DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Computer Architecture
Lecture 5: Compiler Techniques for ILP & Branch Prediction (Chapter 3)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Introduction

- Pipelining becomes universal technique since 1985
  - To overlap execution of instructions and improve performance
  - To exploit *Instruction Level Parallelism (ILP)*
- Two largely separable approaches to exploiting ILP:
  - Hardware-based dynamic approaches
  - Compiler-based static approaches
- To exploit parallelisms over instructions is equivalent to determine dependences over instructions
  - Pipeline hazards
    - Pipeline CPI =Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls
  - If 2 instructions are dependent, they must be executed in order or partially overlapped (cannot be executed simultaneously).

Why Exploiting ILP?

- Can we make CPI closer to 1?
  - If we have *n*-cycle latency, then we need n-1 instructions between a producing instruction and its use
- *Multi-issue Processor*: two or more instructions can be issued (or executed) in parallel
  - The goal is to maximize Instruction per Cycle (IPC)
  - How to reduce the impact of data and control hazards?
  - Basic block ILP

How to Exploit ILP?

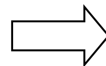
- Two main approaches:
  - Hardware-based dynamic approaches
    - Hardware locates the parallelism in run-time
    - Used in server, desktop processors, tablets, and high-end cell phones. Not used as extensively in IOT space.
    - *Superscalar processors*: Pentium 4, Power, Opteron
  - Compiler-based static approaches
    - Software finds parallelism at compile-time
    - Used in DSP processors (Not as successful outside of scientific applications)
    - *VLIW processors*: Itanium 2, TI DSP

Basic Block ILP

- Basic Block (BB) ILP is limited:
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- We must exploit ILP across multiple basic blocks
  - Using *Loop unrolling* to exploit loop-level parallelism

```
for (i=1; i<=1000, i=i+1)
  x[i] = x[i] + y[i]
```

Loop-level parallelism



```
x[1] = x[1] + y[1]
x[2] = x[2] + y[2]
…
x[1000]=x[1000]+y[1000]
```

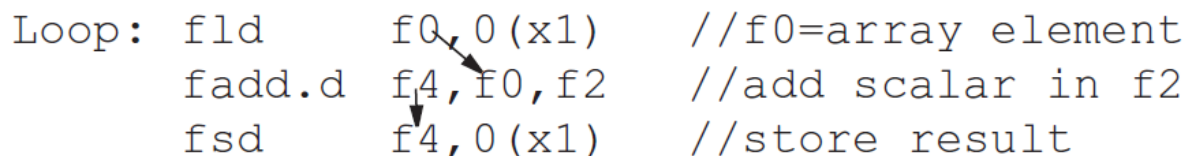
(True) Data Dependence

- Data dependencies are a property of the program
- Data dependence conveys:
  - Possibility of a pipeline hazard
  - Order in which results must be calculated (i.e. program behavior)
  - Upper bound on ILP
- Dependencies that flow through memory locations are difficult to detect
  - Hardware-based dynamic approach is more attractive

Data Dependences Example

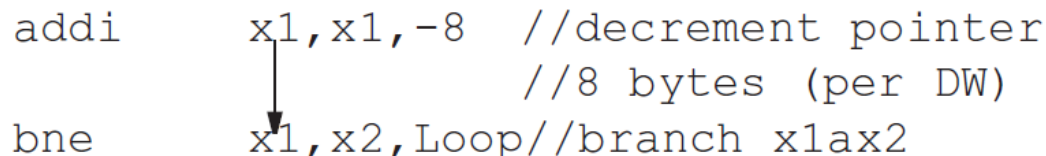
```
Loop:  fld     f0,0(x1)       //f0=array element
       fadd.d  f4,f0,f2       //add scalar in f2
       fsd     f4,0(x1)       //store result
       addi    x1,x1,−8       //decrement pointer 8 bytes
       bne     x1,x2,Loop     //branch x1≠x2
```

- The data dependences in this code sequence involve both floating-point data:



```
Loop:  fld     f0,0(x1)     //f0=array element
       fadd.d  f4,f0,f2     //add scalar in f2
       fsd     f4,0(x1)     //store result
```

and integer data:



```
addi    x1,x1,-8   //decrement pointer
                   //8 bytes (per DW)
bne     x1,x2,Loop//branch x1ax2
```

Overcome Data Dependences

- Maintaining the dependence but avoiding a hazard
  - scheduling the code in HW/SW approach
- Eliminating a dependence by transforming the code
  - primary by software
- Dependence detection
  - by register names: simpler
  - by memory locations: more complicated
    - Two addresses may refer to the same location but look quite different (e.g. 100(R4), 20(R6) may be identical)
    - The effective address of a load/store may changed from instruction to instruction (20(R4), 20(R4) may be different)

Name Dependence

- Two instructions use the same register but no data exchange
  - Not a true data dependence, *but is a problem when reordering instructions or a irregularly pipelined datapath.*
  - *Anti-dependence*: instruction j writes a register that instruction i reads
    - there is an antidependence between fsd and addi on register x1
  - *Output-dependence*: instruction i and instruction j write the same register
    - Ordering must be preserved
- Using *register renaming* to eliminate name dependences

Register Renaming and WAW/WAR

- DIV.D F0, F2, F4
- ADD.D F6, F0, F8
- S.D F6, 0 (R1)
- SUB.D F8, F10, F14
- MUL.D F6, F10, F8

> - WAW: ADD.D/MUL.D
> - WAR: ADD.D/SUB.D, S.D/MUL.D
> - RAW: DIV.D/ADD.D, ADD.D/S.D SUB.D/MUL.D

- DIV.D F0, F2, F4
- ADD.D S, F0, T
- S.D S, 0 (R1)
- SUB.D F8, F10, F14
- MUL.D F6, F10, F8

RAWs are (must be) still there, after register renaming !!

Control Dependence

- Example 1:

```
   add x1,x2,x3
   beq x4,x0,L
   sub x1,x1,x6
L: …
   or  x7,x1,x8
```

- `x1` in `or` instruction depends on `add` or `sub`, relied on the branch is taken or not.
- Violating the control dependence in this example might affect the data flow (i.e. the program behavior).

- Example 2:

```
   add x1,x2,x3
   beq x12,x0,skip
   sub x4,x5,x6
   add x5,x4,x9
skip:
   or  x7,x8,x9
```

- Assume `x4` isn't used after skip. Possible to move `sub` before the branch
- Violate the control dependence might not affect the data flow in this example.

Preserve Control Dependence?

- The two properties critical to program correctness are
  - The exception behavior
  - Data flow
- Branches make data flow dynamic
- Control dependence is not the critical property that must be preserved
  - We may execute instruction that should not have been executed, thereby violating the control dependence, if we can do so without affecting the correctness of the program
  - *Hardware/software speculation*
- Control stalls can be eliminated or reduced by a variety of hardware and software techniques

Summary: ILP and Data Dependencies

- HW/SW must preserve program order: order instructions would execute in if executed sequentially as determined by original source program
  - Dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies

  1) indicates the possibility of a hazard

  2) determines order in which results must be calculated

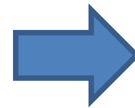
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instructions from the source instruction by the pipeline latency (or instruction latency) of the source instruction
- Example:

  for (i=999; i>=0; i=i-1)

