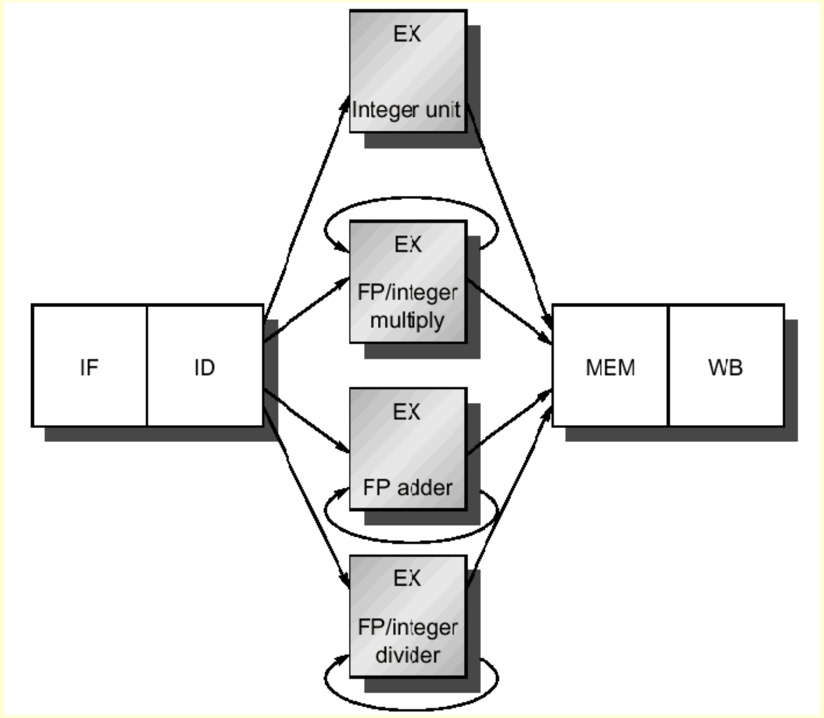
CMSC 611: Advanced Computer Architecture

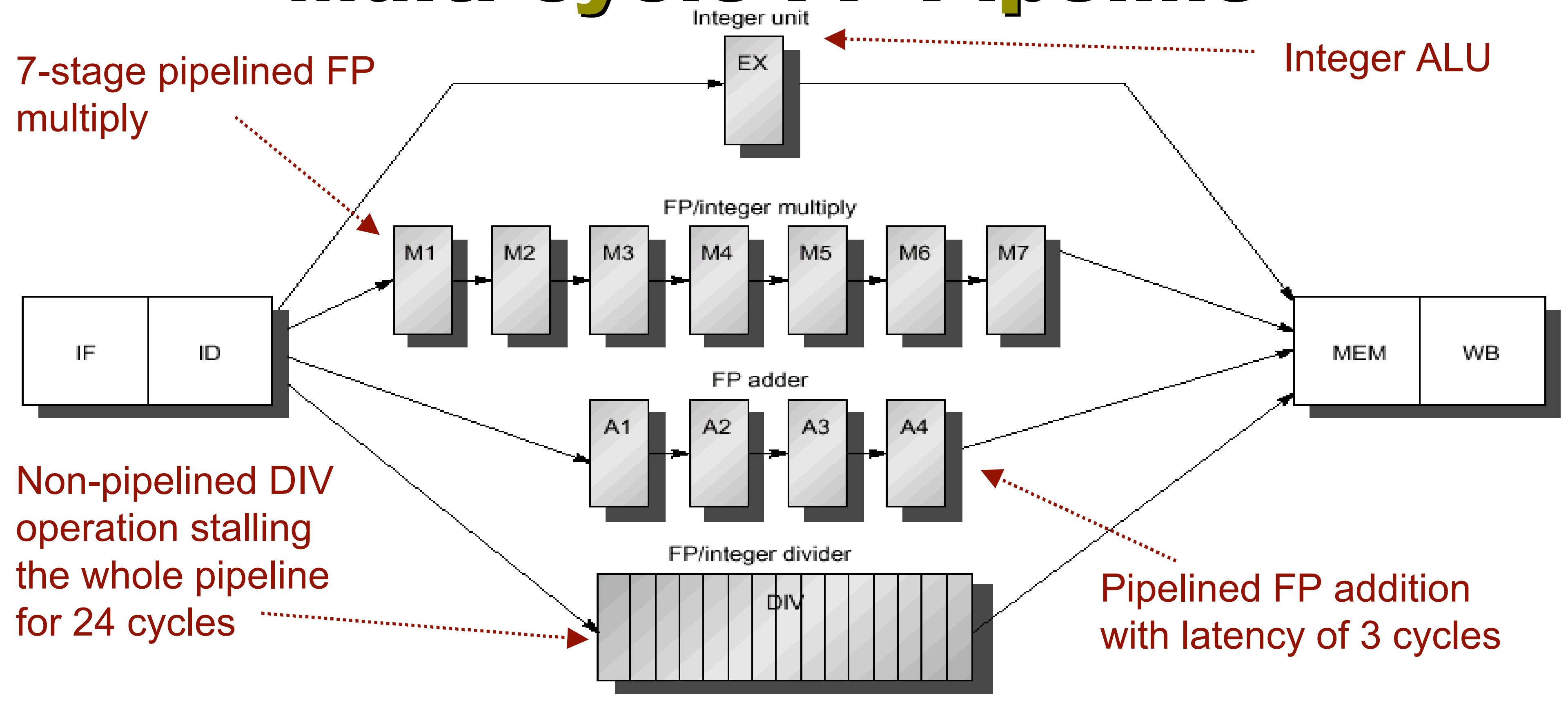
Instruction Level Parallelism

Floating-Point Pipeline

- Impractical for FP ops to complete in one clock
  - (complex logic and/or very long clock cycle)
- More complex hazards
  - Structural
  - Data



Multi-cycle FP Pipeline


Integer unit
EX
Integer ALU
7-stage pipelined FP multiply
FP/integer multiply
M1
M2
M3
M4
M5
M6
M7
IF
ID
MEM
WB
FP adder
A1
A2
A3
A4
Non-pipelined DIV operation stalling the whole pipeline for 24 cycles
FP/integer divider
DIV
Pipelined FP addition with latency of 3 cycles

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	EX	*MEM*	WB			

Example: *blue indicate where data is needed and red when result is available*

Multi-cycle FP: EX Phase

- Latency: cycles between instruction that produces result and instruction that uses it
  - Since most operations consume their operands at the beginning of the EX stage, latency is usually number of the stages of the EX an instruction uses
- Long latency increases the frequency of RAW hazards
- Initiation (Repeat) interval: cycles between issuing two operations of a given type

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

FP Pipeline Challenges

- Non-pipelined divide causes structural hazards
- Number of register writes required in a cycle can be larger than 1
- WAW hazards are possible
  - Instructions no longer reach WB in order
- WAR hazards are **NOT** possible
  - Register reads are still taking place during the ID stage
- Instructions can complete out of order
  - Complicates exceptions
- Longer latency makes RAW stalls more frequent

Instruction	Clock cycle number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4, 0(R2)	IF	ID	EX	MEM	WB												
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
SD 0(R2), F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Example of RAW hazard caused by the long latency

WB Structural Hazard

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
LD F2, 0(R2)							IF	ID	EX	MEM	WB

- At cycle 11, the MULTD, ADDD and LD instructions will try to write back
  - structural hazard if there is only one write port
- Additional write ports are not cost effective since they are rarely used
- Instead
  - Detect at ID and stall
  - Detect at MEM or WB and stall

WAW Data Hazards

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
LD F2, 0(R2)						IF	ID	EX	MEM	WB	
….							IF	ID	EX	MEM	WB

- WAW hazards can be corrected by either:
  - Stalling the latter instruction at MEM until it is safe
  - Preventing the first instruction from overwriting the register
- Correcting at cycle 11 OK unless intervening RAW/use of F2
- WAW hazards can be detected at the ID stage
  - Convert 1st instruction to no-op
- WAW hazards are generally very rare, designers usually go with the simplest solution

Detecting Hazards

- Hazards among FP instructions & and combined FP and integer instructions
- Separate int & fp register files limits latter to FP load and store instructions
- Assuming all checks are to be performed in the ID phase:
  – Check for structural hazards:
    - Wait if the functional unit is busy (Divides in our case)
    - Make sure the register write port is available when needed
  – Check for a RAW data hazard
    - Requires knowledge of latency and initiation interval to decide when to forward and when to stall
  – Check for a WAW data hazard
    - Write completion has to be estimated at the ID stage to check with other instructions in the pipeline
- Data hazard detection and forwarding logic from values stored between the stages

Maintaining Precise Exceptions

- Pipelining FP instructions can cause out-of-order completion
- Exceptions also a problem:

```
DIVF    F0, F2, F4
ADDF    F10, F10, F8
SUBF    F12, F12, F14
```

  - No data hazards
  - What if DIVF exception occurs after ADDF writes F10?

