

Branch Prediction


Chapter 3



More on Dependencies

- We will now look at further techniques to deal with dependencies—which negatively affect ILP—in program code.
- A dependency may be overcome in two ways:
 1. Maintain the dependency but avoid the related hazard, or
 2. transform the program code to eliminate the dependency.

Data Flow and Dependency Detection



- Data flows between instructions either through registers or memory locations.
 - Dependency detection based on *registers* is straightforward since registers are explicitly named in opcodes.
 - Dependencies that flow through *memory* are harder to detect. Aliasing: `100 (R4)` and `20 (R6)` may refer to the same memory location.

Name Dependencies

- Two instructions may use the same register (or memory location)—a name—but *no* data flows between them.

Antidependence

```
⋮  
DIV.D F0, , F4  
ADD.D , F6, F8
```

Output dependence

```
⋮  
DIV.D , F2, F4  
ADD.D , F6, F8
```

- *No* true data dependence. Can reorder instructions or execute in parallel after register renaming.

Control Dependencies

- Every instruction in a program—except for those in the first basic block—are control dependent on a branch instruction.

S2 depends on **p2**:

```
if (p1) {  
    S1;  
}  
if (p2) {  
    S2;  
}
```

S2 depends on **p1** and **p2**:

```
if (p1) {  
    S1;  
    if (p2) {  
        S2;  
    }  
}
```

Dynamic Branch Prediction

- The frequency of branch instructions in program code calls for a closer look at how branch cost can be reduced.
- The predicted-untaken scheme and delay slots are *static* techniques to deal with branches:
 - In these schemes, the action of the CPU does *not* depend on the actual dynamic branching behavior (*jump or fall-through?*) of a given instruction.

1-Bit Branch-Prediction Buffer

- Maintain a branch-prediction buffer (branch history table), recording last behavior of branch:
 - Small memory buffer indexed by least significant bits of branch instruction address.

0x4000003c: BEQZ R2, *label*

LSB	Predictor
⋮	⋮
111100	0
⋮	⋮

Branch untaken

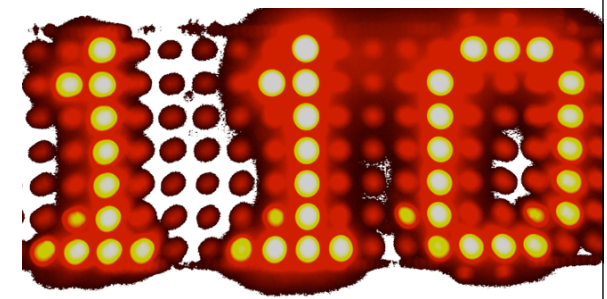
Branch taken

1-Bit Branch-Prediction Buffer

- Branch mispredicted: invert predictor bit and store back into branch-prediction buffer.
 1. Branch may have changed behavior (e.g., loop exit), or
 2. another branch may have overwritten the entry.
- CPU always assumes the predictor bit to be correct and starts fetching instructions from target (1) or fall-through (0).
- Most useful for pipelines that determine branch target address early (and evaluate condition later on).

