Changelog

18 Apr 2023: adjust spacing on control hazard pipeline diagram exercise

18 Apr 2023: fix static prediction: adjust jl to jg, use for_loop_top label at bottom correctly, adjust comments to match assembly

18 Apr 2023: predict: repeat last: have assembly use jnz instead of jz

so far

automating building programs

HW tools for OSes exceptions, virtual memory

OS security policies accounts

networking layers, secure communication

threading+concurrency

HW performance tricks caching, pipelining

last time (1)

data hazard problem

- pipelining changes order of register reads/writes

hazard v dependency

forwarding to resolve data hazard

- value available before it is stored
- add 'shortcut' to send to right place
- add logic to select shortcut conditionally (compare register #s)

last time (2)

forwarding paths + choosing most recent version

combining stalling with forwarding

control hazards

- can't figure out next instruction after fetch?

anonymous feedback (1)

"Are you going to post a study guide for the final? I think this class is a bit all over the place, so it would be really nice to have maybe a practice exam, or some sort of study guide so that we know how to study. Thanks :)"

- probably not going to have a practice exam
- do now have some exams from pilot posted
- likely going to try to look over what's covered in schedule in writing exam
- (roughly equal weight for each week of class)

anonymous feedback (2)

"The exercises in class are actually insanely helpful in helping me understand the topic. Also, you're a goated professor and teach the class so well even though the course material is very difficult."

re: exercises — definitely try to have them for most topics
(also helpful for me to know if everyone's lost)

"What are some of your hobbies?"

hiking, birding, drawing

anonymous feedback (3)

"Some of the TA's do a horrible job of going to their own OH. I sat on discord for over an hour and a half on Monday and none of the scheduled TAs were even on discord. This is not an easy class and when I show up to get help, it should be available as advertised. "

This indeed is something that should not happen...

quiz Q1

cycle time = longest stage time = 500 ps

fetch starts at time 0ps

decode starts at time 500ps

execute starts at time 1000ps

memory starts at time 1500ps

writeback starts at time 2000ps

quiz Q2

no hazards: once pipeline full
start/finish one new instruction every cycle

first instruction finishes after 6 cycles

then 1 instruction finishes every cycle

$\rightarrow (6 + 99\ 999\ 999)$ cycles

$\approx 100\ 000\ 000 \cdot 600$ps $=$ 60 ms

quiz Q3ab

6 to 5 pipeline stages by combining two stages

cycle time likely to be higher

- if third+fourth stage were slowest, maybe slightly less than twice
- if third+fourth stage were fastest, maybe only slightly
- if not higher, than we started with a poor choice of pipeline stages

higher cycle time $\rightarrow$ lower throughput

- assuming stalls are not extremely common, which is likely with forwarding

if pipeline stages are well-balanced before/after change, lower latency

if pipeline stages not well-balanced after, more wasted time $\rightarrow$

quiz Q4

usually fetch new instruction + complete new instruction every cycle

5% take two extra cycles to fetch new instruction

implies later two extra cycles to complete new instruction

10% take one extra cycle

$(1 + .05 * 2 + .1 * 1) = 1.2$ average cycles per instruction

$\times 1000$ps $=$ 1200 ps average time

quiz Q5

if no hazards:

```
addq %r8, %r9    F1 F2 D  E1 E2 E3+M1 M2 W
xorq %r8, %r11      F1 F2 D  E1 E2 E2 M1 M2 W
subq ***, %r10         F1 F2 D  E1 E2 E3 M1 M2 W
```

but subq uses %r9 from addq, so it can't start E1 until after addq computes its value

```
addq %r8, %r9    F1 F2 D  E1 E2 E3+M1 M2 W
xorq %r8, %r11      F1 F2 D  E1 E2 E2 M1 M2 W
subq %r9, %r10         F1 F2 D  D +E1 E2 E3 M1 M2 W
                                        1  2  3
```

jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cmpq %r8, %r9

jne LABEL

(do nothing)

(do nothing)

xorq %r10, %r11

movq %r11, 0(%r12)

…

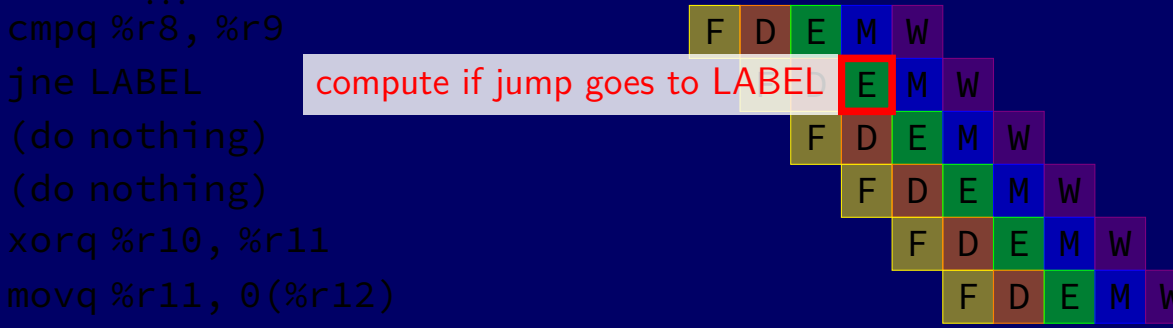
jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
…

jXX: stalling?

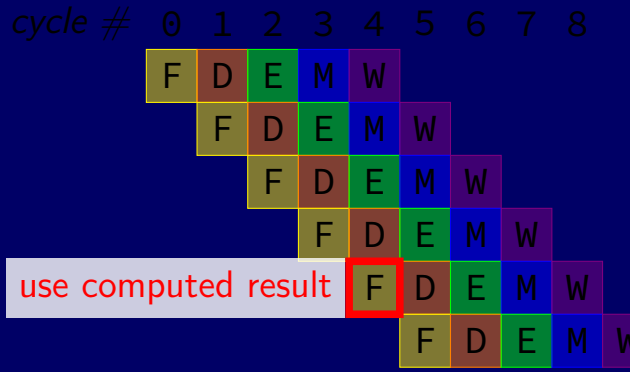
```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```



jXX: stalling?

```
cmpq %r8, %r9
jne LABEL        // not taken
xorq %r10, %r11
movq %r11, 0(%r12)
...
```

cmpq %r8, %r9
jne LABEL
(do nothing)
(do nothing)
xorq %r10, %r11
movq %r11, 0(%r12)
…



making guesses

```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

speculate (guess): jne won't go to LABEL

right: 2 cycles faster!; wrong: undo guess before too late

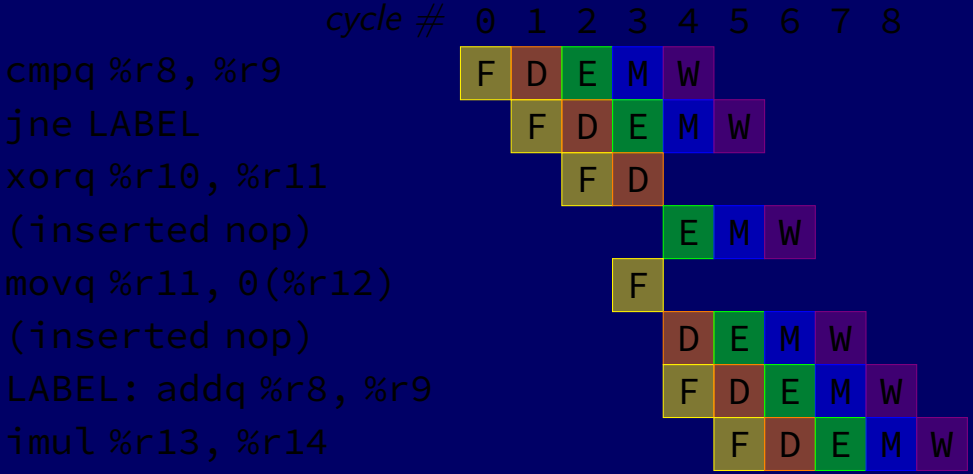
jXX: speculating right (1)

```
        cmpq %r8, %r9
        jne LABEL
        xorq %r10, %r11
        movq %r11, 0(%r12)
        ...

LABEL:  addq %r8, %r9
        imul %r13, %r14
        ...
```

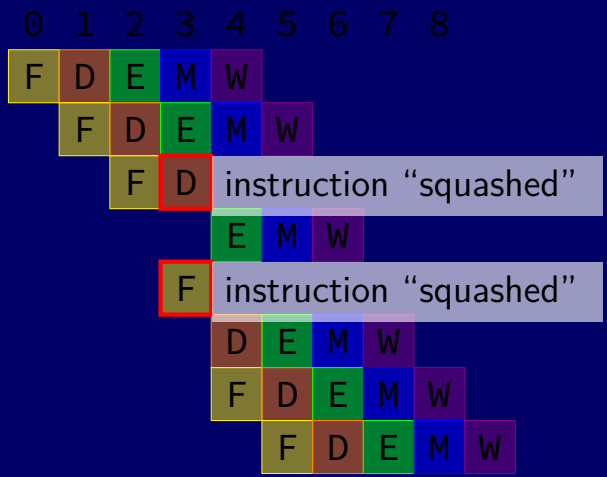
```
cmpq %r8, %r9
jne LABEL
xorq %r10, %r11
movq %r11, 0(%r12)
…
```

jXX: speculating wrong



jXX: speculating wrong

```
cmpq %r8, %r9
jne LABEL
xorq %r10, %r11
(inserted nop)
movq %r11, 0(%r12)
(inserted nop)
LABEL: addq %r8, %r9
imul %r13, %r14
…
```



“squashed” instructions

on misprediction need to undo partially executed instructions

mostly: remove from pipeline registers

more complicated pipelines: replace written values in cache/registers/etc.

performance

hypothetical instruction mix

kind	portion	cycles (predict not-taken)	cycles (stall)
taken jXX	3%	3	3
non-taken jXX	5%	1	3
others	92%	1*	1*

* — ignoring data hazards

performance

hypothetical instruction mix

kind	portion	cycles (predict not-taken)	cycles (stall)
taken jXX	3%	3	3
non-taken jXX	5%	1	3
others	92%	1*	1*

predict: $3 \times .03 + 1 \times .05 + 1 \times .92 =$ 1.06 cycles/instr.

stall: $3 \times .03 + 3 \times .05 + 1 \times .92 =$ 1.16 cylces/instr. ($1.19 \div 1.09 \approx 1.09$x faster)

* — ignoring data hazards

exercise: control hazard timing+forwarding?

