
```

What it does

  - "SW R3, X" instruction is executed regardless
  - "SW R4, X" instruction is executed only if R1 is non-zero

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken



- **Compiler effectiveness for single branch delay slot:**
  - **Fills about 60% of branch delay slots**
  - **About 80% of instructions executed in branch delay slots useful in computation**
  - **About 50% (60% x 80%) of slots usefully filled**

- **"Canceling" branches: increase utilization of delay slot**
  - **Branch delay slot instruction is executed but write-back is disabled if it is not supposed to be executed**
  - **Two variants: branch "likely taken", branch "likely not-taken"**
  - **allows more slots to be filled**

- **Delayed Branch downside:**
  - **What if the pipeline is longer?**
  - **What if multiple instructions are issued per clock (superscalar)**

Branch Prediction - context

- If we have a branch predictor….
  - We want to fetch the correct (predicted) next instruction without any stalls
  - We need the prediction before the preceding instruction has been decoded
  - We need to predict conditional branches
    - Direction prediction
  - And indirect branches
    - Target prediction

Branch Prediction Schemes

Takenness:

- 1-bit Branch-Prediction Buffer
- 2-bit Branch-Prediction Buffer
- Correlating Branch Prediction Buffer
- Tournament Branch Predictor

Hennessy and Patterson 6th ed Appendix C p18-26

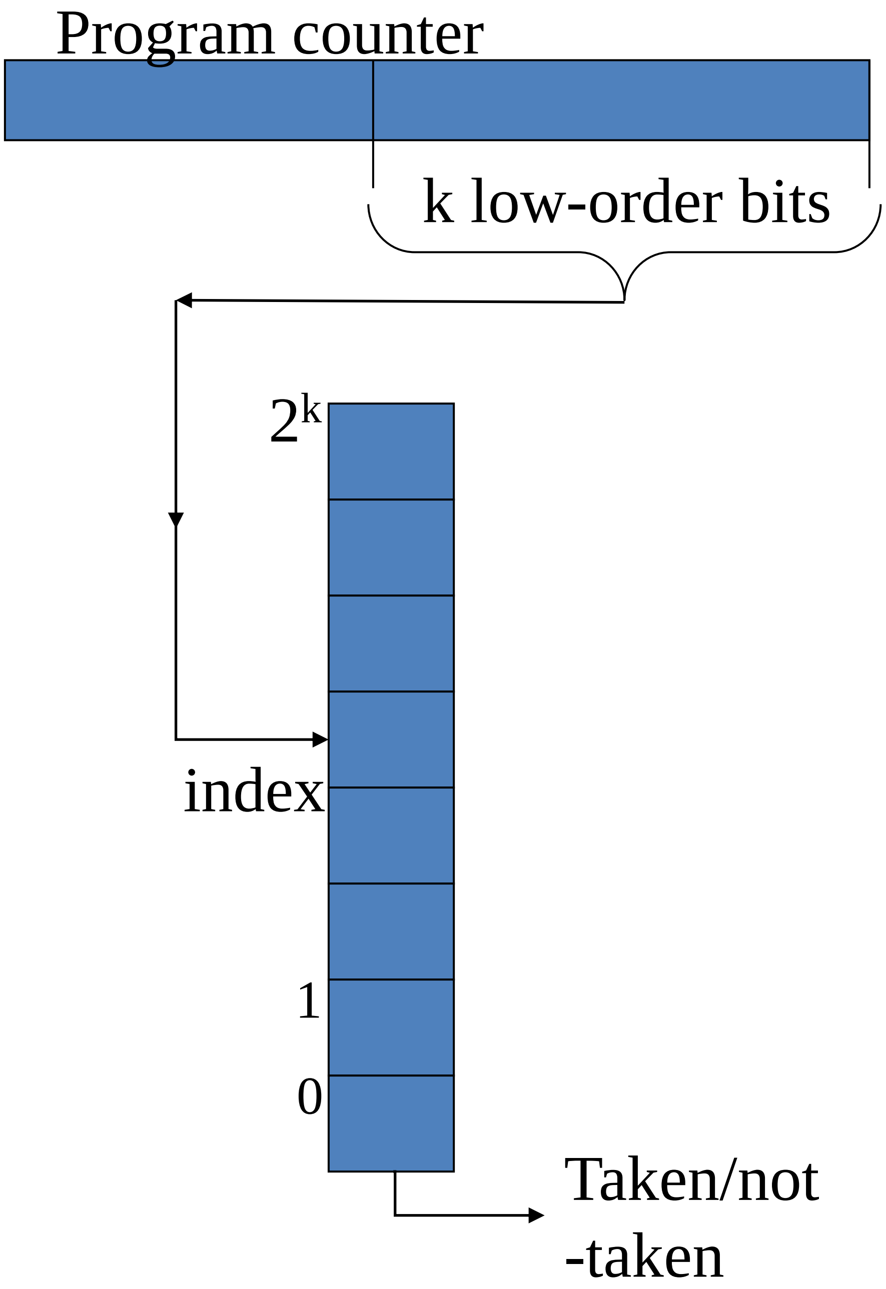
Hennessy and Patterson 6th ed p182-191

Target:

- Branch Target Buffer
- Return Address Predictors

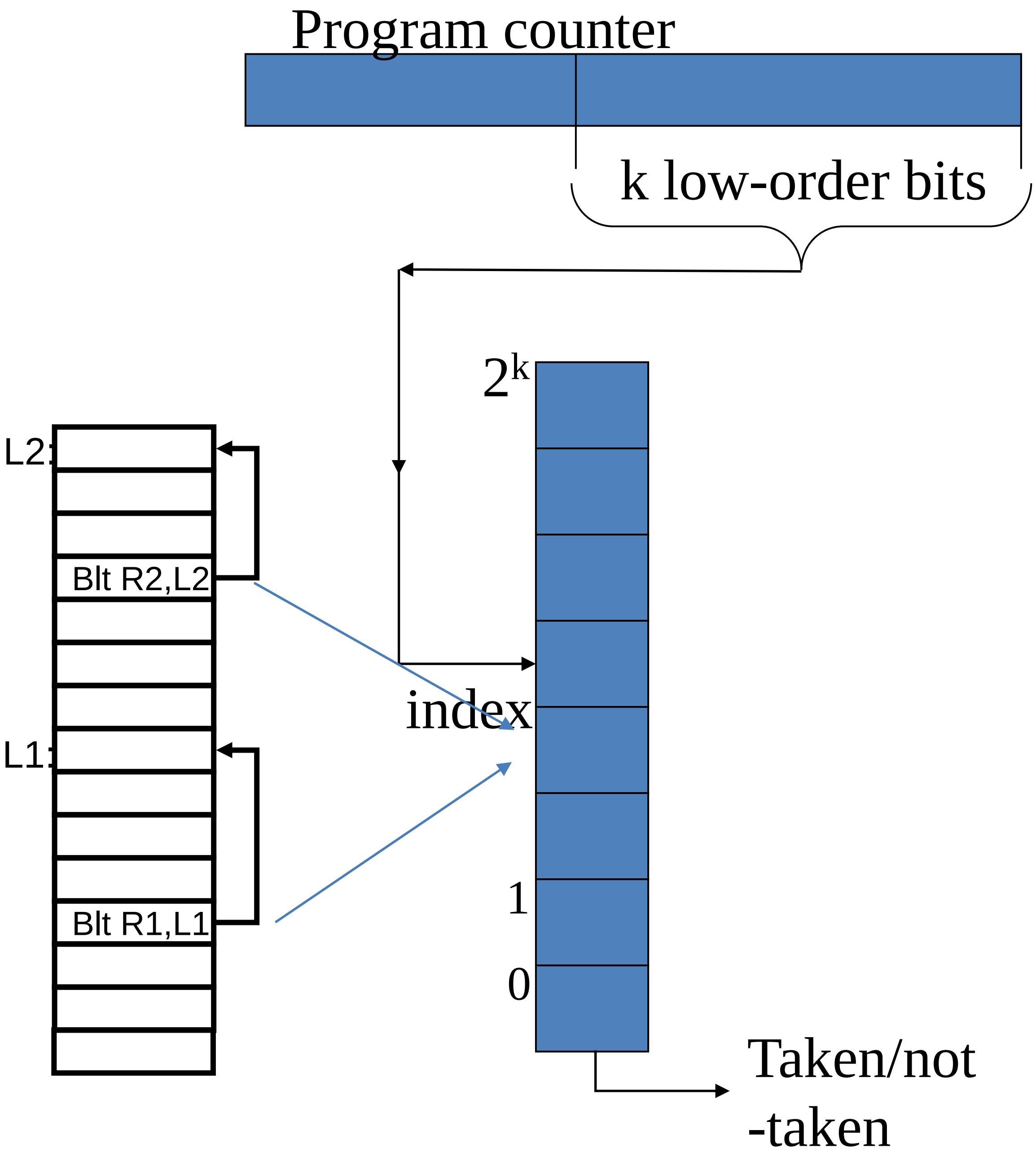
Simplest idea: branch history table (BHT)

- Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check



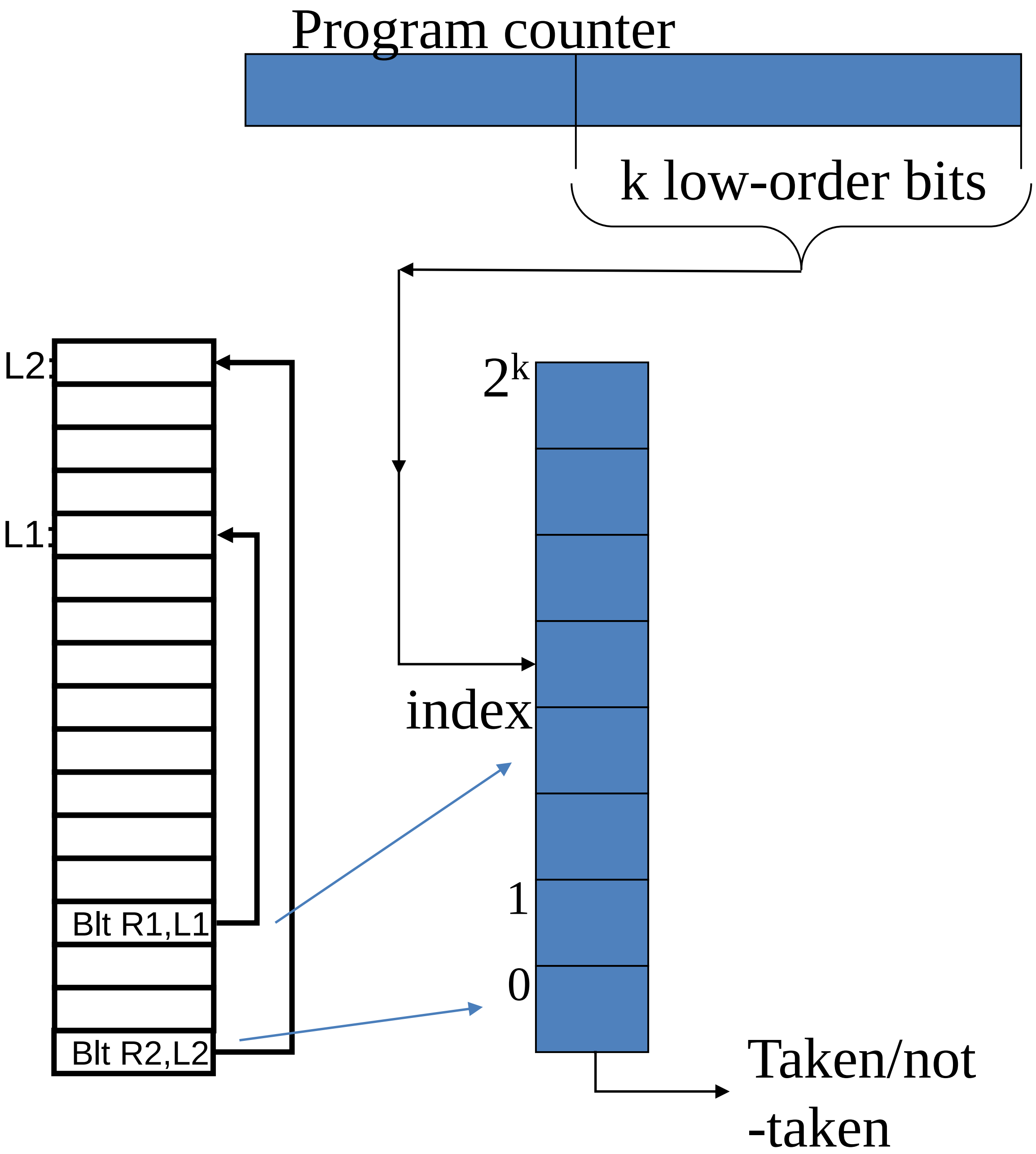
Simplest idea: branch history table (BHT)

- Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)