1-Bit

Prediction Quality



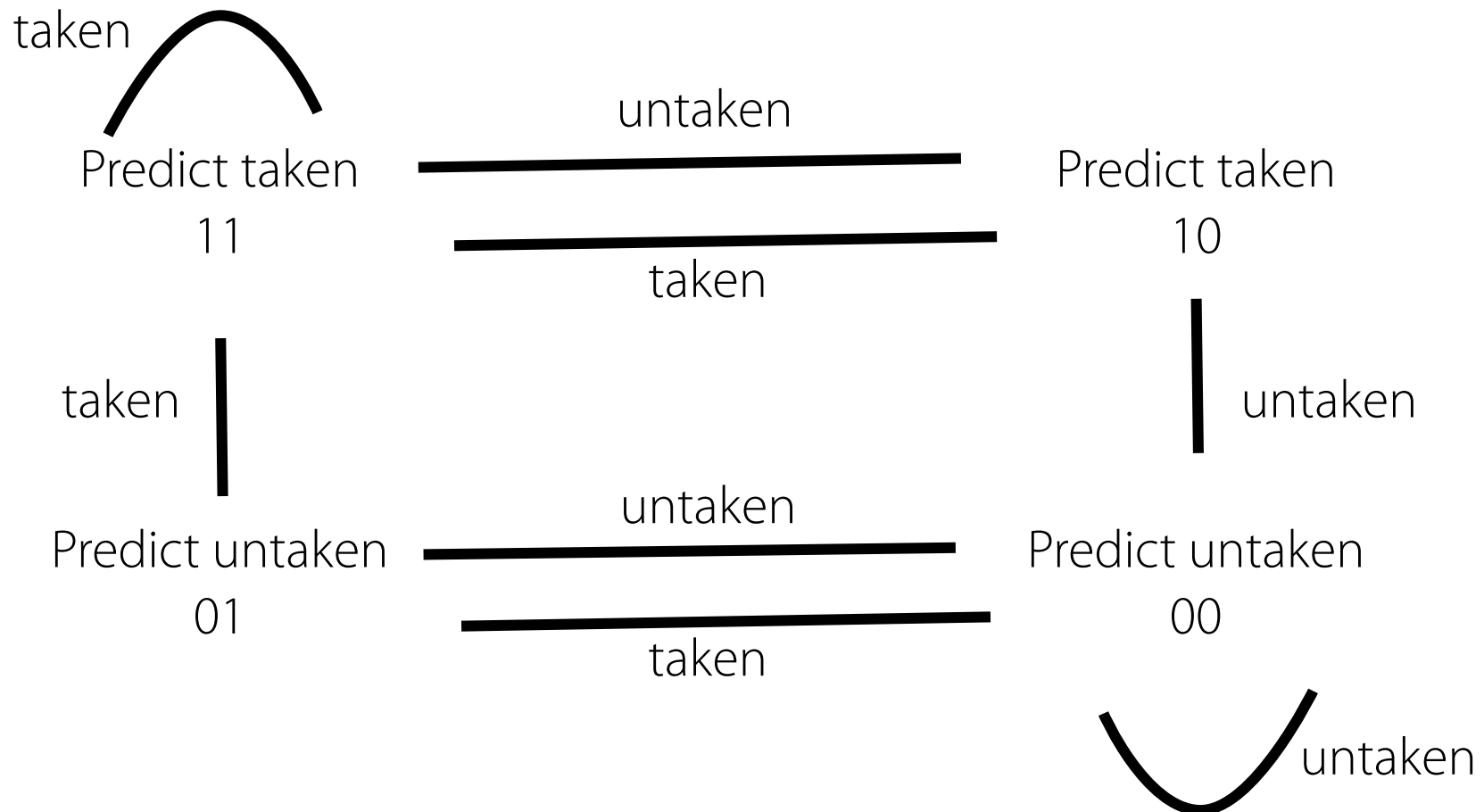
- Assume the following loop is iterated 10 times (the BNEZ is taken 9 times, 10th time untaken).
What is the 1-bit branch-prediction accuracy?

```
loop: ...  
      :  
      BNEZ R2, loop
```