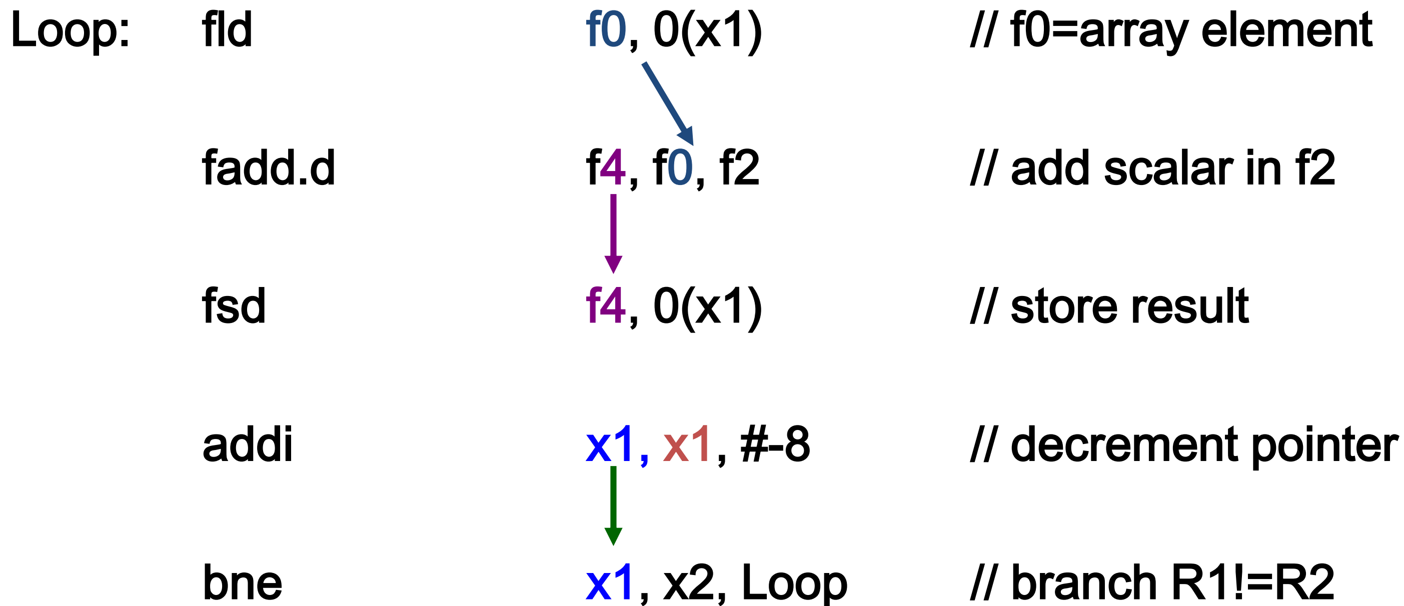
  x[i] = x[i] + s;



```
Loop:  fld      f0,0(x1)
       fadd.d   f4,f0,f2
       fsd      f4,0(x1)
       addi     x1,x1,#-8
       bne      x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Data Dependence Analysis

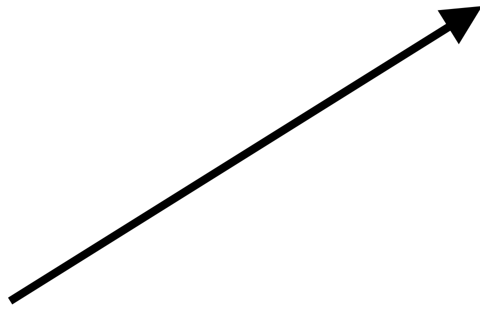


- The arrows show the order that must be preserved for correct execution.
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

Step 1: Insert Stalls w/wo Scheduleing

```
Loop:   fld     f0,0(x1)
        stall
        fadd.d  f4,f0,f2
        stall
        stall
        fsd     f4,0(x1)
        addi    x1,x1,-8
        bne     x1,x2,Loop
```

8 C.C/ iteration



```
Loop:   fld     f0,0(x1)
        addi    x1,x1,-8
        fadd.d  f4,f0,f2
        stall
        stall
        fsd     f4,8(x1)
        bne     x1,x2,Loop
```

7 C.C/ iteration

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Step 2: Loop Unrolling wo Scheduling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   fld      f0,0(x1)
        fadd.d   f4,f0,f2
        fsd      f4,0(x1)      //drop addi & bne
        fld      f6,-8(x1)
        fadd.d   f8,f6,f2
        fsd      f8,-8(x1)     //drop addi & bne
        fld      f0,-16(x1)
        fadd.d   f12,f0,f2
        fsd      f12,-16(x1) //drop addi & bne
        fld      f14,-24(x1)
        fadd.d   f16,f14,f2
        fsd      f16,-24(x1)
        addi     x1,x1,-32
        bne      x1,x2,Loop
```

26 C.C/ 4 iterations

or **6.5 C.C/ iteration**

- note: no reuse any of registers

Step 3: Re-Schedule the Unrolled loop

- Pipeline schedule the unrolled loop:

```
Loop:   fld      f0,0(x1)
        fld      f6,-8(x1)
        fld      f8,-16(x1)
        fld      f14,-24(x1)
        fadd.d   f4,f0,f2
        fadd.d   f8,f6,f2
        fadd.d   f12,f0,f2
        fadd.d   f16,f14,f2
        fsd      f4,0(x1)
        fsd      f8,-8(x1)
        fsd      f12,-16(x1)
        fsd      f16,-24(x1)
        addi     x1,x1,-32
        bne      x1,x2,Loop
```

14 C.C/ 4 iterations

or **3.5 C.C/ iteration**

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  - "*strip mining*" technique
- For large values of n, most of the execution time will be spent in the unrolled loop

5 Loop Unrolling Decisions

1. **Determine loop unrolling** useful by finding that loop iterations were independent (except for maintenance code)
2. **Use different registers** to avoid unnecessary constraints forced by using same registers for different computations
3. **Eliminate the extra test and branch instructions and adjust the loop termination and iteration code**
4. **Determine that loads and stores in unrolled loop can be interchanged** by observing that loads and stores from different iterations are independent
   - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
5. **Schedule the code**, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure (compiler limitation): potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

3.7 Exploiting ILP Using Multiple Issue and Static Scheduling

Getting CPI below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors
- 2 types of superscalar processors issue varying numbers of instructions per cycle
  - use in-order execution if they are statically scheduled, or
  - use out-of-order execution if they are dynamically scheduled
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)

Five Multiple-Issue Processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Basic VLIW

- A VLIW uses *multiple, independent* functional units
- A VLIW packages multiple operations into one very long instruction
  - The burden for choosing and packaging independent operations falls on compiler
  - HW in a superscalar makes these issue decisions is unnecessary
- VLIW depends on enough parallelism for keeping FUs busy
  - *Loop unrolling and then code scheduling*
  - Compiler may need to do local scheduling and global scheduling
- Here we consider a VLIW processor might have instructions that contain 5 operations, including 1 integer (or branch), 2 FP, and 2 memory references
  - 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide

Recall: Unrolled Loop that Minimizes Stalls for Scalar

- Loop:

```
Loop:   fld     f0,0(x1)
        fld     f6,-8(x1)
        fld     f8,-16(x1)
        fld     f14,-24(x1)
        fadd.d  f4,f0,f2
        fadd.d  f8,f6,f2
        fadd.d  f12,f0,f2
        fadd.d  f16,f14,f2
        fsd     f4,0(x1)
        fsd     f8,-8(x1)
        fsd     f12,-16(x1)
        fsd     f16,-24(x1)
        addi    x1,x1,-32
        bne     x1,x2,Loop
```

fld to fadd.d: 1 Cycle
fadd.d to fsd: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.29 clocks per iteration

23 ops in 9 clocks, average 2.5 ops per clock, about 50% efficiency

Note: Need more registers in VLIW

VLIW Problems

- Increase in code size
  - Ambitious loop unrolling
  - Whenever instructions are not full, the unused FUs translate to waste bits in the instruction encoding
    - An instruction may need to be left completely empty if no operation can be scheduled
  - Clever encoding or compress/decompress
- Limitations of the lockstep operation
  - No hazard-detection hardware at all.
- Binary code compatibility
  - Different numbers of functional units and unit latencies require different versions of the code
  - Need re-compilation
  - Solution: Object-code translation or emulation