    - **Aliasing**: possible mispredictions if 2 different branch instructions map to the same BHT entry



Simplest idea: branch history table (BHT)

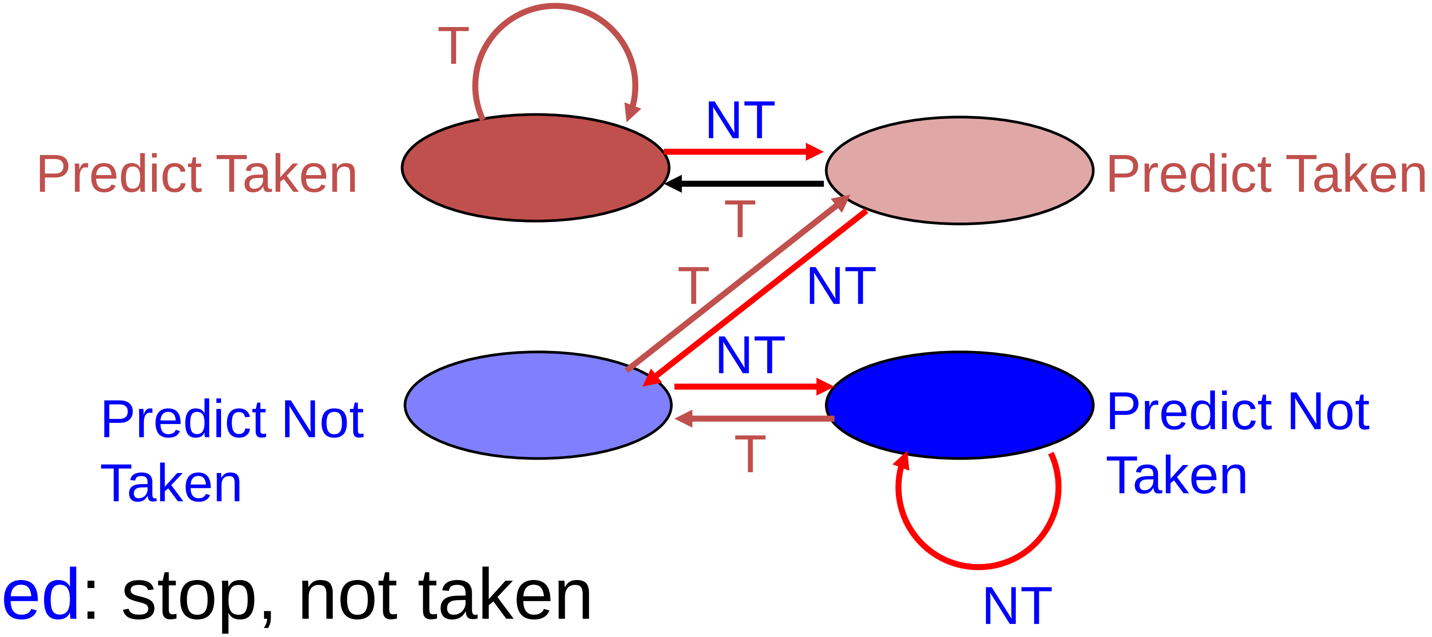
- ***Problem***: in a loop, 1-bit BHT will cause 2 mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Only 80% accuracy even if the loop's branch is taken 90% of the time



Dynamic Branch Prediction

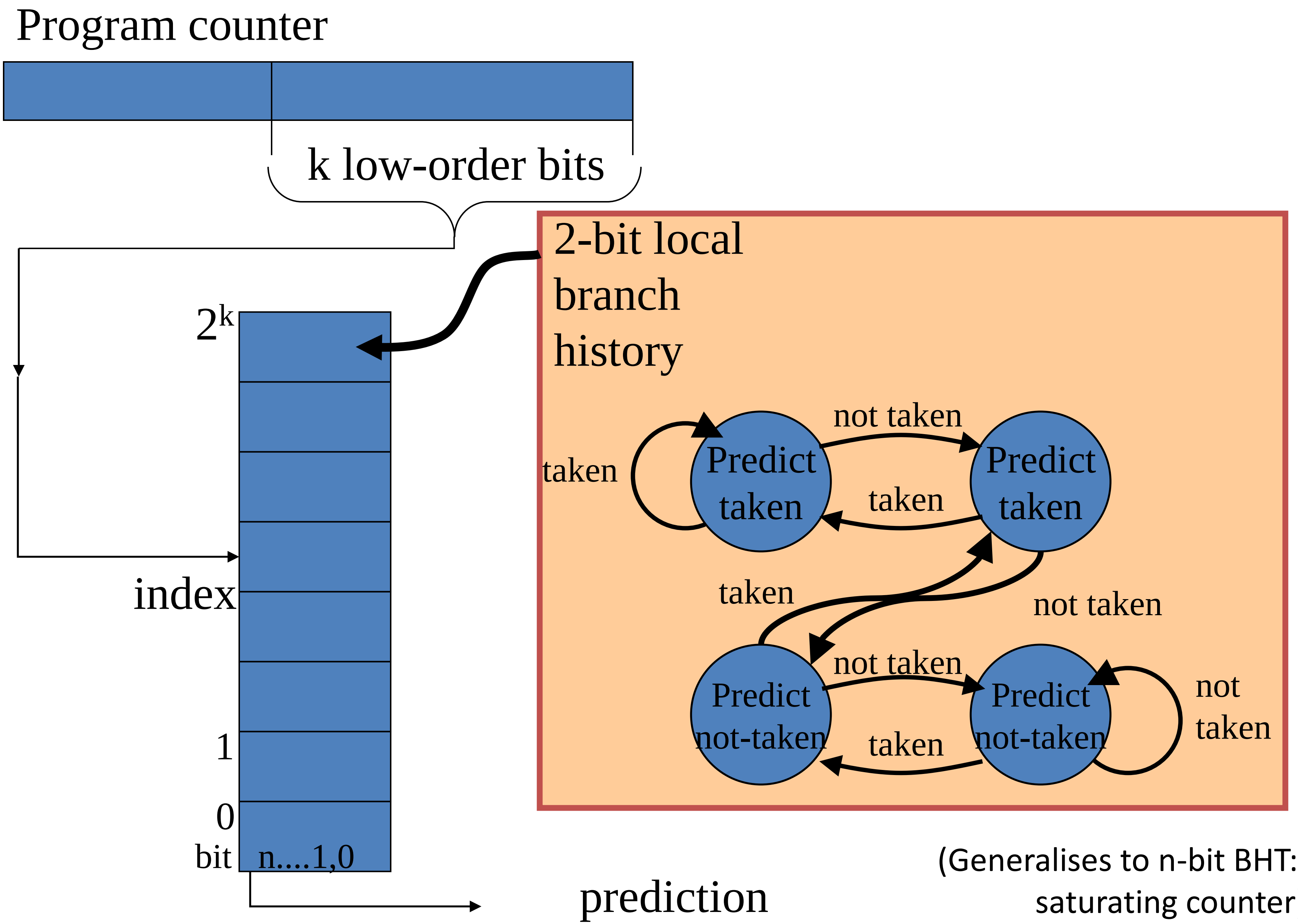
(Jim Smith, 1981)

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:* (Figure 3.7, p. 198)



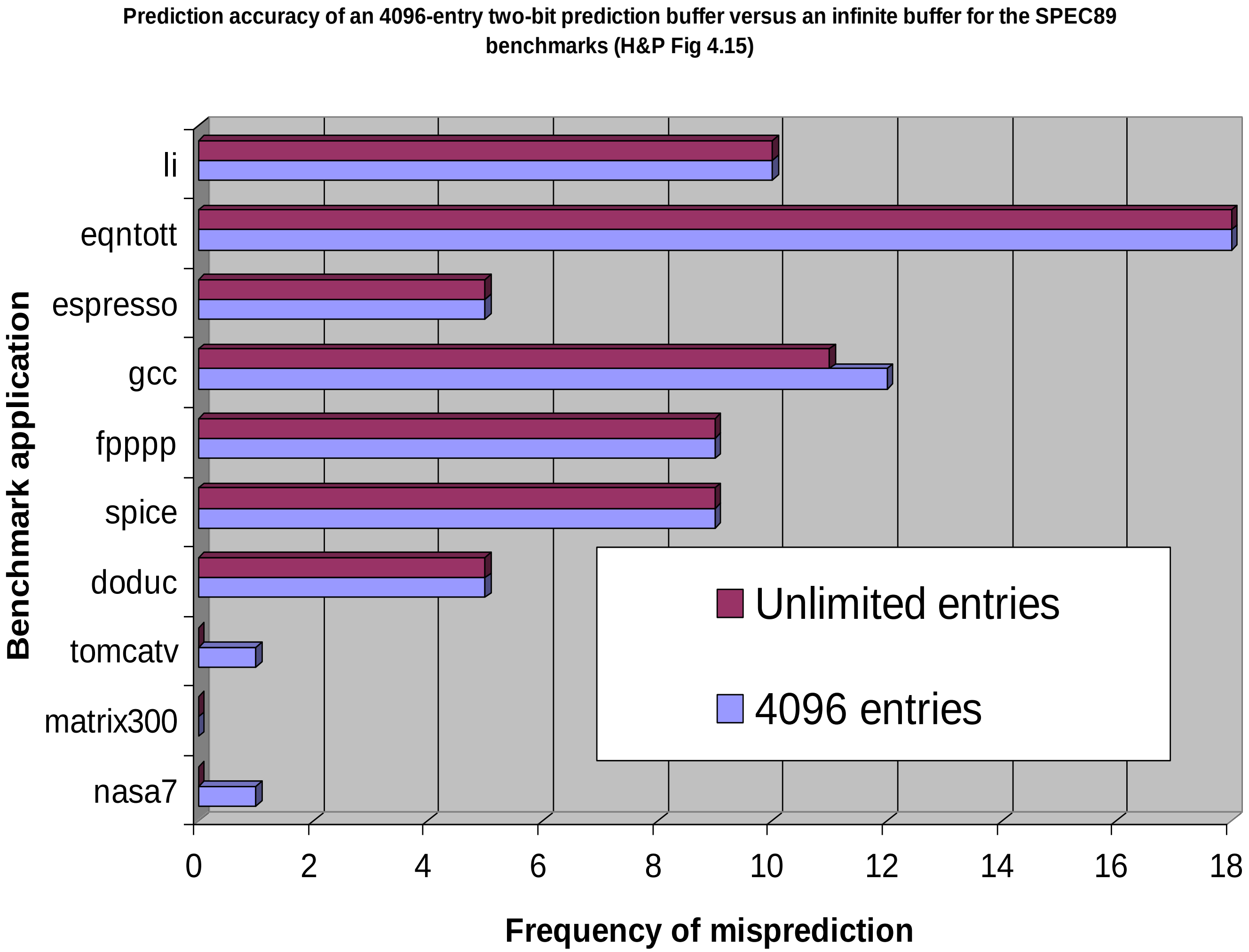
- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

The 2-bit branch history table (BHT)



(Generalises to n-bit BHT: saturating counter)

n-bit BHT - how well does it work?

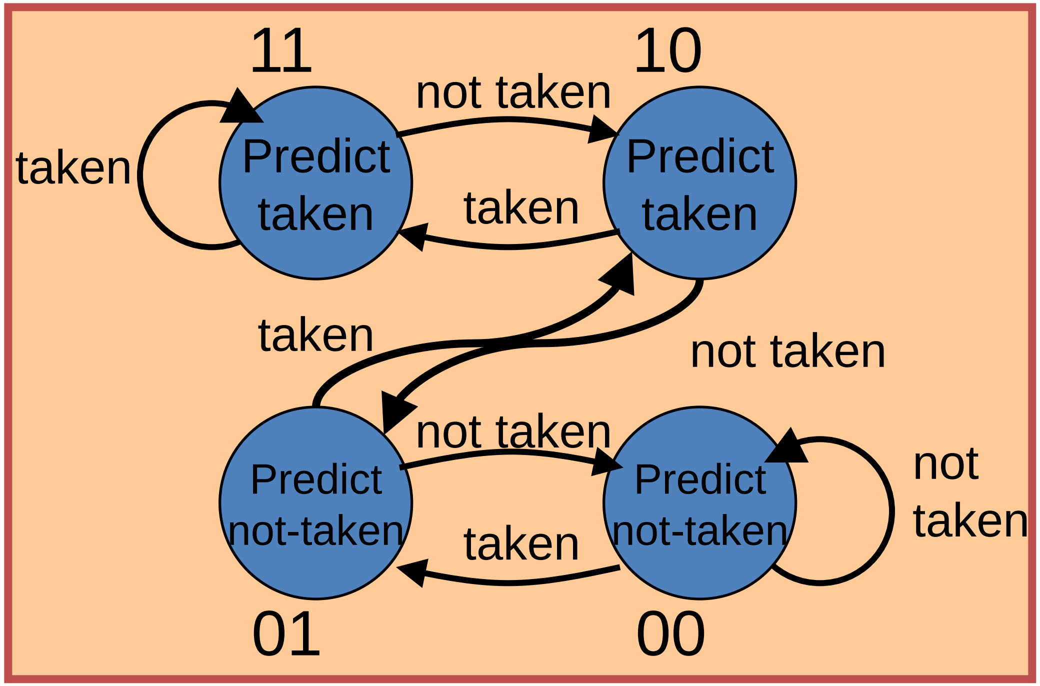


- 2-bit predictor often very good, sometimes awful
- Little evidence that BHT capacity is an issue
- 1-bit is usually worse, 3-bit is not usefully better

N-bit BHT - why does it work so well?