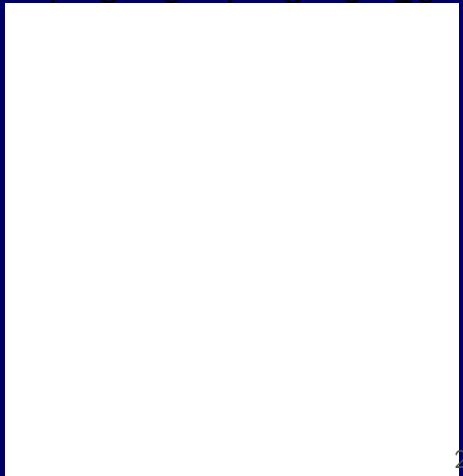
with F/D/E/M/W: what is fetched when? what is forwarded?

	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) `addq %rcx, %r9`												
(2) `jne foo` (predicted taken, actually not)												
(3) `subq %rax, %r9`												
(4) `call bar`												
(5) `bar: pushq %r9`												

[solution]: control hazard timing+forwarding?

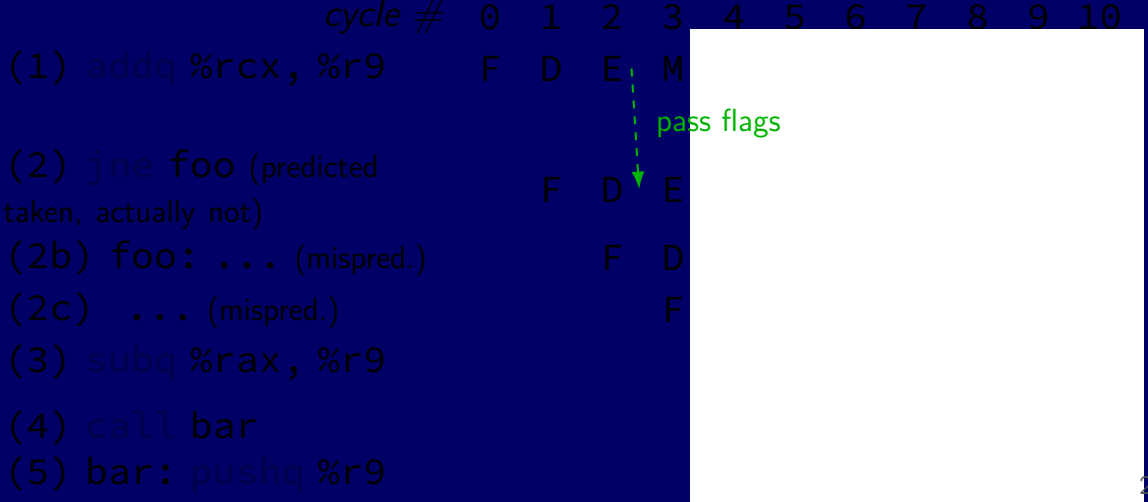
with F/D/E/M/W: what is fetched when? what is forwarded?

	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) `addq %rcx, %r9`		F	D	E	M							
(2) `jne foo` (predicted taken, actually not)				F	D	E						
(2b) `foo: ...` (mispred.)					F	D						
(2c) `...` (mispred.)						F						
(3) `subq %rax, %r9`												
(4) `call bar`												
(5) `bar: pushq %r9`												



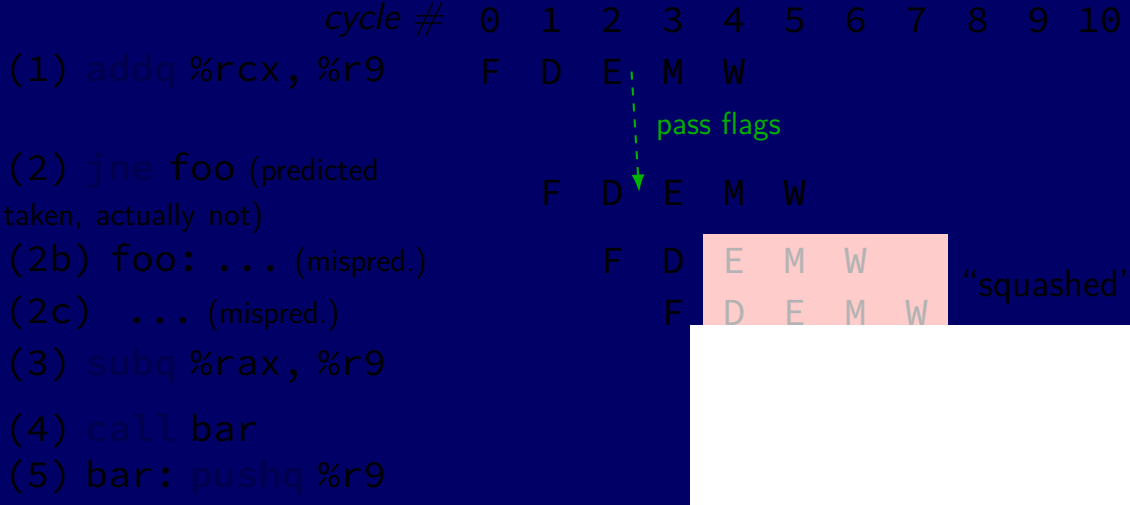
[solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?



[solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?



[solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

[solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

[solution]: control hazard timing+forwarding?

with F/D/E/M/W: what is fetched when? what is forwarded?

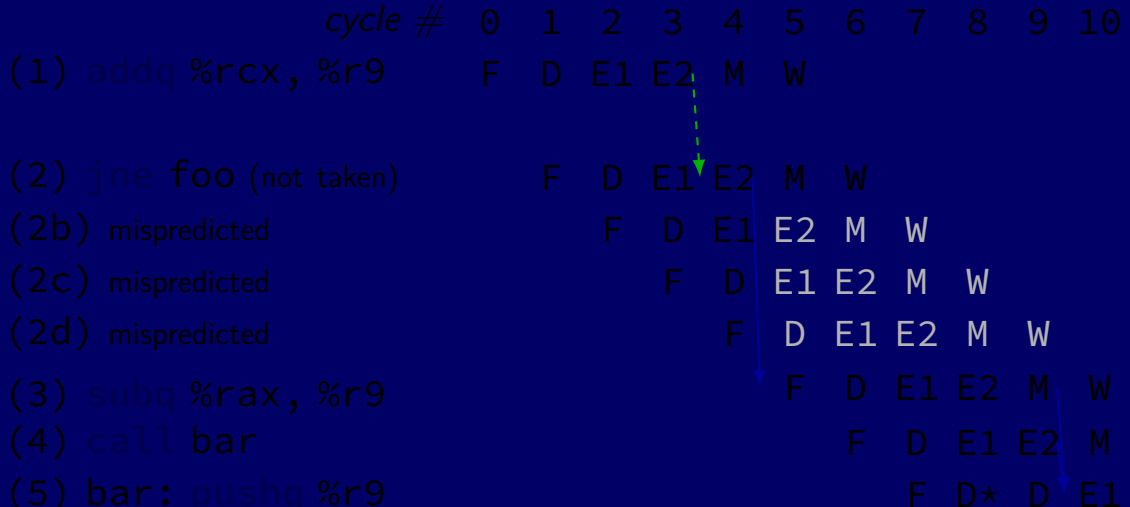
exercise: with different pipeline

with F/D/E1/E2/M/W

	cycle #	0	1	2	3	4	5	6	7	8	9	10
(1) `addq %rcx, %r9`												
(2) `jne foo` (not taken)												
(3) `subq %rax, %r9`												
(4) `call bar`												
(5) `bar: pushq %r9`												

[solution]: with different pipeline

with F/D/E1/E2/M/W



static branch prediction

forward (target $>$ PC) not taken; backward taken

intuition: loops:

```
LOOP: ...
      ...
      je LOOP

LOOP: ...
      jne SKIP_LOOP
      ...
      jmp LOOP
SKIP_LOOP:
```

exercise: static prediction

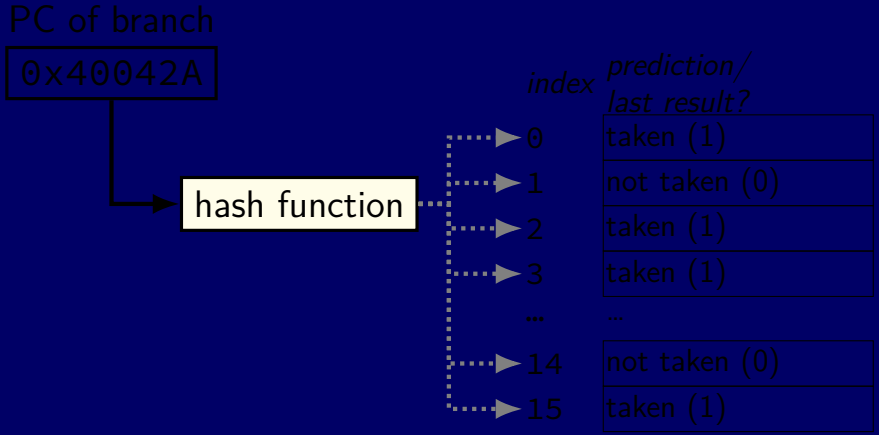
```
.global foo
foo:
    xor %eax, %eax // eax <- 0
foo_loop_top:
    test $0x1, %edi
    je foo_loop_bottom  // if (edi & 1 == 0) goto .Lskip
    add %edi, %eax
foo_loop_bottom:
    dec %edi               // edi = edi - 1
    jg for_loop_top // if (edi > 0) goto for_loop_top
    ret
```

suppose %edi = 3 (initially)