Four FP Exception Solutions

1. Settle for imprecise exceptions
   - Some supercomputers still uses this approach
   - IEEE floating point standard requires precise exceptions
   - Some machine offer slow precise and fast imprecise exceptions
2. Buffer the results of all operations until previous instructions complete
   - Complex and expensive design (many comparators and large MUX)
   - History or future register file

Four FP Exception Solutions

3. Allow imprecise exceptions and get the handler to clean up any miss
   - Save PC + state about the interrupting instruction and all out-of-order completed instructions
   - The trap handler will consider the state modification caused by finished instructions and prepare machine to resume correctly
   - Issues: consider the following example

     Instruction1: Long running, eventual exception

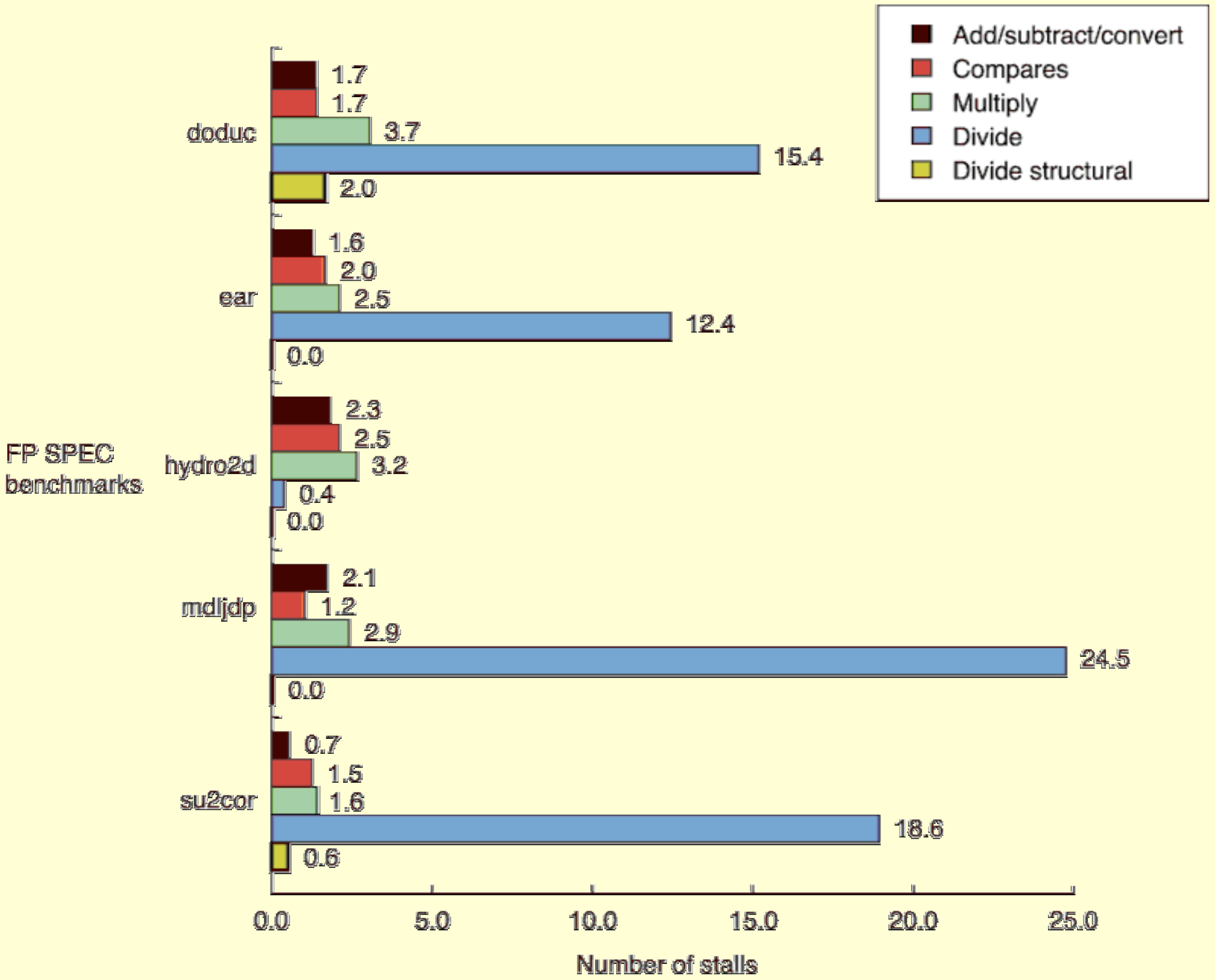
     Instructions 2 … (n-1) : Instructions that do not complete

     Instruction n : An instruction that is finished
   - The compiler can simplify the problem by grouping FP instructions so that the trap does not have to worry about unrelated instructions

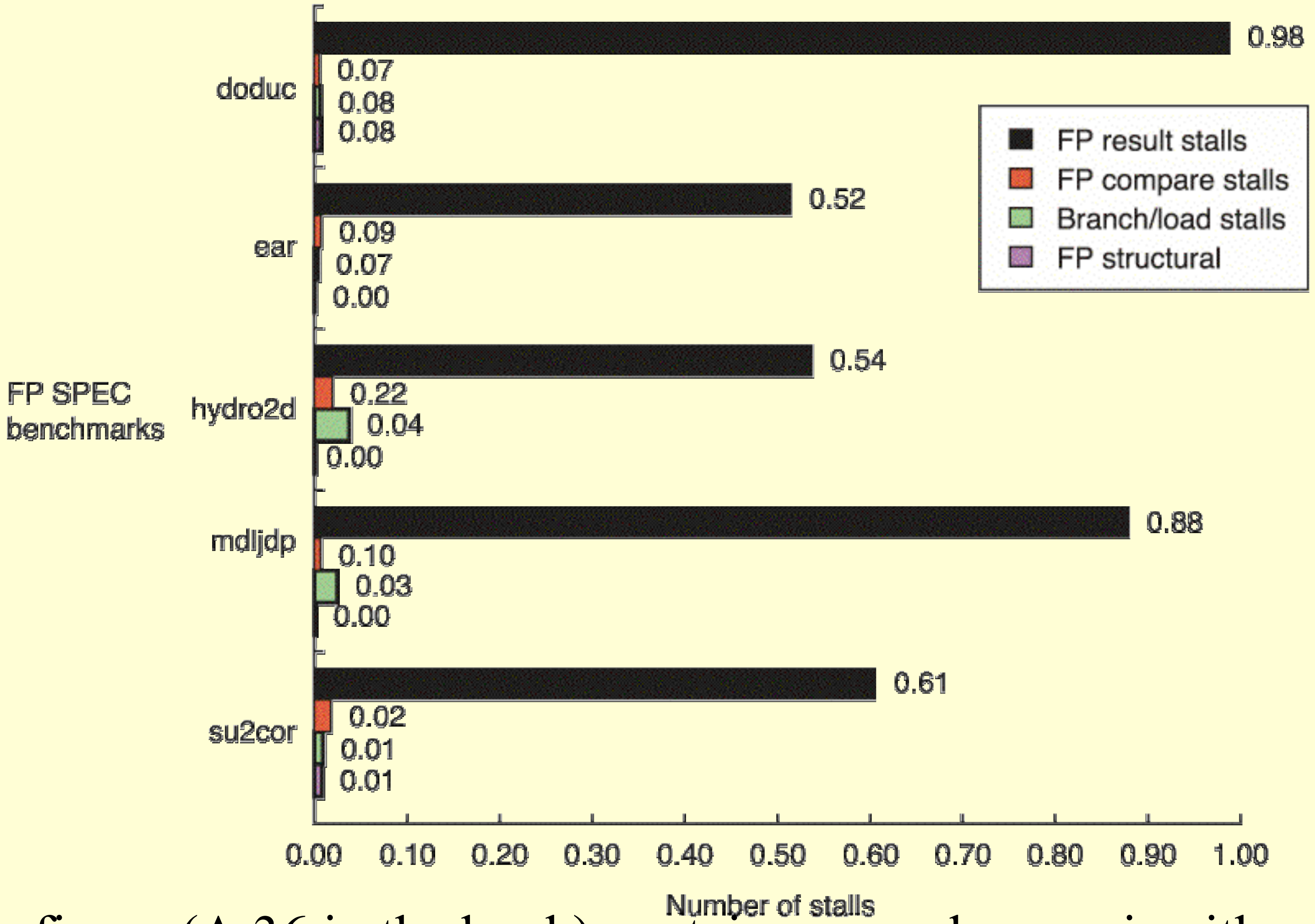
Four FP Exception Solutions

4. Allow instruction issue to continue only if previous instruction are guaranteed to cause no exceptions:
   - Mainly applied in the execution phase
   - Used on MIPS R4000 and Intel Pentium

Stalls/Instruction, FP Pipeline



More FP Pipeline Performance



This figure (A.36 in the book) contains several errors in either graph or data. Only take-home: result stalls are most common by far

Instruction Level Parallelism (ILP)

- Overlap the execution of unrelated instructions
- Both instruction pipelining and ILP enhance instruction throughput not the execution time of the individual instruction
- Potential of IPL within a basic block is very limited
  - in “gcc” 17% of instructions are control transfer meaning on average 5 instructions per branch

