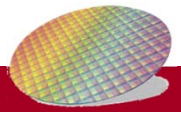
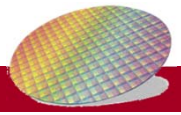
Instruction-Level Parallelism and Its Exploitation



Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits "Instruction Level Parallelism"

- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications
- Hardware-based approaches dominate desktop and server markets
- Compiler-based approaches are popular in PMD market



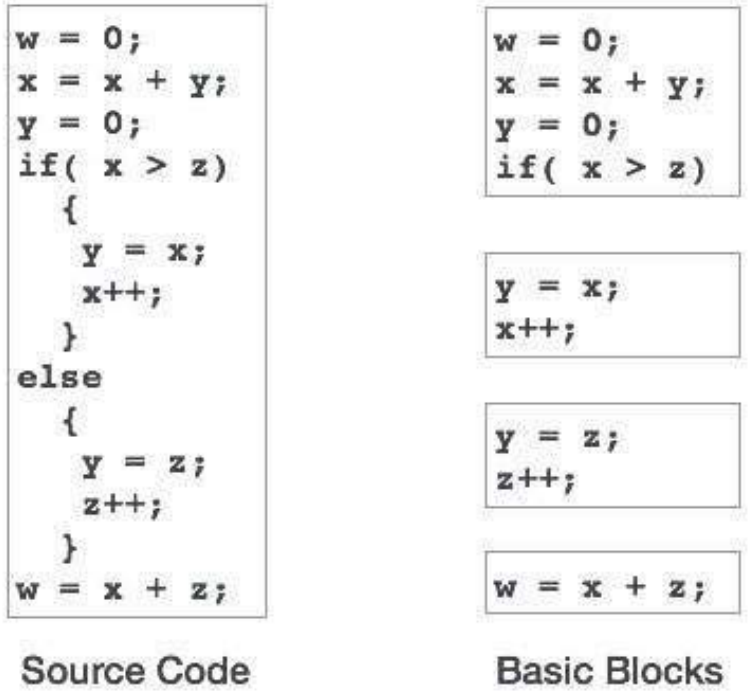
Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI

Pipeline CPI =Ideal pipeline CPI +Structural stalls +Data hazard stalls +Control stalls

- Parallelism within a basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Recall: Basic block
=> straight-line codes with no branches in except to the entry and no branches out except at the exit



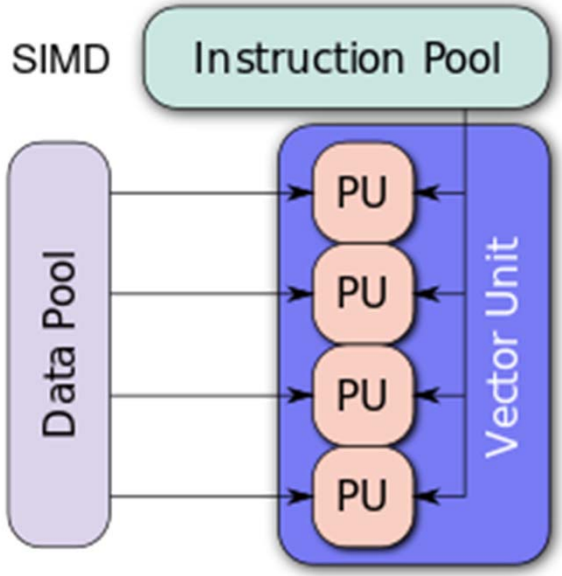
Some techniques to improve ILP (more details in later slides)

- loop-level Parallelism: If a iteration of the loop can overlap with other integration
  - Unroll the loop (statically or dynamically)
  - A simple way to improve ILP

```
for (i= 0; i<=9; i=i+1)
   x[i]= x[i] + y[i];
```

```
x[0]= x[0] + y[0];
x[1]= x[1] + y[1];
   …..
x[9]= x[9] + y[9];
```

- SIMD such as vector processor and GPU operates a small to moderate number of data items in parallel

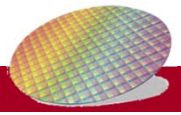


Dependence

- Hazard: next instruction cannot execute in the following cycle
- Dependences

  Dependences ≠ Hazard

  – Dependencies are a property of programs
  – Pipeline organization determines whether a given dependence results in hazard
- Message that data dependence conveys:
  – Possibility of a hazard
  – Order in which results must be calculated
  – Upper bound on exploitable instruction level parallelism
- Two ways to overcome dependence
  – Maintaining the dependence but avoiding a hazard
  – Eliminating a dependence by transforming the code

