Constructive Computer Architecture:

Branch Prediction

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

- Loop length x pipeline width

What fraction of executed instructions are branch instructions?
How frequent are branches?  ARM Cortex 7

Blem et al [HPCA 2013] Spec INT 2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>branch %</th>
<th>load %</th>
<th>store %</th>
<th>other %</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>1.47E+10</td>
<td>16.0</td>
<td>55.6</td>
<td>13.0</td>
<td>15.4</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.41E+10</td>
<td>8.7</td>
<td>34.6</td>
<td>14.4</td>
<td>42.2</td>
</tr>
<tr>
<td>gcc</td>
<td>5.61E+09</td>
<td>10.2</td>
<td>19.1</td>
<td>11.2</td>
<td>59.5</td>
</tr>
<tr>
<td>gobmk</td>
<td>5.75E+10</td>
<td>10.7</td>
<td>25.4</td>
<td>7.2</td>
<td>56.8</td>
</tr>
<tr>
<td>hmmer</td>
<td>1.56E+10</td>
<td>5.1</td>
<td>41.8</td>
<td>18.1</td>
<td>35.0</td>
</tr>
<tr>
<td>h264</td>
<td>1.06E+11</td>
<td>5.5</td>
<td>30.4</td>
<td>10.4</td>
<td>53.6</td>
</tr>
<tr>
<td>libquantum</td>
<td>3.97E+08</td>
<td>11.5</td>
<td>8.1</td>
<td>11.7</td>
<td>68.7</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2.67E+09</td>
<td>11.7</td>
<td>19.3</td>
<td>8.9</td>
<td>60.1</td>
</tr>
<tr>
<td>perlbench</td>
<td>2.69E+09</td>
<td>10.7</td>
<td>24.6</td>
<td>9.3</td>
<td>55.5</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.34E+10</td>
<td>11.5</td>
<td>39.3</td>
<td>13.7</td>
<td>35.5</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>8.2</td>
<td>31.9</td>
<td>10.9</td>
<td>49.0</td>
</tr>
</tbody>
</table>

Every 12\textsuperscript{th} instruction is a branch
How frequent are branches? **X86**

Blem et al [HPCA 2013] Spec INT 2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>core i7; x86 ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>branch %</td>
</tr>
<tr>
<td>astar</td>
<td>5.71E+10</td>
<td>6.9</td>
</tr>
<tr>
<td>bzip2</td>
<td>4.25E+10</td>
<td>11.1</td>
</tr>
<tr>
<td>hmer</td>
<td>2.57E+10</td>
<td>5.3</td>
</tr>
<tr>
<td>gcc</td>
<td>6.29E+09</td>
<td>15.1</td>
</tr>
<tr>
<td>gobmk</td>
<td>8.93E+10</td>
<td>12.1</td>
</tr>
<tr>
<td>h264</td>
<td>1.09E+11</td>
<td>7.1</td>
</tr>
<tr>
<td>libquantum</td>
<td>4.18E+08</td>
<td>13.2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2.55E+09</td>
<td>16.4</td>
</tr>
<tr>
<td>perlbench</td>
<td>2.91E+09</td>
<td>17.3</td>
</tr>
<tr>
<td>sjeng</td>
<td>2.11E+10</td>
<td>14.8</td>
</tr>
<tr>
<td>Average</td>
<td>9.4</td>
<td>31.0</td>
</tr>
</tbody>
</table>

Every 10\textsuperscript{th} or 11\textsuperscript{th} instruction is a branch
How frequent are branches? ARM Cortex 7