Loops: Simple & Common

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

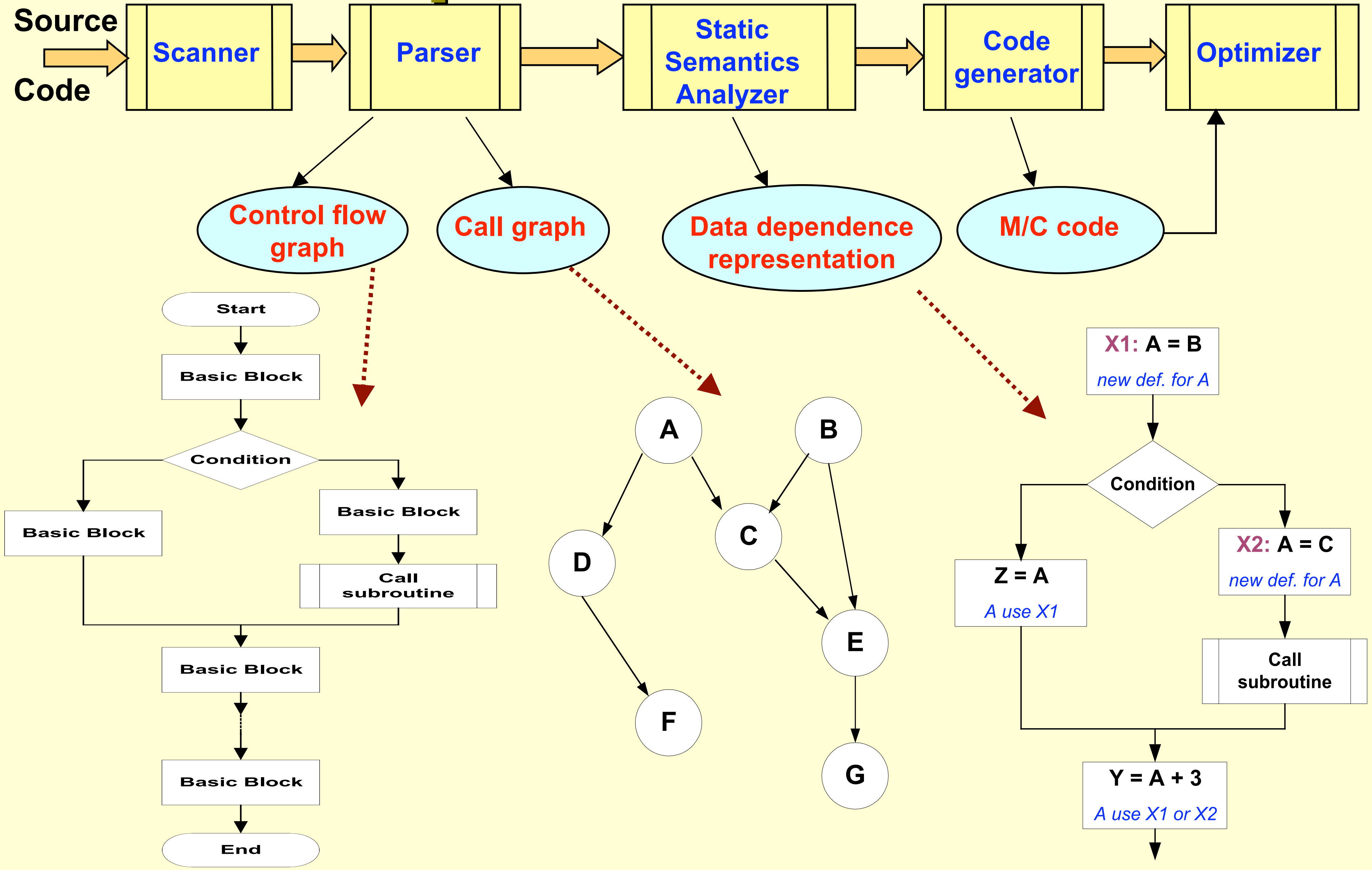
- Techniques like loop unrolling convert loop-level parallelism into instruction-level parallelism
  - statically by the compiler
  - dynamically by hardware
- Loop-level parallelism can also be exploited using vector processing
- IPL feasibility is mainly hindered by data and control dependence among the basic blocks
- Level of parallelism is limited by instruction latencies

Major Assumptions

- Basic MIPS integer pipeline
- Branches with one delay cycle
- Functional units are fully pipelined or replicated (as many times as the pipeline depth)
  - An operation of any type can be issued on every clock cycle and there are no structural hazard

Instruction producing result	Instruction using results	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

Compilation Overview



Motivating Example

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

```
Loop:  LD      F0,x(R1)      ;F0=vector element
       ADDD    F4,F0,F2      ;add scalar from F2
       SD      x(R1),F4      ;store result
       SUBI    R1,R1,8       ;decrement pointer (DW)
       BNEZ    R1,Loop       ;branch R1!=zero
```

Standard Pipeline execution

```
Loop:  LD      F0,x(R1)
       stall
       ADDD    F4,F0,F2
       stall
       stall
       SD      x(R1),F4
       SUBI    R1,R1,8
       stall
       BNEZ    R1,Loop
       stall
```

Smart compiler

```
Loop:  LD      F0,x(R1)
       SUBI    R1,R1,8
       ADDD    F4,F0,F2
       stall   ;F4 latency
       BNEZ    R1,Loop
       SD      x+8(R1),F4
```

Sophisticated compiler optimization reduced execution time from 10 cycles to only 6 cycles

Loop Unrolling

```
Loop:  LD     F0,x(R1)
       ADDD   F4,F0,F2
       SD     x(R1),F4
       SUBI   R1,R1,8
       BNEZ   R1,Loop
```

Replicate loop body 4 times, will need cleanup phase if loop iteration is not a multiple of 4

- 6 cycles, but only 3 are loop body
- Loop unrolling limits overhead at the expense of a larger code
  - Eliminates branch delays
  - Enable effective scheduling
- Use of different registers needed to limit data hazard

```
Loop:  LD     F0,x(R1)
       ADDD   F4,F0,F2
       SD     x(R1),F4     ;drop SUBI & BNEZ
       LD     F6,x-8(R1)
       ADDD   F8,F6,F2
       SD     x-8(R1),F8     ;drop again
       LD     F10,x-16(R1)
       ADDD   F12,F10,F2
       SD     x-16(R1),F12;drop again
       LD     F14,x-24(R1)
       ADDD   F16,F14,F2
       SD     x-24(R1),F16
       SUBI   R1,R1,#32    ;alter to 4*8
       BNEZ   R1,LOOP
```

Scheduling Unrolled Loops

Cycle		Instruction	
1	Loop:	LD	F0,x(R1)
3		ADDD	F4,F0,F2
6		SD	x(R1),F4
7		LD	F6,x-8(R1)
9		ADDD	F8,F6,F2
12		SD	x-8(R1),F8
13		LD	F10,x-16(R1)
15		ADDD	F12,F10,F2
18		SD	x-16(R1),F12
19		LD	F14,x-24(R1)
21		ADDD	F16,F14,F2
24		SD	x-24(R1),F16
25		SUBI	R1,R1,#32
27		BNEZ	R1,LOOP
28		stall	

Loop unrolling exposes more computation that can be scheduled to minimize the pipeline stalls

Understanding dependence among instructions is the key for for detecting and performing the transformation

Cycle		Instruction	
1	Loop:	LD	F0,x(R1)
2		LD	F6,x-8(R1)
3		LD	F10,x-16(R1)
4		LD	F14,x-24(R1)
5		ADDD	F4,F0,F2
6		ADDD	F8,F6,F2
7		ADDD	F12,F10,F2
8		ADDD	F16,F14,F
9		SD	x(R1),F4
10		SD	x-8(R1),F8
11		SUBI	R1,R1,#32
12		SD	x-16(R1),F12
13		BNEZ	R1,LOOP
14		SD	x+8(R1),F1

Inter-instruction Dependence

- Determining how one instruction depends on another is critical not only to the scheduling process but also to determining how much parallelism exists
- If two instructions are parallel they can execute simultaneously in the pipeline without causing stalls (assuming there is not structural hazard)
- Two instructions that are dependent are not parallel and their execution cannot be reordered

Dependence Classifications

- Data dependence (RAW)
  - Transitive: i → j → k = i → k
  - Easy to determine for registers, hard for memory
    - Does 100(R4) = 20(R6)?
    - From different loop iterations, does 20(R6) = 20(R6)?
- Name dependence (register/memory reuse)
  - Anti-dependence (WAR): Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
  - Output dependence (WAW): Instructions i and j write the same register or memory location; instruction ordering must be preserved
- Control dependence, caused by conditional branching

Example: Name Dependence

```
Loop: LD    F0,x(R1)
      ADDD  F4,F0,F2
      SD    x(R1),F4
      LD    F0,x-8(R1)
      ADDD  F4,F0,F2
      SD    x-8(R1),F4
      LD    F0,x-16(R1)
      ADDD  F4,F0,F2
      SD    x-16(R1),F4
      LD    F0,x-24(R1)
      ADDD  F4,F0,F2
      SD    x-24(R1),F4
      SUBI  R1,R1,#32
      BNEZ  R1,Loop
```