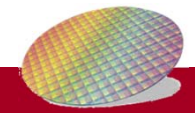


Data Dependence

- Types of Dependence: data dependence, name dependences, control dependences
- Dependent instructions cannot be executed simultaneously
- Data dependency: Instruction *j* is data dependent on instruction *i* if
  - Instruction *i* produces a result that may be used by instruction *j*
  - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*

```
Loop:  L.D      F0, 0(R1)
       ADD.D    F4, F0, F2
       S.D      F4, 0 (R1)
       DADDUI   R1, R1, #-8
       BNE      R1, R2, LOOP
```

```
Loop:  L.D      F0, 0(R1)
       ADD.D    F4, F0, F2
       S.D      F4, 0 (R1)
       DADDUI   R1, R1, #-8
       BNE      R1, R2, LOOP
```



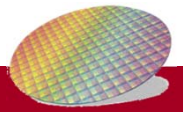
Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- Not true data dependence=> use renaming techniques to resolve

```
Loop:  L.D      F0, 0(R1)
       ADD.D    F4, F0, F2
       S.D      F4, 0 (R1)      Anti-dep on F4

       DADDUI   R1, R1, #-8     Anti-dep on R1
       BNE      R1, R2, LOOP

       L.D      F4, 0(R1)
                  ↓             Output-dep
       ADD.D    F4, F0, F2      on F4
```



Control Dependence

- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
- Constraints imposed by control dep.
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

```
If p1 {
    S1;
};
```

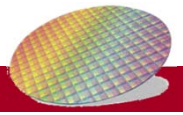
S1 is control dependent o p1

⇨

```
    DADDU   R2, R3, R4
    BEQZ    R2, L1
    LW      R1, 0(R2)
L1:
```

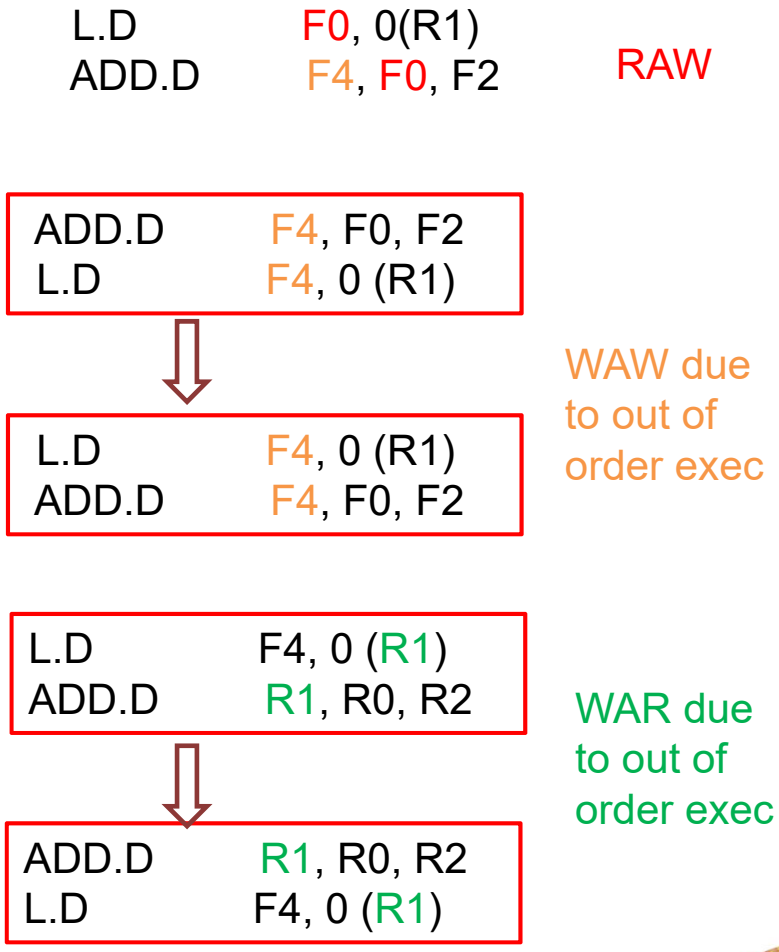
Move LW before BEQZ will make program incorrect