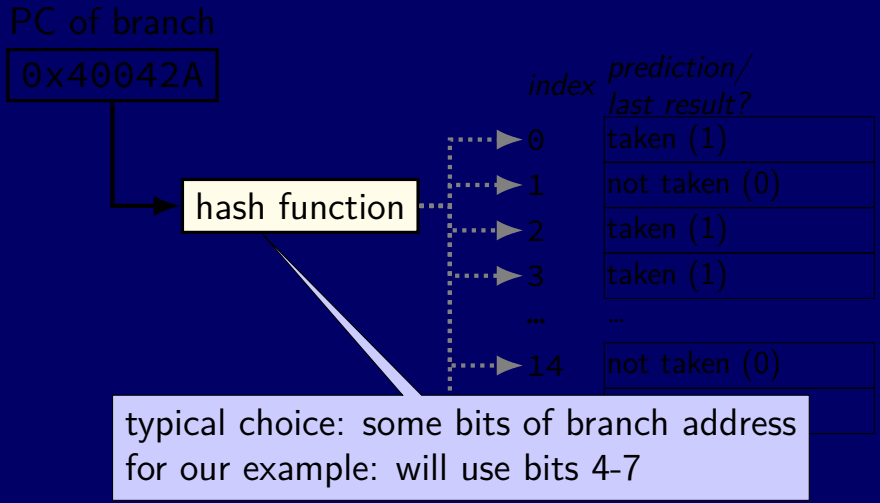
and using forward-taken, backwards-not-taken strategy:

how many mispreditions for je? for jl?