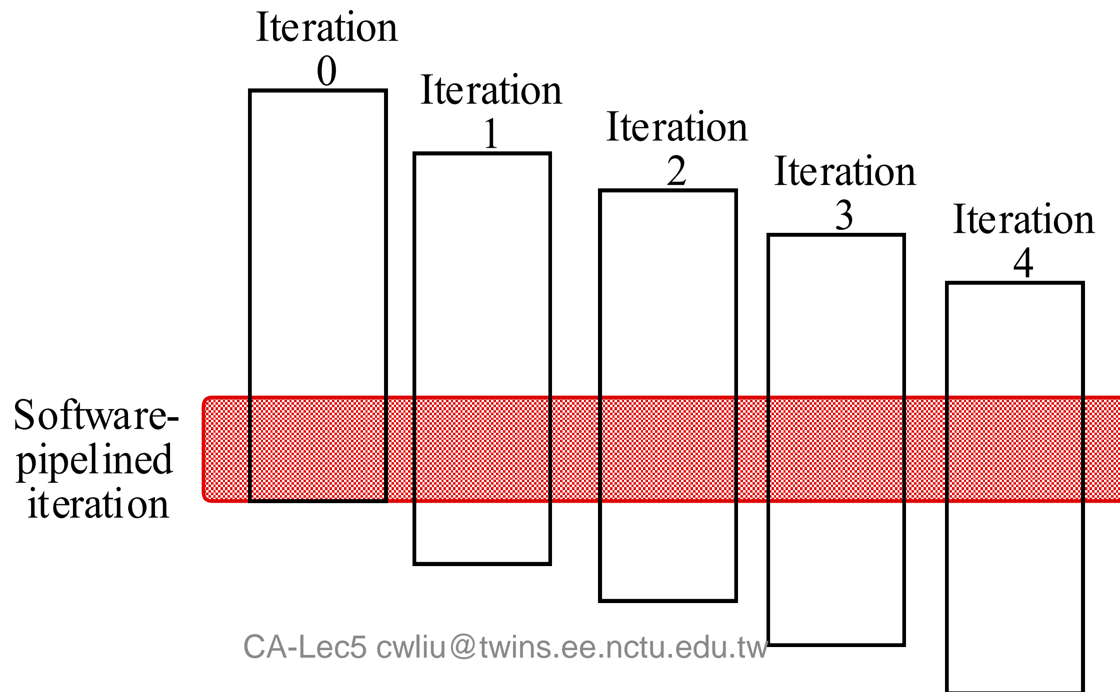
Read the EPIC approach in Appendix H which provides solutions to VLIW problems !!

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- IA-64: instruction set architecture
- 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) => 40% fewer mispredictions?
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 μ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

Another Possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- *Software pipelining*: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop



3.3 Reducing Branch Costs with Advanced Branch Prediction

Control Hazard Avoidance

- Consider Effects of Increasing the ILP
  - Control dependencies rapidly become the limiting factor
  - They tend to not get optimized by the compiler
    - Higher branch frequencies result
    - Plus multiple issue (more than one instructions/sec) → more control instructions per sec.
  - Control stall penalties will go up as machines go faster
    - Amdahl's Law in action - again
- Branch Prediction: helps if can be done for reasonable cost
  - Static by compiler: appendix C (e.g. predict not taken, delay branch)
  - Dynamic by HW: this section

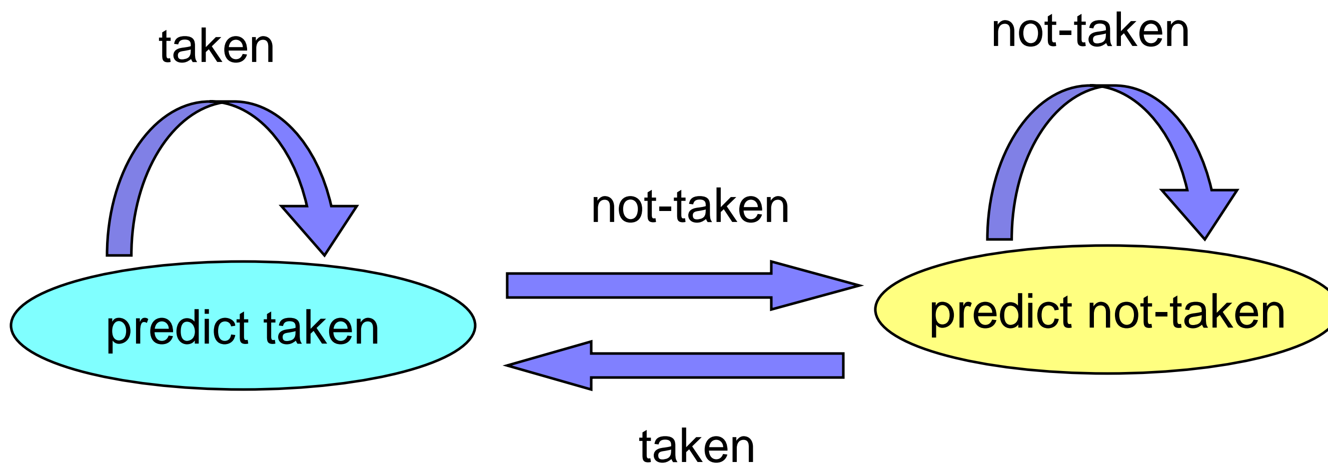
Why Does Branch Prediction Work?

- Underlying algorithm has regularities
- Data that is being operated on has regularities
- Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Dynamic Branch Prediction

- The predictor will depend on the behavior of the branch at run time
- Goals:
  - allow the processor to resolve the outcome of a branch early, prevent control dependences from causing stalls
- Effectiveness of a branch prediction scheme depends not only on the accuracy but also on the cost of a branch
  - BP_Performance = *f* (accuracy, cost of misprediction)
- Branch History Table (BHT)
  - Lower bits of PC address index table of 1-bit values
    - No “precise” address check – just match the lower bits
  - Says whether or not branch taken last time

1-bit Dynamic Hardware Prediction



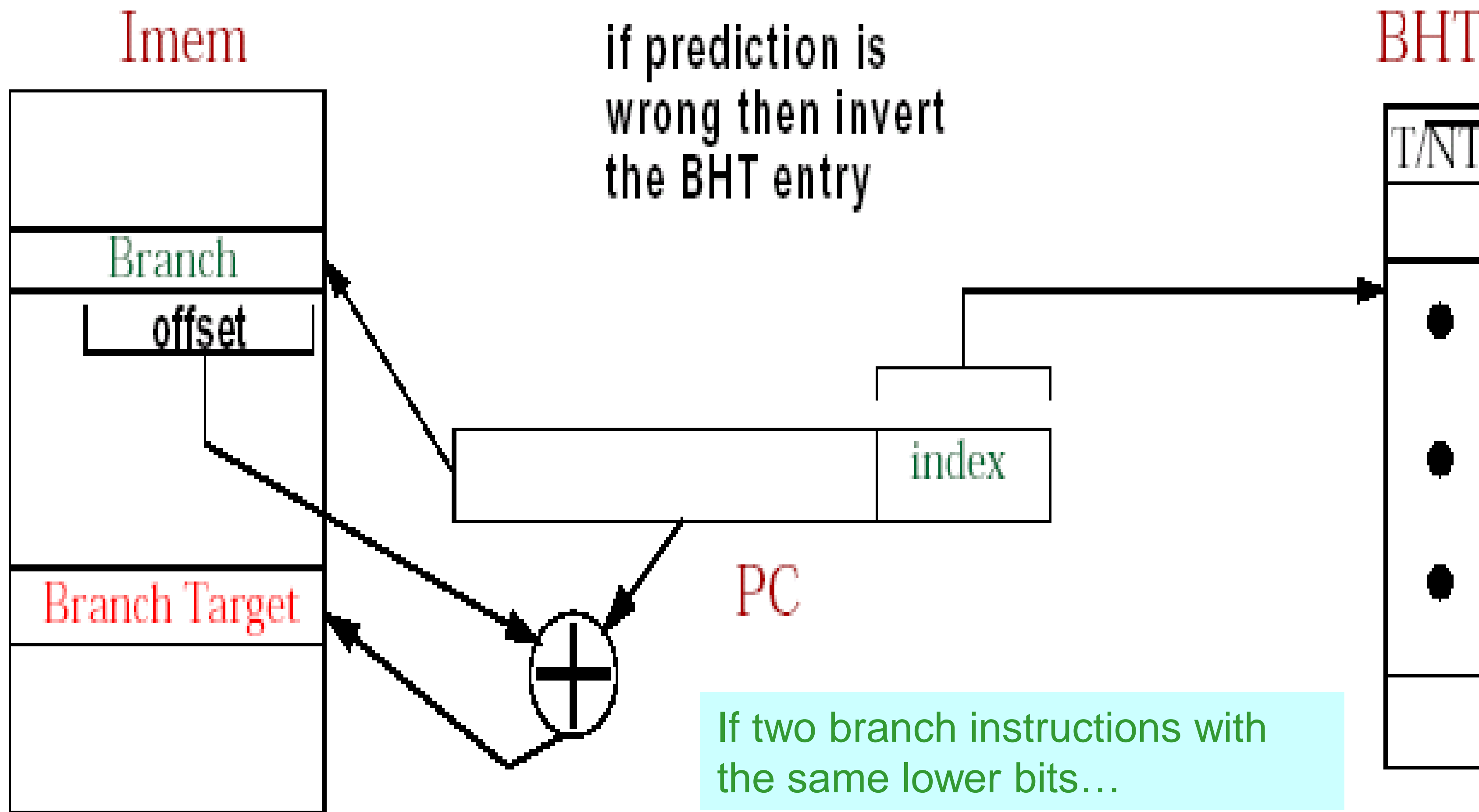
Problem: Loop case

```
LOOP:       LOAD    R1, 100(R2)
            MUL     R6, R6, R1
            SUBI    R2, R2, #4
            BNEZ    R2, LOOP
```