Answer: 80%

2-Bit Branch-Prediction

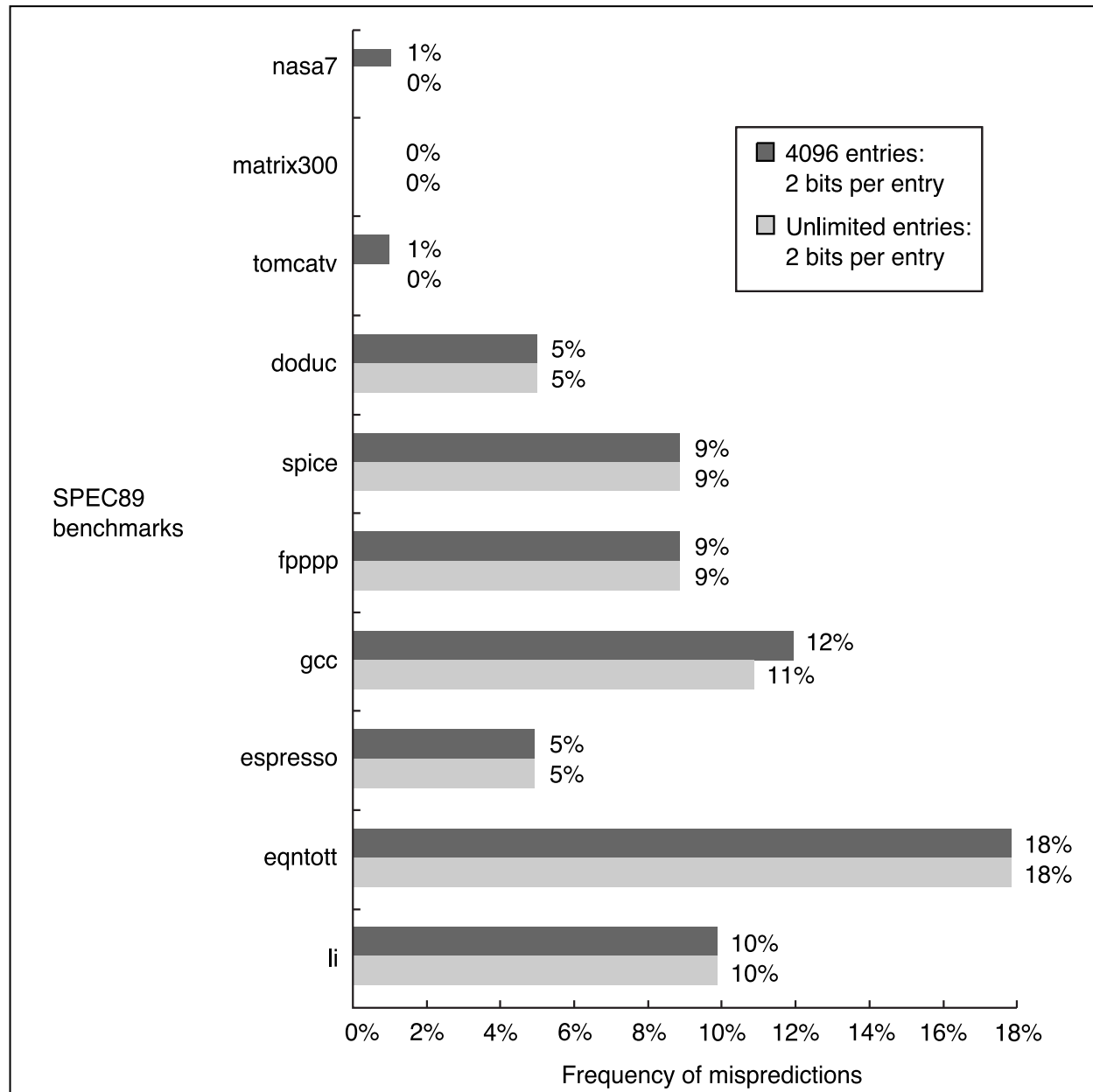
- For a 2-bit branch predictor, a prediction must miss *twice* before it is changed:



n -Bit Branch-Prediction

- 2-bit predictor is a special case of n -bit saturating counter, counter value $\in [0, 2^n-1]$:
 - Increment counter if branch taken; decrement if untaken.
 - Predict branch to be taken if counter value $\geq 2^{n-1}$.
- Accuracy of 4096-entry buffer 82%–99% even for a 2-bit predictor; n -bit predictors found to be *no* significant improvement.

Branch-Prediction Buffer Size



Correlated Branches

- More complex predictor setups are required if *separate* branches are correlated. Consider ($d \equiv R1$):

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

```
b1    BNEZ    R1, l1
      DADDIU  R1, R0, #1
l1:   DADDIU  R3, R1, #-1
b2    BNEZ    R3, l2
      ...
l2:
```

- Branch $b1$ not taken \Rightarrow branch $b2$ not taken.