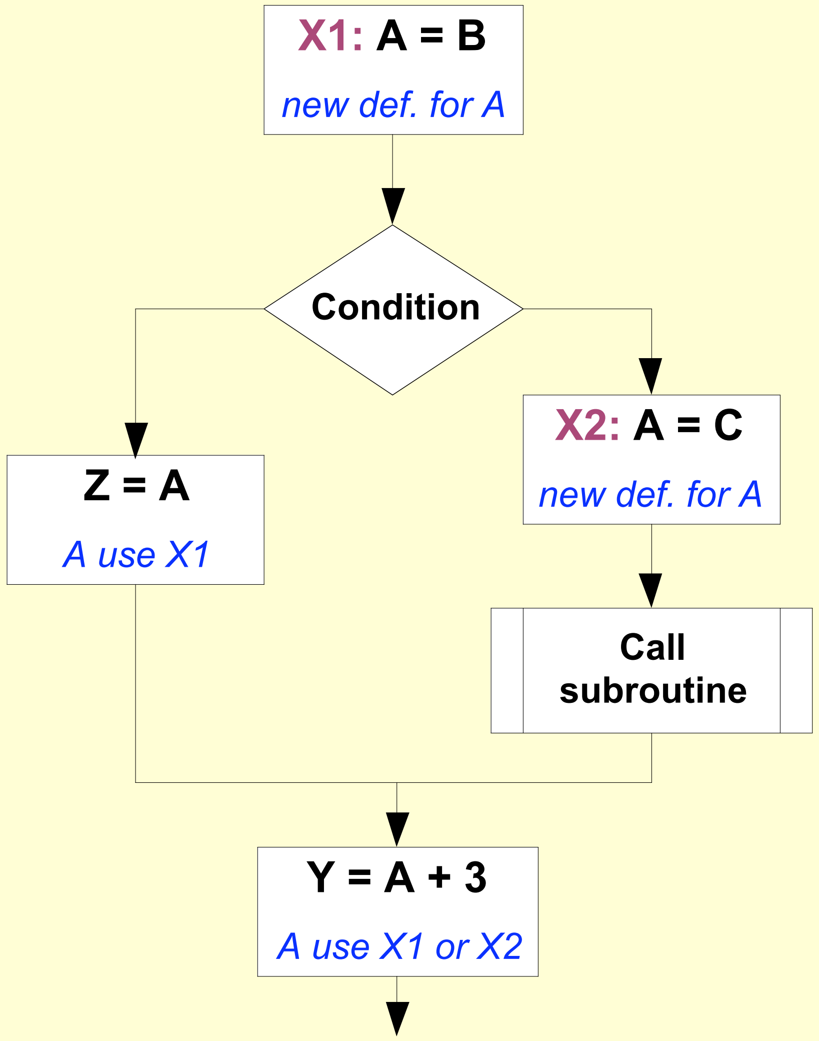
Register renaming

```
Loop: LD    F0,x(R1)
      ADDD  F4,F0,F2
      SD    x(R1),F4
      LD    F6,x-8(R1)
      ADDD  F8,F6,F2
      SD    x-8(R1),F8
      LD    F10,x-16(R1)
      ADDD  F12,F10,F2
      SD    x-16(R1),F12
      LD    F14,x-24(R1)
      ADDD  F16,F14,F2
      SD    x-24(R1),F16
      SUBI  R1,R1,#32
      BNEZ  R1,LOOP
```

- Again Name Dependencies are Hard for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Compiler needs to know that R1 does not change → *0(R1)≠ -8(R1)≠ -16(R1)≠ -24(R1)* and thus no dependencies between some loads and stores so they could be moved

Control Dependence

- Control flow of a program is enforced by conditional branching
- Two constraints on control dependences:
  - An instruction control dependent on a branch cannot be moved before the branch
  - An instruction not control dependent on a branch cannot be moved after the branch
- Observing control dependence is not a must
  - But preserve program correctness in exception behavior and data flow



Preserving Correctness

- Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised

```
BEQZ R2, L1          LW    R1, 0(R2)
LW   R1, 0(R2)       BEQZ  R2, L1
```

  - LW may cause memory protection exception
- Data flow reflects changes to variables (registers and memory) throughout the program

Loop-level Parallelism

- Loop-level parallelism is normally analyzed at the source code level or close to it while most analysis of IPL after code generation
- Loop level analysis involves determining what dependences exist among the operands in the loop across iterations of the loop (bulk is data dependence)

```
for (i=1000; i>0; i=i-1)
   x[i] = x[i] + s;
```

} all dependence are within same iteration and thus loop is parallel

Loop-level Parallelism

- Loop-carried dependence is caused by data dependence between operands modified in consecutive iterations

Example: Assuming A,B,C are distinct and non-overlapping

```
        for (i=1; i<=100; i=i+1) {
            A[i+1] = A[i] + C[i];       /* S1 */
            B[i+1] = B[i] + A[i+1];}    /* S2 */
```

  - S2 uses the value, A[i+1], computed by S1 in the same iteration
  - S1 uses a value computed by S1 in an earlier iteration
- Loop-carried dependence can be efficiently handled by loop unrolling

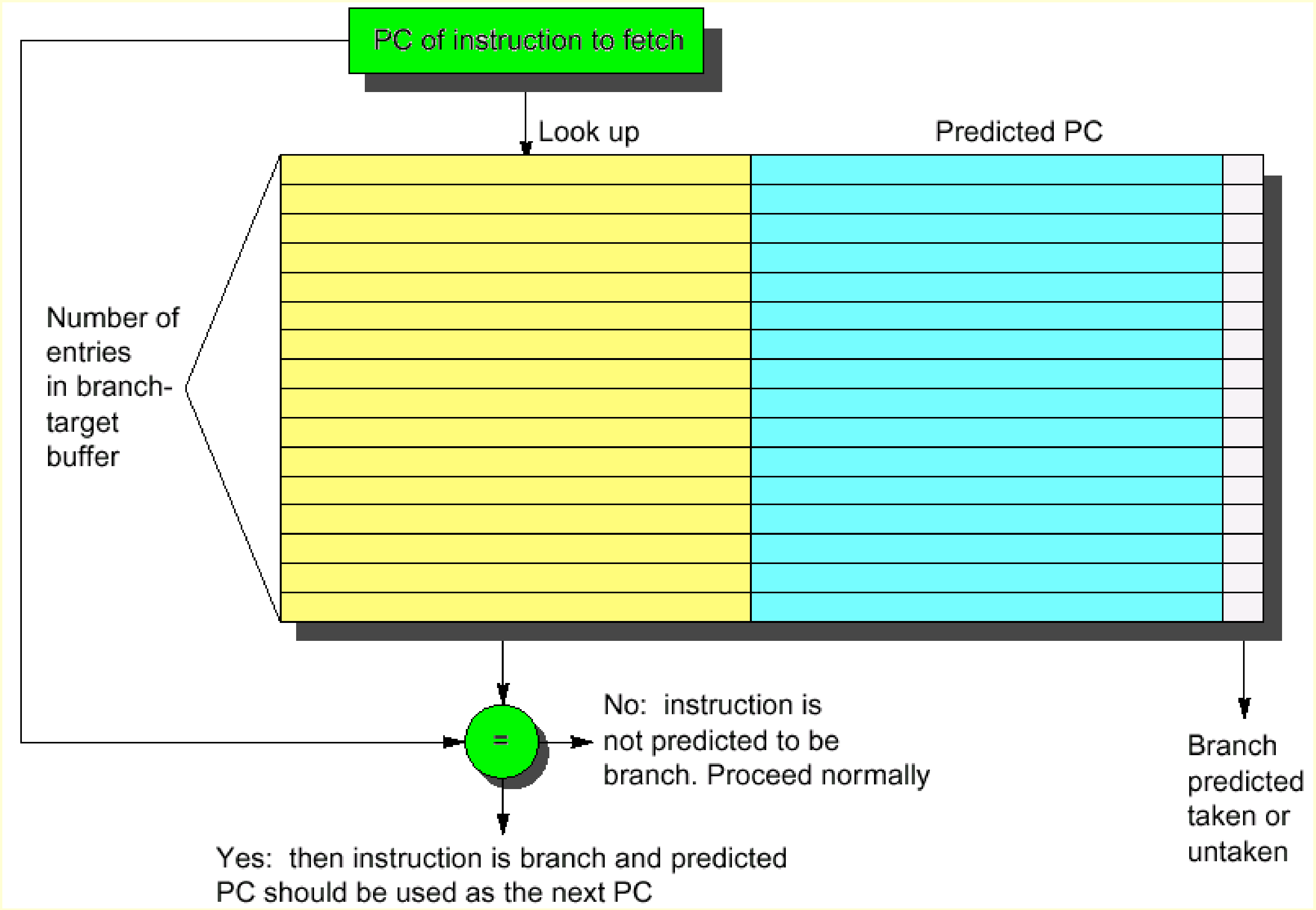
Branching Dilema

- With the increased pipeline throughput, control dependence rapidly becomes the limiting factor to the amount of ILP
- For pipelines that issue n-instructions per clock cycle, the negative impact of stalls caused by control hazards magnifies
- Compiler-based techniques rely on static program properties to handle control hazards
- Hardware-based techniques refer to the dynamic behavior of the program to predict the outcome of a branch