Control dependence prevent from interchanging BEQZ and LW (no data dependence)



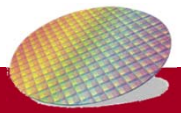
Data Hazard

- According to the order of read and write accesses, data hazards can be classified as
  - Read after write (RAW)
    - J tries to read a source before I writes it
    - Corresponding to true data dependence
  - Write after write (WAW)
    - J tries to write an operand before I write it
    - Corresponding to output dependence
    - Only occur in pipelines that write in more than one stage or with out-or-order execution
  - Write after read (WAR)
    - J tries to write an destination before I read it
    - Corresponding to antidependence
    - Only occur in pipelines with out-or-order execution



Data Dependence vs. Data Hazard

- A dependence is a property of the instructions in a program
  - For example a true dependence arises when one instruction uses the value produced by an earlier instruction.
- A hazard
  - the situation that next instruction cannot execute in the following cycle
  - potential problem in a pipeline that may arise from a dependence.
  - For example, a true data dependence causes a hazard when the value produced by the earlier instruction is not yet available in the register file when the dependent instruction attempts to fetch it. Sometimes forwarding can avoid hazard even there is data dependence.



Critical Property to ensure program correctness

- Critical Property to ensure program correctness: exception behavior & data flow
- Both preserved by maintaining data and control dependence

```
    DADDU  R2, R3, R4
    BEQZ   R2, L1
    LW     R1,0(R2)
L1:
```

Cannot move LW before BEQZ because new exception may happen (R2 may be zero)

```
    DADDU R1,R2,R3
    BEQZ R4,L
    DSUBU R1,R1,R6
L:  …
    OR R7,R1,R8
```

OR instruction dependent on DADDU and DSUBU

Data dependence is not sufficient to preserve correctness

BEQZ is needed to assign correct value to R1 (if R4==0, R1 in DADDU is used, otherwise R1 in DSUBU is used)

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

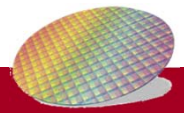
Straightforward MIPS codes ⇒

```
Loop:  L.D     F0,     0(R1)
       ADD.D   F4,F0,F2
       S.D     F4,0(R1)
       DADDUI  R1,R1,#-8
       BNE     R1,R2,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

Number of required cycles to avoid stalls

How this loop will run when it is scheduled on a simple pipeline for MIPS with latencies from Left



Pipeline Stalls

```
Loop:   L.D      F0,0(R1)
        stall
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,0(R1)
        DADDUI R1,R1,#-8
        stall (assume integer load latency is 1)
        BNE R1,R2,Loop
```

How many cycles are needed if no scheduling is used?

9 cycles

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

Pipeline Scheduling

Scheduled code:

```
Loop:   L.D      F0,0(R1)
        DADDUI R1,R1,#-8
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,8(R1)
        BNE R1,R2,Loop
```

How many cycles are needed if pipeline scheduling is used?

7 cycles

Note that actual work is done in three cycles, other are overhead (DADDUI, BNE, and 2 stalls)

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

Loop Unrolling

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

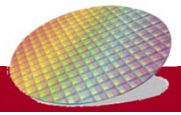
```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

