Lecture #12

****

Review -- 1 min

****

Branch Prediction

f(address, branch direction, branch history,
global branch history,
opcode (e.g. branch v. return),
...)

Do the math

2-bit works well

Laves the question -- need both target and
prediciton

****

Outline - 1 min

****

other predictors
Branch Target Buffer
Combining Branch Strategies
Multiple Issue
More ILP -- HW and SW

****

Lecture - 20 min

****

Other Prediction

Correlating Branches:

Hypothesis--recent branches are correlated
don't just look at history of current branch
also look at taken/not taken history of past nearby branches

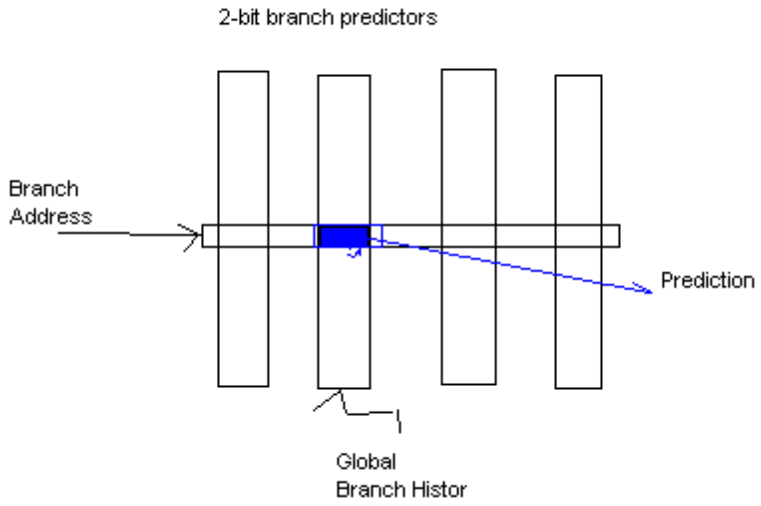
Idea: Record M most recently executed distrinct branches
as taken/not taken, use pattern to select
the proper branch history table

(m,n) predictor

record last m branches to select from 2^m history tables each with n-bit predictors

old 2 bit predictor is (0,2) predictor

Picture



Slide: results

--> big drops for a few

moderate drops for others

QUESTION: why do we treat past branches at this address different than past branches at nearby addresses? (we treat the former as what gets looked up; we treat the latter as "context" – to look up correct table)

→ "nearby branches" is global; no table lookup needed. → practical to use it as part of context (to get current addr's branch history would require table lookup)

→ easy to compress current addr behavior into reasonable function (output of state machine); not so obvious how to treat nearby branches as predictors → need table lookup for that case

Re-evaluating Correlation

SPEC may be non-representative

tiny number of branches responsible for 90% of taken branches

	#=90% of branches
compress	13
eqntott	5
gcc	2020
mpeg	532
real gcc	3214

--> small programs -- thousands of bits for 10 predictions
(seems wrong)

--> real programs and OS more like GCC
(little benefit for extra complexity)

BRANCH TARGET BUFFER

Need both Destination and Prediction

Table: indexed by low bits of PC
returns <tag, predicted PC>

o need to check tag (too expensive to guess wrong)
(also prediction comes back before decode -- would get predictions for non-branch instructions)

o store only TAKEN branches
nontaken = no prediction = increment PC

o combine with branch prediction
1. add prediction FSM to table

2. keep separate branch prediction table
(Each entry of prediction table
smaller than entry of BTB
---> keep many prediction FSMs
per BTB entry)
People really do #2

Branch Folding
---

use BTB to store next instruction, not address
of next instruction

address--><tag, destination+4, mem(destination)

1: IF for branch
--> access BTB
-- returns destination instruction
-- updates PC to instruction(dest+1)
ID for dest, IF for dest+1

--> 0 cycle branch!
(Not just 0 cycle penalty)

Works for unconditional jumps

What about Conditional?--condition never evaluated
solution -- execute jump in parallel with dest instruction
(squash if mispredict)
-- works if they use different resources
-- main conflict: register file
IBM/Moto/Apple POWER architecture -- separate
condition codes

Return Address Stack

Push address on call
pop on return

I said what I wanted to last time -- what is interesting is that branch address is not good predictor of destination -- call pattern is, so use that instead.

****
Admin - 3 min
****
HW2b Friday
Don't forget about projects
Vector discussion moving down a couple weeks
(I'll either skip it or use it to tie together ILP and memory design)

****
Lecture - 24 min
****

CPI < 1 -- Multiple Issue
---

2 Variations
1. Superscaler
Issue varying # instructions per cycled
Scheduled by compiler + HW
2. VLIW -- very long instruction word
fixed number of istructions scheduled by compiler

No commecial VLIW right now
Joint HP-Intel agreement 1998?