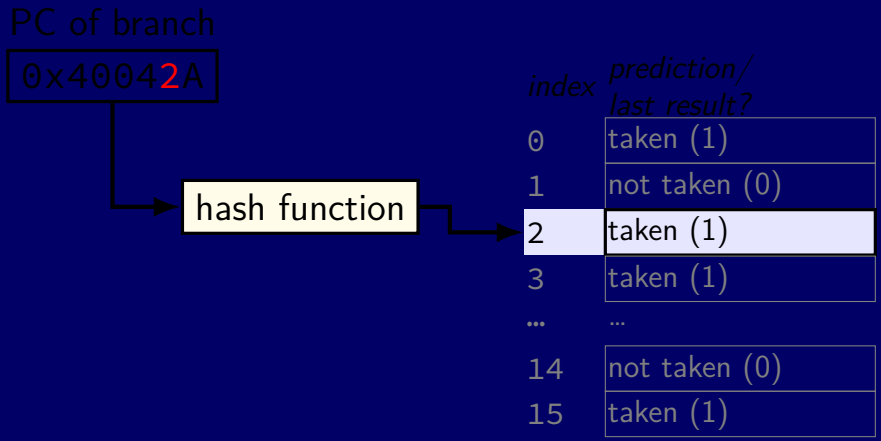
predict: repeat last



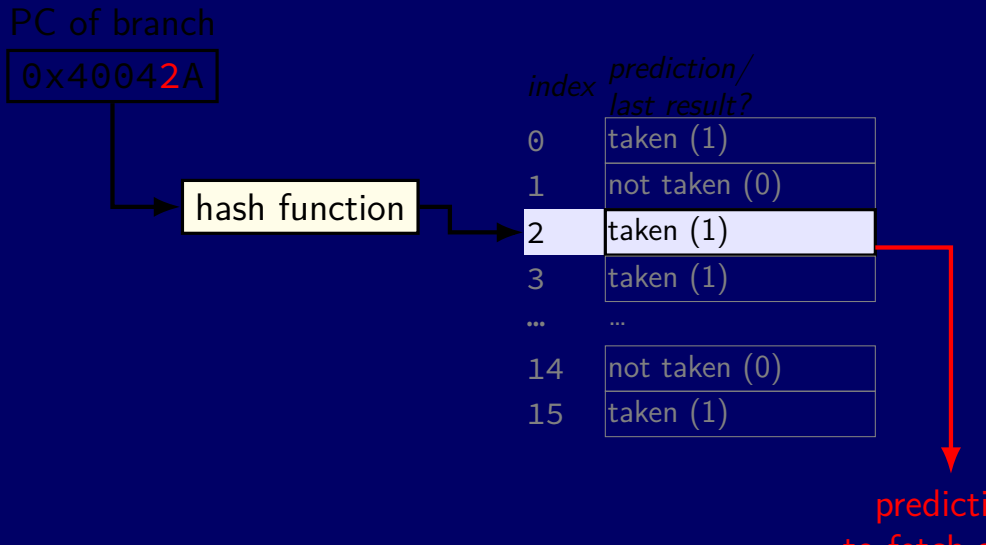
predict: repeat last



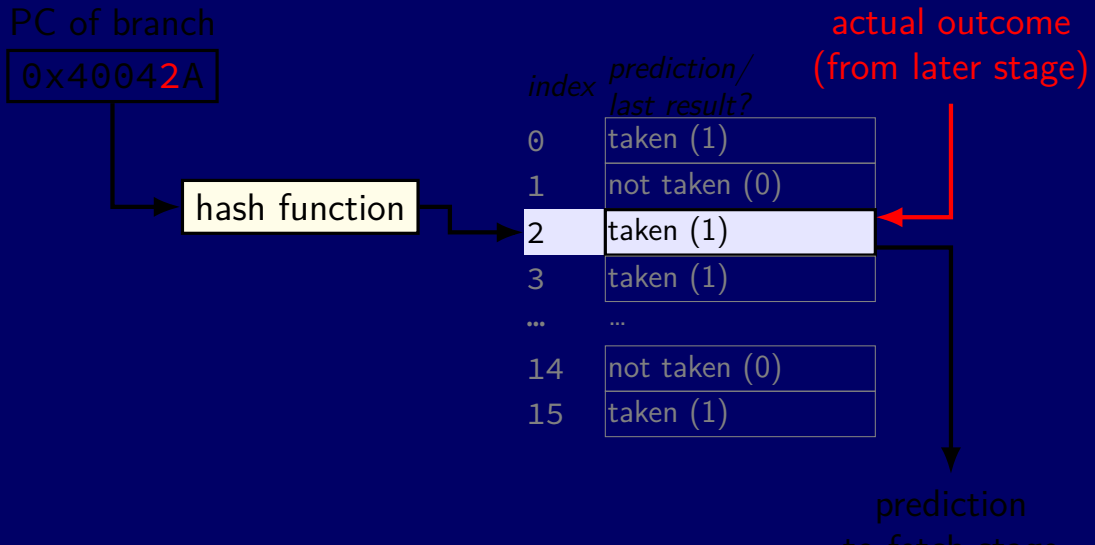
predict: repeat last



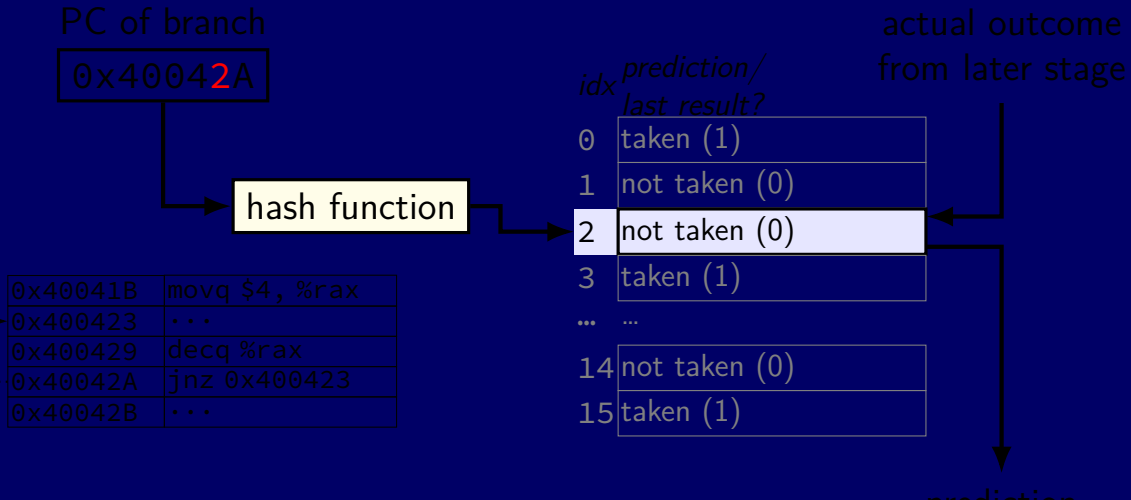
predict: repeat last



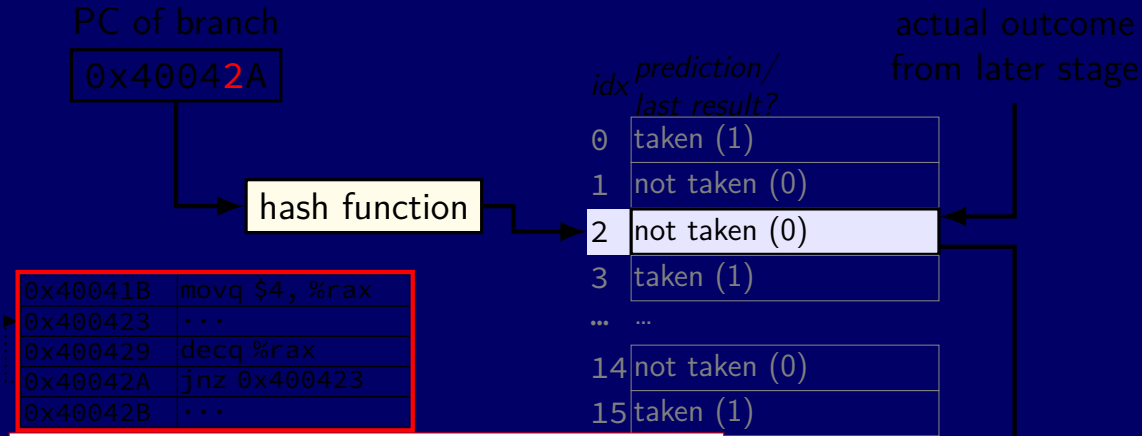
predict: repeat last



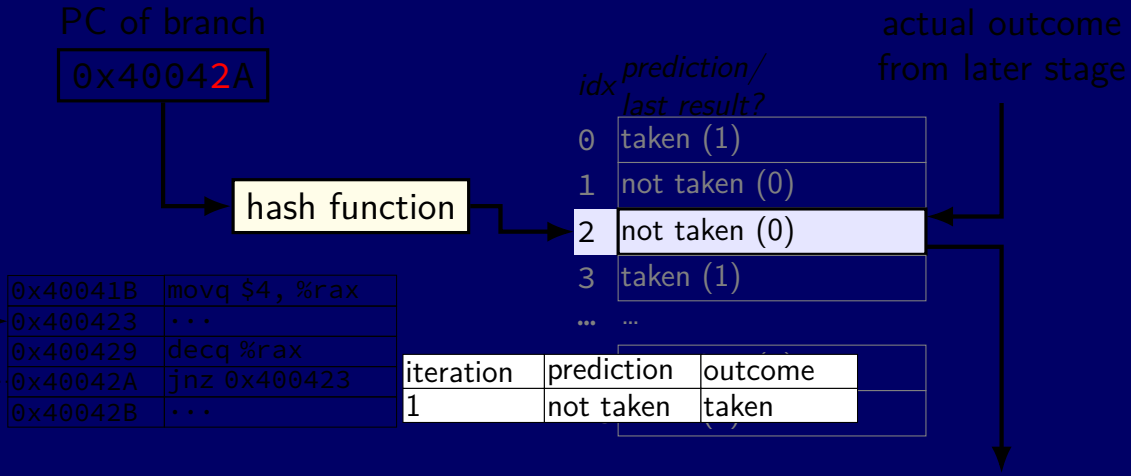
example


PC of branch
0x40042A
hash function
idx
prediction/
last result?
0 taken (1)
1 not taken (0)
2 not taken (0)
3 taken (1)
... ...
14 not taken (0)
15 taken (1)
actual outcome
from later stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ...
prediction
to fetch stage

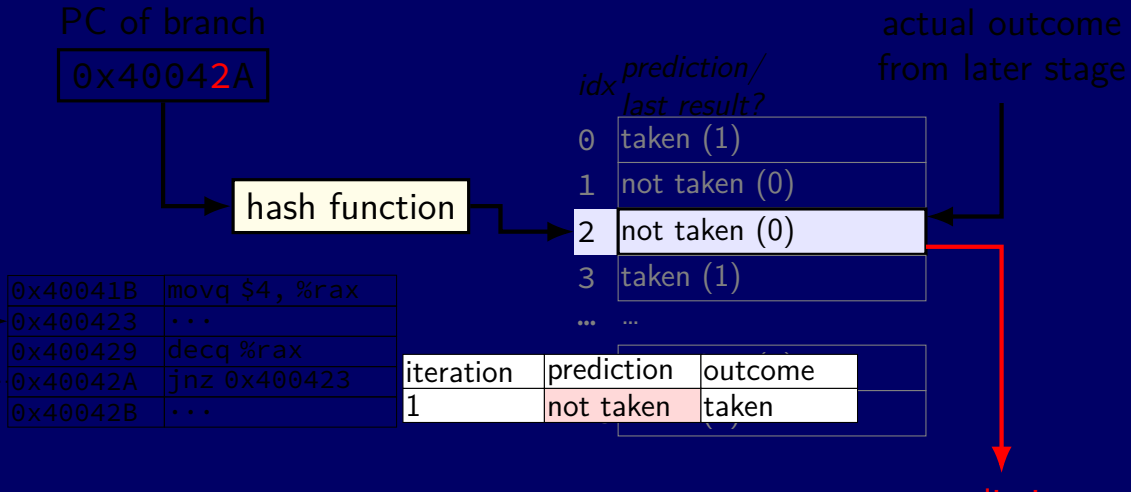
example


PC of branch
0x40042A
hash function
idx
prediction/
last result?
0 taken (1)
1 not taken (0)
2 not taken (0)
3 taken (1)
... ...
14 not taken (0)
15 taken (1)
actual outcome
from later stage
prediction
to fetch stage
0x40041B movq $4, %rax
0x400423 ...
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ...
assembly version of:
i = 4; do { ...; i -= 1; } while (i)

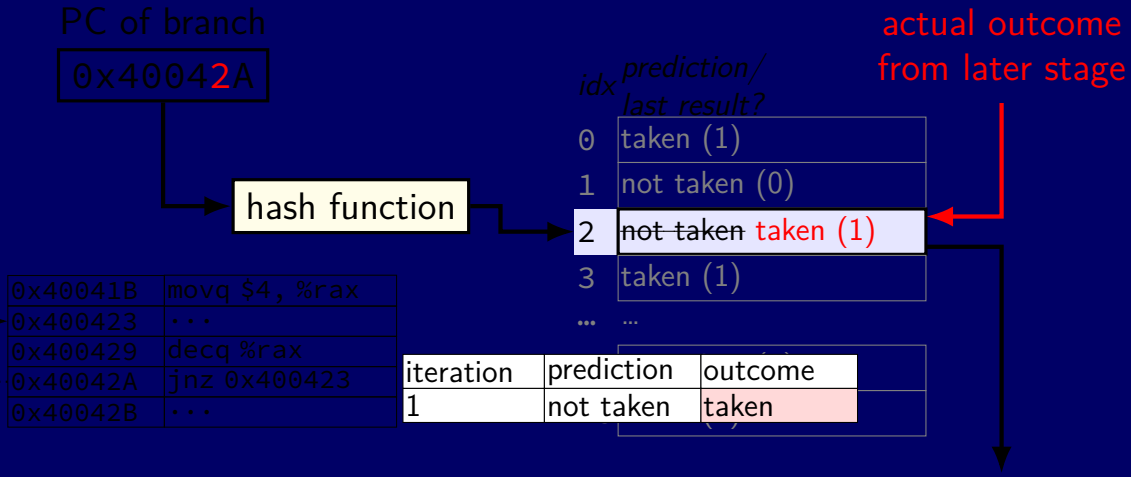
example


PC of branch
0x40042A
hash function
idx
prediction/ last result?
0 taken (1)
1 not taken (0)
2 not taken (0)
3 taken (1)
actual outcome from later stage
0x40041B movq $4, %rax
0x400423 ···
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ···
iteration prediction outcome
1 not taken taken
prediction to fetch stage

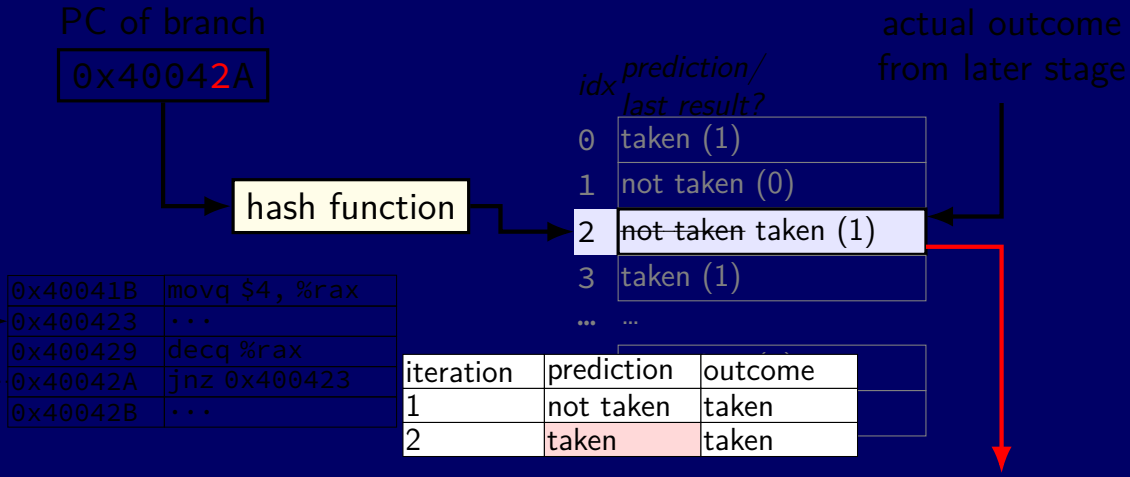
example


PC of branch
0x40042A
hash function
idx
prediction/ last result?
0 taken (1)
1 not taken (0)
2 not taken (0)
3 taken (1)
actual outcome from later stage
0x40041B movq $4, %rax
0x400423 ···
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ···
iteration prediction outcome
1 not taken taken
prediction to fetch stage

