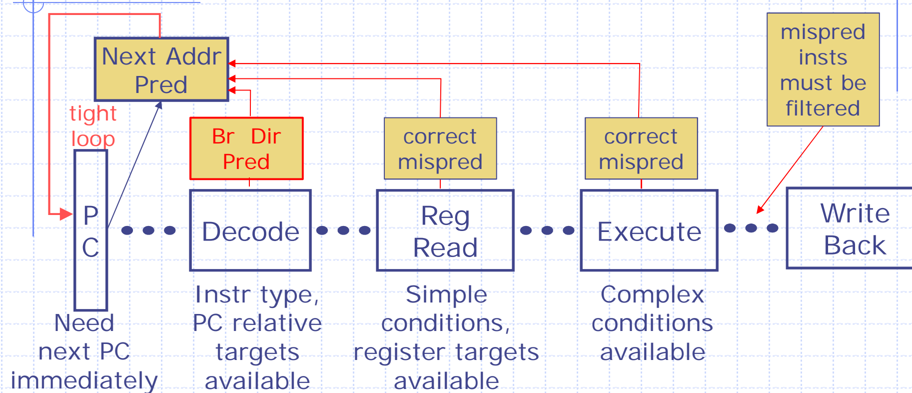
Constructive Computer Architecture:

Branch Prediction: Direction Predictors

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Multiple Predictors: BTB + Branch Direction Predictors



- 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

On ¬taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly ¬taken
		0	0	Strongly ¬taken

?

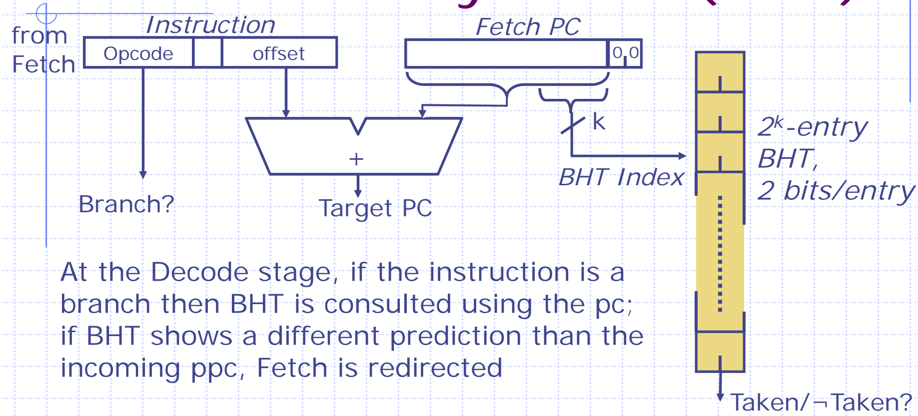
Direction prediction changes only after two successive bad predictions

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

Exploiting Spatial Correlation

Yeh and Patt, 1992

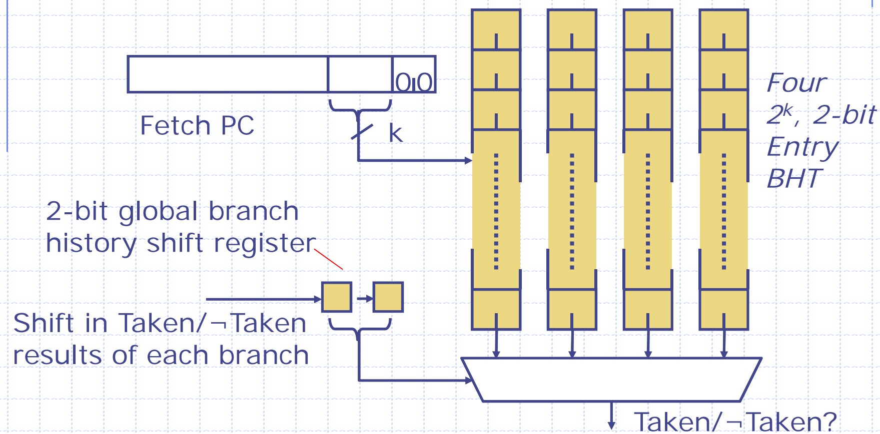
```
if (x[i] < 7) then
        y += 1;
if (x[i] < 5) then
        c -= 4;
```

If first condition is false then so is second condition

History register, H, records the direction of the last N branches executed by the processor and the predictor uses this information to predict the resolution of the next branch

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Where does BHT fit in the processor pipeline?