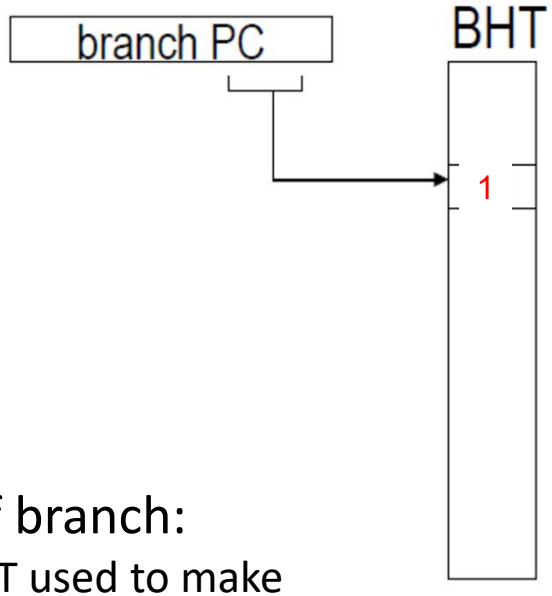
```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

14 Cycles total, 3.5 clock cycles per element
Compared to 9 cycles without scheduling, and 7 cycles with scheduling



Recall: Branch history table with 1-bit predictor

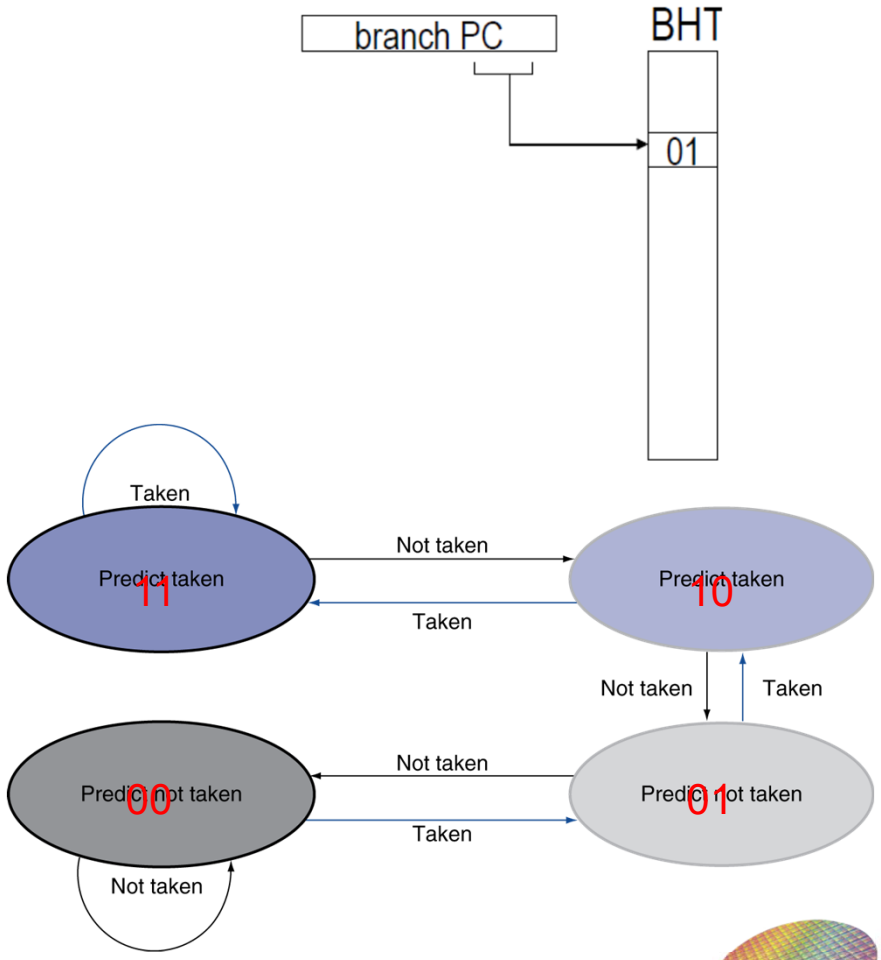
- Branch history table :
  - Is a table of predictors
  - a small memory indexed by the lower portion of the address of the branch instruction (Uses low-order bits of branch PC to choose entry)
  - Contains a bit says whether the branch was recently taken or not
- No full address check => save HW, but may be wrong
- 1-bit predictor:
  - When 0=> predict not taken,
  - When 1=>predict taken,
  - Change states when incorrectly predicted
- Has limited size
  - 2 bits by N (e.g. 4K entries)