Blem et al [HPCA 2013] Spec FP 2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>branch %</th>
<th>load %</th>
<th>store %</th>
<th>other %</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>3.84E+11</td>
<td>13.5</td>
<td>1.4</td>
<td>0.5</td>
<td>84.7</td>
</tr>
<tr>
<td>cactusADM</td>
<td>1.02E+10</td>
<td>0.5</td>
<td>51.4</td>
<td>17.9</td>
<td>30.1</td>
</tr>
<tr>
<td>leslie3D</td>
<td>4.92E+10</td>
<td>6.2</td>
<td>2.0</td>
<td>3.7</td>
<td>88.1</td>
</tr>
<tr>
<td>milc</td>
<td>1.38E+10</td>
<td>6.5</td>
<td>38.2</td>
<td>13.3</td>
<td>42.0</td>
</tr>
<tr>
<td>tonto</td>
<td>1.30E+10</td>
<td>10.0</td>
<td>40.5</td>
<td>14.1</td>
<td>35.4</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>12.15</strong></td>
<td><strong>4.68</strong></td>
<td><strong>1.95</strong></td>
<td><strong>81.22</strong></td>
<td></td>
</tr>
</tbody>
</table>

Every 8th instruction is a branch
How frequent are branches? **X86**

Blem et al [HPCA 2013] Spec FP 2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>core i7; x86 ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>branch %</td>
</tr>
<tr>
<td>bwaves</td>
<td>3.41E+10</td>
<td>3.2</td>
</tr>
<tr>
<td>cactusADM</td>
<td>1.05E+10</td>
<td>0.4</td>
</tr>
<tr>
<td>leslie3D</td>
<td>6.25E+10</td>
<td>4.9</td>
</tr>
<tr>
<td>milc</td>
<td>3.29E+10</td>
<td>2.2</td>
</tr>
<tr>
<td>tonto</td>
<td>4.88E+09</td>
<td>7.1</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td><strong>3.6</strong></td>
</tr>
</tbody>
</table>

Every 27\textsuperscript{th} instruction is a branch
Observations

- Control transfer happens every $8^{th}$ to $30^{th}$ instruction

- Static vs dynamic predictors: Does the prediction depend upon the execution history?

- Processors often use more than one predictor and it takes considerable effort to
  - Integrate a prediction scheme in the pipeline
  - Understand the interactions between various schemes
  - Understand the performance implications

- There is a plethora of branch prediction schemes – their importance grows with the depth of processor pipeline

  *we will start with the basics...*
RISC V Branches & Jumps

Each instruction fetch depends on some information from the preceding instruction:

1. Is the preceding instruction a taken branch?
2. If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Direction known after</th>
<th>Target known after</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JALR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQ/BNE ...</td>
<td>After Exec</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>

A predictor can redirect the pc only after the relevant information required by the predictor is available.
Overview of control prediction

Given (pc, ppc), a misprediction can be corrected (used to redirect the pc) as soon as it is detected. In fact, pc can be redirected as soon as we have a “better” prediction.
Static Branch Prediction

- Since most instructions do not result in a control transfer, pc+4 is a good predictor.
- Overall probability a branch is taken is ~60-70% but:

  ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  - bne0 (preferred taken)
  - beq0 (not taken)

  ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
  - reported as ~80% accurate

  ... but our ISA is fixed!
Dynamic Branch Prediction

learning based on past behavior

Temporal correlation
- The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation
- Several branches may resolve in a highly correlated manner (a preferred path of execution)
Next Address Predictor: Branch Target Buffer (BTB)

Even small BTBs are very effective.

Zero-size BTB behaves like the pc+4 nap.

- BTB remembers recent target PC’s for a set of control instructions
  - Fetch: looks for the pc and the associated target in BTB; if pc is not found then ppc is pc+4
  - Execute: checks prediction, if wrong poisons the wrong-path instructions; updates the BTB for jumps and taken-branches

BTB permits ppc to be determined before the instruction is decoded.

BTB area can be reduced by making tags arbitrarily small!
Next Addr Predictor interface

```plaintext
interface AddrPred;
    method Addr nap(Addr pc);
    method Action update(Addr pc, Addr nextPC, Bool taken);
endinterface
```

- **Predictor training:** On a pc misprediction, pc and epoch are updated and the relevant information is passed to the next address predictor
  - *nap:* simple look up
  - *update:* On a pc misprediction, if the jump or branch at the pc was taken then the BTB is updated with the new (pc, nextPC) otherwise the pc entry is deleted.