Correlated Branches

```

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

```

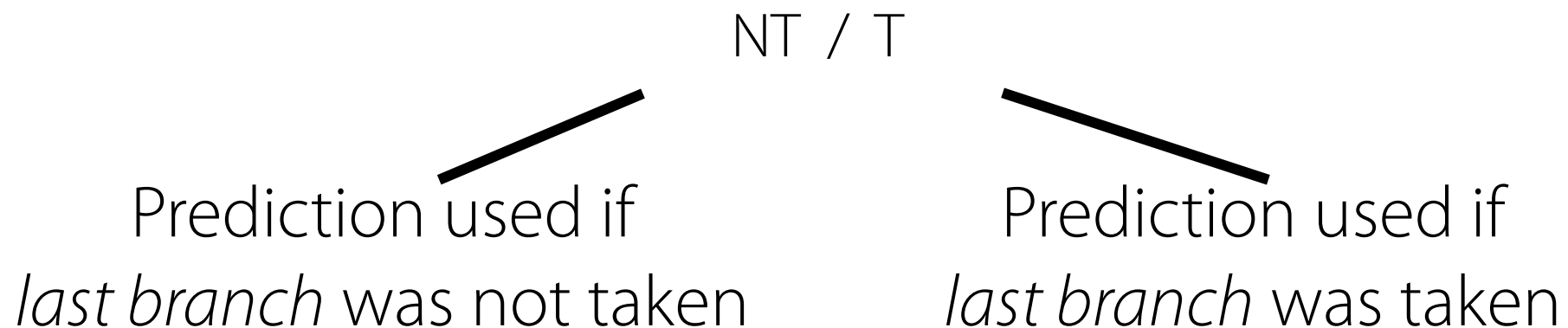
b1    BNEZ    R1, l1
      DADDIU  R1, R0, #1
l1:   DADDIU  R3, R1, #-1
b2    BNEZ    R3, l2
      ...
l2:
    
```

<i>d = ?</i>	<i>b1</i>	<i>b1</i>	<i>b1</i>	<i>b2</i>	<i>b2</i>	<i>b2</i>
	prediction	action	new prediction	prediction	action	new prediction
<i>d = 2</i>	NT	T	T	NT	T	T
<i>d = 0</i>	T	NT	NT	T	NT	NT
<i>d = 2</i>	NT	T	T	NT	T	T
<i>d = 0</i>	T	NT	NT	T	NT	NT

1-Bit Predictor with 1-Bit Correlation



- Each branch is assigned two prediction bits:



- Note: The *last branch* in general is *not* the branch being predicted.

Correlated Prediction

```

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

```

b1    BNEZ    R1, l1
      DADDIU  R1, R0, #1
l1:   DADDIU  R3, R1, #-1
b2    BNEZ    R3, l2
      ...
l2:
    
```

<i>d</i> = ?	<i>b1</i> prediction	<i>b1</i> action	<i>b1</i> new prediction	<i>b2</i> prediction	<i>b2</i> action	<i>b2</i> new prediction
<i>d</i> = 2	NT/NT	T	T/NT	NT/NT	T	NT/T
<i>d</i> = 0	T/NT	NT	T/NT	NT/T	NT	NT/T
<i>d</i> = 2	T/NT	T	T/NT	NT/T	T	NT/T
<i>d</i> = 0	T/NT	NT	T/NT	NT/T	NT	NT/T