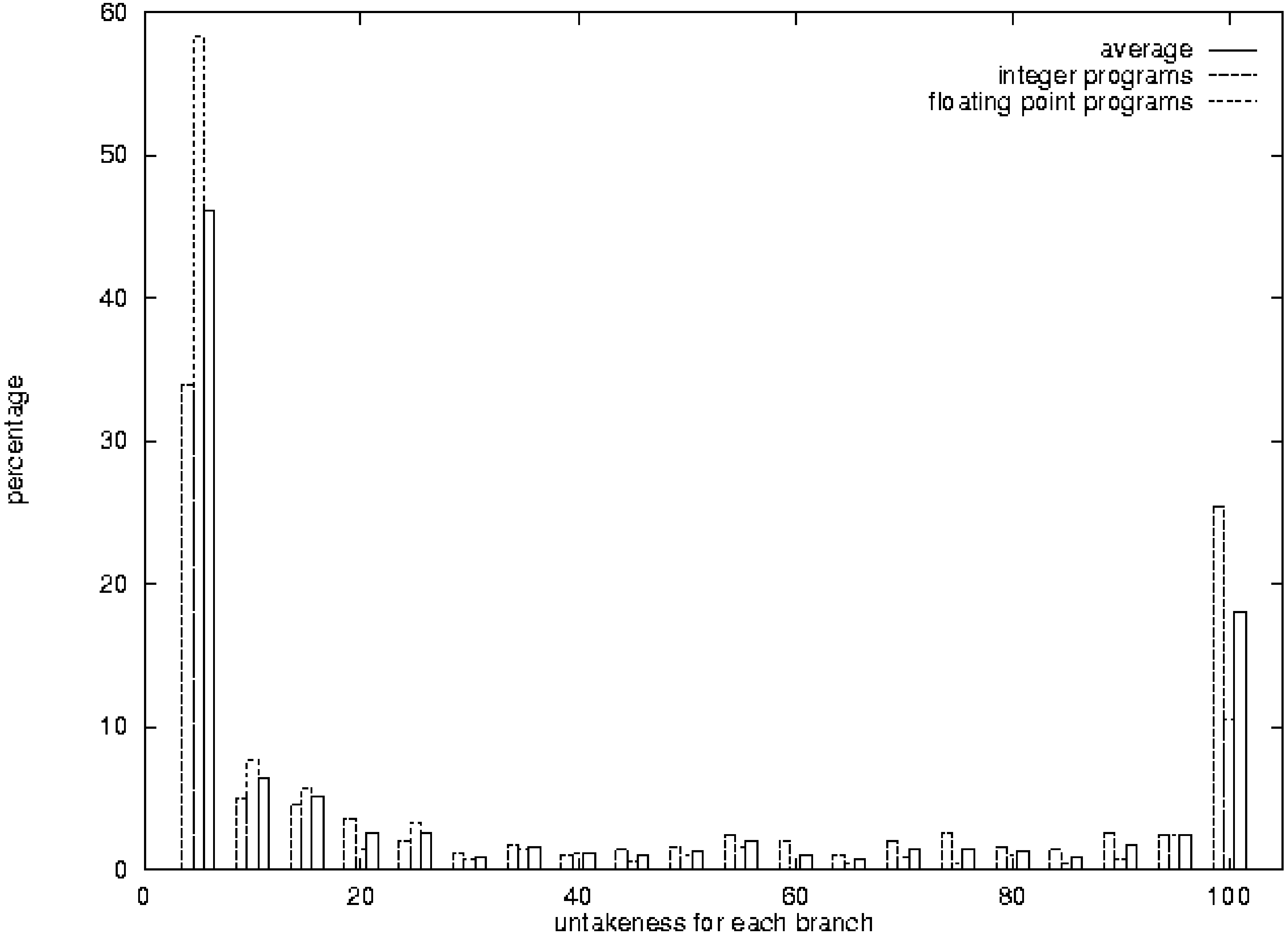
- n-bit BHT predictor essentially based on a saturating counter: taken increments, not-taken decrements
- predict taken if most significant bit is set

- Most branches are highly *biased:* either almost-always taken, or almost-always not-taken
- Works badly for branches which aren't



Often called the "bimodal" predictor

Bias



Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

Is local history all there is to it?

- The bimodal predictor uses the BHT to record "local history" - the prediction information used to predict a particular branch is determined only by its memory address
- Consider the following sequence:

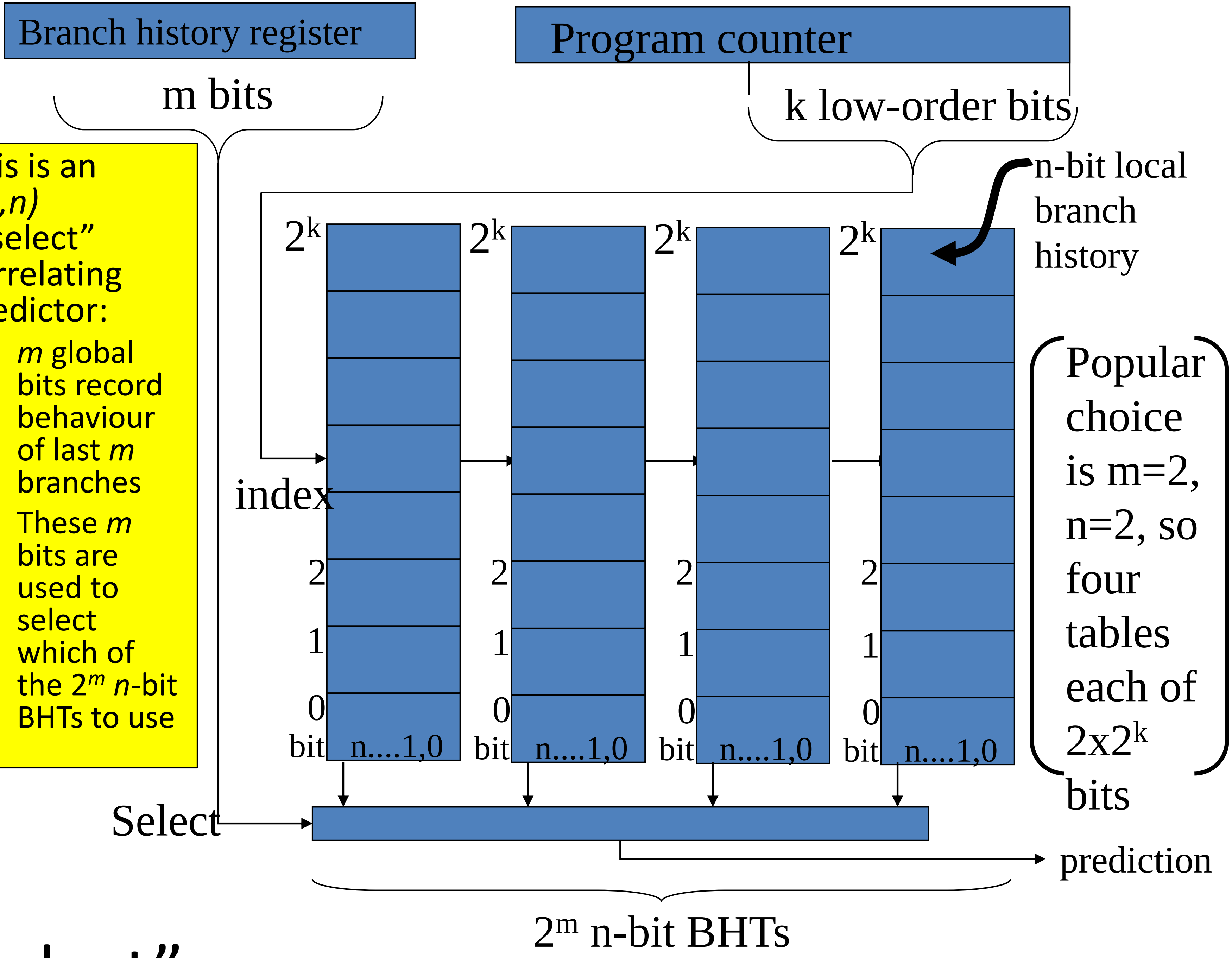
```
if (C1)  then
    S1;
endif
if (C2) then
    S2;
endif
if (C3) then
    S3;
endif
```

  - **It is very likely that condition C2 is correlated with C1 - and that C3 is correlated with C1 and C2**
  - **How can we use this observation?**

Global history

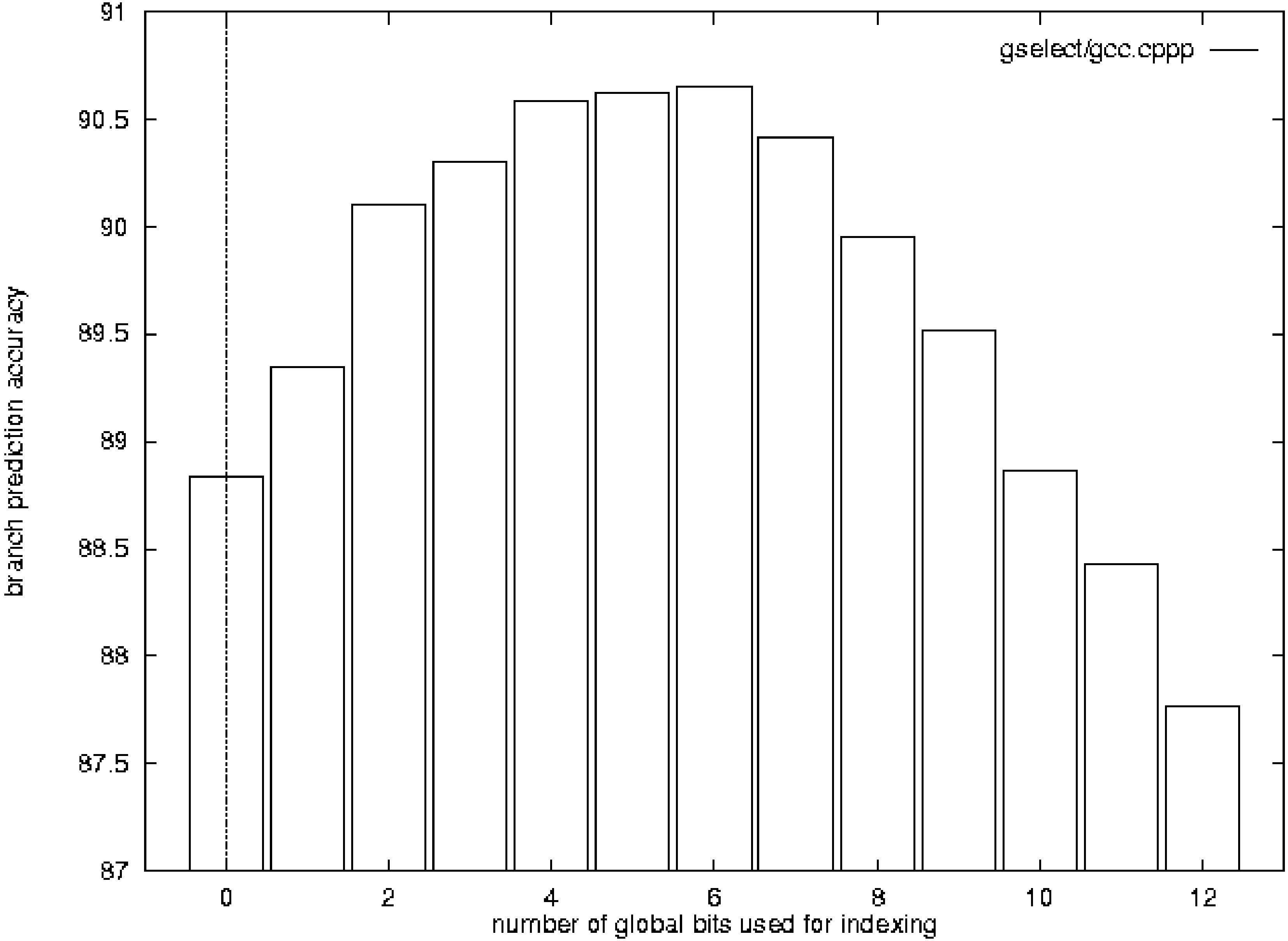
- Definition: Global history. The taken - not-taken history for all previously-executed branches.
