Reducing the branch delay

```
bne $2,$3,foo  
addu $2,$4,$5  
```
Branch bypassing – easy case shaves one cycle off branch penalty
Branch bypassing – back-to-back deps
Branch handling in decode → lots of ugly paths might as well execute branch in EXE stage + use better branch prediction
Branch Prediction from 10,000 ft

Invariant: is_correct(PC) → is_correct(Instr[PC])

On restart (branch misprediction) must –
  a. kill all incorrectly fetched instructions (to ensure correct execution)
  b. refill pipeline (takes # cycles == latency of pipeline up to execute stage)
### Aside: Decoupled Execution

_Buffeting Smooths Execution and Improves Cycle time by Reducing Stall Propagation_

The front end runs ahead .. stalls + cache misses are overlapped.

<table>
<thead>
<tr>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE</td>
</tr>
<tr>
<td>FIFO</td>
</tr>
<tr>
<td>BE</td>
</tr>
</tbody>
</table>

The front end runs ahead .. stalls + cache misses are overlapped.

without decoupling .. stalls + cache misses are not overlapped.
Branch Predicted-Taken Penalty

Back End

Instruction Cache

Instr FIFO

br dir predictor

br. imm

br

Squash Speculatively Fetch Instructions That Follow Branch
Branch Misprediction Penalty

Front End

Instr Cache

“Guess” PC

Instr FIFO

PC FIFO

Back End

dcache

branch resolution login

arch PC

“restart at address X”

Basic Pentium III Processor Misprediction Pipeline

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Fetch</td>
<td>Decode</td>
<td>Decode</td>
<td>Decode</td>
<td>Rename</td>
<td>ROB Rd</td>
<td>Rdy/Sch</td>
<td>Dispatch</td>
<td>Exec</td>
</tr>
</tbody>
</table>

Pentium 3 – ~10 cycles

Basic Pentium 4 Processor Misprediction Pipeline

Pentium 4 – ~20 cycles

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC Nxt IP</td>
<td>TC Fetch</td>
<td>Drive</td>
<td>Alloc</td>
<td>Rename</td>
<td>Que</td>
<td>Sch</td>
<td>Sch</td>
<td>Sch</td>
<td>Disp</td>
<td>Disp</td>
<td>RF</td>
<td>RF</td>
<td>Ex</td>
<td>Flgs</td>
<td>Br Ck</td>
<td>Drive</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since misprediction penalty is larger, we first focus on branch (direction) prediction

- Static Strategies:
  - #1 predict taken (34% mispredict rate)
  - #2 predict (backwards taken, forwards not) (10%, 50%) mispredict rate
    - same backwards behavior as #1
    - better forwards behavior (50%-50% branches)

  penalty: #1 taken 2 cycle \(\sim\) taken 20 cycle
  #2 taken 20 cycle \(\sim\) taken 0 cycle

#1 forward branch ave execution time = 50\% \times 2 + 50\% \times 20 = 11 cycles
#2 forward branch ave execution time = 50\% \times 20 + 50\% \times 0 = 10 cycles
Since misprediction penalty is larger, we first focus on branch (direction) prediction

- **Static Strategies:**
  
  #3 profile (see next slide for misprediction %’s)
  
  - choose a single prediction for each branch and encode in instruction
  
  - some studies show that sample runs are fairly representative of inputs in general
  
  - negative: extra programmer burden

See next slide for misprediction rates
Each branch is permanently assigned a probable direction.

To do better we would need to change the prediction as the program runs!

Profiling Based Static Prediction

15% ave. (specint92), 9% ave. (specfp92) misp rate
A note on prediction/misprediction rates

Qualitatively, ratio of misprediction rates is better indicator of predictor improvement.

