Processor Design (III)

Hung-Wei Tseng
Branch prediction to reduce the overhead of control hazards
Why do we need to stall for branch instructions

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor

1. The target address when branch is taken is not available for instruction fetch stage of the next cycle

You need a cheatsheet for that

2. The target address when branch is not-taken is not available for instruction fetch stage of the next cycle

You need to predict that

3. The branch outcome cannot be decided until the comparison result of ALU is out

4. The next instruction needs the branch instruction to write back its result

A. 0
B. 1
C. 2
D. 3
E. 4
Branch Target Buffer

- The processor needs a “cheat sheet” for where the branch is going without calculating it.
Dynamic branch prediction

- A 2-bit counter for each branch
- Predict taken if the counter value $\geq 2$
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4
  - If we guess right — no penalty
  - If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

<table>
<thead>
<tr>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420 0x8048324 11</td>
</tr>
<tr>
<td>0x400464 0x8048392 10</td>
</tr>
<tr>
<td>0x400578 0x804850a 00</td>
</tr>
<tr>
<td>0x41000C 0x8049624 01</td>
</tr>
</tbody>
</table>

PC = 0x400420

Taken!
Performance of 2-bit counter

- 2-bit state machine for each branch

\[
\text{for}(i = 0; i < 10; i++) \{ \\
\quad \text{sum } += \text{a}[i]; \\
\}
\]

90% accuracy!

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
- \(1 + 20\% \times (1 - 90\%) \times 2 = 1.04\)
local 2-bit predictor

i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
        a[i] += i;
} while ( ++i < 100) // Branch Y

• What’s the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
Local 2-bit predictor

```c
i = 0;
do {
    if (i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while (++i < 100) // Branch Y
```

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For branch Y, almost 100%, For branch X, only 50%
What’s the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%
Local 2-bit predictor

\[ i = 0; \]
\[ \text{do } \{ \]
\[ \quad \text{if}(i \% 2 \neq 0) \quad \text{// Branch X, taken if } i \% 2 = 0 \]
\[ \quad \text{a}[i] *= 2; \]
\[ \quad \text{a}[i] += i; \]
\[ \} \text{ while ( ++i < 100) // Branch Y} \]

- Taken 3 (11)
- Taken 2 (10)
- Not Taken 0 (00)
- Not Taken 1 (01)

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For branch Y, almost 100%,
For branch X, only 50%

Can we capture the pattern?
Branch prediction using global history
2-level global predictor

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Global predictor: predictor using results from all branches
- Local predictor: predictor tracking states/history for each branch
- Each entry in the history table has its own counter.

First level

3-bit GHR = 101 (T, NT, T)

2nd level

\[ 2^3 \text{ entries} \]

Pentium Pro uses this predictor
Performance of the 2-bit global predictor

i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>GHR</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>000</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>001</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>011</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>110</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>101</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>011</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>110</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>101</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>011</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>110</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>101</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>011</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>110</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>101</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>011</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>110</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>101</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>011</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Nearly perfect after this
Branch prediction & your code
Demo revisited

- Why the sorting the array speed up the code despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Branch performance

- Why the performance is better when option is not “0”
  1. The amount of dynamic instructions needs to execute is a lot smaller
  2. The amount of branch instructions to execute is smaller
  3. The amount of branch mis-predictions is smaller
  4. The amount of data accesses is smaller

A. 0
B. 1
C. 2
D. 3
E. 4

```cpp
std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Without sorting</th>
<th>With sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>The prediction accuracy of X before threshold</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>The prediction accuracy of X after threshold</td>
<td>50%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Demo: popcount

- How many 1s in binary representations
- Applications
  - Hamming weight
  - Encryption/decryption

```c
int main(int argc, char *argv[]) {
    uint64_t key = 0xdeadbeef;
    int count = 1000000000;
    uint64_t sum = 0;
    for (int i=0; i < count; i++) {
        sum += popcount(RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```
Four implementations

- Which of the following implementations will perform the best on modern pipeline processors?
Why is B better than A

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  1. B has lower dynamic instruction count than A
  2. B has significantly lower branch mis-predictions than A
  3. B has significantly fewer branch instructions than A
  4. B can incur fewer data hazards

A. 0
B. 1
C. 2
D. 3
E. 4

inline int popcount(uint64_t x)
{
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
Why is B better than A

**A**

```c
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

**B**

```c
inline int popcount(uint64_t x) { 
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
and $t2, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
bne $t1, $zero, LOOP
```

```
and $t2, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
shr $t4, $t1, 1
shr $t5, $t1, 2
shr $t6, $t1, 3
shr $t1, $t1, 4
and $t7, $t4, 1
and $t8, $t5, 1
and $t9, $t6, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
add $t3, $t3, $t2
bne $t1, $zero, LOOP
```

**4*n instructions**

**13*(n/4) = 3.25*n instructions**

Only one branch for four iterations in A
Two versions of B

Before re-ordering:

```assembly
and $t2, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
and $t2, $t1, 1
add $t3, $t3, $t2
shr $t1, $t1, 1
and $t2, $t1, 1
add $t3, $t3, $t2
shr $t4, $t1, 1
shr $t5, $t1, 2
shr $t6, $t1, 3
shr $t1, $t1, 4
and $t7, $t4, 1
and $t8, $t5, 1
and $t9, $t6, 1
add $t3, $t3, $t7
add $t3, $t3, $t8
add $t3, $t3, $t9
bne $t1, $zero, LOOP
```

After re-ordering:

```assembly
and $t2, $t1, 1
shr $t4, $t1, 1
shr $t5, $t1, 2
shr $t6, $t1, 3
shr $t1, $t1, 4
and $t7, $t4, 1
and $t8, $t5, 1
and $t9, $t6, 1
add $t3, $t3, $t2
add $t3, $t3, $t7
add $t3, $t3, $t8
add $t3, $t3, $t9
bne $t1, $zero, LOOP
```

Lots of back-to-back data dependencies — likely to introduce data hazards

This re-ordering is only possible after you “unrolled” your loop — this technique is called “loop unrolling”
Why is B better than A

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  1. B has lower dynamic instruction count than A
  2. B has significantly lower branch mis-predictions than A
  3. B has significantly fewer branch instructions than A
  4. B can incur fewer data hazards

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

```
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```
Why is C better than B

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - C has lower dynamic instruction count than B
  - C has significantly lower branch mis-predictions than B
  - C has significantly fewer branch instructions than B
  - C can incur fewer data hazards

A. 0  B. 1  C. 2  D. 3  E. 4

B

```
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

C

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
Announcement

- Homework #2 due next Monday
- Quiz due next Monday
- Midterm
  - Next Wednesday — only from 8am-9:20a
  - Will have review on next Monday