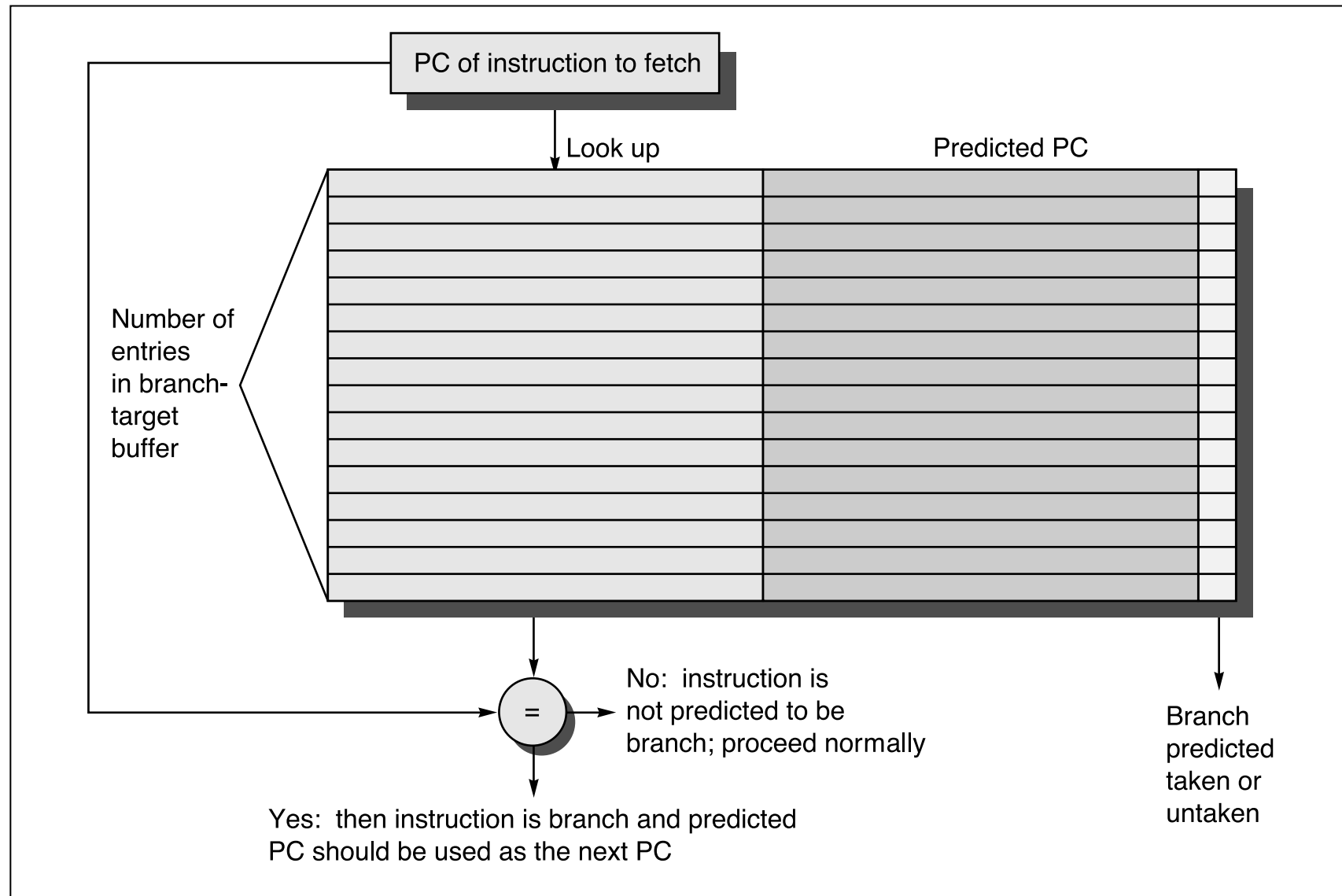
Branch Target Buffers

- In the 5-stage pipeline, the branch prediction buffer is accessed during stage ID (when the CPU knows it is indeed dealing with a branch).
 - By the end of ID, the CPU knows enough to fetch the next instruction (does not wait for actual branch outcome).
 - This is *still 1 cycle too late* to fill the pipeline without disruption.

Branch Target Buffers

- A branch target buffer (BTB) is accessed at stage IF, *before* the CPU knows that the fetched instruction is a branch.
 - *Before decoding*, the CPU uses the current PC as a key to perform a BTB lookup.
 - If the lookup is a hit, the CPU knows the predicted PC (stored in the BTB) at the end of stage IF—just in time.

Branch Target Buffers



Branch Target Buffer Timing

