COSC4201
Instruction Level Parallelism

Prof. Mokhtar Aboelaze

York University

Based on Slides by

Prof. L. Bhuyan (UCR)

Prof. M. Shaaban (RIT)

Outline

- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

ILP Concept

- Data and control hazards put a limit on the ability of the processor to exploit parallelism.
- There are two approaches to exploit ILP, software and hardware based.
  - Hardware: Depends on the hardware to locate and exploit parallelism between the instructions, scoreboard and Tomasulo's algorithm(dynamic)
  - Software: Depends on the compiler to exploit ILP (static)

Vector Processing

- Consider the following loop

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

- No dependence between iterations.
- All iterations can be done in parallel (if no structural hazards).
- Vector processing is ideal for such a case.
- Not widely used in general purpose applications.

Data Dependences and Hazards

° The Objective is to determine *parallel instructions*, those instructions that can run in parallel.

° If they can run in parallel, then they can overlap in a pipeline.

° There are three different type of dependence, data dependence, name dependence, and control dependence.

Data Dependence

° An instruction *j* is data dependent on instruction *i, if either*

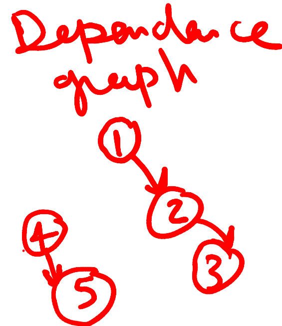
- instruction *i produces a data that is used by j,*
- Instruction *j* is data dependent on *k*, and *k* is data dependent on *I*

° True dependence, we cannot ignore or avoid, data must be produced before it is consumed.

Data Dependence

- The arrows show the data dependence.
- Either the compiler must not schedule the instruction this way, or the processor interlock detects it and stall.
- Dependence is a property of the program, whether it results in a hazard or not, that depends on the pipeline
- Much more difficult is data passed through memory rather than registers

```
Loop:LOAD   F0,0(R1)
     ADD    F4,F0,F2
     SD     F4,0(R1)
     ADD    R1,R1,#-8
     BNE    R1,R2,Loop
```

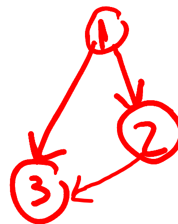


Data Dependence

```
1  ADD.D      F2, F4, F6
2  ADD.D      F10, F6, F8
3  ADD.D      F12, F12, F14
```

No dep.

```
1  ADD.D      F2, F4, F6
2  ADD.D      F10, F2, F8
3  ADD.D      F12, F10, F2
```



Name Dependence

- There are two types of name dependence, *antidependence*, and *name dependence*.
  - Anitdependence between instructions *i*, and *j* when *j* writes a register or memory location that *i* reads.
  - An output dependence is when two instructions write to the same register or memory location.
- There is no value being transmitted between instructions, could be solved by *register renaming*

Data Hazard

- A dependence may result in a data hazard
- RAW (read After Write)
- WAW (Write After Write)
- WAR (Write After Read)

Conrol Dependence

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

- S1 is control dependent on p1
- Can not move S1 before if, and can not move another instruction in the loop.
- We can violate control dependence and execute instruction that is not suppose to, if that will maintain the ***program correctness***.
- We mostly care about *exceptions behavior* and *data flow*

Control Dependence

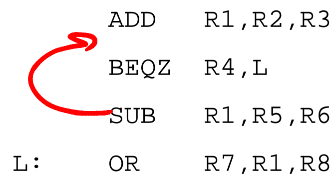
```
      ADD   R2,R3,R4
      BEQZ  R2,L1
      LW    R1,0(R2)
L1:   ….
```

No data dependence, can we move LW before BEQZ?

```
      ADD   R1,R2,R2
      BEQZ  R12,L1
      SUB   R4,R5,R6
      ADD   R5,R4,R9
L1:   OR    R7,R8,R9
      ADD   R9,R5,R10
```

What if R4 is not used after L1, can we move it up before branch

Control Dependence



```
        ADD   R1,R2,R3
        BEQZ  R4,L
        SUB   R1,R5,R6
L:      OR    R7,R1,R8
```

The value of R1 depends on the BEQZ.