In Fetch phase of branch:
– Predictor from BHT used to make prediction
• When branch completes:
– Update corresponding Predictor

Recall: BHT with 2-Bit Predictor

- BHT in 2-bit predictor is a table of “Predictors”
  – 2-bit saturating counters indexed by PC address of Branch
- Four states: 00, 01, 10, 11
  – 00,01=> predict not taken, 10, 11=> prediction taken
- Only change prediction on two successive mispredictions



Execution Pattern	T	T	N	T	T	T	N	T
Predictor value at time of prediction	0	1	2	1	2	3	3	2
Predicted branch	N	N	T	N	T	T	T	T
Prediction result in steady state (Correct, or incorrect)	I	I	I	I	C	C	I	C

Correlating or Two-level Predictors

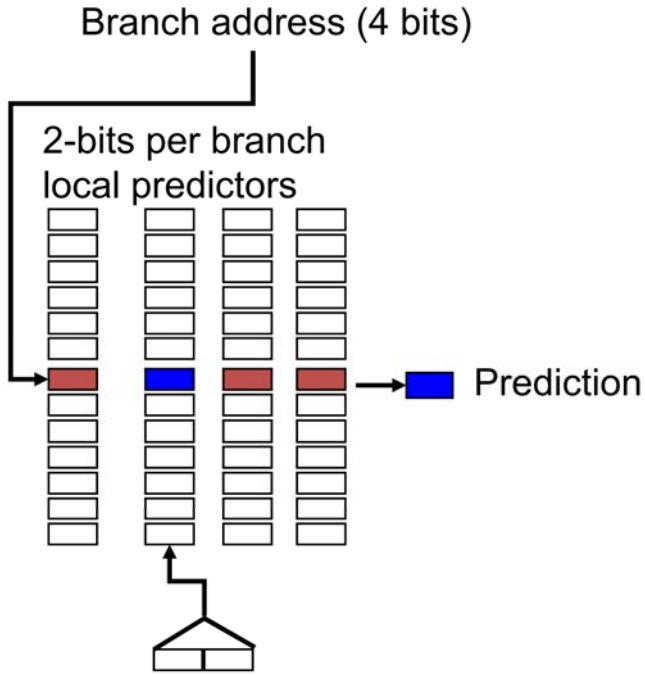
- Motivation: branches may depend on other branches, local predictor fail to catch this