Branch Target Cache

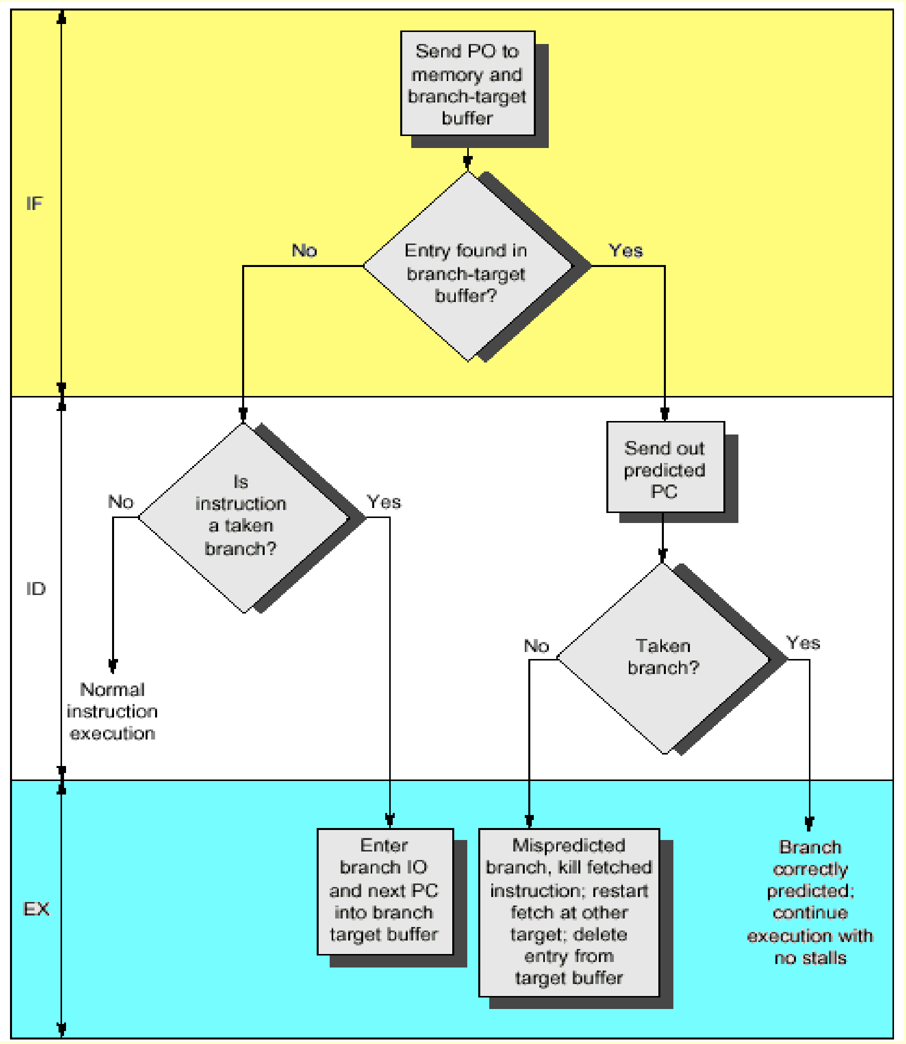
- Predict not-taken: still stalls to wait for branch target computation
- If address could be guessed, the branch penalty becomes zero
- Cache predicted address based on branch address
- Complications for complex predictors: do we know in time?

Branch Target Cache


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch. Proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

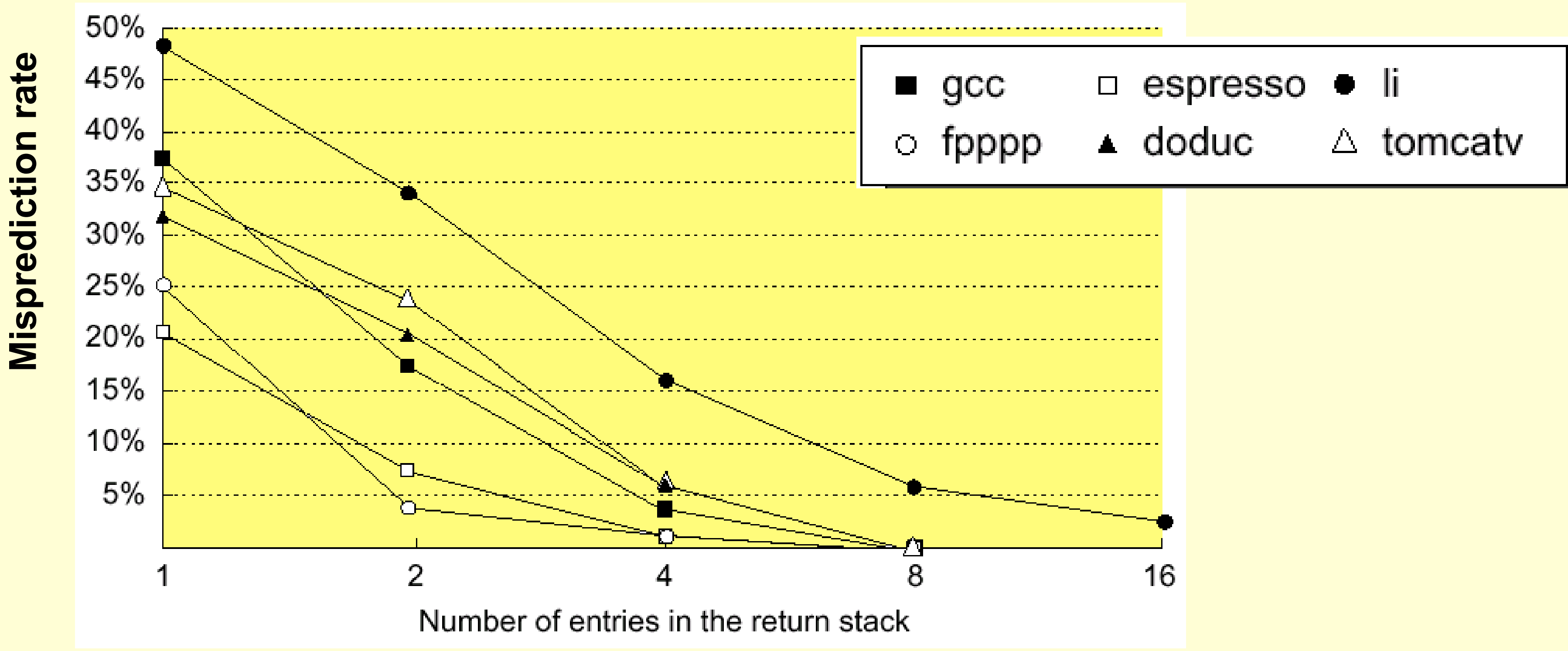
Handling Branch Target Cache

- No branch delay if the a branch prediction entry is found and is correct
- A penalty of two cycle is imposed for a wrong prediction or a cache miss
- Cache update on misprediction and misses can extend the time penalty
- Dealing with misses or misprediction is expensive and should be optimized


IF
ID
EX
Send PO to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Send out predicted PC
Normal instruction execution
Taken branch?
No
Yes
Enter branch IO and next PC into branch target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Return Address Cache

- Branch target caching can be applied to expedite unconditional jumps (branch folding) and returns for procedure calls
- For calls from multiple sites, not clustered in time, a stack implementation of the branch target cache can be useful

