Today

- Reducing Control Costs
  - Branch Prediction
  - Branch Target Buffer
  - Conditional Operations
  - Speculation
Control Flow

• Previously saw data hazards on control force stalls
  – for multiple cycles
• With superscalar, may be issuing multiple instructions per cycle
• Makes stalls even more expensive
  – wasting n slots per cycle
  – e.g.
    • with 7 instructions / branch
    • issue 7 instructions, hit branch, stall for instructions to complete...

Control/Branches

• Cannot afford to stall on branches for resolution
• Limit parallelism to basic block
  – average run length between branches
Idea

• Predict branch direction
• Execute as if that’s correct
• Commit/discard results after know branch direction
• Use ideas seen for precise exceptions to separate
  – working values
  – architecture state

Goal

• Correctly predicted branches run as if they weren’t there (noops)

• Maximize the expected run length between mis-predicted branches
Expected Run Length

- \( E(l) = L_1 + L_2 P_1 + L_3 P_1 P_2 + L_4 P_1 P_2 P_3 \)
- \( L_i = l, P_i = P \)
- \( E(l) = l(1 + p + p^2 + p^3 + \ldots) \)
- \( E(l) = l/(1-p) \)
- \( E(l) = 1/(\text{probability of mispredict}) \)

Expected Run Length

- \( P = 0.9 \quad 10l \)
- \( p = 0.95 \quad 20l \)
- \( p = 0.98 \quad 50l \)
- \( p = 0.99 \quad 100l \)

- Halving mispredict rate, doubles run length
IPC

- Run for E(l) instructions
- Then mispredict
  - waste ~ pipeline delay cycles (and all work)
- Pipe delay: d
- Base IPC: n
- E(l)/n cycles issue n
- d cycles issue nothing useful
- $\text{IPC} = \frac{E(l)}{E(l)/n + d} = \frac{n}{1 + d/n/E(l)}$

Branch Prediction

- Previous runs
- (dynamic) History
- Correlated
Previous Run

- **Hypothesis**: branch behavior is largely a characteristic of the **program**.
  - Can use data from previous runs (different input data set)
  - to predict branch behavior
- **Fisher**: Instructions/mispredict: 40-160
  - even with different data sets

Data Prediction

- Example shows value (and validity) of feedback
  - run program
  - measure behavior
  - use to inform compiler, generate better code
- **Static/procedural analysis**
  - often cannot yield enough information
  - most behavioral properties are undecidable
Branch History Table

- **Hypothesis**: we can predict the behavior of a branch based on its recent past behavior.
  - If a branch has been taken, we’ll predict it’s taken this time.
- To exploit dynamic strength, would like to be responsive to changing program behavior.

Branch History Table

- **Implementation**
  - Saturating counter
    - count up branch taken; down on branch not taken
  - Predict direction based on majority (which side of mid-point) of past branches
- **Saturation**
  - keeps counter small (cheap)
    - typically 2b
  - limits amount of history kept
    - time to “learn” new behavior
Correlated Branch Prediction

• **Hypothesis**: branch directions are highly correlated
  – a branch is likely to depend on branches which occurred before it.

• Implementation
  – look at last m branches
    • shift register on branch directions
  – use a separate counter to track each of the $2^m$ cases
  – contain cost: only keep a small number of entries and allow aliasing

Branch Prediction

• …whole host of schemes and variations proposed in literature
Prediction worked for Direction...

- Note:
  - have to decode instruction to figure out if it’s a branch
  - then have to perform arithmetic to get branch target
- So, don’t know where the branch is going until cycle (or several) after fetch
  - IF ID EX

Branch-Target Buffer

- Take it one step back and predict target address
- Cache
  - in parallel with Memory Fetch (IF)
  - stores predicted target PC
    - and branch prediction
  - tagged with PC to avoid aliasing
Reducing Number of Branching

• A mispredicted branch costs more than a few cycles in these wide-issue machines
  – potentially n*d
• Especially in cases of reconvergent flow and even branch probabilities

![Diagram](image)

Conditional Operations

• Idea: create guarded operations
  – only change register if some result holds
• e.g.
  – 8b saturating add
    • $c = a + b$
    • if ($t_1 > 255$) $c = 255$
    • if ($t_1 < 0$) $c = 0$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1,R2,R3</td>
<td>SUB R4,R1,#255</td>
<td>CMOVP R1,#255,R4</td>
</tr>
<tr>
<td>COMVN R1,#0,R1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditional Operation Prospect

• For unpredictable branch ($p \approx 0.5$)
  – $E_{wasted\ issue\ slots} = p^n*d \ (n*d/2)$

• With conditional move
  – assume $l$ cycles inside conditional clauses
  – one sided if:
    • $E_{wasted} = p*l \ (l/2)$
  – two sided (both length $l$)
    • $E_{wasted} = l$

• Net benefit for short guarded blocks
  – on wide issue machines

Speculation

• Branch prediction allows us to continue executing
• still have to deal with branch being wrong
• In simple pipelined ISA
  – outstanding branch resolved before writeback
Speculation

• Wide-issue ISA?
  – Likely to have more instructions in flight than mean latency between branches (nd>1)
  – to exploit parallelism, need to continue computing assuming the chosen path is correct
    • means making result values visible to subsequent instructions which may be wrong if control flow goes another way

Old Problem

• Mostly the same problem as precise exceptions
  – want to continue computing forward with tenative values
  – want to preserve old state so can roll-back to known state
Revisit Re-Order

Speculation and Re-Order Buffer

- Compute and bypass values from re-order buffer
- At end of re-order buffer
  - commit to RF (architectural state) in proper program order after branches resolved
  - if branch wrong,
    - nullify it’s effect (results predicated upon)
      - flush re-order buffer (pipeline)
    - direct control flow back to correct branch direction
Details

- As before,
  - exception delivery must be deferred until can commit instruction
  - memory operations require re-order/bypass/commit as well
- History/Future File work
  - ...but transfer time may be more critical in this case

Register Update Unit (RUU)

- Simplescalar uses this
- FIFO unit for instruction management serves for both issue and in-order commit
- Decode: fills empty slots
- Issue: picks next set of runnable instructions
- Execution results return here
- Commit: completed instructions in order from head of FIFO
• Needs to hold all outstanding instructions
  – from: considering for issue
  – to: completion and final RF writeback
• Needs to be relatively large
• Complex?
Reading

• Thursday: available ILP, costs
  – finish HP4 (at least 4.7)
  – quantifying
• Tuesday: VLIW
  – Fisher/VLIW and retrospective

Big Ideas

• Interruptions in Control Flow limit our ability to exploit parallelism
• There is structure in programs
  – predictability in control flow
• Make the common case fast
• Predict/guess common case control flow
  – to generate larger blocks
• Nullify effects of erroneous instructions when guess wrong