```
If (aa==2)
   aa=0;
If (bb==2)
   bb=0;
If (aa!=bb)
 ....
```

(if branch 1 and branch 2 are true =>

Branch 3 is false

Branch 3 depends on the result of branch 1 and 2



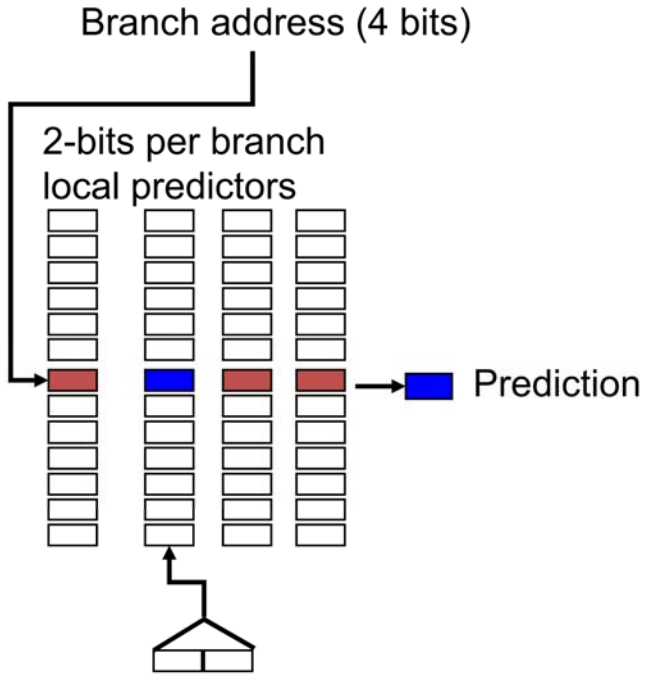
- Taken or not taken of recently executed branches is related to the behavior of present branch as well as history of branch behavior
  - by adding global information, performance improved
  - Correlating branch predictors also look at other branches for clues.

Correlating or Two-level Predictors

- Example: (m,n) =(2, 2) predictor : 2-bit global and 2-bit local predictors
- 2-bit global branch history (then there are 4 possibilities)
  - (T-T, NT-T, NT-NT, NT-T).
- For each possibility, we need to use 2-bit predictor
- Total required bit