example

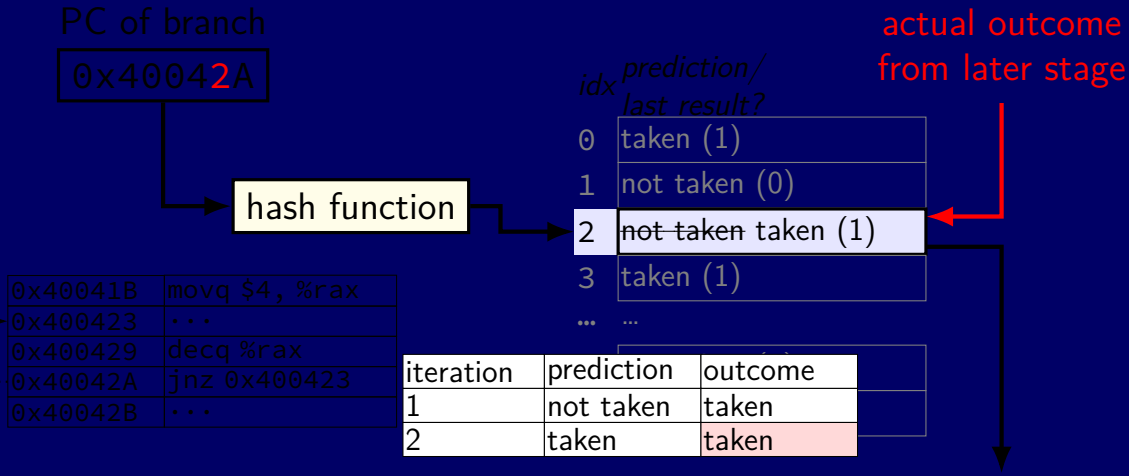

PC of branch
0x40042A
hash function
idx
prediction/ last result?
0 taken (1)
1 not taken (0)
2 not taken taken (1)
3 taken (1)
actual outcome from later stage
0x40041B movq $4, %rax
0x400423 ···
0x400429 decq %rax
0x40042A jnz 0x400423
0x40042B ···
iteration prediction outcome
1 not taken taken
prediction to fetch stage

example



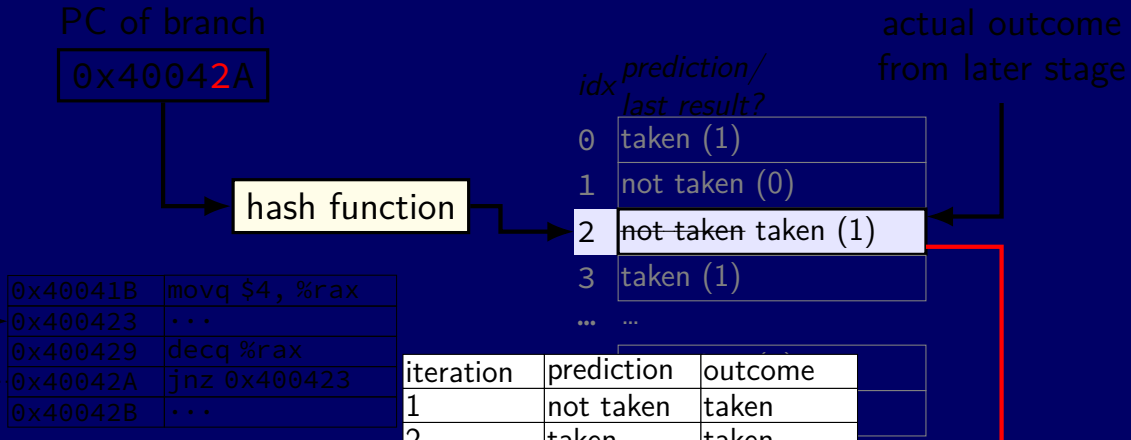
iteration	prediction	outcome
1	not taken	taken
2	taken	taken

example



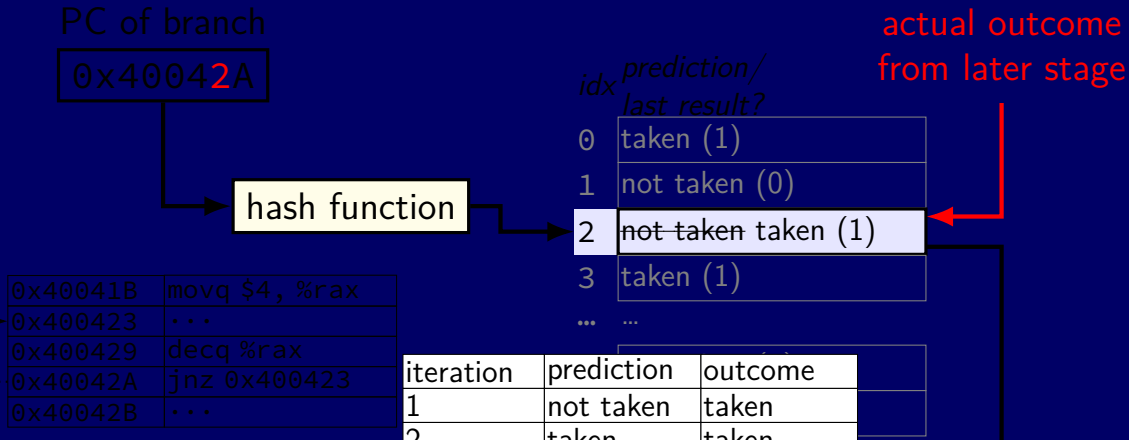
iteration	prediction	outcome
1	not taken	taken
2	taken	taken

example



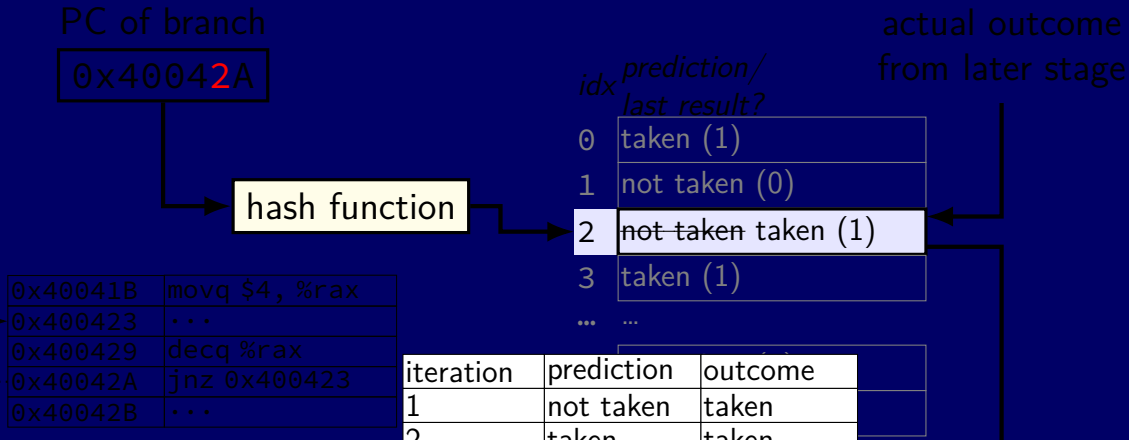
iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken

example



iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken

example



iteration	prediction	outcome
1	not taken	taken
2	taken	taken
3	taken	taken
4	taken	not taken
1	not taken	taken

collisions?

two branches could have same hashed PC

nothing in table tells us about this

versus direct-mapped cache: had *tag bits* to tell

is it worth it?

adding tag bits makes table *much* larger and/or slower

but does anything go wrong when there's a collision?

collision results

possibility 1: both branches usually taken

- no actual conflict — prediction is better(!)

possibility 2: both branches usually not taken

- no actual conflict — prediction is better(!)

possibility 3: one branch taken, one not taken

- performance probably worse

1-bit predictor for loops

predicts first and last iteration wrong

- example: branch to beginning — but same for branch from beginning to end

everything else correct

exercise

use 1-bit predictor on this loop

executed in outer loop (not shown) many, many times

what is the conditional branch misprediction rate?

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
…	…	…	…	…
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	

mod 3: correct for i=2,5,8,…,49 (16/50)
break: correct for i=2,3,…,48 (48/50)
overall: 64/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
…	…	…	…	…
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	

mod 3: correct for i=2,5,8,…,49 (16/50)
break: correct for i=2,3,…,48 (48/50)
overall: 64/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
…	…	…	…	…
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	

mod 3: correct for i=2,5,8,…,49 (16/50)
break: correct for i=2,3,…,48 (48/50)
overall: 64/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
…	…	…	…	…
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	

mod 3: correct for i=2,5,8,…,49 (16/50)
break: correct for i=2,3,…,48 (48/50)
overall: 64/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	???	T	???
1	break	???	N	???
1	mod 3	T	N	
2	break	N	N	✓
2	mod 3	N	N	✓
3	break	N	N	✓
3	mod 3	N	T	
4	break	N	N	✓
…	…	…	…	…
48	mod 3	N	T	
49	break	N	N	✓
49	mod 3	T	N	
50	break	N	T	
0	mod 3	N	T	

mod 3: correct for i=2,5,8,…,49 (16/50)
break: correct for i=2,3,…,48 (48/50)
overall: 64/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

beyond local 1-bit predictor

can predict using more historical info

whether taken last several times

- example: taken 3 out of 4 last times $\rightarrow$ predict taken

pattern of how taken recently

- example: if last few are T, N, T, N, T, N; next is probably T
- makes two branches hashing to same entry not so bad

outcomes of last N conditional jumps ("global history")

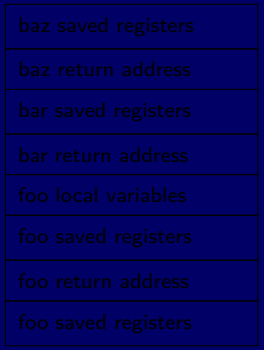
- take into account conditional jumps in surrounding code
- example: loops with if statements will have regular patterns

predicting ret: ministack of return addresses

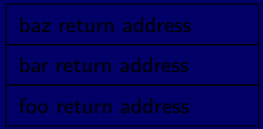
predicting ret — ministack in processor registers

push on ministack on call; pop on ret

ministack overflows? discard oldest, mispredict it later

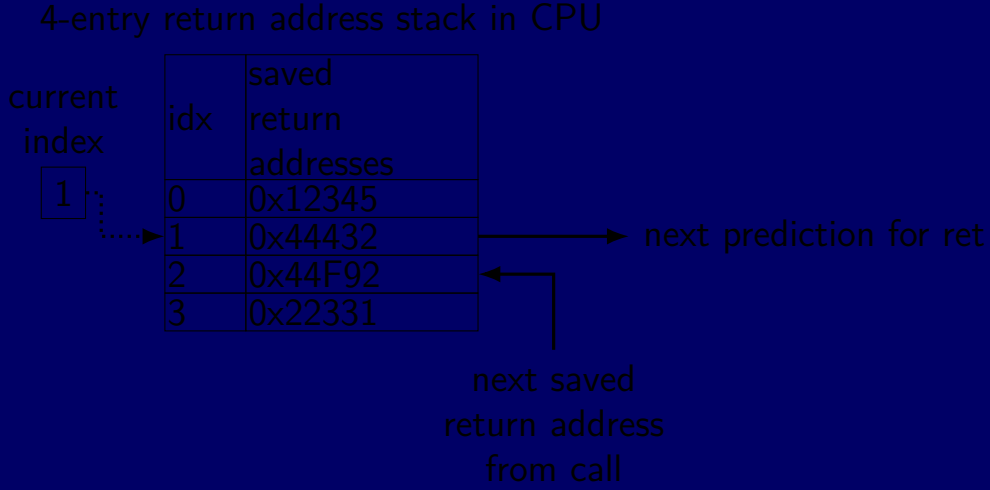


stack in memory



(partial?) stack in CPU registers

4-entry return address stack



on `call`: increment index, save return address in that slot

1-cycle fetch?

assumption so far:

1 cycle to fetch instruction + identify if jmp, etc.

often not really practical

especially if:

- complex machine code format
- many pipeline stages
- more complex instruction cache
- (future idea) fetching 2+ instructions/cycle

branch target buffer

what if we can't decode LABEL from machine code for `jmp` LABEL or `jle` LABEL fast?