  - **Idea: use global history to improve branch prediction**
- Compromise: use *m* most recently-executed branches
  - Implementation**: keep an *m*-bit Branch History Register (BHR) - a shift register recording taken - not-taken direction of the last m branches**
- Question: How to combine local information with global information?

- This is an *(m,n)* "gselect" correlating predictor:
  - *m* global bits record behaviour of last *m* branches
  - These *m* bits are used to select which of the 2^m *n*-bit BHTs to use



"Gselect"

How many bits of branch history should be used?



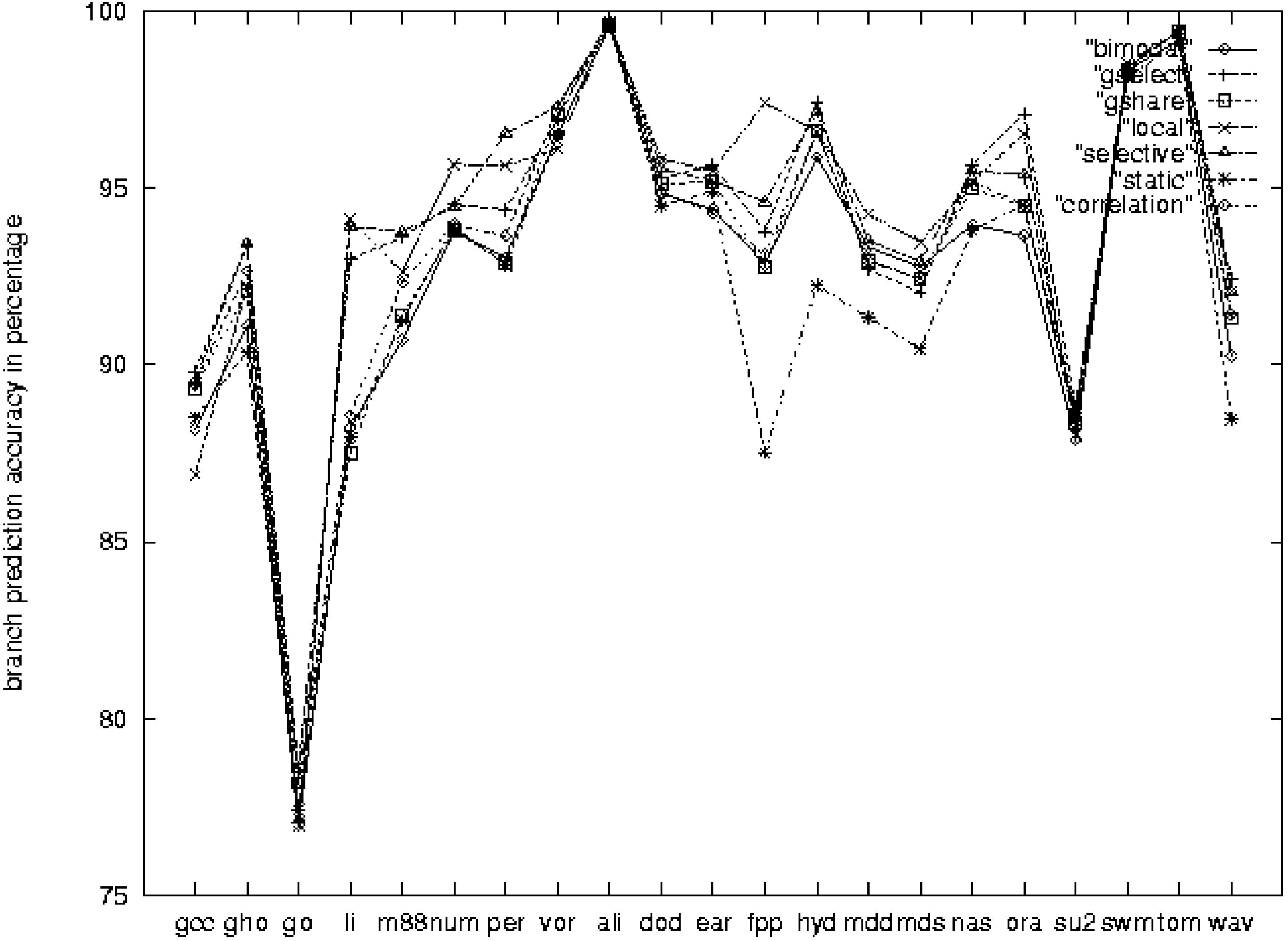
- (2,2) is good, (4,2) is better, (10,2) is worse

Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

Variations

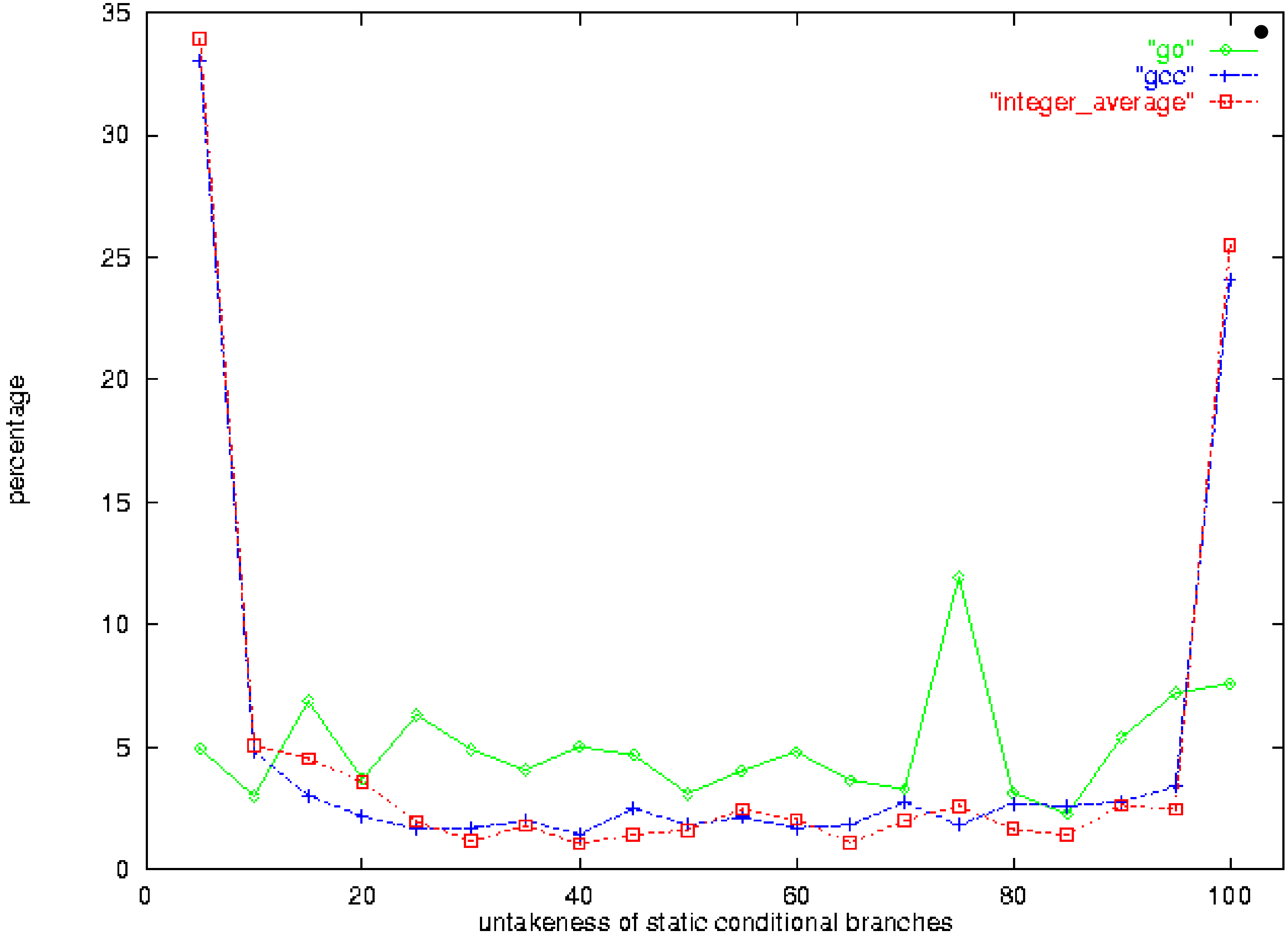
- There are many variations on the idea:
  - ***gselect***: many combinations of *n* and *m*
  - ***global***: use *only* the global history to index the BHT - ignore the PC of the branch being predicted (an extreme (n,m) gselect scheme)
  - ***gshare***: arrange bimodal predictors in single BHT, but construct its index by XORing low-order PC address bits with global branch history shift register - claimed to reduce conflicts
  - ***Per-address Two-level Adaptive using Per-address pattern history* (PAp)**: for each branch, keep a *k*-bit shift register recording its history, and use this to index a BHT *for this branch* (see Yeh and Patt, 1992)