Data dependence by itself is not sufficient to maintain correctness of the program

Outline

- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

Loop Unrolling

° Consider the following loop X=X+s

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

```
Loop:  L.D        F0, 0 (R1)     ;F0=array element
       ADD.D      F4, F0, F2     ;add scalar in F2  (constant)
       S.D        F4, 0(R1)      ;store result
       DADDUI     R1, R1, # -8   ;decrement pointer 8 bytes
       BNE        R1, R2,Loop    ;branch R1!=R2
```


Loop Unrolling

i.e followed immediately by .. →

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

No scheduling

			Clock cycle
Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0 (R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1,R2, Loop	9
	stall		10

10(9) cycles per iteration

Scheduling

			Clock cycle
Loop:	L.D	F0, 0(R1)	1
	DADDUI	R1, R1, # -8	2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0 (R1)	6
	BNE	R1,R2, Loop	7
	stall		8

8(7) cycles per iteration

Loop Unrolling

No scheduling

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

Assuming the array size is multiple of 2, i.e. the number of loop iterations is a multiple of 4

We eliminated the stalls because of the branching

28 cycles for 4 iterations (7 per iteration)

Loop Unrolling

When scheduled for pipeline

```
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),F12
        BNE       R1,R2, Loop
        S.D       F16, 8(R1), F16    ;8-32 = -24
```


Loop Unrolling

- A basic instruction block is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence.
- The amount of parallelism in the basic block is limited by the dependence and the size of the block
- Loop unrolling is increasing the size of the block, decreasing the “control” dependence by eliminating branches.

Loop Unrolling

- Determine that unrolling the loop would be useful by finding that the loop iterations where independent.
- Determine that it was legal to move S.D after DADDUI and BNE; find the correct S.D offset.
- Use different registers (rename registers) to avoid constraints of using the same register for different computations
- Eliminate extra tests and branches and adjust loop maintenance code.
- Determine that loads and stores can be interchanged by observing that loads and stores in different iterations are inependent..
- Schedule the code, preserving any dependencies needed to give the same result as the original code.

Outline

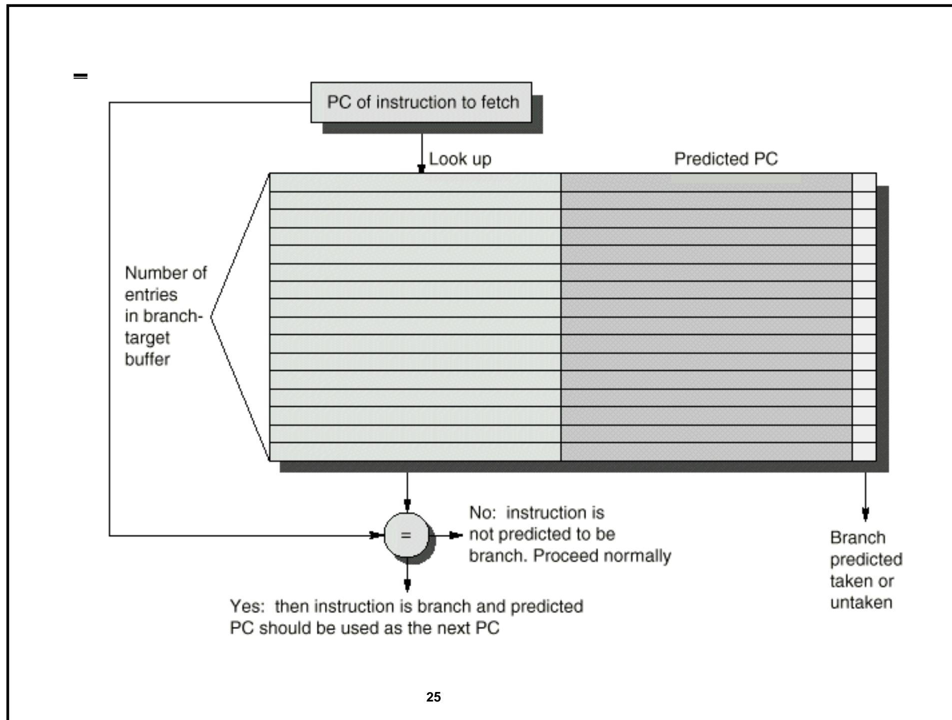
- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