Superscaler

Fetch and Issue Multiple Instructions per Cycle

```
IF    ID    EX    MEM WB
IF    ID    EX    MEM WB
IF    ID    EX    MEM WB
IF    ID    EX    MEM WB
      IF    ID    EX    MEM WB
      IF    ID    EX    MEM WB
      IF    ID    EX    MEM WB
      IF    ID    EX    MEM WB
            IF
            ...
```

Typicaly restrict issue --

instead of 5 arbitrary instructions in parallel:

e.g. 2 integer

2 FP

1 branch

Observations

1. ILP -- #issue * pipeline depth

   e.g. in above instruction need 20 instructions that can execute at once

2. Superscaler == Superpipelined

<viewgraph -- get picture from limits of ILP paper>

```
IF    IF    IF    IF    ID    ID    ID    ...
      IF    IF    IF    IF    ID    ID    ...
            IF    IF    IF    IF
```

-- Still need 20 instructions executing at once

3. Complex issue logic

   Examine n opcodes, 3*n registers

   hazard cases proportional to n^2

   pipelining -- limited by clock skew, latch delay

   superscaler limited by issue logic (which may slow clock)

both limited by ILP

4. Increased Branch Hazard
    May slow issue (2 or more cycles)
        --> more cycles lost
    Multiple insructions per clock
        --> each clock cycle lost costs multiple
            instructions
    (notice -- same problem for superpipelined)

5. increase load delay
    same reasons

5. Use lots of hardware
    memory BW for instructions
    register ports
    memory ports
    hazard detection

Approach 2: VLIW
---

avoid issue difficulties by scheduling parallel instrucitons in compiler

e.g. 16 instructions packed into a 168 bit word

Characteristics
    1. all n instuctions proceed in lock step
        (if 1 stalls all stall)
    2. compiler must ensure correct scheduling
        of parallel instructions
        (instructions in same word independent
        by definition)
    3. new HW with different FU's
        --> must recompile

no binary compatibility across generations

Appealing to researchers in architecture -- make HW run fast, punt hard issues to compiler

#2 --> need "hero" compiler (e.g. find much more than basic block worth of parallel instructions)
2M lines of code typical

#3 significant practical concern

Increasing ILP in SW and HW

SW: Trace Scheduling
HW: Annulling instructions
HW: Speculative Execution

Trace Scheduling

Loop unrolling says -- common case is loop is taken
so copy code from after conditional branch to before it

Can do same thing for non-loop conditional branches
called *Trace Scheduling*

picture: tree of execution paths
Trace sceduling -- optimize for most likely path

Like loop unrolling, 2 steps
1. trace selection (~unrolling)
--> create straight line code down likely path

example

```
                if A==0
                        // A1
                        <insr>; <insr>;...
                else
                        //A2
-->likely               <instr>; <instr>; ...
                        if B==0
                                // B1
-->likely                       <instr>...
                        else
                                //B2
                                <instr>...
                //C
                <instr>...


                BEQ R1, A1
        A2:     <instr>
                <instr>
                ...
                BNEZ R2, B2
        B1:     <instr>
                <instr>
                ...
        C:      <instr>
                ...


        A1:     <instr>
                ...
                j C

        B2:     <instr>
                ...
                j C
```

example: if untaken branch faster than taken branch
make the straight line
2. trace compaction (~scheduling)

-->avoid hazards by moving instructions as early as
possible

Trace scheduling harder, less common than loop unrolling
o harder to tell by inspection if branch taken/not taken
(need high prob. branch to make it worth while)
o less regular code -- harder to move around

Annulling Instructions

Tech trends:
branches getting more expensive v. regular instructions
-- superscaler/pipelined --> more cycles delay
-- superscaler --> more instructions per cycle lost
Most vendors have added to new instruction
CMOV -- Alpha, MIPS, PowerPC, SPARC
annul any following instr -- PA-RISC
(-->probably a good idea)

Idea -- change control dependence into a data dependence

Conditional instruction evaluates condition (e.g. register === 0)
if true -- do instruction
if false -- no effect (no register update; no exception)

example: conditional move

```
        BNEZ R1, L
        MOV R2, R3
    L: ....
```

becomes

```
        CMOVZ R2, R3, R1
```

Speculative Execution (HW Support)

Ties together

- o Dynamic branch prediction
- o register renaming
- o in-order commit

Advantages of HW v. Software

1. Disambiguate memory references
2. better branch prediction
3. binary compatibility

Implementation:

1. Branch Target Buffer
   --> don't stall on instructions
   (even if condition depends on pending instructions)
2. renaming + in-order commit via Reorder Buffer
   --> don't commit until branch direction known

Key point: look at reorder buffer

	State	Value
1. ADDD ...		
2. SUB ...		
3. BNEZ R1, LOOP	T	NT
4. MULTD ...		
5. ADD ...		

QUESTION: need to restore PC after mispredicted branch (or exception)

How?

1. Add PC field to reorder buffer (DA: doubles size)
2. keep reorderPC -- increment on every committed instruction; update on taken branch

****
Summay - 1 min
****