The steady-state prediction behavior will mispredict on the first and last loop iterations

BHT Prediction

Useful only for the target address is known before CC is decided



Problem with the Simple BHT

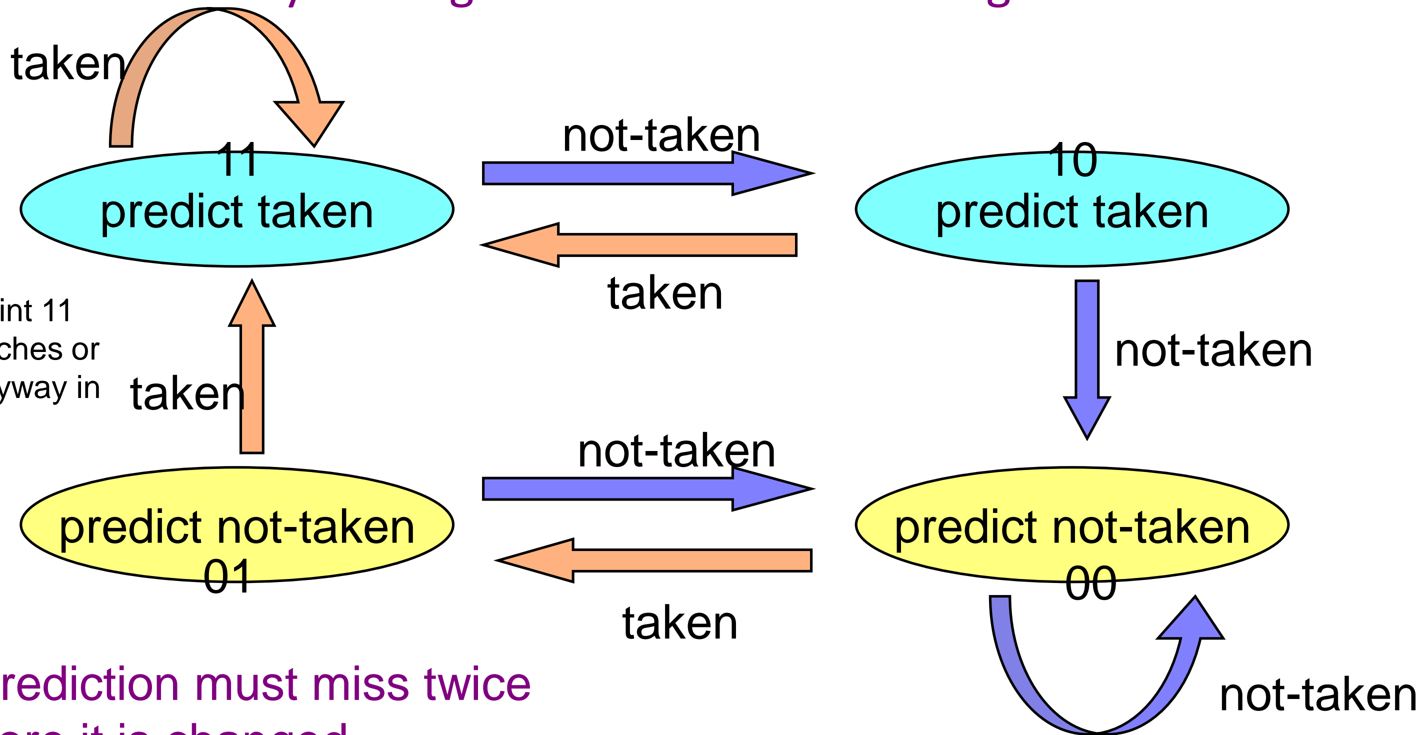
clear benefit is that it's cheap and understandable

- Aliasing
  - All branches with the same index (lower) bits reference same BHT entry
    - Hence they mutually predict each other
    - No guarantee that a prediction is right. But it may not matter anyway
  - Avoidance
    - Make the table bigger - OK since it's only a single bit-vector
    - This is a common cache improvement strategy as well
      - Other cache strategies may also apply
- Consider how this works for loops
  - Always mispredict twice for every loop
    - One is unavoidable since the exit is always a surprise
    - However previous exit will always cause a mis-prediction the first try of every new loop entry

N-bit Predictors

idea: improve on the loop entry problem

- 2-bit counter implies 4 states
  - Statistically 2 bits gets most of the advantage



Compiler could hint 11 init. on loop branches or it will go to 11 anyway in the 4th iteration