$$2^{m} \times n \times \#\ \text{of entries selected by a branch address}$$

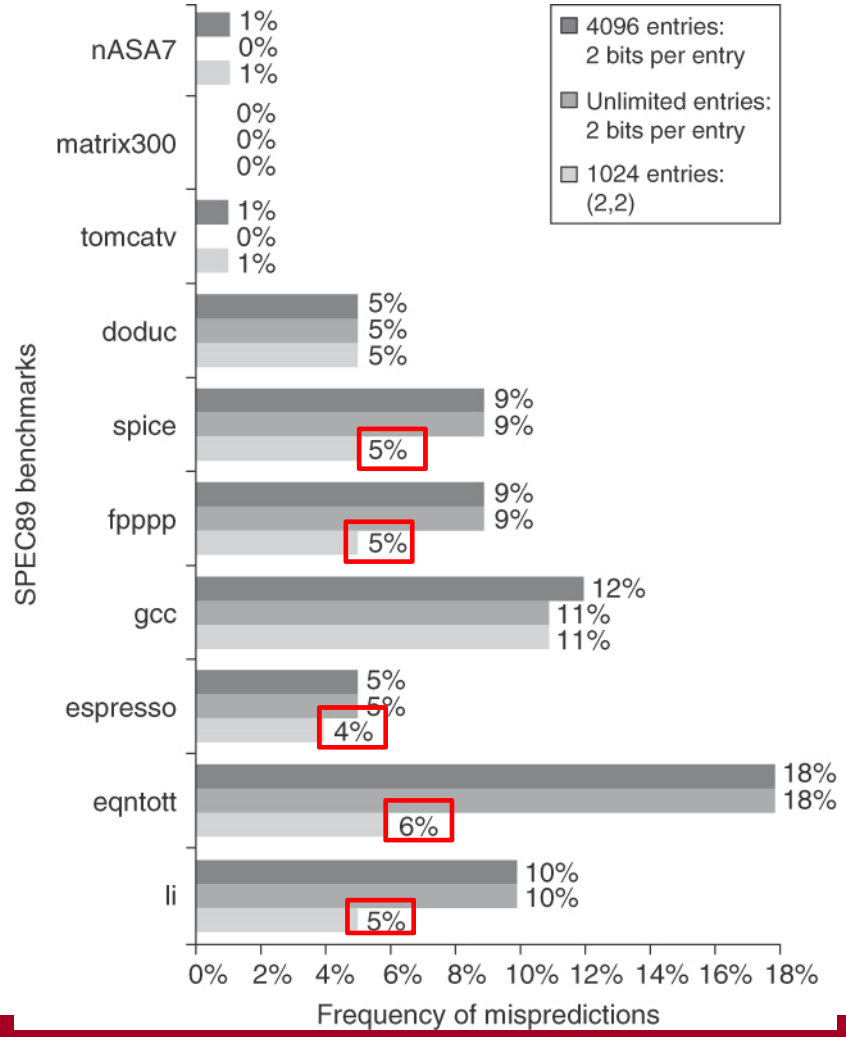
2-bit global branch *history* is an 2-*bit* shift register that records the last 2 branches encountered by the processor



Performance of Correlating Branch Prediction

- Case 1: noncorrelating predictor 2-bit preditor, 4096 entries (= 8192 bits)
- Case 2: noncorrelating 2-bit predictor with unlimited entries
- Case 3: 2-bit predictor with 2 bits of global history, 1024 entries (=8192 bits)

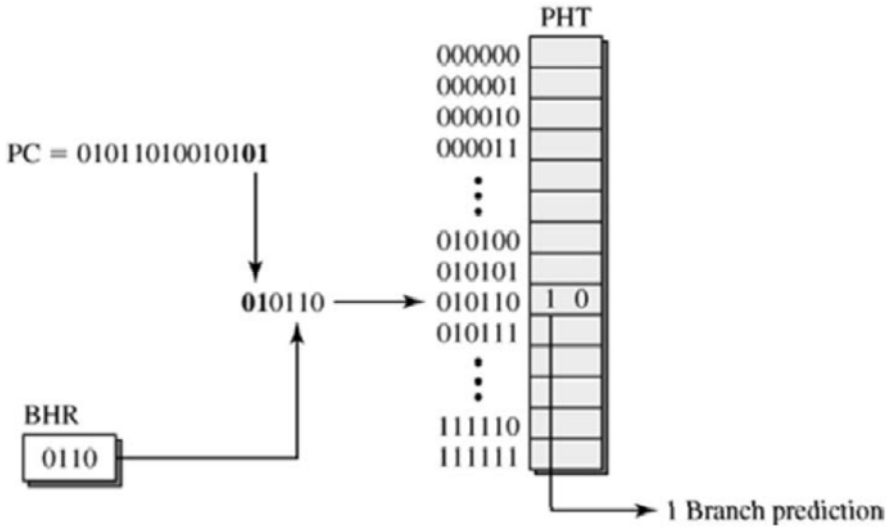
- With same number of state bits, (2,2) performs better than noncorrelating 2-bit predictor.
- Outperforms a 2-bit predictor with infinite number of entries

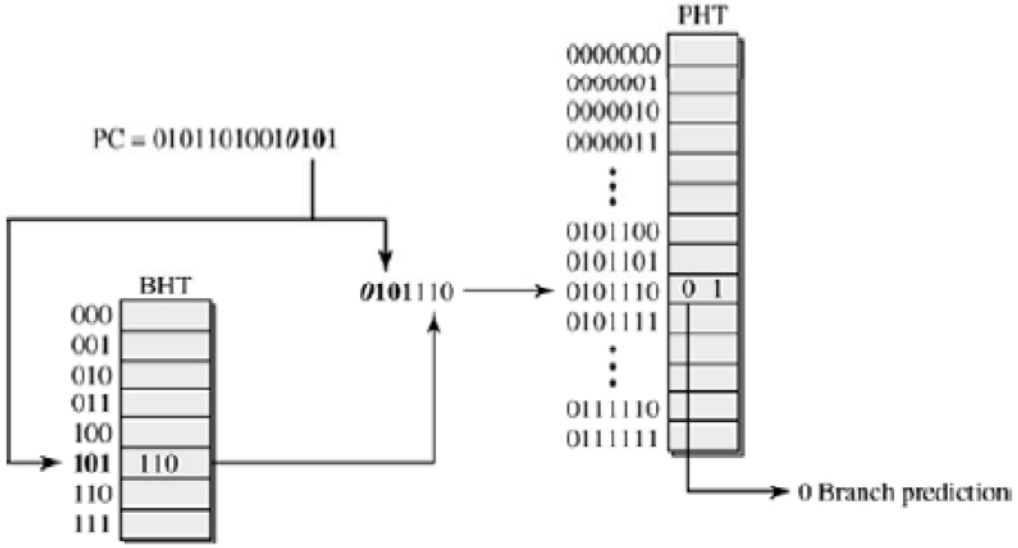