15% ave. (specint92), 9% ave. (specfp92) misp rate

(assumes misprediction probability independent between branches)

<table>
<thead>
<tr>
<th>Prediction Rate (p)</th>
<th>Misprediction Rate</th>
<th># Consecutive Branches Predicted Correctly (w/ 50% prob)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>50%</td>
<td>1</td>
</tr>
<tr>
<td>78%</td>
<td>22%</td>
<td>2.78</td>
</tr>
<tr>
<td>85%</td>
<td>15%</td>
<td>4.26</td>
</tr>
<tr>
<td>91%</td>
<td>9%</td>
<td>7.34</td>
</tr>
<tr>
<td>95%</td>
<td>5%</td>
<td>13.5</td>
</tr>
<tr>
<td>96%</td>
<td>4%</td>
<td>16.98</td>
</tr>
<tr>
<td>98%</td>
<td>2%</td>
<td>34.3</td>
</tr>
</tbody>
</table>

Bernoulli Process:  
\[ p^k = 0.5 \]

\[ k = \log(0.5)/\log(p) \]

2% makes a huge difference here
Compiler also can take advantage of Static Prediction / Profiling / Knowledge

- Static Strategies:
  - #1 predict taken (34% mispredict rate)
  - #2 predict backwards taken, forwards not not (10%, 50% mispredict rate)
  - #3 profile (see previous slide)
  - #4 delayed branches
    always execute instructions after branches avoids need to flush pipeline after branch
Observation: Static Prediction is limited because it only uses instructions as input + has a fixed prediction
Dynamic Prediction: More inputs allow it to adjust the branch direction prediction over time

branch (direction) predictor

branch info
mispredict
instr

feedback

Instr FIFO

PC FIFO

restart
new pc
branch info

Dynamic Prediction: More inputs allow it to adjust the branch direction prediction over time
Dynamic Prediction: More detailed

branch (direction) predictor

Instruction Cache

 EA

Instr FIFO

FIFO

br. descr

Feedback

GPC

+4

New pc branch-info

Restart

pc

Dynamic Prediction: More detailed
Dynamic Branch Prediction – Track Changing Per-Branch Behavior

- Store 2 bits per branch
- Change the prediction after two consecutive mistakes!

**BP state:**

\[(\text{next prediction } \text{taken}/\neg\text{taken}) \times (\text{last branch } \text{taken}/\neg\text{taken})\]

**note:** this is not strictly a saturating up-down counter
Why two bits?

- One bit is wishy-washy on loops

Two mispredictions per loop execution with single-bit prediction

(No data – either use what is left over from before or initialize on i. fill with “predict taken” for backwards branches)

Single Bit Predictor Analysis
Why two bits?

• One bit is wishy-washy on loops

One misprediction per loop execution with two-bit prediction

Two Bit Predictor Analysis
n-bit implementation

blindly write into this hash table; branches may alias but that’s “ok”
Accuracy of simple dynamic branch predictor: 4096-entry 2-bit predictor on Spec89

somewhat old benchmarks – probably need slightly larger predictors to do this well on current benchmarks

vs profiling
Limits of 2-bit prediction

- ∞ table does not help much on spec89

- reportedly, more bits does not help significantly either.
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then 
    y += 1;
if (x[i] < 5) then 
    c -= 4;

If first condition false, second condition also false

History bit: H records the direction of the last branch executed by the processor

Two sets of BHT bits (BHT0 & BHT1) per branch instruction

H = 0 (not taken) ⇒ consult BHT0
H = 1 (taken) ⇒ consult BHT1

Adapted from Arvind and Asanovic’s MIT course 6.823, Lecture 6
Accuracy with 2 bits of global history

Less storage than 4k x 2bit but better accuracy (for these benchmarks)
Two-Level Branch Predictor

Pentium Pro (1995) uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)

Fetch PC

2-bit global branch history shift register

Shift in Taken/¬Taken results of each branch

Taken/¬Taken?
Benefit of longer histories for fixed-iteration loops with small iteration counts

- Unary encoding of branch patterns

No mispredictions per (5-iter) loop execution with \( \geq 5 \)-bits of history

Prediction  | T  | T  | T  | T  | T  | N  | T  | T  | T  | T  | T  | T  | T  | T  | T  | N  
-------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
Outcome      | T  | T  | T  | T  | T  | N  | T  | T  | T  | T  | T  | N  | T  | T  | T  | T  | T  | N  