A prediction must miss twice before it is changed

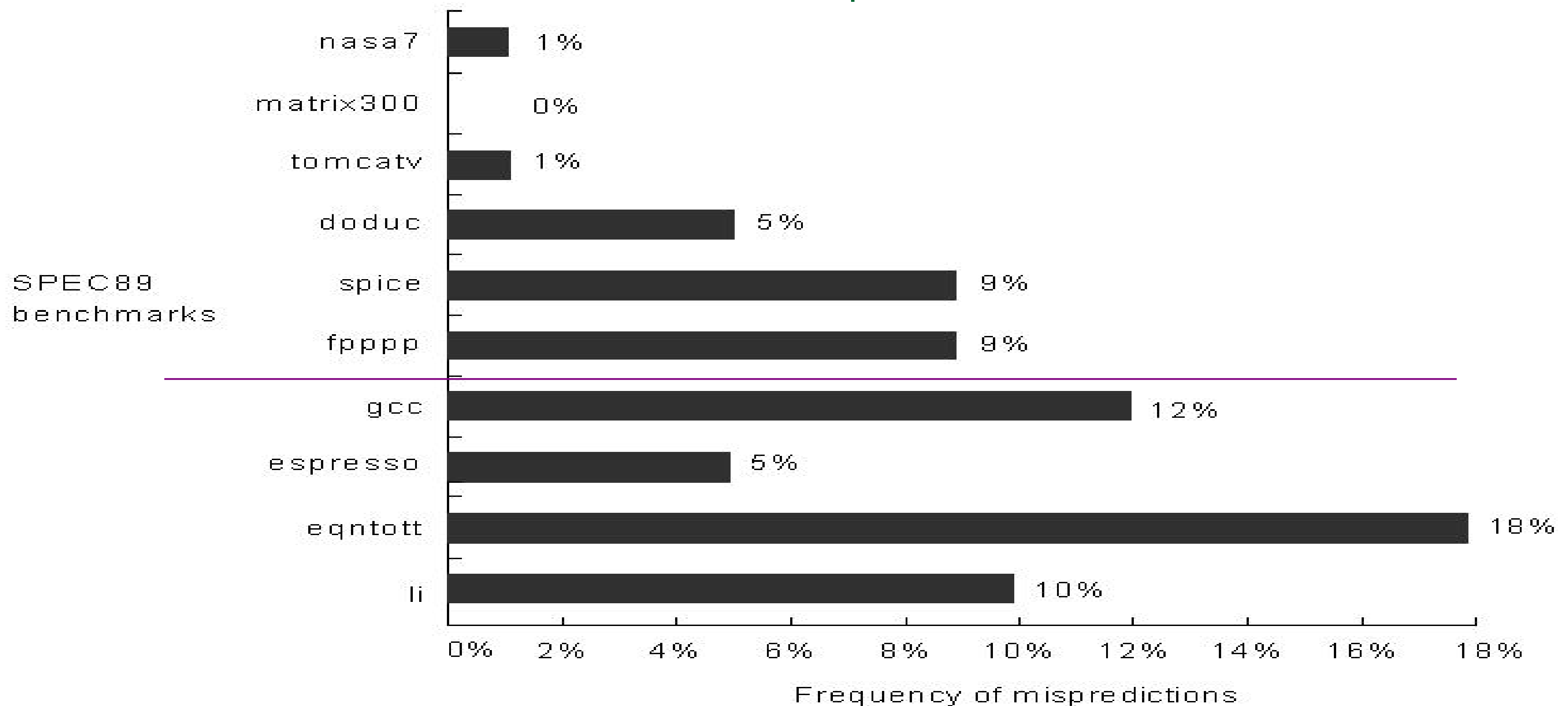
Only the loop exit causes a mispredict

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

BHT Accuracy

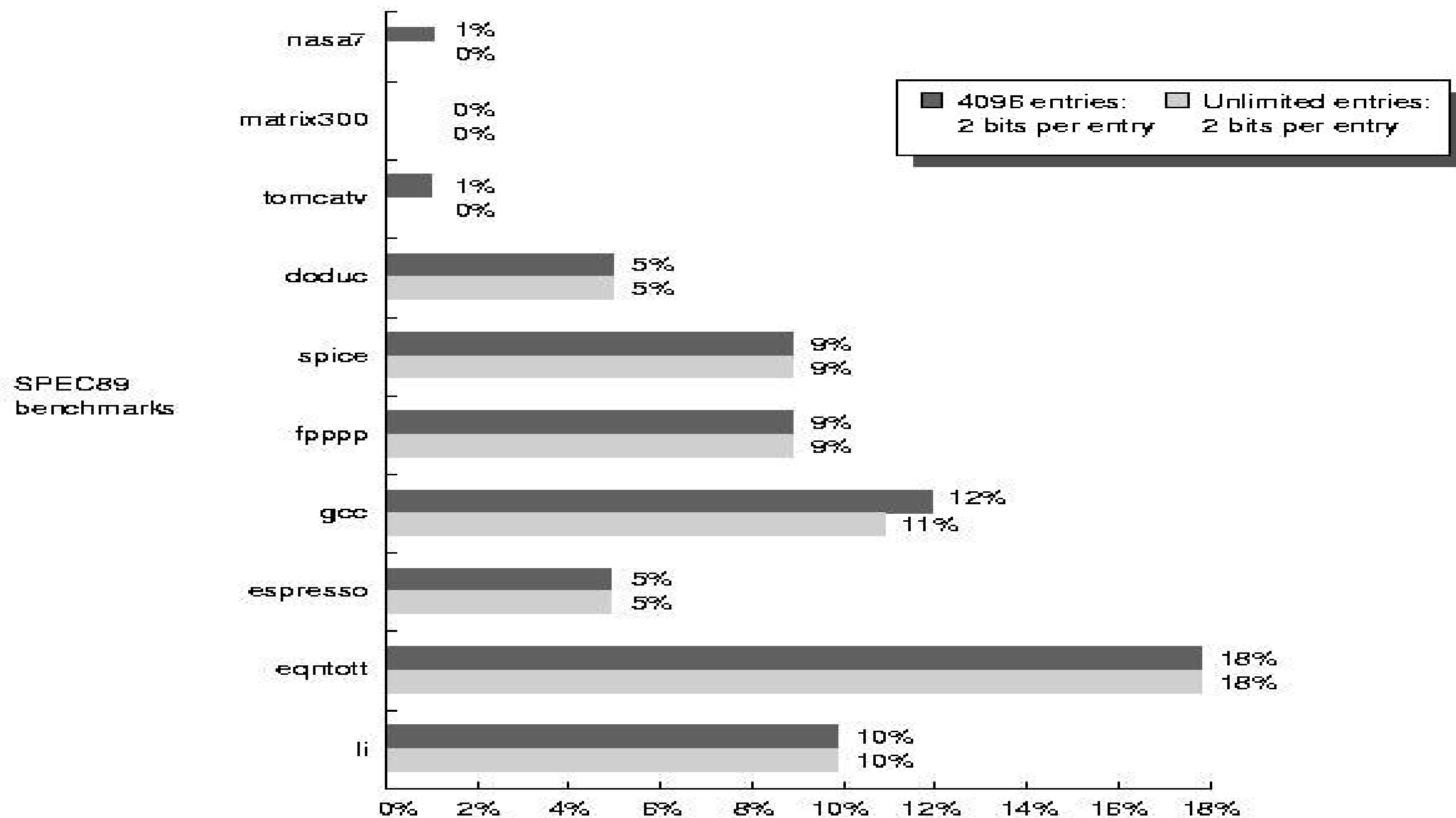
- Mispredict because either:
  - Wrong guess for that branch (accuracy)
  - Got branch history of wrong branch when index the table (size)

4K of BPB with 2-bit entries misprediction rates on SPEC89@IBM Power



To Increase the BHT Size

- 4096 about as good as infinite table
- The hit rate of the buffer is clearly not the limiting factor for an enough-large BHT size



The worst case for the 2-bit predictor

aa and bb are assigned to R1 and R2

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

```
        addi    x3, x1, -2
        bnez    x3, L1          //branch b1 (aa!=2)
        add     x1, x0, x0      //aa=0
L1:     addi    x3, x2, -2
        bnez    x3, L2          //branch b2 (bb!=2)
        add     x2, x0, x0      //bb=0
L2:     sub     x3, x1, x2      //x3=aa-bb
        beqz    x3, L3          //branch b3 (aa==bb)
```

if the first 2 branches are untaken, then the 3rd will always be taken

➔ the behavior of branch b3 is correlated with the behavior of branches b1 and b2

Improve Accuracy By Correlating Predictors

- *Correlating predictors* or 2-level (*m,n*) *predictors*
  - (*m*,*n*) predictor means record last m branches to select between 2^m history tables each with n-bit predictor
  - Correlation = To record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
  - *m*-bit shift register keeping T/NT status of last m branches
  - *n*-bit Predictor = To determine which way to go
- Simple 2-bit BHT is just a (0,2) predictor

2-Level (*m*, *n*) BHT

- Use the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch
- Total bits for the (m, n) BHT prediction buffer with *p*-bit index:

$$2^m \times n \times 2^p$$

  - p bits of buffer index = 2^P-entry n-bit BHT
  - 2^m banks of memory selected by the global branch history (which is just a shift register) - e.g. a column address

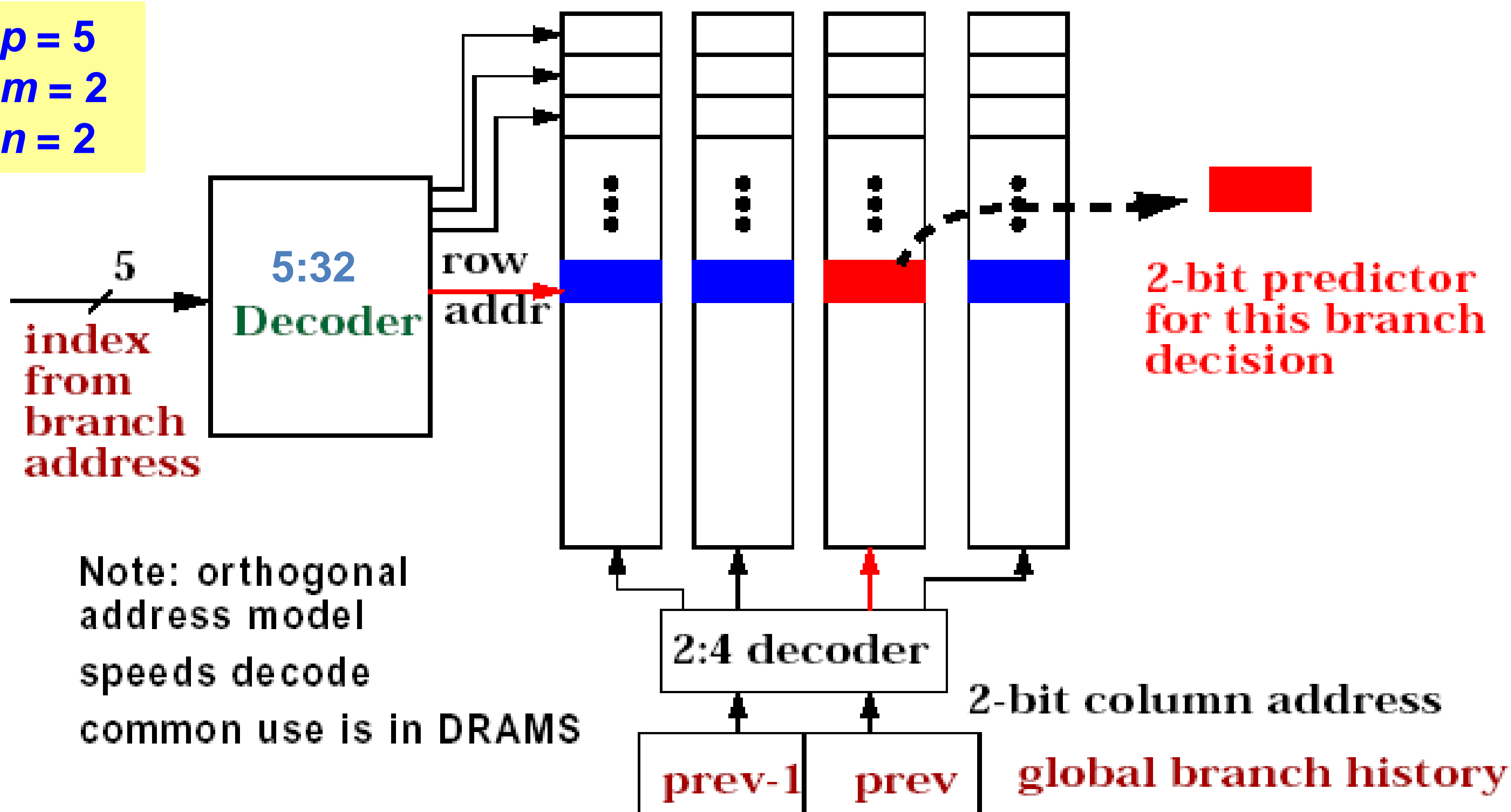
– Use p bits of the branch address to select row