*Code is given at the end*
method Action update(Addr pc, Addr nextPC, Bool taken);

let index = getIndex(pc);
let tag = getTag(pc);

if(taken) begin
  validArr[index] <= True;
  entryPcArr.upd(index, tag);
  ppcArr.upd(index, nextPc);
end

else if(tag == entryPcArr.sub(index))
  validArr[index] <= False;
endmethod
Integrating BTB in the 4-Stage pipeline

... 
AddrPred  btb <- mkBtb

rule fetch;
  iMem.enq(pc[1]);
  let ppcF = nap(pc[1]); pc[1] <= ppcF;
  f2d.enq(Fetch2Decode(pc:pc[1], ppc:ppcF, epoch:epoch[1]));
endrule

rule decode; ... //no change
rule execute; ...
rule writeBack; ... //no change
4-Stage-pipeline without Branch predictors execute

rule execute;
let x = d2e.first; ...
if (epoch[0] != inEp) begin e2w.enq(Invalid); d2e.deq; end
else begin
let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
  if (eInst.iType == Ld)
    dMem.enq(MemReq{op:Ld, addr:eInst.addr, data:?});
  else if (eInst.iType == St) begin
    dMem.enq(MemReq{op:St, addr:eInst.addr,
                      data:eInst.data}); end
  let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
  if (x.ppc != nextPC) begin pc[0] <= eInst.addr;
      epoch[0] <= !epoch[0]; end
  btb.update(pcE, nextPC, eInst.brTaken);
endrule

for btb training
Suppose we maintain a table of how a particular Br has resolved before. At the decode stage we can consult this table to check if the incoming (pc, ppc) pair matches our prediction. If not redirect the pc
Branch Prediction Bits
Remember how the branch was resolved previously:

- Assume 2 BP bits per instruction
- Use saturating counter

<table>
<thead>
<tr>
<th>On ¬taken</th>
<th>On taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1 0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0 1</td>
<td>Weakly ¬taken</td>
</tr>
<tr>
<td>0 0</td>
<td>Strongly ¬taken</td>
</tr>
</tbody>
</table>

Direction prediction changes only after two successive bad predictions.
Two-bit versus one-bit Branch prediction

Consider the branch instruction needed to implement a loop

- with one bit, the prediction will always be set incorrectly on loop exit
- with two bits the prediction will not change on loop exit

A little bit of hysteresis is good in changing predictions
Branch History Table (BHT)

At the Decode stage, if the instruction is a branch then BHT is consulted using the pc; if BHT shows a different prediction than the incoming ppc, Fetch is redirected.

4K-entry BHT, 2 bits/entry, ~80-90% correct direction predictions
4-Stage-pipeline without Branch predictors fetch, decode

```haskell
rule fetch;
  iMem.enq(pc[1]);
  let ppcF = nap(pc[1]); pc[1] <= ppcF ;
  f2d.enq(Fetch2Decode(pc:pc[1], ppc:ppcF, epoch:epoch[1]))
endrule
rule decode;
  let inst = iMem.first; let x = f2d.first;
  if(epoch[1] != x.inEp) begin iMem.deq; f2d.deq end //wrongpath
else begin
  let dInst = decode(inst);
  let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
  if(!stall) begin
    ...fetch register values
    d2e.enq(Decode2Execute{pc: x.pc, ppc: x.ppc,
      dIinst: dInst, epoch: x.epoch,
      rVal1: rVal1, rVal2: rVal2});
    sb.insert(dInst.rDst); iMem.deq; f2d.deq end
  endrule
```

October 25, 2017

http://csg.csail.mit.edu/6.175
4-Stage-pipeline without Branch predictors execute

rule execute;
let x = d2e.first; ...
if(epoch[0] != inEp) begin e2w.enq(Invalid); d2e.deq; end
else begin
let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
if(eInst.iType == Ld)
dMem.enq(MemReq{op:Ld, addr:eInst.addr, data:?});
else if (eInst.iType == St) begin
dMem.enq(MemReq{op:St, addr:eInst.addr,
data:eInst.data}); end
let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
if (x.ppc != nextPC) begin pc[0] <= eInst.addr;
epoch[0] <= !epoch[0]; end
e2w.enq(Valid Exec12Exec2(eInst:eInst, pc:pcE));
d2e.deq;
end
endrule
4-Stage-pipeline without Branch predictors \textit{writeback}