- Each suits some programs well but not all

Horses for courses



Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

Extreme example - “go”

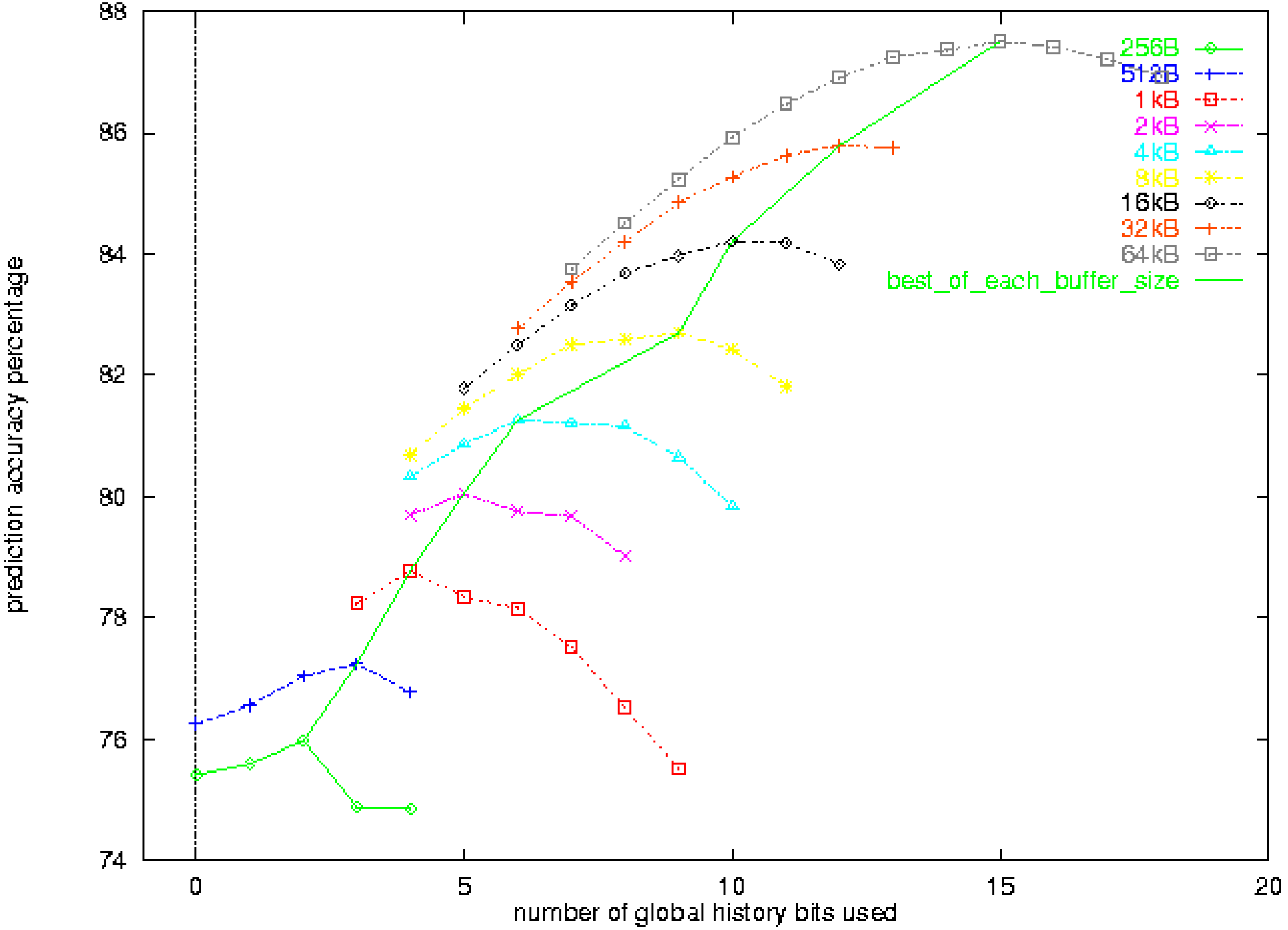


- “go” is a SPEC95 benchmark code with highly-dynamic, highly-correlated branch behaviour

- The bias of “go”s branches is more-or-less evenly spread between 0% taken and 100% taken
- All known predictors do badly

Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

Some dynamic applications have highly-correlated branches



- For “go”, optimum BHR size (m) is much larger

Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

Re-evaluating Correlation

- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

program	branch %	static	# = 90%
compress	14%	236	13
eqntott	25%	494	5
gcc	15%	9531	2020
mpeg	10%	5598	532
real gcc	13%	17361	3214

- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases?

Tournament Predictors

- Motivation for correlating branch predictors is that the 2-bit predictor failed on important branches; by adding global information, performance improved

- Tournament predictors: use 2 predictors,
  - one based on global information
  - the other based on local information
  - and combine with a selector
  - The selector is driven by a predictor….

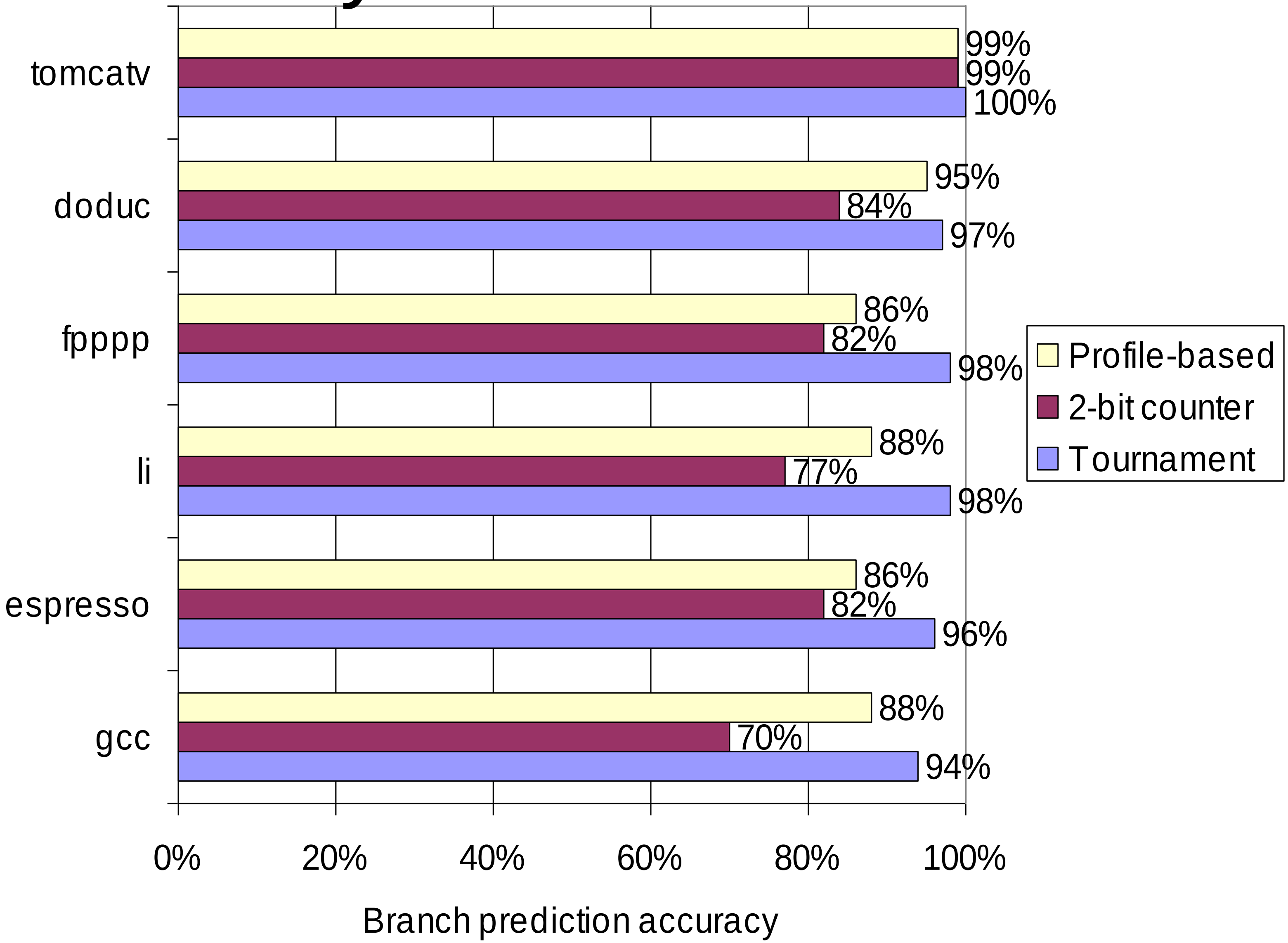
- Hopes to select the right predictor for the right branch

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!