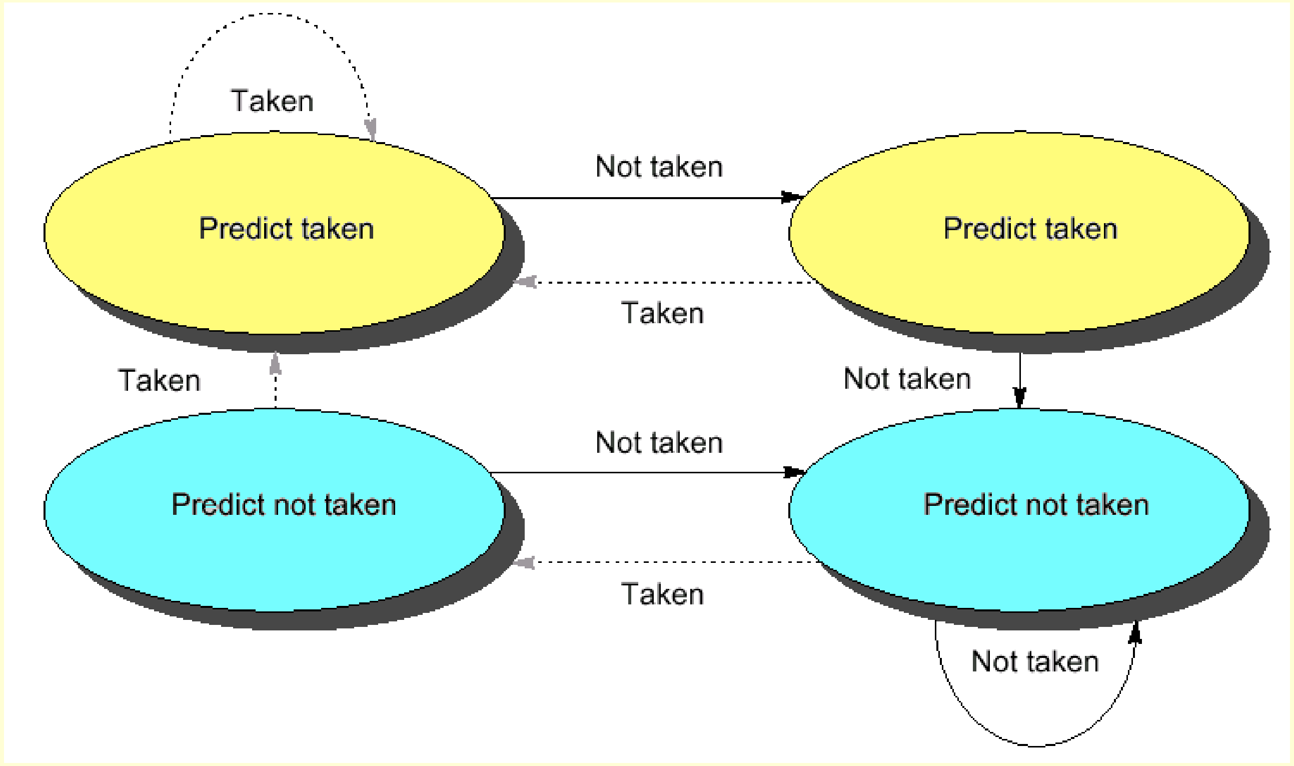


Basic Branch Prediction

- Simplest dynamic branch-prediction scheme
  - use a branch history table to track when the branch was taken and not taken
  - Branch history table is a small 1-bit buffer indexed by lower bits of PC address with the bit is set to reflect the whether or not branch taken last time
- Performance = f(accuracy, cost of misprediction)
- Problem: in a loop, 1-bit branch history table will cause two mispredictions:
  - End of loop case, when it exits instead of looping
  - First time through loop on next time through code, when it predicts exit instead of looping

2-bit Branch History Table

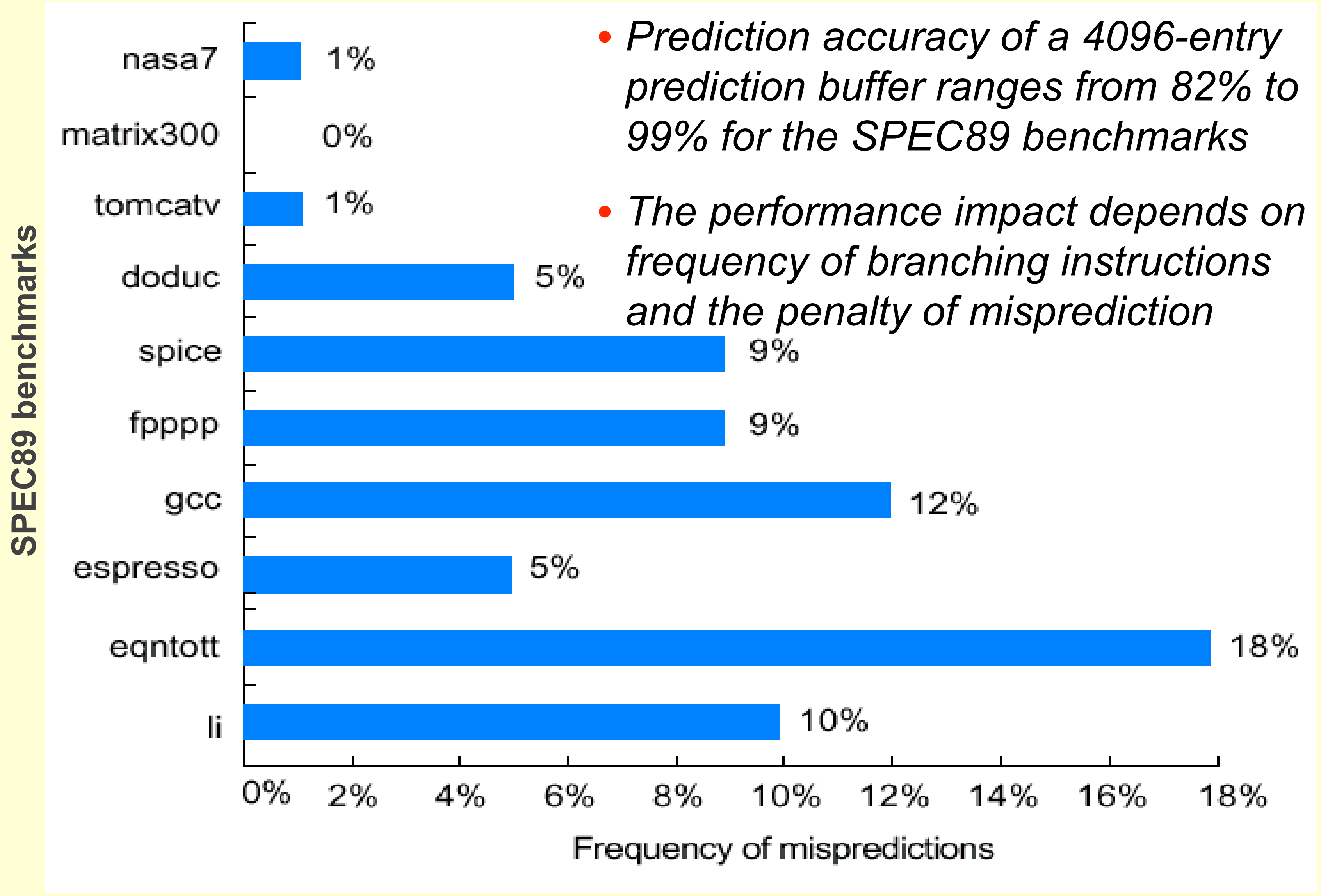
- A two-bit buffer better captures the history of the branch instruction
- A prediction must miss twice to change



N-bit Predictors

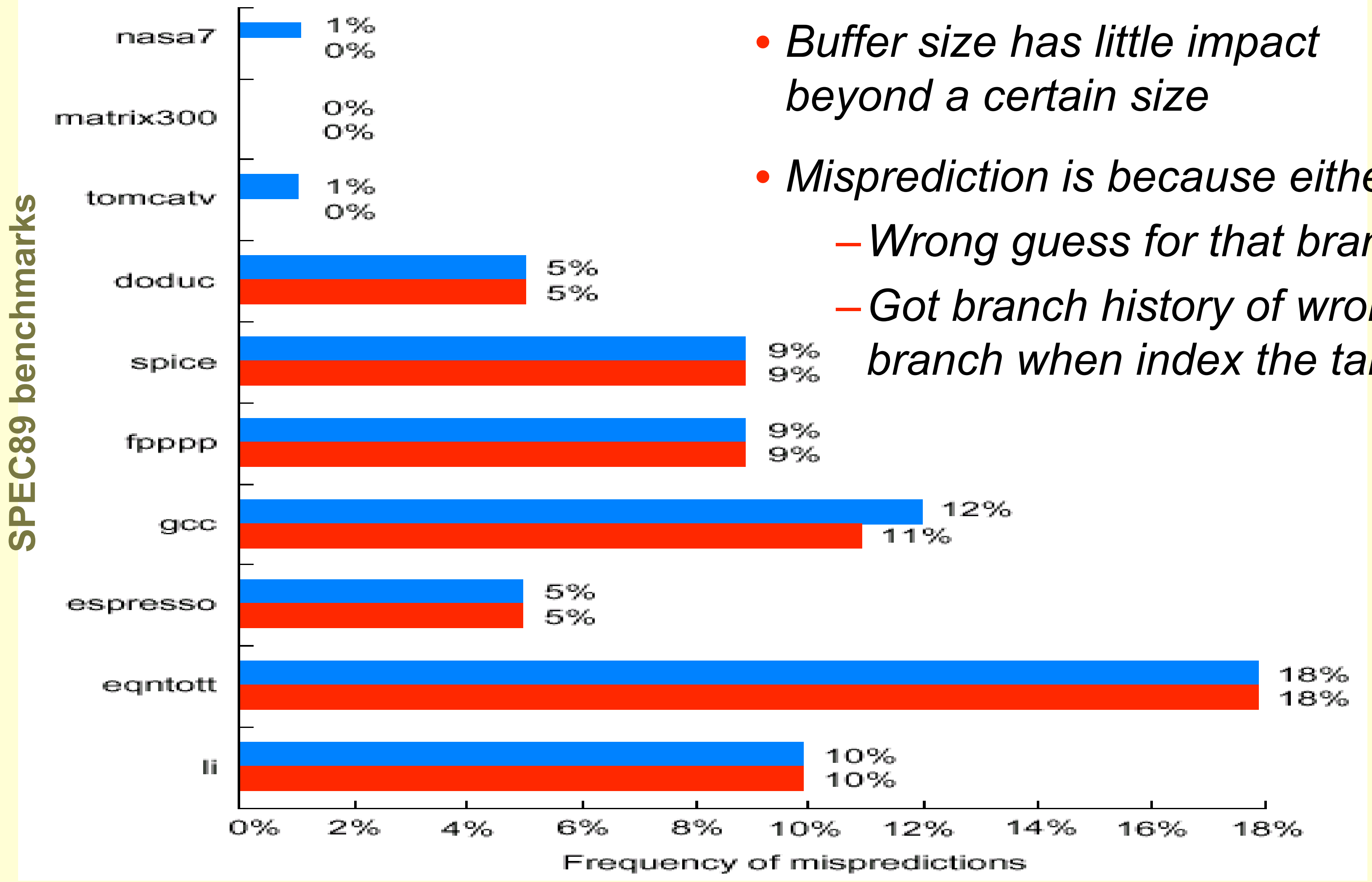
- 2-bit is a special case of n-bit counter
  - For every entry in the prediction buffer
  - Increment/decrement if branch taken/not
  - If the counter value is one half of the maximum value (2n-1), predict taken
- Slow to change prediction, but can change

