CSE 240A: Next Steps

- We’ve covered:
  - Technology
  - History
  - Performance Analysis

- Goal: Grok Modern Sequential Processor Design
  - Recap of 80’s architecture
    - Single-Issue, In-order Stall-Based Pipeline
    - Think about Exceptions
  - Front Ends
  - Back Ends
    - In-Order Superscalars (Ultra Sparc I, III, P5)
    - Out-of-Order Superscalars (MIPS R10K, Alpha 21264, P6)
Old formulation of branch paths w/o prediction
Cleaner formulation of Branching

Invariant: is_correct(PC) \rightarrow 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

Buffering Smooths Execution and Improves Cycle time by Reducing Stall Propagation

F6 S1 C7 F6
(f=fetch, s=stall, c=cache miss, e=execute)
E2 C8 E8

w/ decoupling: front end runs ahead .. stalls + cache misses are overlapped.

w/o decoupling .. stalls + cache misses are not overlapped.
Pipelined Front End

branch direction predictor

Instruction Cache

GPC

+4

bne+ $2,$3,foo

br. imm

EA

Instr FIFO

pc

PC FIFO

checker

new pc

restart
Branch Predicted-Taken Penalty

Back End

- GPC
- Branch Predicted-Taken Penalty
- Instruction Cache
- Instr FIFO
- br dir predictor
- br. imm
- EA
- pc
- Squash Speculatively Fetch Instructions That Follow Branch
- Checker

Branch: br
Branch Misprediction Penalty

The diagram illustrates the process of handling a branch misprediction penalty in a processor pipeline. The Front End includes the Instr Cache, "Guess" PC, and PC FIFO. The Back End includes the Instr FIFO, dequeue, dcache, branch resolution, login, arch PC, and an input that says "restart at address X".

The Basic Pentium III Processor Misprediction Pipeline is shown with 10 stages:

1. Fetch
2. Fetch
3. Decode
4. Decode
5. Decode
6. Rename
7. ROB Rd
8. Rdy/Sch
9. Dispatch
10. Exec

Pentium 3 – ~10 cycles

The Basic Pentium 4 Processor Misprediction Pipeline is shown with 20 stages:

1. TC Nxt IP
2. TC Fetch
3. Drive
4. Alloc
5. Rename
6. Que
7. Sch
8. Sch
9. Disp
10. Disp
11. RF
12. Ex
13. Flgs
14. Br Ck
15. Drive

Pentium 4 – ~20 cycles

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 ~taken 20 cycle
      #2 taken 20 cycle ~taken 0 cycle

#1 forward branch ave execution time = 50% * 2 + 50% * 20 = 11 cycles
#2 forward branch ave execution time = 50% * 20 + 50% * 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:

\[ p^k = 0.5 \]

\[ k = \frac{\log(0.5)}{\log(p)} \]

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 (10%, 50% mispredict rate)
  - #3 profile (see previous slide)
  - #4 delayed branches

  always execute instructions after branches avoids need to flush pipeline after branch

  → eliminates branch taken penalty and branch direction prediction penalty for loops where branch direction is determined early enough
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

Instruction Cache

branch info
mispredict
instr

feedback

GPC

Instr FIFO

EA

PC FIFO

restart
new pc
branch info

+4
Dynamic Prediction: More detailed
Dynamic Branch Prediction –
Track Changing Per-Branch Behavior

- Store 1 bit per branch – the last outcome.
- Use that to predict!

Diagram:
- States: 0 and 1
- Transitions:
  - From 0 to 1 on 'taken' outcome
  - From 1 to 0 on 'taken' outcome
  - From 0 to 0 on 'not taken' outcome
  - From 1 to 1 on 'not taken' outcome

Predictions:
- Predict Taken
- Predict Not Taken
1-bit Predictor Loop Behavior

wishy washy on loops

Prediction Outcome
T T T T T T N T T T N
T T T T T T N T T T N
(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
Two Bit Dynamic Predictor

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

```
01
¬ taken

11
  taken

10
  ¬ taken
  taken

01
  ¬ taken
  taken

00
  ¬ taken
  taken

¬ taken

00
  ¬ taken

11
  taken
```

Predict Taken

Predict Not Taken
Two bit dynamic predictors

- Better behavior 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”

Branch (direction) Predictor

compute
state
transition

branch info (pc/descr/outcome)

(many cycles later)

read
branch descr

blindly write into this hash table; branches may alias but that’s “ok”

Branch History Table (BHT)

write

branch descr

prediction

is c. branch?

“Guess” PC

ICache

4k n-bit counters

n-bits

Branch History Table (BHT)

compute state transition

branch info (pc/descr/outcome)

branch descr

prediction

is c. branch?

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

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) \( \Rightarrow \) consult BHT0
- \( H = 1 \) (taken) \( \Rightarrow \) consult BHT1

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

Equal storage than 4k x 2bit but better accuracy (for these benchmarks)
Pentium Pro (1995) uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)
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 >= 5-bits of history

Global History/ Prediction
0 1 1 1 1 -> N
1 1 1 1 0 -> T
1 1 1 0 1 -> T
1 1 0 1 1 -> T
1 0 1 1 1 -> T

Doesn’t work for many-iteration loops – but relative error is smaller! Issue: exponential space (see next slide)
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
  - good for memorizing those constant loop constants
  - we don’t store it for each branch, but across all branches.

B. local predictor: track last 10 choices of a single branch
  1024 10-bit entries containing history for that entry
  This history indexes into 1K 3-bit saturating counters

Like correlating predictor with 4096 1-entry tables
Tournament, Correlating, Local Predictor Performance

Consecutively Correctly Predicted Branches w/ 50% probability:

- 2-bit: ~10
- Correlating: ~18
- Tournament: ~25

Spec89 (size presumably in Kbit)
Top N List of Ways to Avoid Branch-Taken Penalties

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
Top N List of Ways to Avoid Branch-Taken Penalties

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?
4. Anulled Branches

**Top N List of Ways to Avoid Branch-Taken Penalties**

*Positive/Negatives?*

+ 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

Instruction Cache

Instr FIFO

positive/negatives?

fix BTB guess?

+4

BTB

GPC

“btb override”

Instruction Cache

EA

feedback

restart new pc branch info

Branch Target Buffer

BTC override

branch info mispredict

instr

pc

pc

pc
BTB Design #1

PC of instruction to fetch

Look up

Predicted PC

Number of entries in branch-target buffer

Positive/Negatives?

Yes: then instruction is branch and predicted PC should be used as the next PC

No: instruction is not predicted to be branch; proceed normally

Override

to i-cache
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.
BTB does not always do a good job of predicting ra’s of functions that have multiple call sites.

Idea: maintain a stack of ra’s

% of return addresses mispredicted