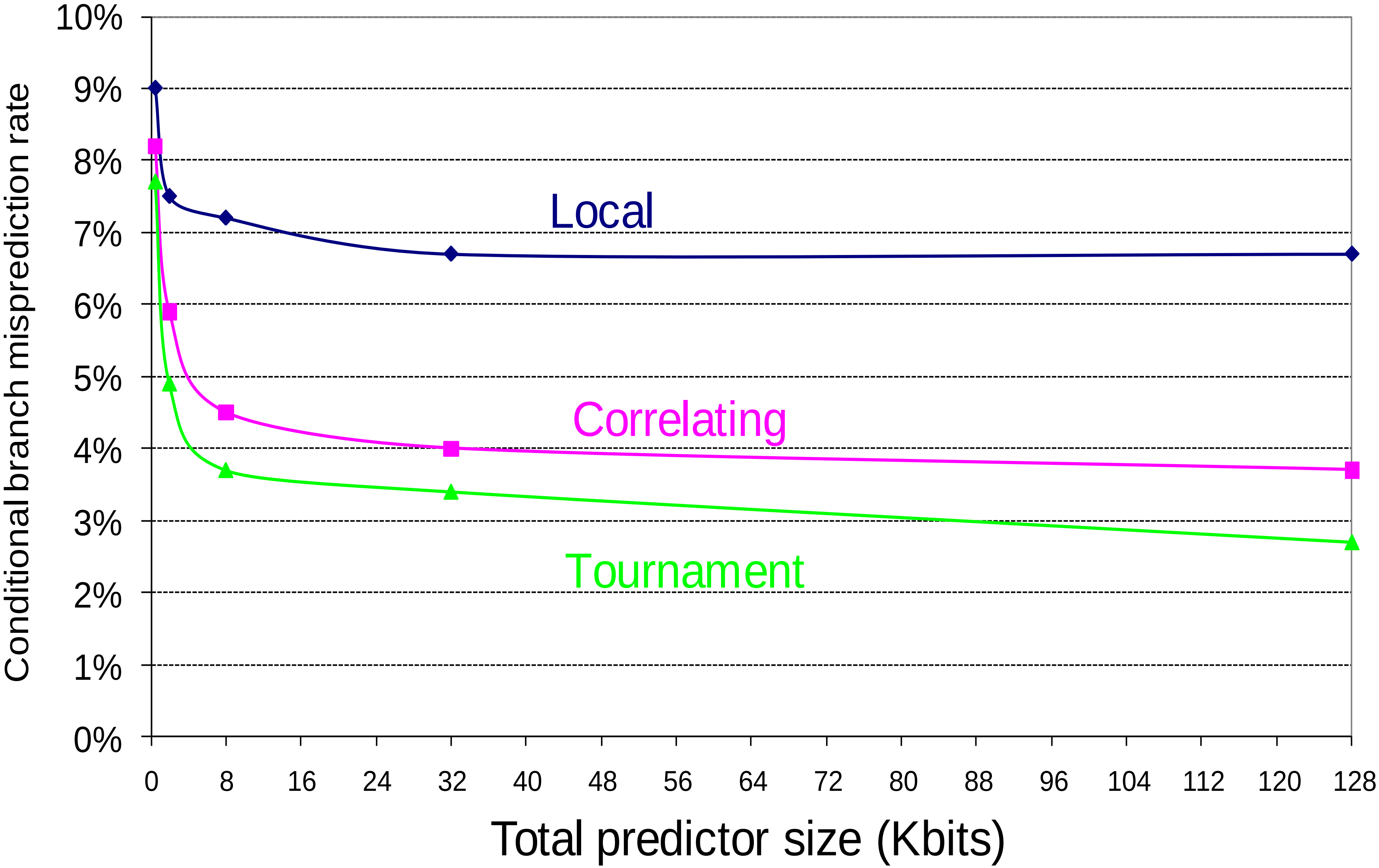
  (~180,000 transistors)

Accuracy of Branch Prediction



- Profile: branch profile from last execution
(static in that the prediction is in encoded in the instruction, but derived from the real execution profile)
- A good dynamic predictor can outperform profile-driven static prediction by a large margin

Accuracy v. Size (SPEC89)



Tournament is not just a better predictor; it delivers a better prediction with fewer transistors
It's another example of combining two different optimisations, each good for different situations

Summary

- Prediction seems essential (?)
  - Fine-Grained Multi-Threaded (FGMT) processors can avoid control hazards
  - Predicated Execution can reduce number of branches, number of mispredicted branches
  - Delayed branches and cancelling branches can help, at least in simple pipelines
- Two questions: branch ***takenness***, branch ***target***

Takenness:

- Branch History Table: 2 bits for loop accuracy
  - Saturating counter (bimodal) scheme handles highly-biased branches well
  - Some applications have highly dynamic branches
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: try two or more competitive solutions and pick between them

Target:

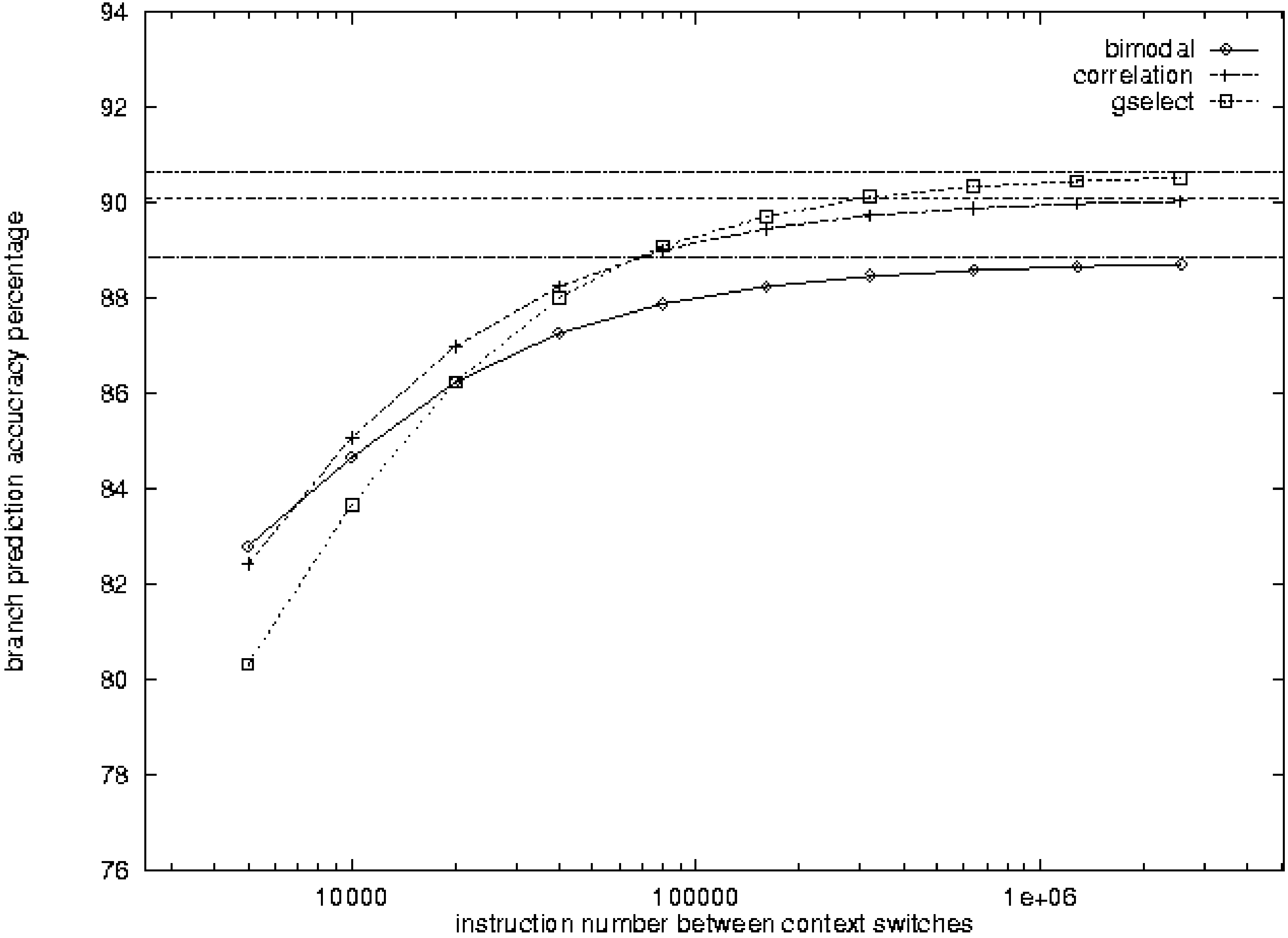
- Next time!