– Use m-bit shift register to select to select column

– Get the n predictor bits in the entry to make the decision

(2,2) Predictor Implementation

4 banks = each with 32 2-bit predictor entries



Example of Correlating Branch Predictors

d is assigned to R1

```
if (d==0)
    d = 1;
if (d==1)
    …
```

```
        BNEZ   R1, L1          ;branch b1 (d!=0)
        DAAIU  R1, R0, #1      ;d==0, so d=1
L1:     DAAIU  R3, R1, #-1
        BNEZ   R3, L2          ;branch b2 (d!=1)
…
L2:
```

Example of Correlating Branch Predictors (Cont.)

initial value of d	d==0?	b1	value of d before b2	d==1?	b2
0	YES	not taken	1	YES	not taken
1	NO	taken	1	YES	not taken
2	NO	taken	2	NO	taken

1-bit predictor initialized to NT

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

All the branches are mispredicted !!!

Example of Correlating Branch Predictors (Cont.)

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

(1,1) predictor

Use 1-bit correlation + 1-bit prediction with initialized to NT/NT

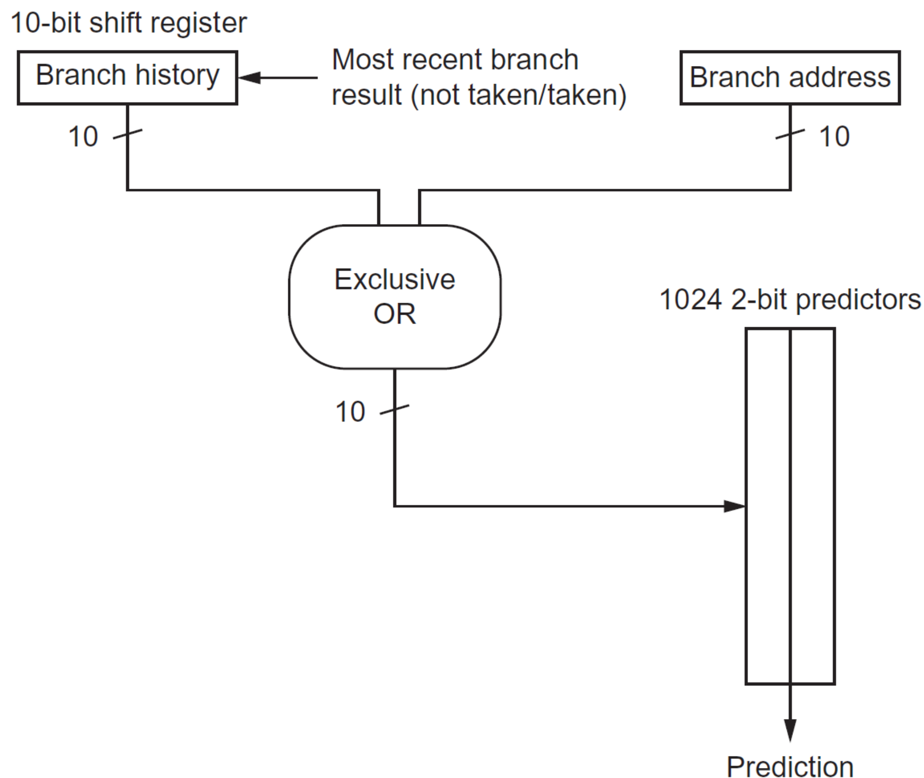
d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

Comparison of Different 2-bit Predictors

SPEC89 benchmarks	4096 entries: 2 bits per entry (8K bits)	Unlimited entries: 2 bits per entry	1024 entries: (2,2) (8K bits)
nASA7	1%	0%	1%
matrix300	0%	0%	0%
tomcatv	1%	0%	1%
doduc	5%	5%	5%
spice	9%	9%	5%
fpppp	9%	9%	5%
gcc	12%	11%	11%
espresso	5%	5%	4%
eqntott	18%	18%	6%
li	10%	10%	5%

Frequency of mispredictions (0%, 2%, 4%, 6%, 8%, 10%, 12%, 14%, 16%, 18%)

McFarling's Gshare Predictor



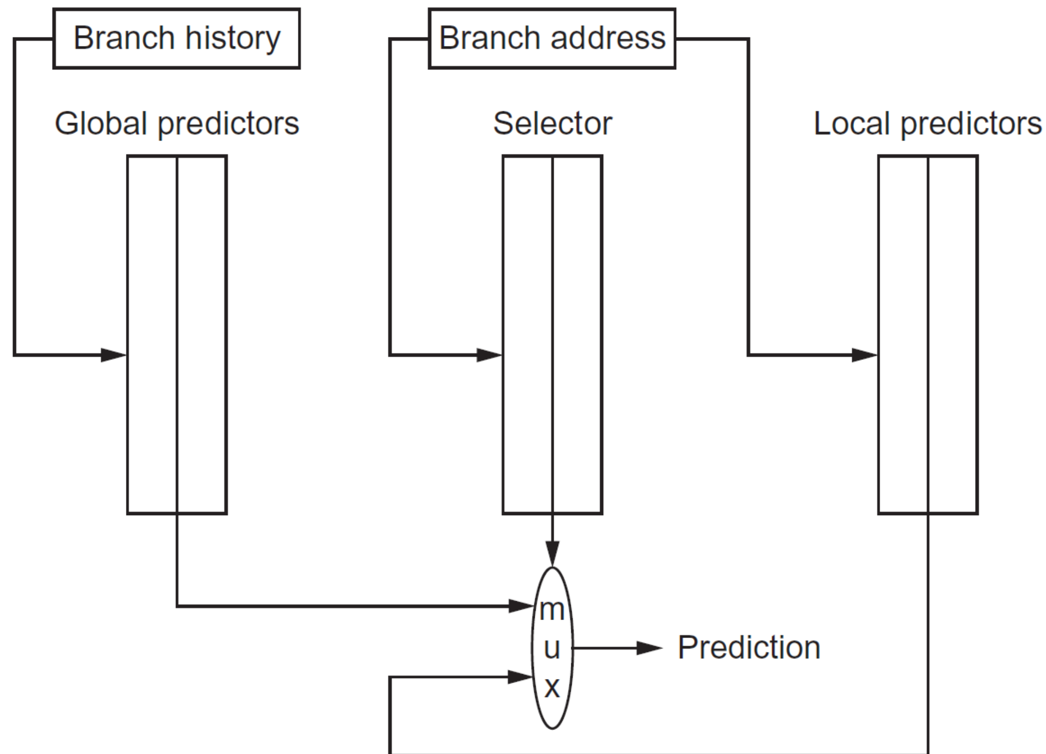
- The best-known example of a correlating predictor: gshare predictor
  - The index is formed by combining the address of the branch and the most recent conditional branch outcomes using an exclusive-OR, which essentially acts as a hash of the branch address and the branch history.
  - The hashed result is used to index a prediction array of 2-bit counters

