Lecture #11

****
Review -- 1 min
****
Want Precise Exceptions
Reorder Buffer --> In-order completion

****
Outline - 1 min
****
Reorder Buffer Example
Branch Prediction
- Branch Prediction
- Branch Target Buffer
- Return Stack

****
Lecture - 20 min
****

Simple Reorder Buffer Example

		Issue	FinisheEx	WB	Commit
ADDD	F0, F2, F4				
SUBD	F6, F0, F8				

Reorder Buffer

Entry	Busy	Instr	State	Destination	Value
1					
2					

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Add1							
Add2							

Register Status

	F0	F2	F4	F6	F8	...
Reorder #:						

reorder buffer
get class to step through how reorder buffer would be implemented (when combined with Tomasulo) in particular -- on issue -- what do you write into register result status? (A: reorder buffer entry -- write
change CDB into reorder buffer (changes CDB to "**go to** " bus)
Renaming -- new name for register is its reorder bufer entry

Key idea – why do you write reorder number into Qj rather than functional unit number. B/c when instruction issues, it may have completed ececution, but not committed yet. In that case, the only place to get that data item is from reorder buffer.

Point of this -- precise exceptions
On exception

    When instruction reaches commit
    handle exception
    restart next instruction

QUESTION: A STUMPER
how do you determine the correct PC to restart
Answer 1: store PC with each instruction in reorder buffer

    DA: doubles amount of storage

Answer 2: keep a counter; update it on a taken branch

    increment it on each committed instruction

Branch Prediction

Motivation:

1. Solving other bottlenecks
Tomasulo --> can start instructions as fast as you can
supply them
--> work hard to supply instructions at every cycle
--> branch prediction
2. Superscalar
multiple instructions per cycle -->
multiple instructions lost per cycle of
branch delay

Approaches
----
1. Branch Prediction Table
Predict taken/not taken early
(but need both taken/not taken and destination
this still useful -- destination can be in instruction
wheras taken/not taken may depend on earlier, not finished
instructions

2. Branch Target Buffer
Predict taken/not taken AND destination

3. Return stack
#1 and #2 are caches indexed by branch address
returns' destination not predicted by where the
return lives, it is predicted by where
you called from

****
Admin - 3 min
****

HW2a today
HW2b next week
Project checkpoint – coming up

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

Prediction
History – a bunch of branch algorithms got invented in ad-hoc ways. Then theory to understand what is going on came along. Turns out this theory is really general – branch prediction, prefetching, data compression (that's where it comes from). I'll start with theory and then describe specific algorithms in that context.

when you get to a branch at some address, you want to predict whether it is taken or not taken. How would you do this?

You might ask, what happens most often when I am in this situation? (leave "this situation" deliberately vague)

```
if(takenCount[thisSituation] > notTakenCount[thisSituation]){
        predict taken
else
        predict not
```

"This situation" = context. Use whatever you want for context
Good candidates

- address of branch
- whether the last time you were here branch was taken or not
- whether the last time you were at some other branch, branch was taken or not
- direction of target (forwards/backwards)
- whether there have been an even or odd number of instructions since last taken branch

…

So generic strategy is: encode context as bit string
use context to lookup result count

predict what is more common result

This is “markov” prediction
given state XXXXXXX and count T=.43 and NT=.57
that says probability of next being in state
XXXXXXXT is 43%
XXXXXXXN is 57
If you look at a fixed number of context bits, you get a markov state machine with known probabilities of transition

This is the best you can do with fixed # context bits

Prediction by partial matching
You would think that more bits of context menas better predictions, right?
But take it to an extreme – 1M bits of context
2^1M possible contexts
It takes you a long time before you can even make your first prediction
Seldom get into same state twice → seldom have a good guess

PPM says: keep predictions for 1M bits of context, 999999 bits of context, 999998 bits f context, …
Use the most context for which there is a valid count

→ faster “warm up” → recognize “patterns” more quicly

This is what gets used in text compression:
given that I’ve just seen “Th” what do I expect to see next?

e	1338
a	781
i	301
u	12
y	1

(notice – no longer just predicting “yes”/”no” – predict probability of each of several possibilities

Same idea works for prefetching
given that I’ve just read addresses 2112, 2113, 2114, 2115, what do I expect to read next?

Practical Prediction

2 problems

1) finite amount of state
   limited number of contexts/counters
   limited counter size

   bigger means more expensive and slower

2) what to use for context?
   (finite amount of state → only a few history counters → only get to use a few bits of context to pick a counter → choose context carefully)

1. Branch Prediction Table

Table: <Address, recent branch history> -->Taken/Not Taken prediction

Engineering – can't keep different counters for all addresses:
→ Address: Lower bits of PC Adress

Several addresses will map to same counters. Do we need a tag?
NO: (No address check)
Incorrect address-->poor prediction
Bigger table --> fewer conflict misses

Engineering – when you reach an address, how do you come up with the rest of the context (e.g. history of taken/not taken branches for that address) → look it up????
Seems pretty slow – lookup history(address), lookup prediction(address, history)

Engineering – consider what you will see when you lookup
history(address) = <TTTTTT>
Do you really need to now use this full context to lookup the prediction?
Do you really need to keep 2 big counters for this entry?