Appendix: slides not covered in video

Warm-up effects and context-switching

- In real life, applications are interrupted and some other program runs for a while (if only the OS)
- This means the branch prediction is regularly trashed
- Simple predictors re-learn fast
  - in 2-bit bimodal predictor, all executions of given branch update the same 2 bits
- Sophisticated predictors re-learn more slowly
  - for example, in (2,2) gselect predictor, prediction updates are spread across 4 BHTs
- *Selective* predictor may choose fast learner predictor until better predictor warms up

Warm-up...



- Best predictor takes 20,000 instructions to overtake bimodal

Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (http://www.cs.berkeley.edu/~zhendong/cs252/project.html)

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 a large commercial transaction processing (TP) workload!
  - 21264 avg. 17 mispredictions per 1000 instructions
  - 21164 avg. 15 mispredictions per 1000 instructions

- Why?
  - TP code is much larger than the benchmarks
  - the 21164 holds twice as many branch predictions based on local behavior (2K vs. the 21264's 1K local predictor)

Branch direction prediction: topics not covered

- Yeh and Patt's "Two-Level Adaptive Branch Predictor" (and Yeh/Patt classification GAg,GAp,Pap)
  - Tse-Yu Yeh, Yale N. Patt: **Alternative Implementations of Two-Level Adaptive Branch Prediction**. ISCA 1992: 124-134
- Seznec and Michaud's TAGE predictor
  - André Seznec. 2011. **A new case for the TAGE branch predictor**. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)
- Neural branch predictors eg
  - Daniel A. Jiménez and Calvin Lin. 2002. **Neural methods for dynamic branch prediction**. ACM Trans. Comput. Syst. 20, 4 (November 2002), 369–397.

Hennessy and Patterson 6th ed p188

Student question

- Hello I was wondering if - in the o-o-o pipeline with an RUU - the way predication works is that we have the instructions that are predicated on a particular predicate register (i.e. those that will execute only if their predicate condition is true) depend on the predicate register in the RUU in the same way that an instruction depends on its operands.
- Once the required predicate register value becomes available (either from the register file or an FU), the instruction is either trashed from the RUU or made eligible for dispatch (assuming its other dependencies are resolved).
- One advantage is that we do not use the FU's needlessly as we would with a branch misprediction. Also, unlike on a branch misprediction, only a few entries in the RUU are flushed (those whose predicate condition is false) as opposed to the whole RUU. To guarantee that only a few entries are flushed, we must only use predication for a small number of instructions.
- Is all the above correct? Many thanks!

- This all makes complete sense.
- Of course you **might** try to execute predicated instructions speculatively - you could start them off, and then decide whether to commit the result at commit time when the condition is known.
- The trouble with that is that if you guessed wrong, you will have to flush as it's possible the register result of the predicated instruction might have been forwarded to another instruction, erroneously.
- There is a menu of techniques that might fix this. For example, see
- Predicate Prediction for Efficient Out-of-order Execution paper.dvi (psu.edu)
- There is a subtlety (explained in the paper above) [and I think it applies to the scheme you propose] that predicated register writes create ambiguity in dependence:
- 1: r1 <- a
- 2: r2 <- b
- 3: (p1) r2 <- r1
- 4: r4 <- r2
- Should instruction 4 be dispatched when instruction 2 writes-back, or should it wait for instruction 3? But we removed instruction 3 from the RUU!
- (one might comment that conditional branches create ambiguity in dependence.... it's almost as if we are translating predication into control dependence on the fly).
- Paul

Student question

- In lecture 3.1, at the end we look at tournament predictors to choose which branch prediction method to use. How do we know that the predictor used for the tournament predictor is best?
- I assume the overhead is too much but disregarding that, can't the same reasoning for using a tournament predictor be applied to the tournament predictor itself? Hence, is there any data for using a tournament predictor to choose between which tournament predictor to use?

- I think the logic of using a local predictor as the tournament "selector" is that for each branch, we predict whether it's best to use local history or global history.
- But you might say "it depends" - the answer to this question might depend on the context in which this branch is executed.
- A good example of "context" is the function's caller.
- Consider this example:

```
void f(int i) {
  if (i &1) S1 else S1;}
void g() {
  x = rand();
  f(x);}
void h() {
  x = rand();
  if (x&1)
     f(x)  else
   S3}
```

- In this example, the condition in f is highly correlated with the outcome of the condition in g (ie it's the same).
- So when f is called from g the global history is helpful, while when it's called from h the global history is useless.
- However in this example, the condition is hard to predict anyway - so you might as well use history always. Your scheme would only help us if the local prediction were actually useful.
- So I think this example shows that an advantage is possible but depends on some pretty complicated circumstances - so its advantage would be rather thin?
- Another thing I think this example reinforces is that using global history is particularly good for repeated or redundant tests - where it doesn't help avoiding the misprediction - it avoids mispredicting again.
- You might also wonder whether the taken/not-taken history of recent branches is the most useful way to distinguish the relevant contexts?

Student question

What types of programmes are the different variations of global history tables suited for?

Out of the 4 variations (gselect, global, gshare, pAp), would you mind describing a feature of a body of code which would make it most suited for each of these variations?

After having a read of this article (https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf) which was very useful btw, I recommend you check it out, I see that each is a continuation of the idea that global information really doesn't contain a lot of valuable information, so I suppose a followup to this is why even bother using global information?

I think they offer a variety of compromises between

(1) covering as many different branches in the program as possible

(2) exploiting "global" context (ie providing different predictions for the same branch in different contexts)

(3) learning quickly

(4) handling periodic branches with a useful range of different periods (the simplest periodic branch is a while-loop exit branch - perhaps the loop is usually executed N times. Can the predictor provide a perfect prediction? For N=2, N=3?

There are also some subtleties I think with branch predictor aliasing: suppose two different frequently-executed branches happen to have the same low-order address bits, so they map to the same BHT entry in the local half of a tournament predictor. It might be clever to arrange the tournament predictor so that they are not aliased in the "global" half.

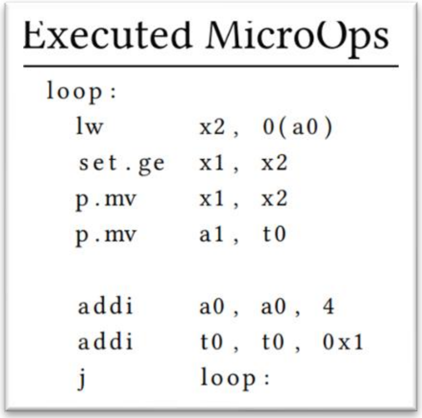
Student question

In section 4.2 [of the SonicBOOM article], it talks about the micro-ops that have been created to implement predication. It says it uses a predicate register to determine "whether to execute the original op, or to perform a copy operation from the stale register to the destination register".

I do not understand why it would perform that copy. Surely if the predicate is false, it should simply do nothing in the instruction?

Great question! This is quite subtle, and involves just how the o-o-o register renaming mechanism works.

Consider their example:



Executed MicroOps

```
loop:
   lw        x2,  0(a0)
   set.ge    x1,  x2
   p.mv      x1,  x2
   p.mv      a1,  t0

   addi      a0,  a0,  4
   addi      t0,  t0,  0x1
   j         loop:
```

- The instruction "p.mv x1 x2" conditionally updates x1 (the running maximum) with x2, the array value just read.
- Register x1 is stored in a physical register (say P_m), allocated during register renaming earlier in the computation (initially, when the "max" is initialised to zero).
- Register x2 is allocated to a physical register (say P_q) at the load instruction.
- During register renaming (ie during issue) a new physical register (say P_n) is allocated for the result of this instruction, and the issue-side register alias table is set to point to this P_n register.
- So if the instruction is not active (predicate p is false), we still need to copy x1 from P_m to P_n.
- If the instruction *is* active, we copy P_q to P_n instead.
- Afterwards, the issue-side register alias table maps logical register x1 to P_n.
- On later iterations of the loop this all happens again, but when the loop exits, the register alias table tells us which physical register the final value of x1 is actually in.

Provocative question

Suppose we can observe the power consumption of a processor while it is decoding a message using a secret key. Perhaps we can trigger the device to repeat the operation many times.
Suppose the code for the decryption algorithm contains conditional branches, that depend on the key.
Can we deduce anything about the secret key? Should we worry? Can we prevent it?

Your answer goes here….