- BHT can only be used after instruction decode
- We still need the next instruction address predictor (e.g., BTB) at the fetch stage
- *Predictor training:* On a pc misprediction, information about redirecting the pc has to be passed to the fetch stage. However for training the branch predictors information has to be passed even when there is no misprediction

Multiple predictors in a pipeline

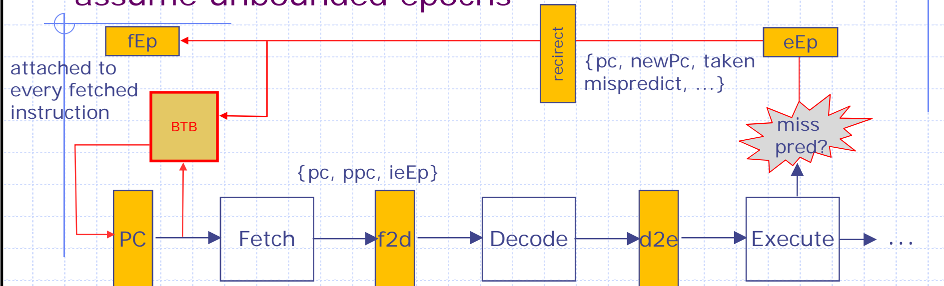
- At each stage we need to take two decisions:
  - Whether the current instruction is a *wrong path instruction*. Requires looking at epochs
  - Whether the prediction (ppc) following the current instruction is good or not. Requires consulting the prediction data structure (BTB, BHT, ...)
- Fetch stage must correct the pc unless the redirection comes from a known wrong path instruction
- Redirections from Execute stage are always correct, i.e., cannot come from wrong path instructions

Dropping *vs* poisoning an instruction

- Once an instruction is determined to be on the wrong path, the instruction is either dropped or poisoned
- Drop: If the wrong path instruction has not modified any book keeping structures (e.g., Scoreboard) then it is simply removed
- Poison: If the wrong path instruction has modified book keeping structures then it is poisoned and passed down for book keeping reasons (say, to remove it from the scoreboard)
- Subsequent stages know not to update any architectural state for a poisoned instruction