Performance of 2-bit Branch Buffer



- *Prediction accuracy of a 4096-entry prediction buffer ranges from 82% to 99% for the SPEC89 benchmarks*
- *The performance impact depends on frequency of branching instructions and the penalty of misprediction*

Optimal Size for 2-bit Branch Buffers



- *Buffer size has little impact beyond a certain size*
- *Misprediction is because either:*
  - *Wrong guess for that branch*
  - *Got branch history of wrong branch when index the table*

🟦 **4096 entries (2 bits/entry)** 🟥 **Unlimited entries (2 bits/entry)**

Correlating Predictors

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

→

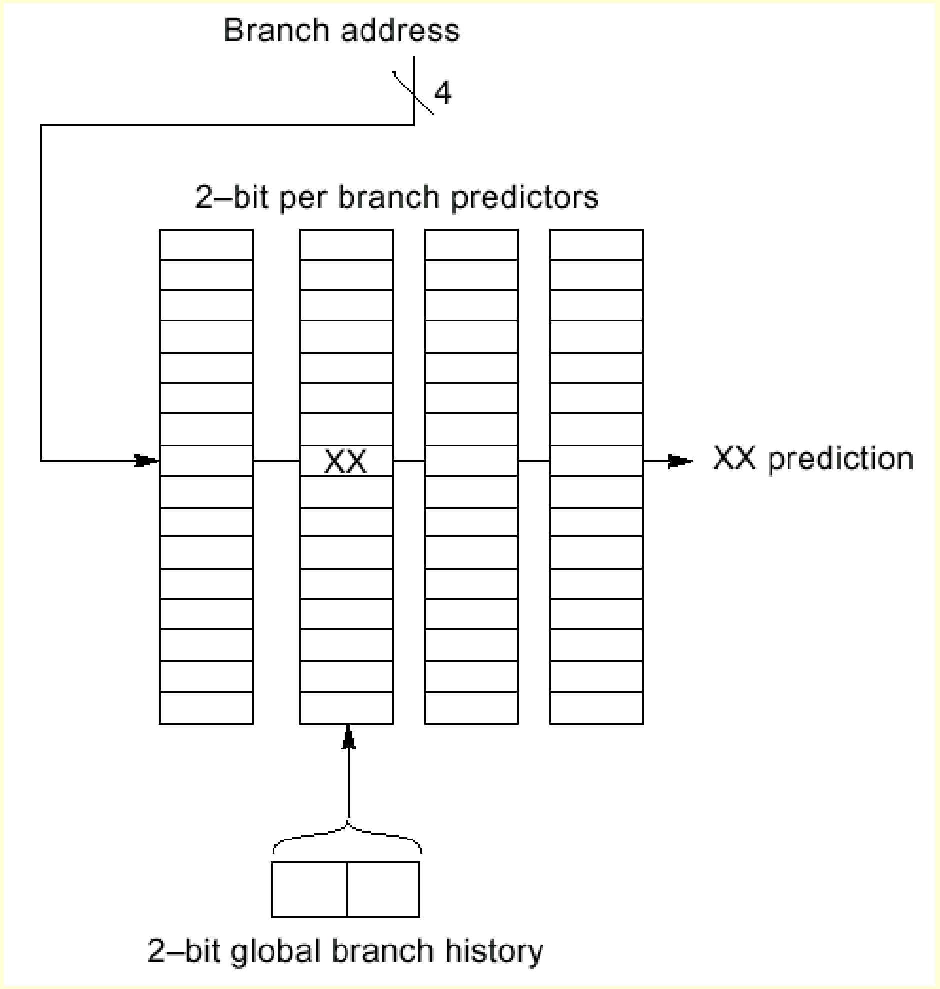
```
     DSUBUI   R3, R1, #2
     BNEZ     R3, L1        ; branch b1 (aa!=2)
     ANDI     R1, R1, #0    ; aa=0
L1:  SUBUI    R3, R2, #2
     BNEZ     R3, L2        ; branch b2 (bb!=2)
     ANDI     R2, R2, #0    ; bb=0
L2:  SUBU     R3, R1, R2    ; R3=aa-bb
     BEQZ     R3, L3        ; branch b3 (aa==bb)
```

- The behavior of branch b3 is correlated with the behavior of b1 and b2
- Clearly of both branches b1 and b2 are untaken, then b3 will be taken
- A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior
- Branch predictors that use the behavior of other branches to make a prediction are called correlating or two-level predictors

Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch

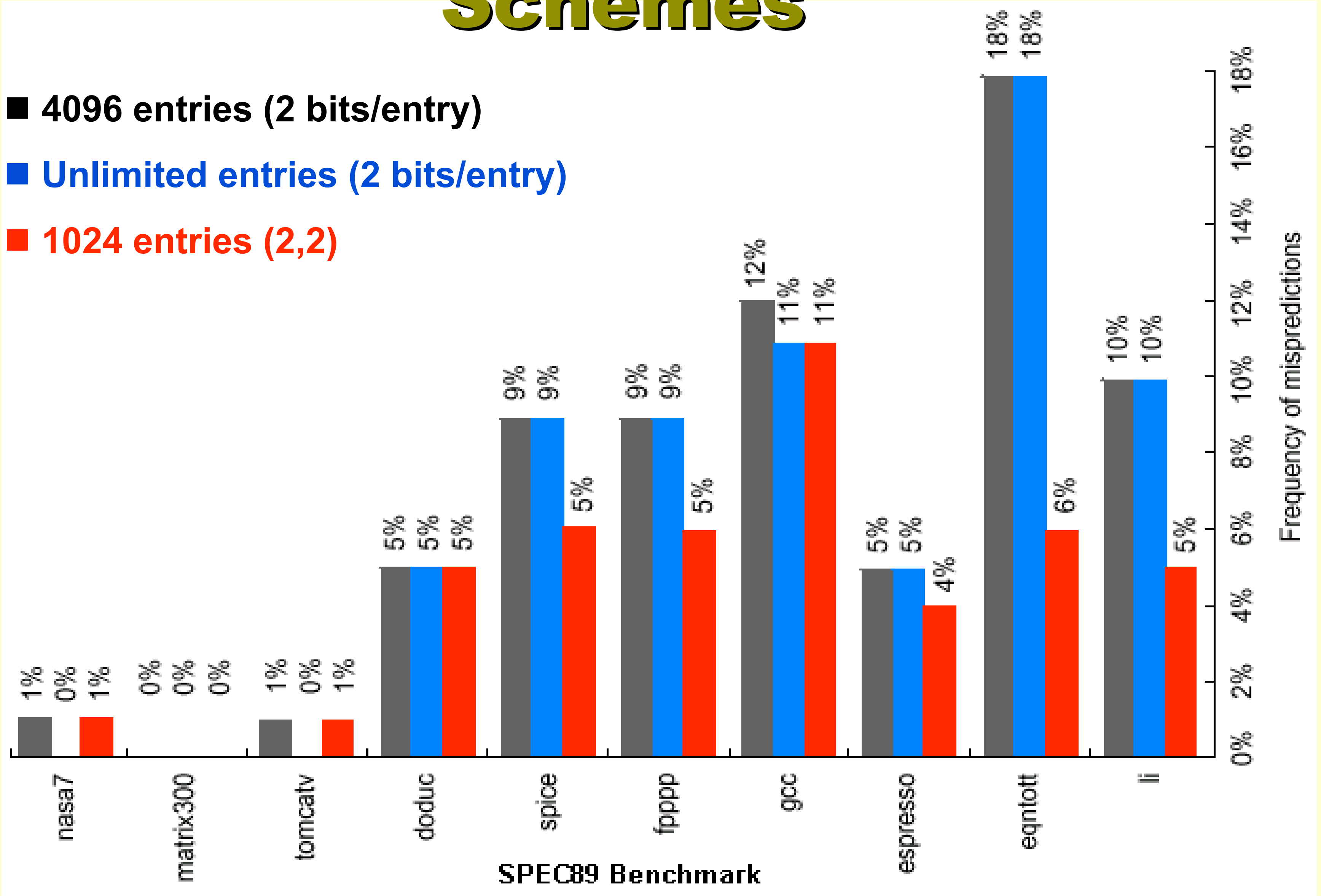
(2,2) Correlating Predictors

- Record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
- (m,n) predictor means record last m branches to select between 2m history tables each with n-bit counters
  - Old 2-bit branch history table is a (0,2) predictor
- In a (2,2) predictor, the behavior of recent branches selects between, four predictions of next branch, updating just that prediction