Introduction

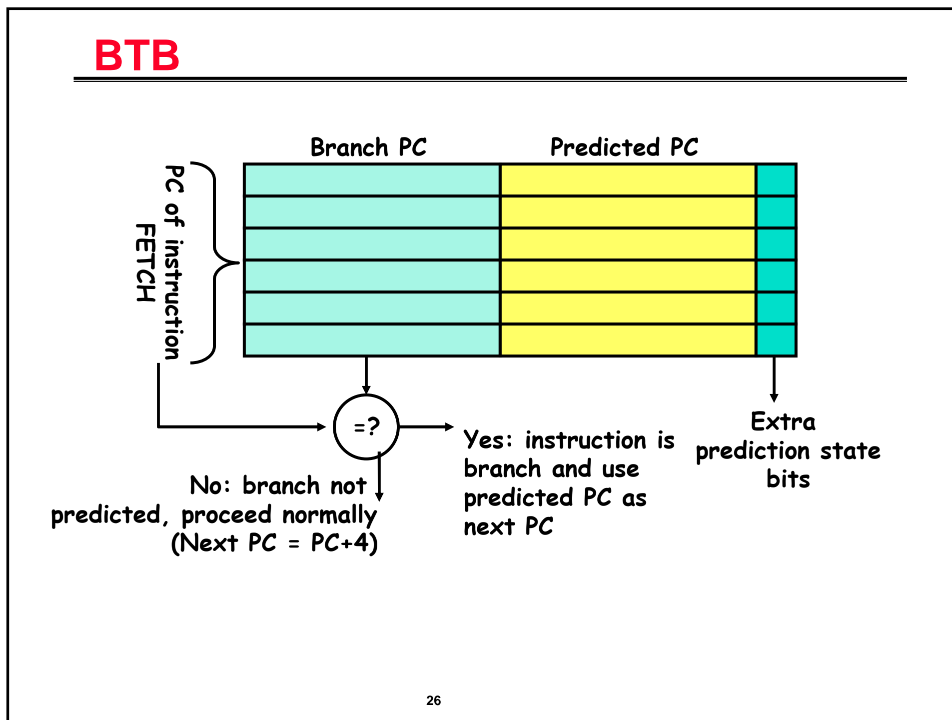
- Dynamic scheduling deals with data dependence improving, the limiting factor is the control dependence.
- Branch prediction is important for processors that maintains a CPI of 1, but it is crucial for processors who tries to issue more than one instruction per cycle (CPI < 1).
- We have already studied some techniques (delayed branch, predict not taken), but these do not depend on the dynamic behavior of the code.

Branch History Table

- A small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains only 1-bit, to predict taken or untaken
- If the prediction is incorrect, the prediction bit is inverted.
- In a loop, it mispredicts twice
  - 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



BTB



Branch Target Buffer

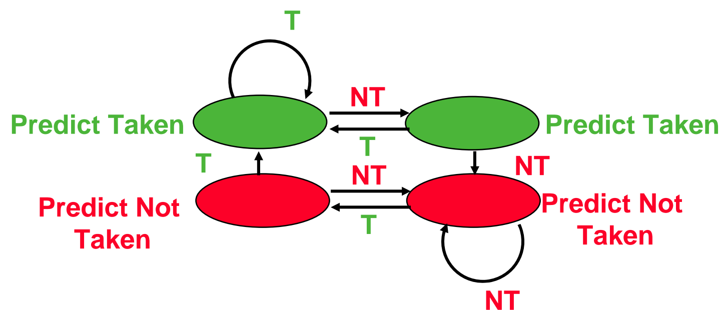
° Predicting by itself is not that useful unless we know the target address.

° We access the buffer in the IF stage.

° We must know if that entry is for that particular branch or not (unlike the prediction, we send the target PC address before we know if the instruction is a branch or not.

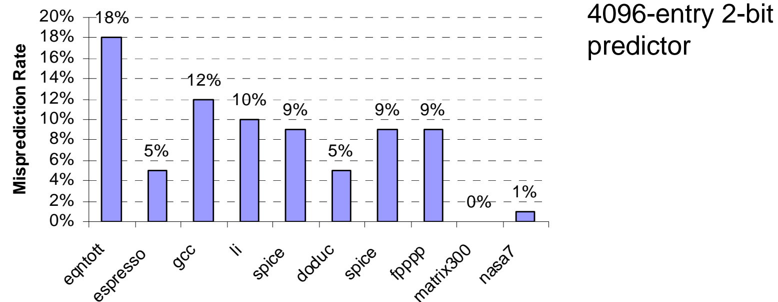
° We only need to store the predicted taken branch (untaken is similar to no branch). That might cause complication when we use a 2-bit predictor since we have to store information for taken and untaken (mat use two separate buffers).