Tournament Predictors

The most popular one

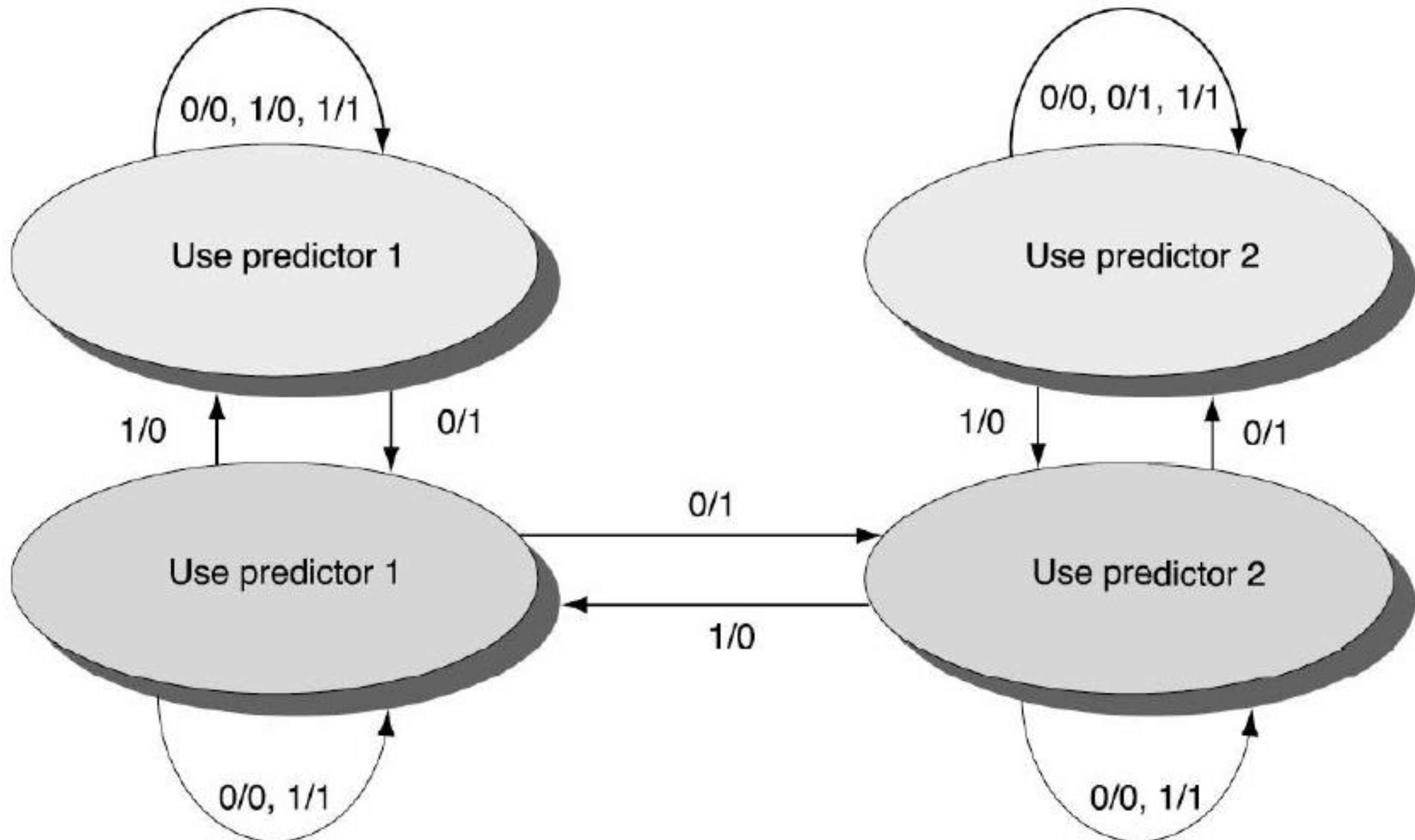
- Recall that the correlator is just a local predictor
- Adaptively combine local and global predictors
  - Multiple predictors
    - One based on global information: Results of recently executed m branches
    - One based on local information: Results of past executions of the current branch instruction
  - Selector to choose which predictors to use
    - E.g.: 2-bit saturating counter, incremented whenever the "predicted" predictor is correct and the other predictor is incorrect, and it is decremented in the reverse situation
- Advantage
  - Ability to select the right predictor for the right branch

Tournament Predictors

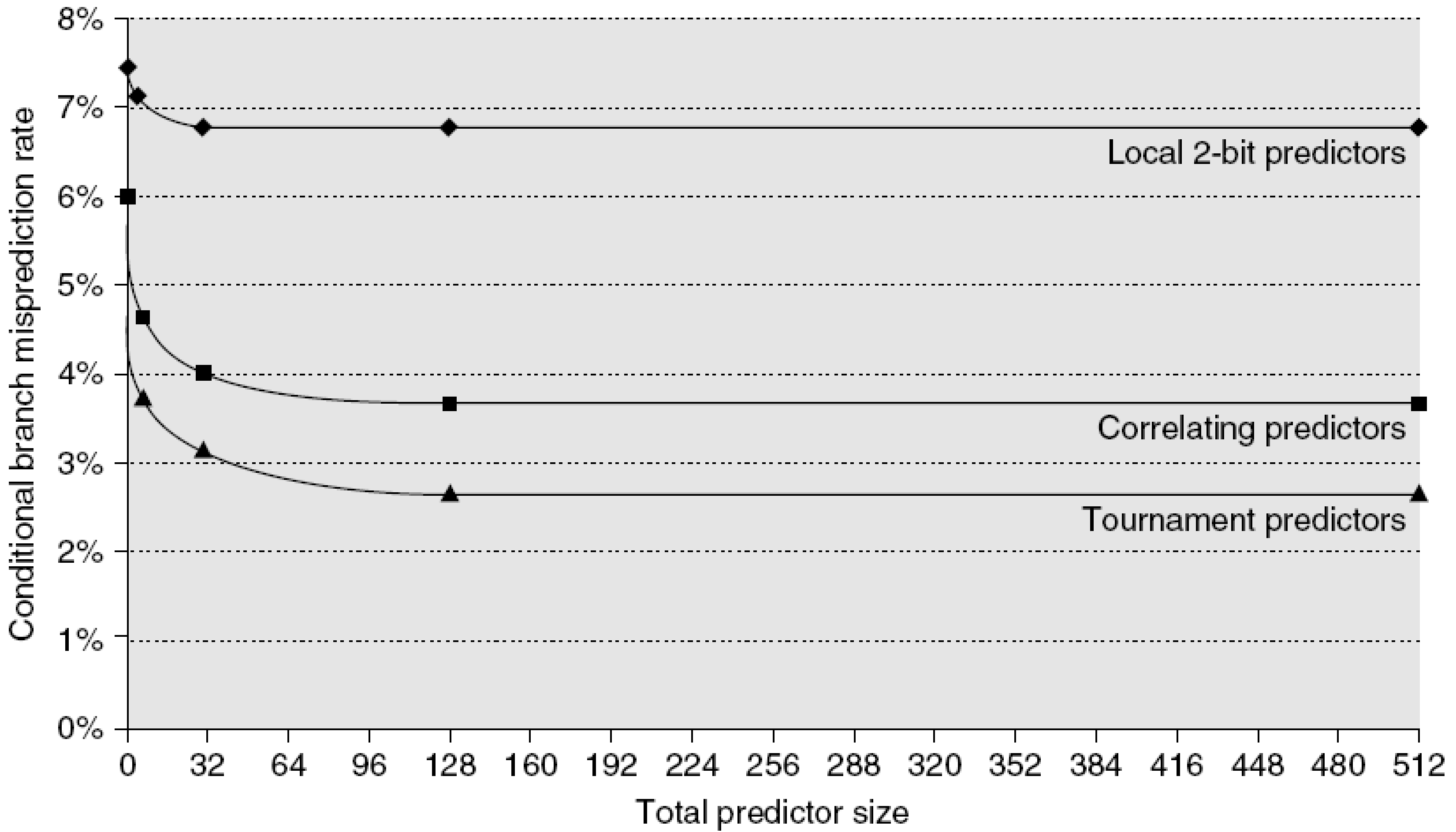


- A global predictor uses the most recent branch history to index the predictor
- A local predictor uses the address of the branch as the index.