Global vs Local History Predictors

- Global:
  BHR: A shift register for global history
  -Shift in latest result in each cycle
  - provides global context

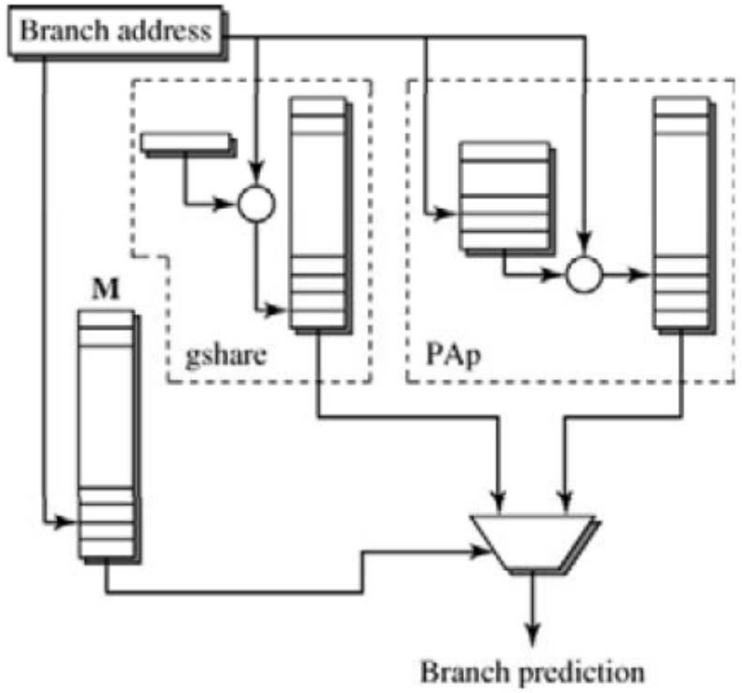
Local:
BHT: keeps track of local history
- select entry based on PC bits & shift in latest result in each cycle



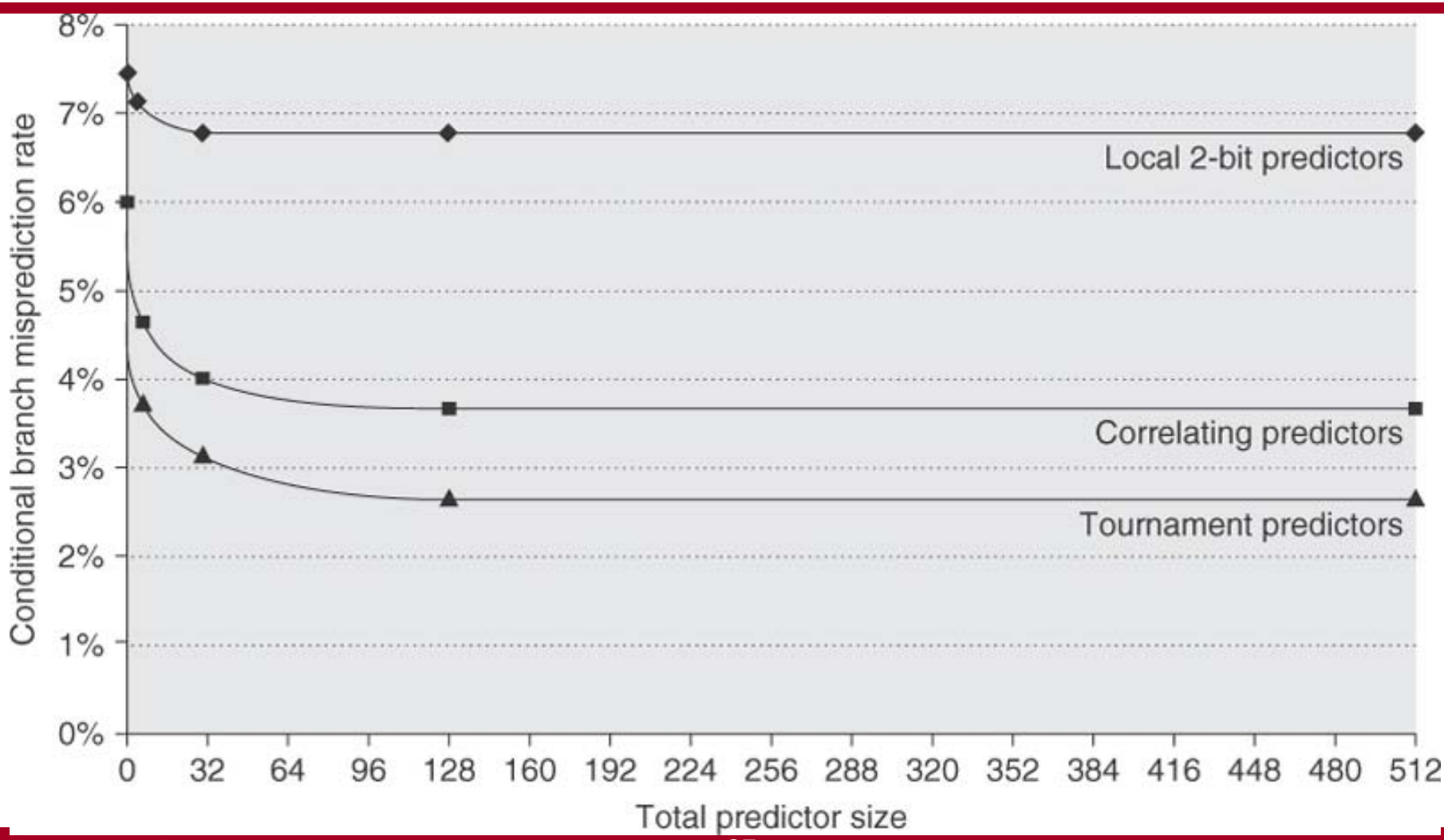


Tournament Predictors

- Tournament predictors: use two predictors, 1 based on global information and 1 based on local information, and combine with a selector
  - use two predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Hopes to select right predictor for right branch (or right context of branch)



The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased.



Backup Slides