\textbf{rule} writeback;
\begin{verbatim}
let vx = e2w.first;
if (vx matches tagged Valid .x) begin
  let pcE=x.pc; let eInst=x.eInst;
  if (isValid(eInst.dst)) begin
    let data = eInst.iType==Ld ? dMem.first: eInst.data;
    rf.wr(fromMaybe(?., eInst.dst), data);
  end
  if(eInst.iType == Ld) dMem.deq;
end
sb.remove; e2w.deq;
\end{verbatim}
endrule
BTB predictor

module mkBtb(AddrPred);
  RegFile#(BtbIndex, Addr) ppcArr <- mkRegFileFull;
  RegFile#(BtbIndex, BtbTag) entryPcArr <- mkRegFileFull;
  Vector#(BtbEntries, Reg#(Bool))
    validArr <- replicateM(mkReg(False));

  function BtbIndex getIndex(Addr pc) = truncate(pc>>2);
  function BtbTag getTag(Addr pc) = truncateLSB(pc);

  method Addr nap(Addr pc);
    BtbIndex index = getIndex(pc);
    BtbTag tag = getTag(pc);
    if(validArr[index] && tag == entryPcArr.sub(index))
      return ppcArr.sub(index);
    else return (pc + 4);
  endmethod

  method Action update(Addr pc, Addr nextPC, Bool taken);
endmodule
4-Stage-pipeline with BTB
fetch, decode

```verilog
rule fetch;
    iMem.enq(pc[1]);
    let ppcF = btb.nap(pc[1]); pc[1] <= ppcF ;
    f2d.enq(Fetch2Decode(pc:pc[1], ppc:ppcF, epoch:epoch[1]))
endrule

rule decode;
    let inst = iMem.first; let x = f2d.first;
    if(epoch[1] != x.inEp) begin iMem.deq; f2d.deq end //wrongpath
else begin
    let dInst = decode(inst);
    let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
    if(!stall) begin
        ...fetch register values
        d2e.enq(Decode2Execute{pc: x.pc, ppc: x.ppc,
                              dInst: dInst, epoch: x.epoch,
                              rVal1: rVal1, rVal2: rVal2});
        sb.insert(dInst.rDst); iMem.deq; f2d.deq end
endrule
```
4-Stage-pipeline with BTB

execute

rule execute;

let x = d2e.first; ...

if(epoch[0] != inEp) begin e2w.enq(Invalid); d2e.deq; end
else begin

let eInst = exec(dInstE, rVal1E, rVal2E, pcE);

if(eInst.iType == Ld)
    dMem.enq(MemReq{op:Ld, addr:eInst.addr, data:?});
else if (eInst.iType == St) begin
    dMem.enq(MemReq{op:St, addr:eInst.addr, data:eInst.data}); end

let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
if (x.ppc != nextPC) begin pc[0] <= eInst.addr;
    epoch[0] <= !epoch[0];
    btb.update(pcE, nextPC, eInst.brTaken); end

    e2w.enq(Valid Exec12Exec2(eInst:eInst, pc:pcE));
    d2e.deq; end
endrule
4-Stage-pipeline with BTB writeback

```haskell
rule writeback;
  let vx = e2w.first;
  if (vx matches tagged Valid .x) begin
    let pcE=x.pc; let eInst=x.eInst;
    if (isValid(eInst.dst)) begin
      let data = eInst.iType==Ld ? dMem.first: eInst.data;
      rf.wr(fromMaybe(?, eInst.dst), data);
    end
    if(eInst.iType == Ld) dMem.deq;
  end
  sb.remove; e2w.deq;
endrule
```