State Transition Diagram for Tournament Predictor



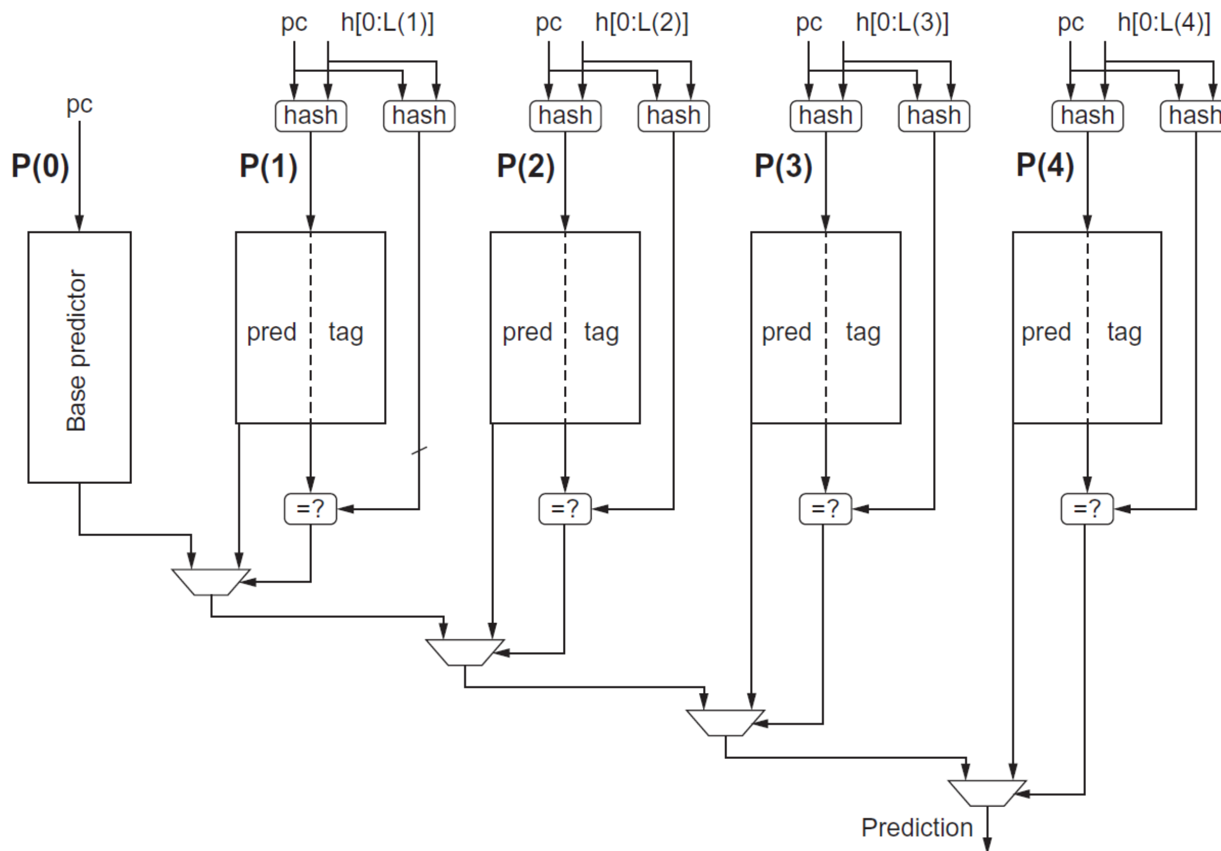
Branch Prediction Performance



Tagged Hybrid Predictors (TAGE)

- Need to have predictor for each branch and history
  - Problem: this implies huge tables
  - Solution:
    - Combine multiple predictors that track whether a prediction is likely to be associated with the current branch
    - Use hash tables, whose hash value is based on branch address and branch history
    - Longer histories may lead to increased chance of hash collision, so use multiple tables with increasingly shorter histories
    - Use PPM (Prediction by Partial Matching) algorithm

5-component Tagged Hybrid Predictor



- Five prediction tables: P(0), P(1), . . . P(4), where P(*i*) is accessed using a hash of the PC and the history of the most recent *i* branches (kept in a shift register, h, just as in gshare).
- P(0) always matches because it uses no tags and becomes the default prediction.
- A small tag of 4–8 bits is good enough (100% matches are not required).
- A prediction from P(1), . . . P(4) is used only if the tags match the hash of the branch address and global branch history.

DEPT. OF ELECTRC
ENGINEERING
INST. OF ELECTRC

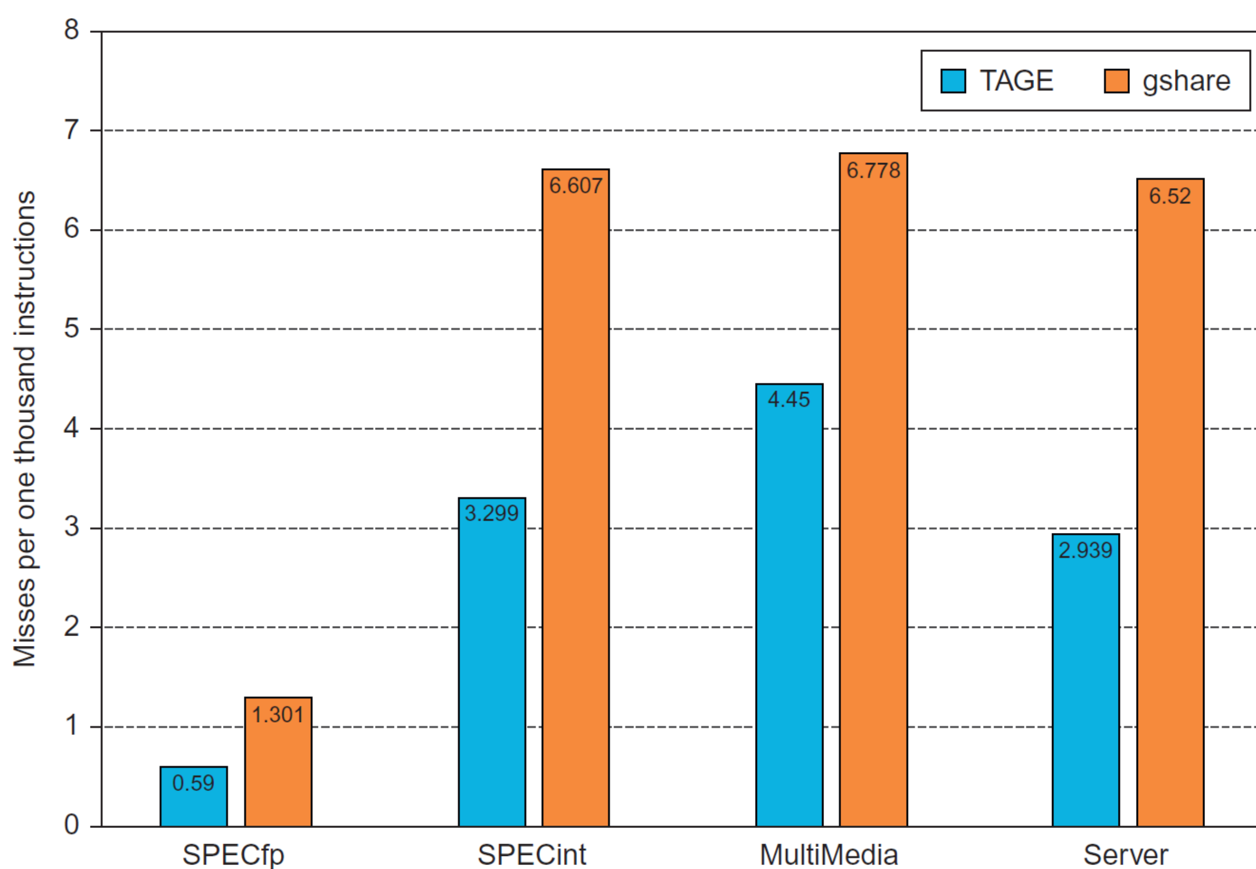


Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

- Compared to gshare, tagged hybrid predictors are more complex to implement and are probably slightly slower because of the need to check multiple tags and choose a prediction result.
- For deeply pipelined processors with large penalties for branch misprediction, the increased accuracy outweighs those disadvantages.