No – the answer is (almost always): predict T

For a given address, we can lookup the context, and pretty much figure out the right prediction for the context *a priori*

address → history for that address → prediction
table lookup hard-wired

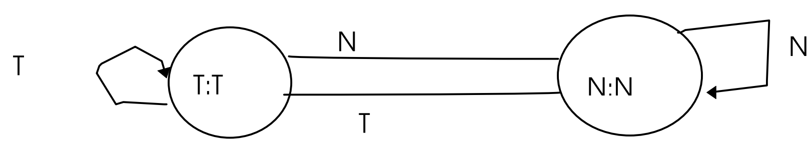
Encode history as a state machine. Prediction encoded by current state; next state transition from result of branch

Prediction State Machines

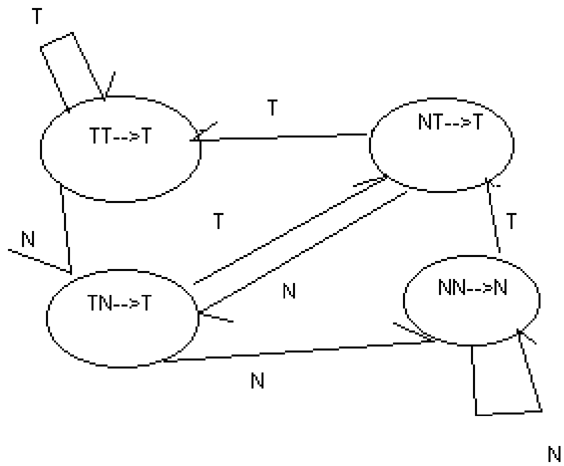
Several ways to build 2-bit predictor
--> talk about how you should decide

0-bit
"Are more branches taken or not taken"
-->predict taken

1-bit
"Given that last branch was "taken" or "not taken"
are more brnaches taken or not



2-bit "Given last two branches..."



(This doesn't look like either of the
two pictures in book -- I'll
come back to that in a second)

N-Bit

Given pattern XXXXX which is more likely
XXXXXT or XXXXXN?

You could do this experiment.

Nonuniform Prediction

Intuition:

o TN and NT states don't buy us much accuracy
o FSM does wrong thing for common case of branches

--> Don't map past states directly to FSM
(Make it more likely that 3-branch series maps on to correct
2-branch state)

Notion -- after a long series of branches in one direction,
make it take 2 mispredictions to change prediction

Patterson Text: Saturating 2-bit counter

```
 |----    ---n--->      --n-->      --n-->            --|
t    (T)           (T?)          (N?)          (N)       n
```

```
|----     <--t--          <--t--        <--t--          <----|
```

Patterson Picture

```
T       ---n-->              T?
^       <--t---
|                            |
t                            n
|                            |/
N?      ---n--->             N
        <---t---
```

SLIDE:

Allen Smith Jan 1984 _Computer_

	IBM_CPL	IBM_BUS	IBM_SCI	IBM_SUP	PDP11	CDC6400	
state == past 2 branches	93.3		96.5	86.6	83.4	97.5	90.6
state== past5 branches	94.7		97.1	92.2	83.9	98.2	95.7
saturating 2-bit counter	93.8		96.2	91.3	80.2	97.8	86.4
patterson picture	93.8		96.2	91.3	80.2	97.8	89.1

Answer:

1. Winner depends on workload
2. Nonuniform history NOT a clear win
3. Patterson picture MAY be slightly better than saturating
4. more states somewhat better
5. Might not use saturating counter for more states

Bottom line: 2-bit prediction works really well and is widely used

Other Prediction context
----

k-bit prediction takes addres + result of last k branches at that address to predict next branch

Could use other stuff to predict next branch.

(Other contexts)

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

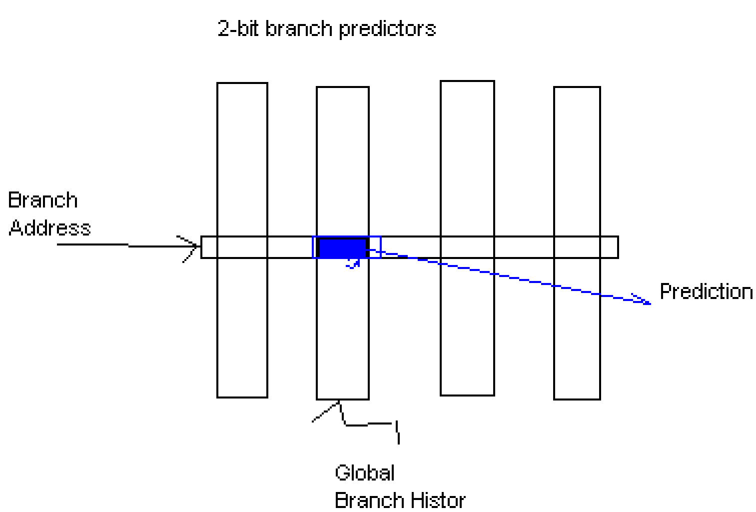
Idea: Record M most recently executed distrinct branches
as taken/not taken, use pattern to select
the proper branch history table

(m,n) predictor
use (address, result lastM branches) as “context” to look up
most relevant prediction; use n-bit counter to

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

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 POWER architecture -- separate conditino codes

Return Address Stack

    Push address on call
    pop on return

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

Branch prediction
    f(
    forward/backwards branch, (static prediction)
    past history @ current PC,
    past history @ other branches,
    call/return pattern,
    ...)