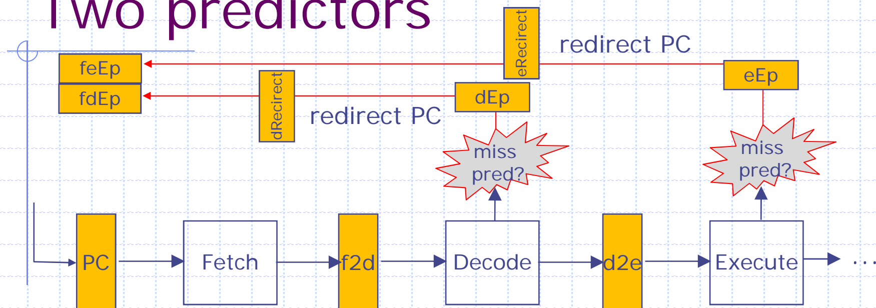
N-Stage pipeline – BTB only

assume unbounded epochs

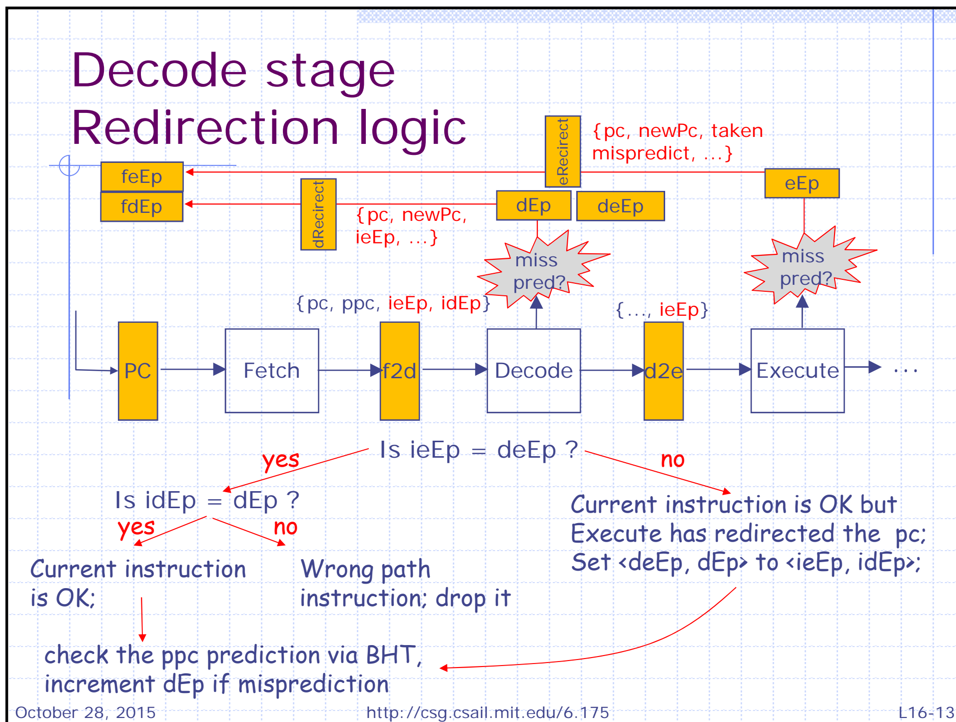


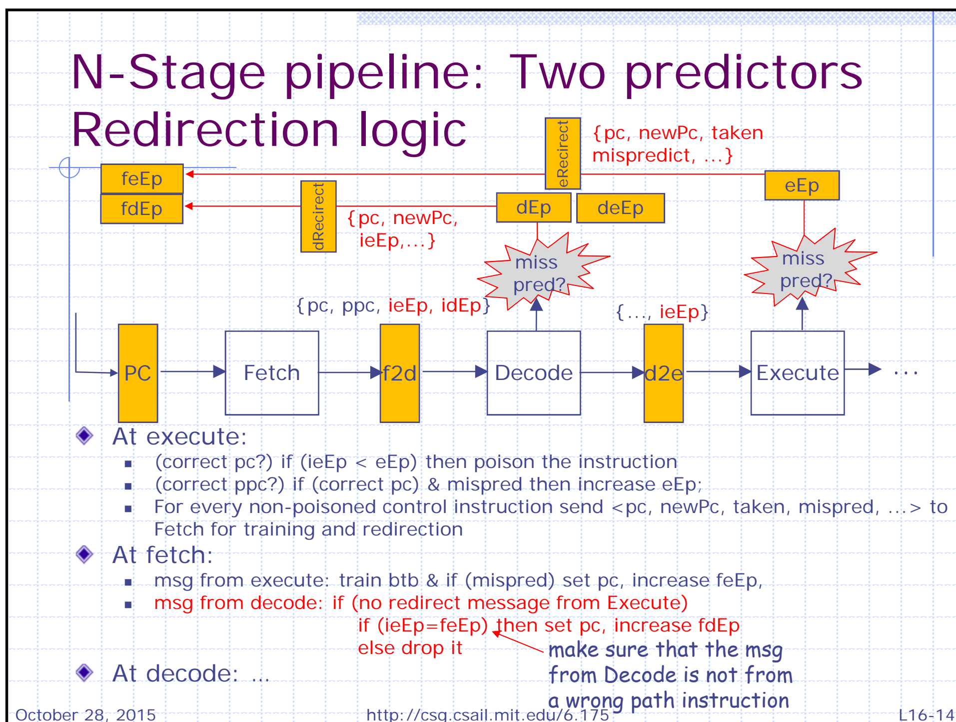
- At Execute:
  - (correct pc?) if (ieEp < eEp) then mark the instruction as poisoned
  - (correct ppc?) if (correct pc) & mispred then increase eEp
  - For every control instruction send <pc, newPc, taken, mispred, ...> to Fetch for training and redirection
- At Fetch:
  - msg from Execute: train BTB with <pc, newPc, taken, mispred> and if msg from Execute indicates misprediction then set pc, increase fEp

N-Stage pipeline: Two predictors

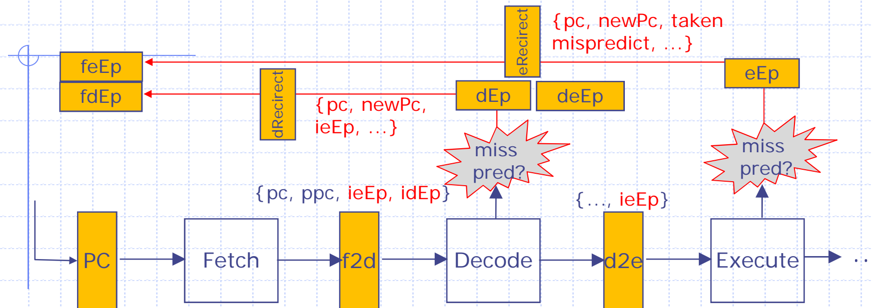


- Both Decode and Execute can redirect the PC; Execute redirect should never be overruled
- We will use separate epochs for each redirecting stage
  - feEp and deEp are estimates of eEp at Fetch and Decode, respectively. deEp is updated by the incoming eEp
  - fdEp is Fetch's estimates of dEp
  - Initially all epochs are set to 0
- Execute stage logic does not change