- will happen in more complex pipelines

what if we can't decode that there's a RET, CALL, etc. fast?

BTB: cache for branch targets

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branch targets

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

BTB: cache for branch targets

idx	valid	tag	ofst	type	target	(more info?)
0x00	1	0x400	5	Jxx	0x3FFFF3	…
0x01	1	0x401	C	JMP	0x401035	---
0x02	0	---	---	---	---	---
0x03	1	0x400	9	RET	---	…
…	…	…	…	…	…	…
0xFF	1	0x3FF	8	CALL	0x404033	…

valid	…
1	…
0	…
0	…
0	…
…	…
0	…

```
0x3FFFF3:  movq %rax, %rsi
0x3FFFF7:  pushq %rbx
0x3FFFF8:  call 0x404033
0x400001:  popq %rbx
0x400003:  cmpq %rbx, %rax
0x400005:  jle 0x3FFFF3
…          …
0x400031:  ret
…          …
```

indirect branch prediction

for instructions like: `jmp *%rax` or `jmp *(%rax, %rcx, 8)`

simple idea: record what happened last time, predict the same

Intel Haswell extension:

- lookup in based on hash of *last several jump addresses*
- allows different predictions based on context of `jmp *%rax`
- really useful for polymorphism

backup slides

exercise: forwarding paths (2)

cycle # 0 1 2 3 4 5 6 7 8

```
addq %r8, %r9
subq %r8, %r9
ret (goes to andq)
andq %r10, %r9
```

in subq, %r8 is _____ addq.

in subq, %r9 is _____ addq.

in andq, %r9 is _____ subq.

in andq, %r9 is _____ addq.

A: not forwarded from

beyond 1-bit predictor

devote *more space* to storing history

main goal: rare exceptions don't immediately change prediction

example: branch taken 99% of the time

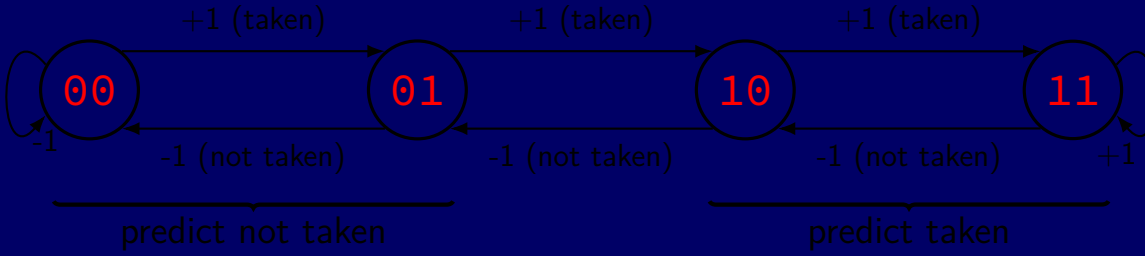
1-bit predictor: wrong about 2% of the time

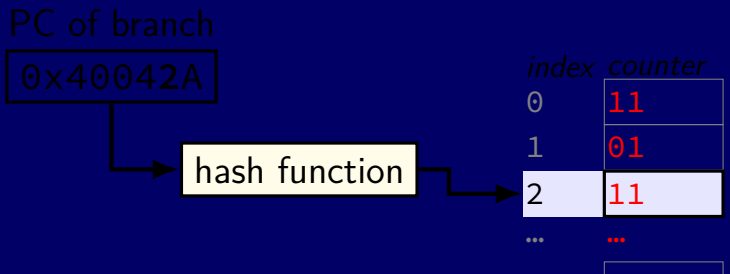
- 1% when branch not taken
- 1% of taken branches right after branch not taken

new predictor: wrong about 1% of the time

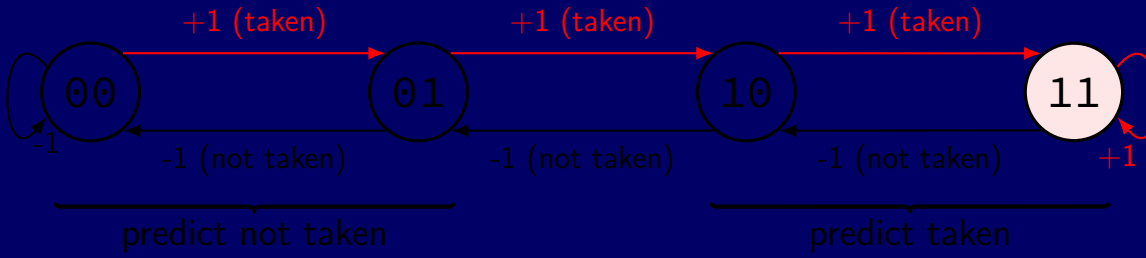
- 1% when branch not taken

2-bit saturating counter



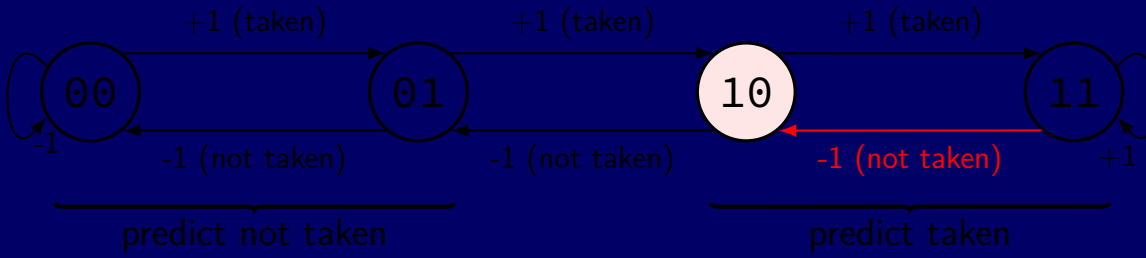


2-bit saturating counter



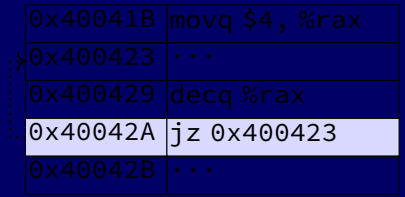
branch always taken:
value increases to ‘strongest’ taken value

2-bit saturating counter



branch almost always taken, then not taken once:
still predicted as taken

example



iter.	table before	prediction	outcome	table after
1	01	not taken	taken	10
2	10	taken	taken	11
3	11	taken	taken	11
4	11	taken	not taken	10
1	10	taken	taken	11
2	11	taken	taken	11
3	11	taken	taken	11
4	11	taken	not taken	10
1	10	taken	taken	11
...	...	...	...	...

generalizing saturating counters

2-bit counter: ignore one exception to taken/not taken

3-bit counter: ignore more exceptions

`000` $\leftrightarrow$ `001` $\leftrightarrow$ `010` $\leftrightarrow$ `011` $\leftrightarrow$ `100` $\leftrightarrow$ `101` $\leftrightarrow$ `110` $\leftrightarrow$ `111`

`000-011`: not taken

`100-111`: taken

exercise

use 2-bit predictor on this loop

executed in outer loop (not shown) many, many times

what is the conditional branch misprediction rate?