2-bit Predictor



Red: stop, not taken
Green: go, taken
Adds *hysteresis* to decision making process
How many misses per loop ?

2-bit Prediction



4096-entry 2-bit predictor

Mispredict because either:

- **Wrong guess for that branch**
- **Got branch history of wrong branch when index the table**

Correlating Branch Predictors

° The 2-bit predictor uses the recent behavior of the branch to predict the behavior of this branch in the future.

° Sometimes, it is useful to look at the recent behavior of **other** branches

° Consider the following example

Correlating Branch Predictors

B1 if (aa==2)
aa=0;
B2 if (bb==2)
bb=0;
B3 if (aa!==bb){

```
        DSUBUI  R3, R1, #2
        BENZ    R3, L1      ; b1  (aa!=2)
        DADD    R1, R0, R0  ; aa==0
L1:     DSUBUI  R3, R1, #2
        BNEZ    R3, L2      ; b2  (bb!=2)
        DADD    R2, R0, R0  ; bb==0
L2:     DSUBUI  R3, R1, R2  ; R3=aa-bb
        BEQZ    R3, L3      ; b3  (aa==bb)
```


Correlating Branch Predictor Ex:

B1 if (d==0)
d=1;
B2 if (d==1)
{

d

```
        BNEZ    R1, L1      ; d == 0 ?
        DADD    R1, R0, #1  ; YES d==1
L1:     DADD    R3, R1, #-1
        BNEZ    R3, L2      ; b2  (bb!=2)

L2:
```

If b1 not taken, b2 is not taken for sure

Initial d	d==0?	B1	d befoe b2	d==1	b2
0	Y	NO	1	Y	NO
1	N	Taken	1	Y	NO
2	N	Taken	2	N	Taken

1-bit Predictor Ex:

Initial d	d==0?	B1	d befoe b2	d==1	B2
0	Y	NO	1	Y	NO
1	N	Taken	1	Y	NO
2	N	Taken	2	N	Taken

d	B1 Pred	B1 action	newB1 pred	B2 pred	B2 action	new B2 pred
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

Miss on every prediction

Correlating Branch Predictor

- Consider a branch that uses 1-bit for correlation.
- Every branch have 2 separate prediction, one if the previous one is taken, and another if not taken
- The prediction is on the form of **NT/T**
- What will be the result of such a predictor on the previous example.

Correlating Branch Predictor Ex:

Initial d	d==0?	B1	d befoe b2	d==1	b2
0	Y	NO	1	Y	NO
1	N	Taken	1	Y	NO
2	N	Taken	2	N	Taken

d	b1 Pred	b1 action	newb1 pred	b2 pred	b2 action	new b2 pred
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	NT	**NT**/T	NT	NT/T

Misprediction on first try only

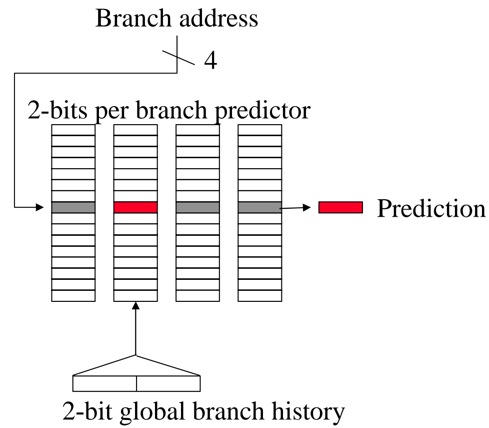
Correlating Branch Predictor

- The previous predictor is called (1,1) predictor.
- It uses one bit for history (last branch), to choose among two I-bit branch predictors.
- In general a predictor could me (m,n) predictor.
- It uses the last m branch to choose among 2^m branch predictors each is n-bit predictor.