Decode stage
Redirection logic
eRecirect
{pc, newPc, taken
mispredict, …}
feEp
fdEp
dRecirect
{pc, newPc,
ieEp, …}
dEp
deEp
eEp
miss
pred?
miss
pred?
{pc, ppc, ieEp, idEp}
{…, ieEp}
PC
Fetch
f2d
Decode
d2e
Execute
…
Is ieEp = deEp ?
yes
no
Is idEp = dEp ?
yes
no
Current instruction
is OK;
Wrong path
instruction; drop it
Current instruction is OK but
Execute has redirected the pc;
Set <deEp, dEp> to <ieEp, idEp>;
check the ppc prediction via BHT,
increment dEp if misprediction
October 28, 2015
http://csg.csail.mit.edu/6.175
L16-13


N-Stage pipeline: Two predictors
Redirection logic
eRecirect
{pc, newPc, taken
mispredict, …}
feEp
fdEp
dRecirect
{pc, newPc,
ieEp,…}
dEp
deEp
eEp
miss
pred?
miss
pred?
{pc, ppc, ieEp, idEp}
{…, ieEp}
PC
Fetch
f2d
Decode
d2e
Execute
…
At execute:
(correct pc?) if (ieEp < eEp) then poison the instruction
(correct ppc?) if (correct pc) & mispred then increase eEp;
For every non-poisoned control instruction send <pc, newPc, taken, mispred, …> to
Fetch for training and redirection
At fetch:
msg from execute: train btb & if (mispred) set pc, increase feEp,
msg from decode: if (no redirect message from Execute)
if (ieEp=feEp) then set pc, increase fdEp
else drop it
make sure that the msg
from Decode is not from
a wrong path instruction
At decode: …
October 28, 2015
http://csg.csail.mit.edu/6.175
L16-14

One bit epoch does not work



- The decode redirect which is issues in eEp should only kill instructions in the same eEp in Fetch
- Suppose a message has red eEpoch and sits for a long time in dRedirect then by the time Fetch reads it eEpoch may have changed to green and again to red. In such a situation the message in dRedirect should be discarded
- For one-bit epoch solution see Khan, Wright and Zhang

Discussion

- The number of entries in BTB is small both because of the need for fast access and the need to store the target address (small and fat)
- The number entries in BHT is large (thin and tall)
- We can keep the history bits for branches in the BTB also to improve performance; alternatively we can set the branches to be always-taken
- Jumps through registers (JALR) are problematic and perhaps should not be kept in the BTB

Uses of Jump Register (JALR)

- Switch statements (jump to address of matching case)

  BTB will work well only if the same case is used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB will work well only if the same function is called repeatedly, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB is not likely to work because a function is called from many distinct call sites!

How can we improve subroutine call transfers?

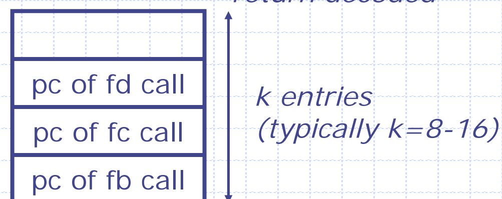
Subroutine Return Stack

- A small structure to accelerate JR for subroutine returns is typically much more accurate than BTBs

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

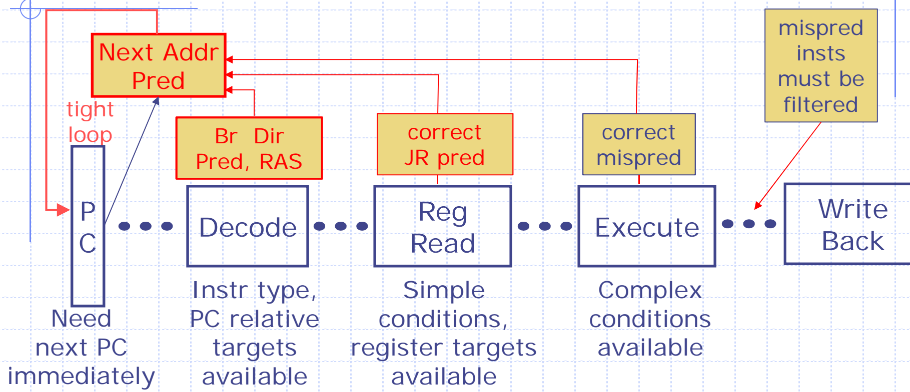
Push call address when function call executed

Pop return address when subroutine return decoded



Don't keep these instructions in BTB

Multiple Predictors: BTB + BHT + Ret Predictors



- Multiple predictors are common; one of the PowerPCs has all the three predictors
- Performance analysis is quite difficult – depends upon the sizes of various tables and program behavior
- The system must work even if every prediction is wrong