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

exercise soln (1)

i=	branch	predicted	outcome	correct?
0	mod 3	01 (N)	T	
1	break	01 (N)	N	✓
1	mod 3	10 (T)	N	
2	break	00 (N)	N	✓
2	mod 3	01 (N)	N	✓
3	break	00 (N)	N	✓
3	mod 3	00 (N)	T	
4	break	00 (N)	N	✓
...	...	...	...	...
48	mod 3	00 (N)	T	
49	break	00 (N)	N	✓
49	mod 3	01 (N)	N	✓
50	break	00 (N)	T	
0	mod 3	00 (N)	T	

mod 3: correct for i=1,2,4,5,7,8,...,49 (33/50)

mod 3: ends up always predicting not ta

break: correct for i=2,3,...,48 (49/50)

break: ends up always predicting not tak

overall: 82/100

```
int i = 0;
while (true) {
  if (i % 3 == 0) goto next;
  ...
next:
  i += 1;
  if (i == 50) break;
}
```

branch patterns

```
i = 4;
do {
    ...
    i -= 1;
} while (i != 0);
```

typical pattern for jump to top of do-while above:

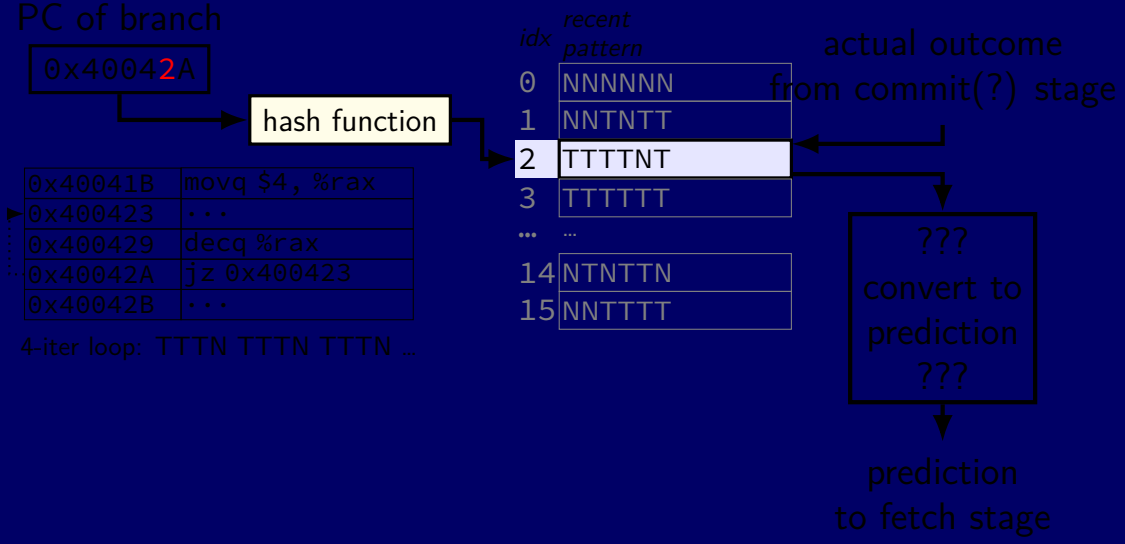
TTTN TTTN TTTN TTTN TTTN...(T = taken, N = not taken)

goal: take advantage of recent pattern to make predictions

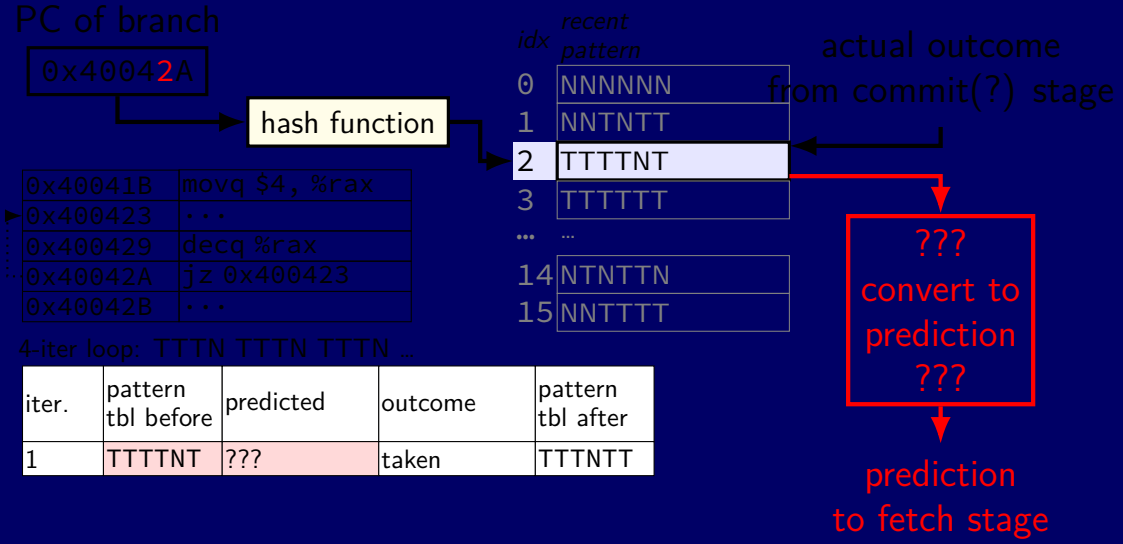
just saw 'NTTTNT'? predict T next

'TNTTTN'? predict T; 'TTNTTT'? predict N next

local pattern predictor (incomplete)



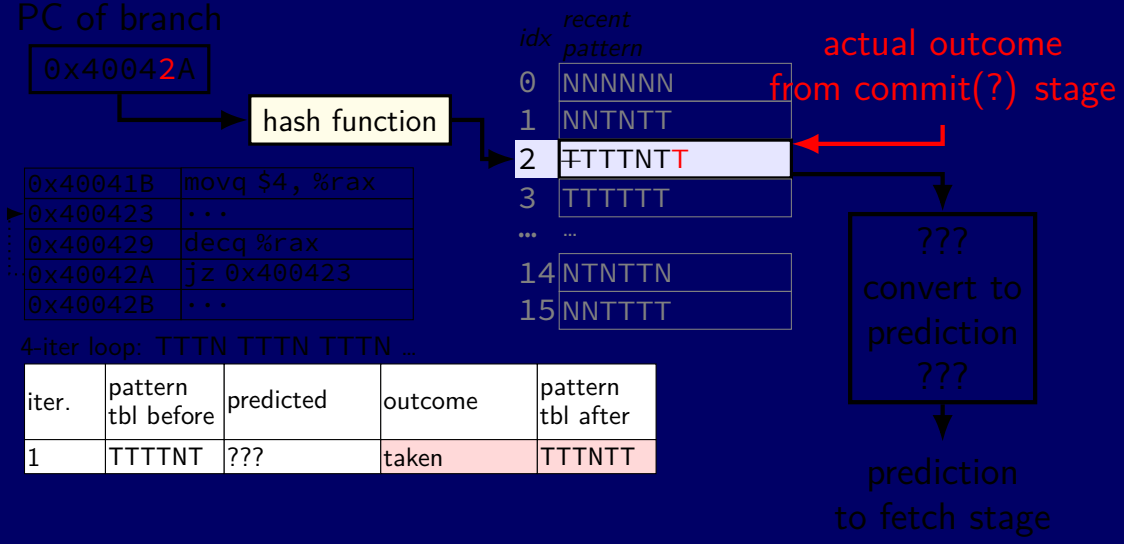
local pattern predictor (incomplete)



4-iter loop: TTTN TTTN TTTN …

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT

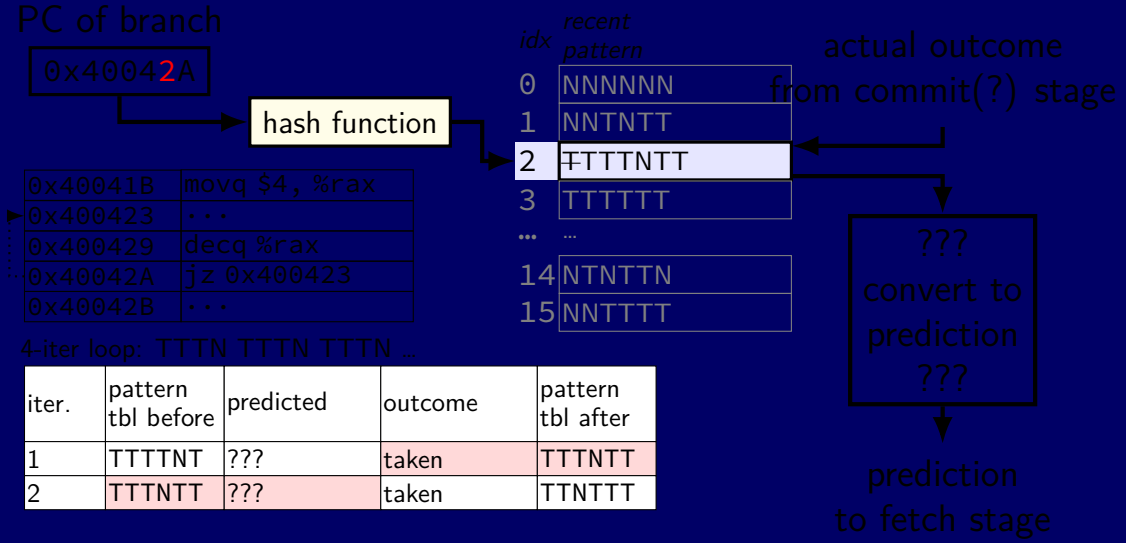
local pattern predictor (incomplete)



4-iter loop: TTTN TTTN TTTN …

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT

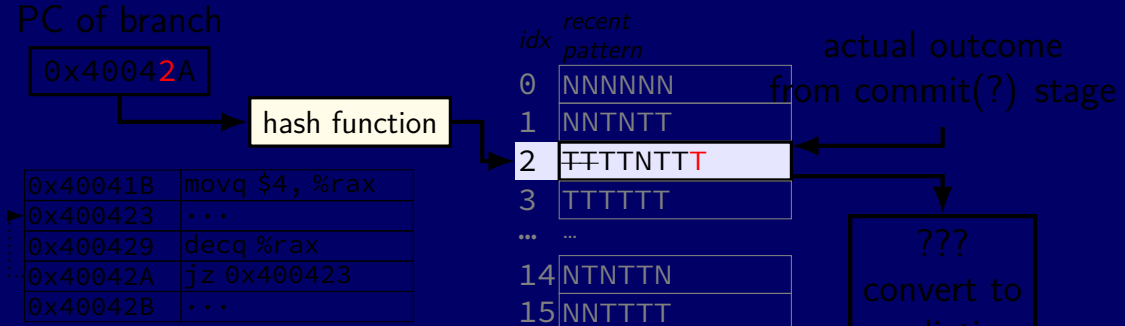
local pattern predictor (incomplete)