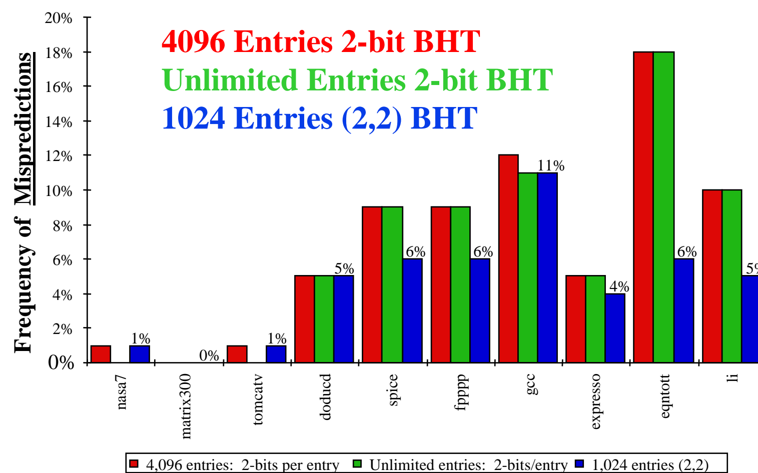
(2,2) Predictor

(2,2) predictor

– Behavior of recent branches selects between four predictions of next branch, updating just that prediction



Accuracy



Tournament Predictors

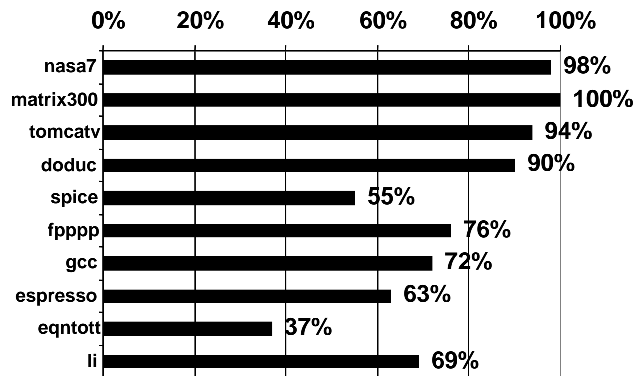
- Our first predictor, used only local information for prediction.
- The Correlating branch predictor used global information to improve performance.
- Tournament predictors uses more than one predictor (usually one local and another global), and use a selector to choose among these 2 predictors.

Tournment Predictor – Example Alpha 21264

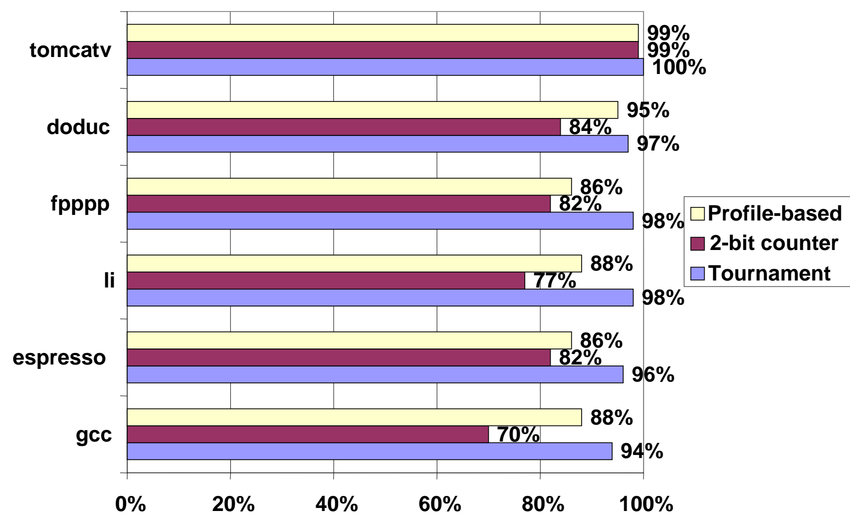
Tournament predictor using, say, 4K 2-bit counters indexed by local branch address, based on which predictor was most effective recently. Chooses between:

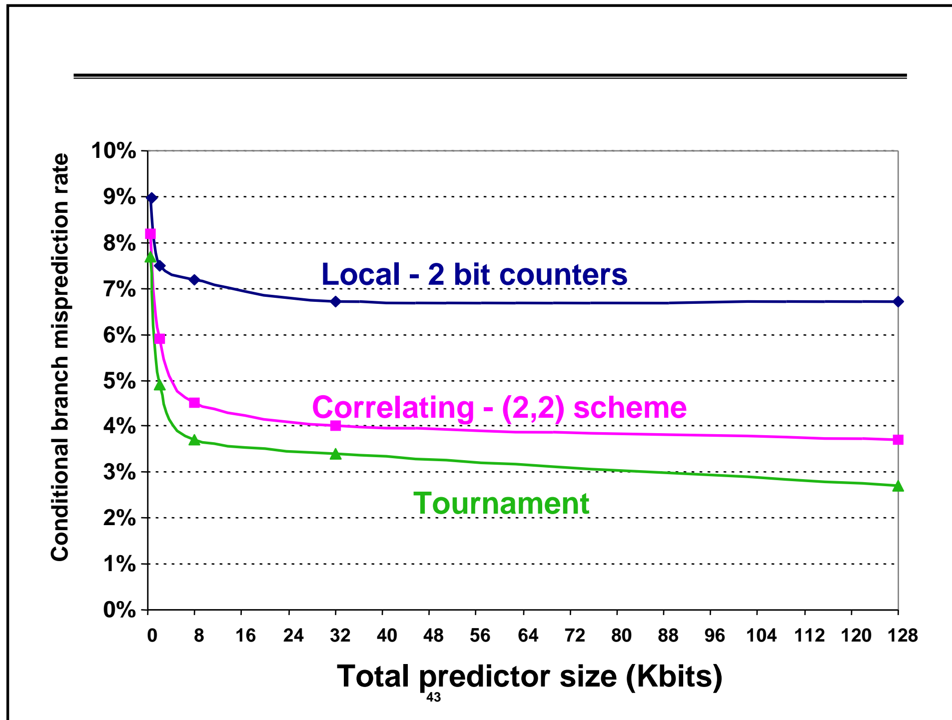
- Global predictor
  - 4K entries index by history of last 12 branches (2^{12} = 4K)
  - Each entry is a standard 2-bit predictor
- Local predictor
  - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
  - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters

Percentage of prediction by local predictor

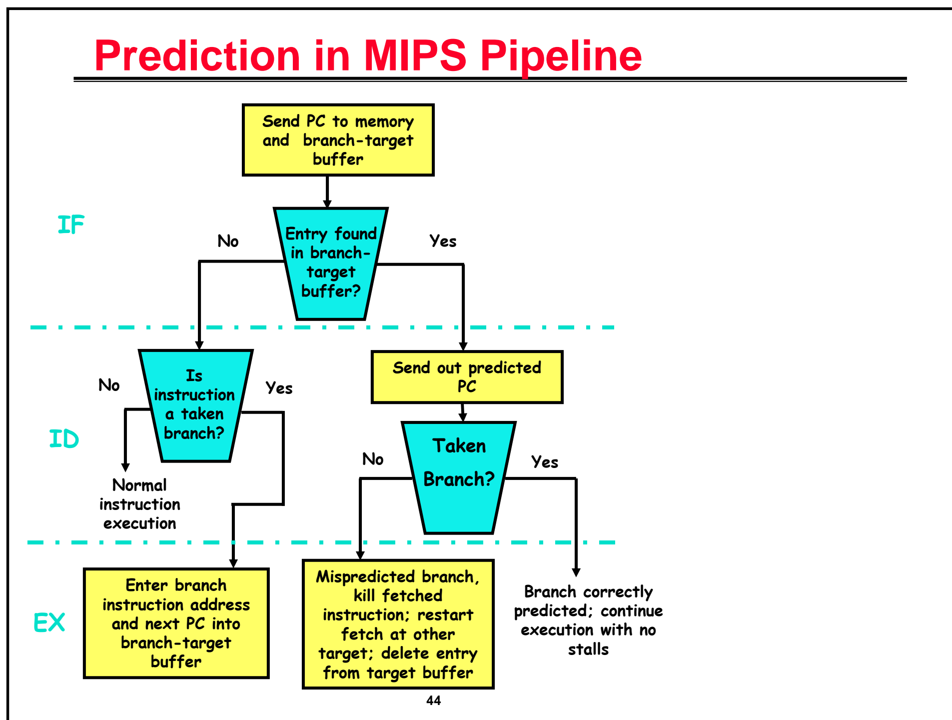


Accuracy of Prediction





Prediction in MIPS Pipeline



Integrated Instruction Fetch Unit

° Considering *fetch* as a single stage in a pipeline is no longer valid.

° An integrated instruction fetch unit is a part of many modern processors.

° The instruction fetch unit integrates several functions.

- Integrated branch prediction
- Instruction fetch (for multiple instruction per clock)
- Instruction memory access and buffering to deal with the complication of multiple fetches and crossing cache lines, …