Total size = 2^m □ n □ # prediction entries selected by branch address

Accuracy of Different Schemes



Example

- Assume that d has values 0, 1, or 2 (alternating between 0, 2)
- Assume that the sequence will be executed repeatedly
- Ignore all other branches including those causing the sequence to repeat
- All branches are initially predicted to untaken state

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

```
        BNEZ      R1, L1          ; branch b1 (d!=0)
        DADDI     R1, R0, #1      ; d==0, sp d=1
L1:     DSUBUI    R3, R1, #1
        BNEZ      R3, L2          ; branch b2 (d!=1)
….
L2:
```

Example

With a single bit predictor

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 branches are mispredicted*

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

$\Rightarrow$

```
      BNEZ     R1, L1          ; branch b1 (d!=0)
      DADDI    R1, R0, #1      ; d==0, sp d=1
L1:   DSUBUI   R3, R1, #1
      BNEZ     R3, L2          ; branch b2 (d!=1)
….
L2:
```

Example

With one bit predictor with one bit of correlation

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

- *Except for first iteration, all branches are correctly predicted*

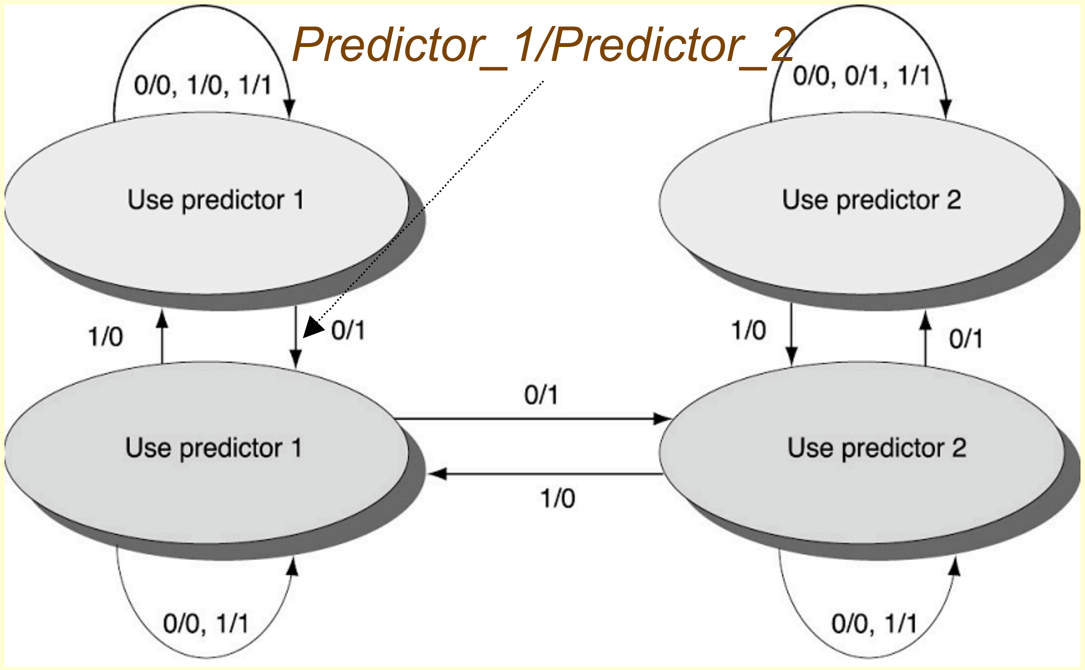
```
if (d==0)
   d=1;
if (d==1)
```

$\Rightarrow$

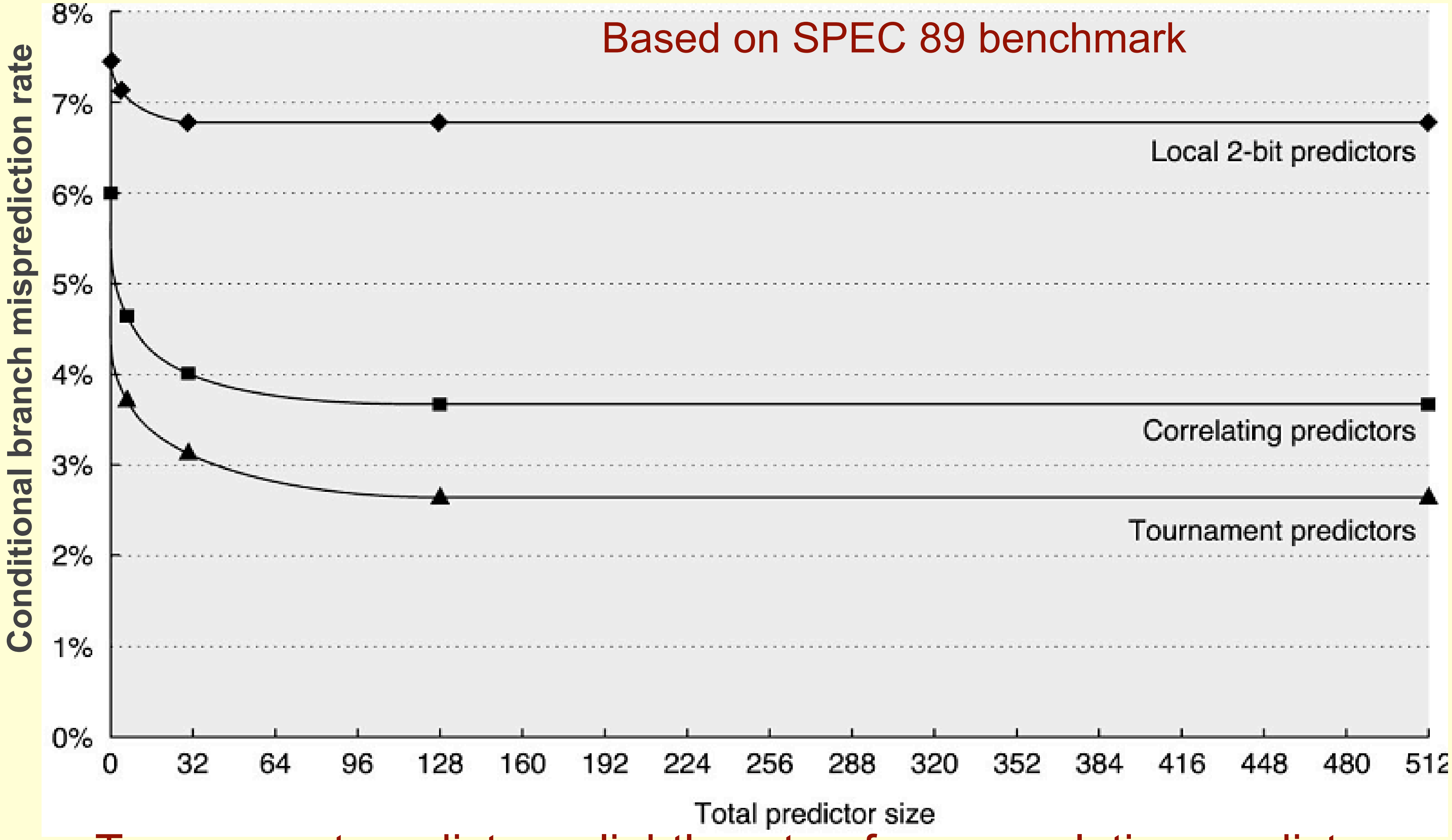
```
     BNEZ     R1, L1        ; branch b1 (d!=0)
     DADDI    R1, R0, #1    ; d==0, sp d=1
L1:  DSUBUI   R3, R1, #1
     BNEZ     R3, L2        ; branch b2 (d!=1)
….
L2:
```

Tournament Predictors

- Multilevel branch predictors use several levels of branch prediction tables together with an algorithm to choose among them
- Tournament selectors are the most popular form of multilevel branch predictors (e.g. DEC Alpha 21264)
- Tournament predictors combines both local and global predictor
- Selection between the two predictors are based on a selector (2-bit counter)
- Make a transition with two wrong prediction using the current table for which the correct prediction would have been possible using the other predictor



Performance of Tournament Predictors



Tournament predictors slightly outperform correlating predictors