History Table / Prediction

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

 Doesn't work for many-iteration loops – but relative error is smaller!
Alpha 21264 Tournament Predictor

“Predictor-Predictor”: 4K 2-bit counters indexed by branch address
- chooses between two predictors:

A. **global predictor**: 4K 2-bit counters indexed by 12-bit global history

B. **local predictor**
1024 10-bit entries containing history for that entry
   This history indexes into 1K 3-bit saturating counters
Tournament, Correlating, Local Predictor Performance

Consecutively Correctly Predicted Branches w/ 50% probability:

- 2-bit ~10
- correlating ~18
- tournament ~25

Spec89 (size presumably in Kbit)
Pentium 4 3.2 GHz Spec2000 Misprediction Rates

\[\text{gzip} \quad \text{vpr} \quad \text{gcc} \quad \text{mcf} \quad \text{crafty} \quad \text{wupwise} \quad \text{swim} \quad \text{mgrid} \quad \text{applu} \quad \text{mesa}\]

Branch mispredictions per 1000 instructions

Note: The metric is slightly different here, but P4 has some of the best branch prediction because it needs it – extremely long pipeline.
Predicted-Taken Penalty

Back End

br dir predictor

Instruction Cache

Instr FIFO

br. imm

EA

+4

PC

PC FIFO

pc

checker

restart

new pc

br
1. Unroll thy loops

Unrolling loops reduces the number of backwards-taken branches in the program, and thus many of the predicted taken branches.

Matters most when loop bodies are small.

*Positive/Negatives?*  

red arcs = common case in this example
Top N List of Ways to Avoid Branch-Taken Penalties

2. Unroll+
Reorder code into common paths and off-paths.

- Often need profiling to get this kind of information.

-+ Avoid branch-taken penalties with the same accuracy limits as static branch prediction.
- Often more instructions added to off-paths
3. Delayed Branches

- Requires extra work that is independent of the branch that can be scheduled often not available.

- Architecturally fixed number of delay slots.

- Messy semantics – branches within branch delay slots? Exceptions?

Positive/Negatives?
Top N List of Ways to Avoid Branch-Taken Penalties

4. Anulled Branches

top:

a ←
b ←
c
d
bne b, top
e

killed if branch not taken!

+ Filler instruction are automatically independent of branch because they come from the next iteration of the loop. It is easier to fill these than standard delayed branches.

- Architecturally fixed number of delay slots.
- Messy semantics – branches within branch delay slots? Exceptions?
Top N List of Ways to Avoid Branch-Taken Penalties

5. Fetch Ahead (So as Not to Fall Behind)

+ Fetch unit can fetch more instructions per cycle than the backend can consume, filling the FIFO more quickly. Then, the front end can afford to spend a few cycles on each taken branch.
Top N List of Ways to Avoid Branch-Taken Penalties

6. Branch Target Buffer
Branch Target Buffer

branch (direction) predictor

branch info
mispredict
instr

btb override

fix BTB
guess?

+4

Instruction Cache

Instr FIFO

PC FIFO

restart
new pc
branch info

feedback

Branch Target Buffer Positive/Negatives?
BTB Design #1

- 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

Positive/Negatives?
Simple, Fast “Next-Ptr” BTB design – a la Alpha 21264

BTB selects next *fetch block* to access. Update mechanism (not shown) may include some hysteresis ala 2-bit predictor, and does not need to be on the critical path.

(The red line is the critical path – [in the Raw tile, this was the critical path of the design] - which can be optimized down to the latency through the SRAM, a Mux, and a latch.)

Compared to the I-Cache, the BTB SRAM is smaller (e.g. 512 x 9b versus 512 x 256b or 1024*10b versus 1024 x 128b) and should have a smaller access time and/or lower latency than i-cache.

Positive/Negatives?