4-iter loop: TTTN TTTN TTTN …

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT

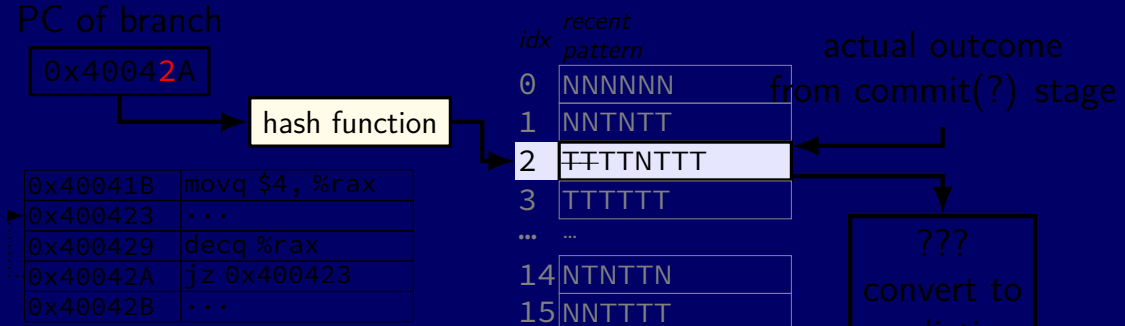
local pattern predictor (incomplete)



4-iter loop: TTTN TTTN TTTN …

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT

local pattern predictor (incomplete)



4-iter loop: TTTN TTTN TTTN …

iter.	pattern tbl before	predicted	outcome	pattern tbl after
1	TTTTNT	???	taken	TTTNTT
2	TTTNTT	???	taken	TTNTTT
3	TTNTTT	???	taken	TNTTTT
4	TNTTTT	???	not taken	NTTTTN

recent pattern to prediction?

easy cases:

just saw TTTTTT: predict T

just saw NNNNNN: predict N

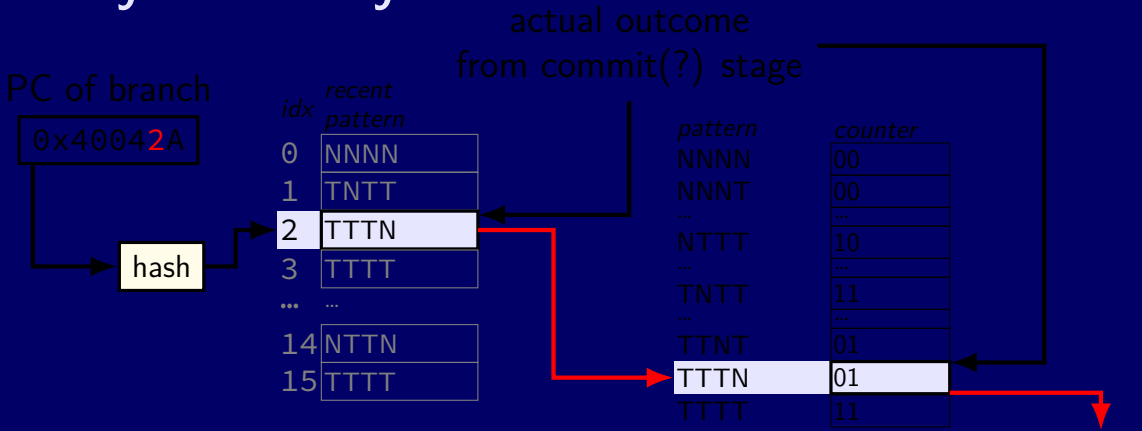
just saw TNTNTN: predict T

hard cases:

TTNTTTT

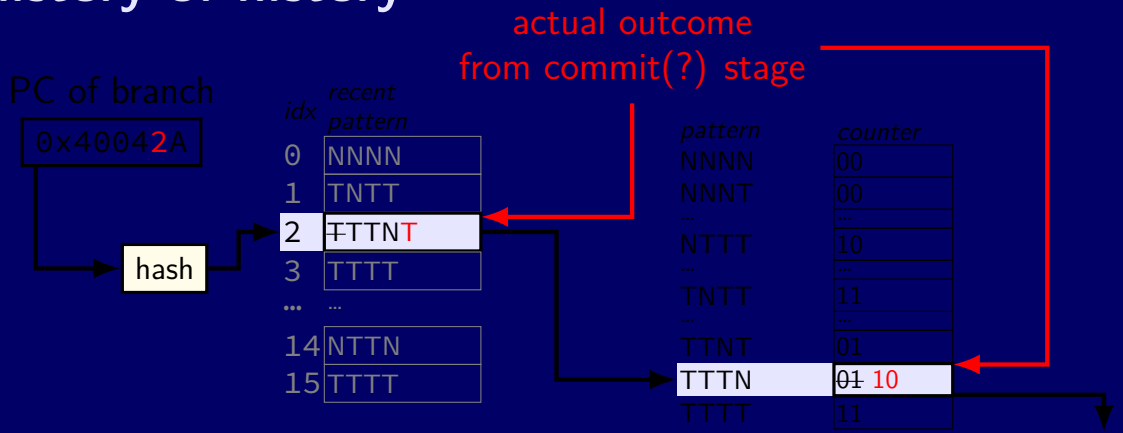
- predict T? loop with many iterations
  (NTTTTTTTNTTTTTTTNTTTTTT…)
- predict T? if statement mostly taken
  (TTTTNTTNTTTTTTTTTNTTTT…)

history of history



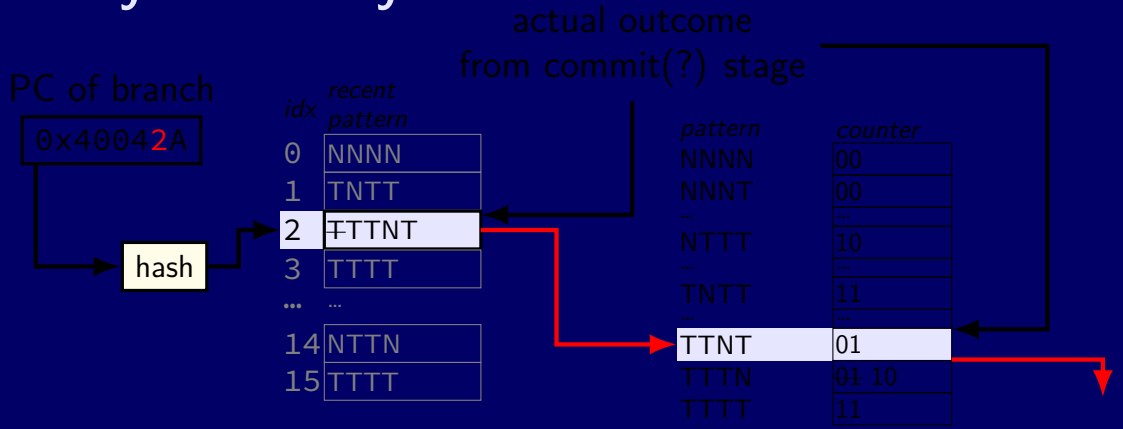
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

history of history



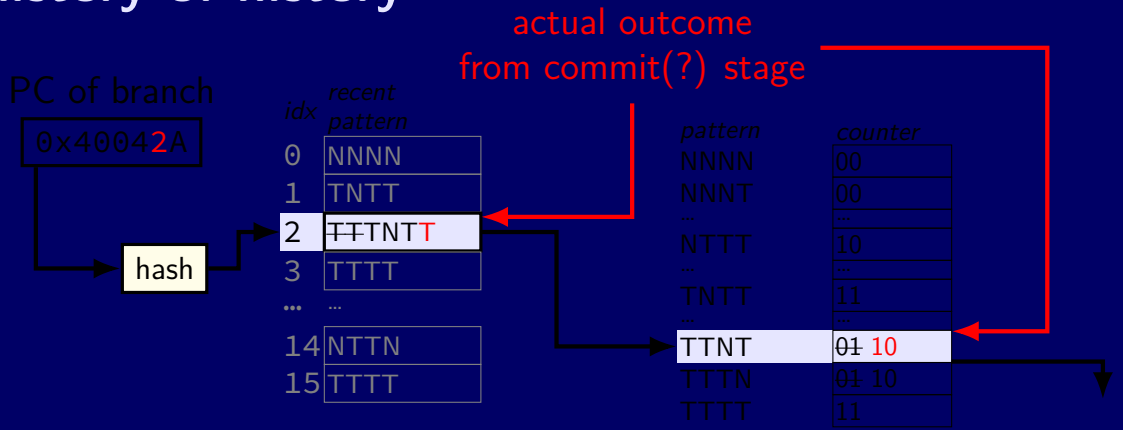
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT

history of history



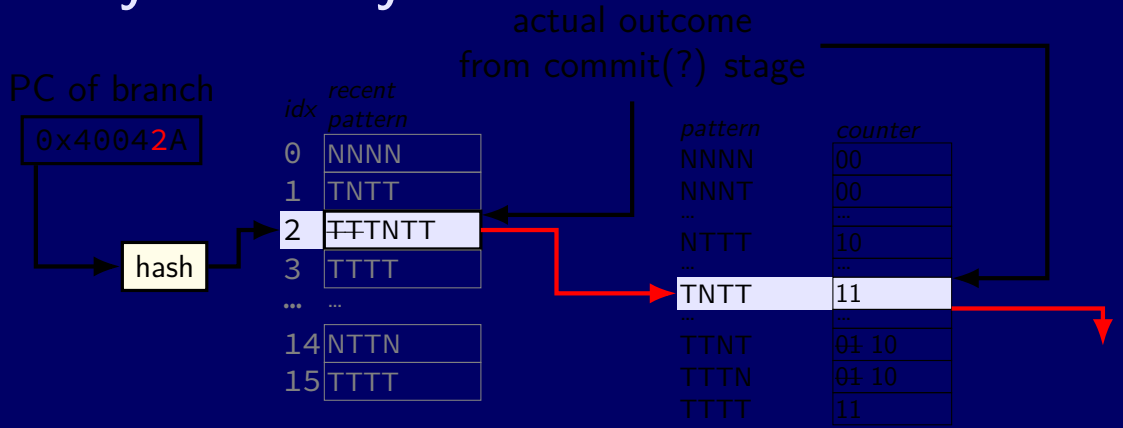
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT

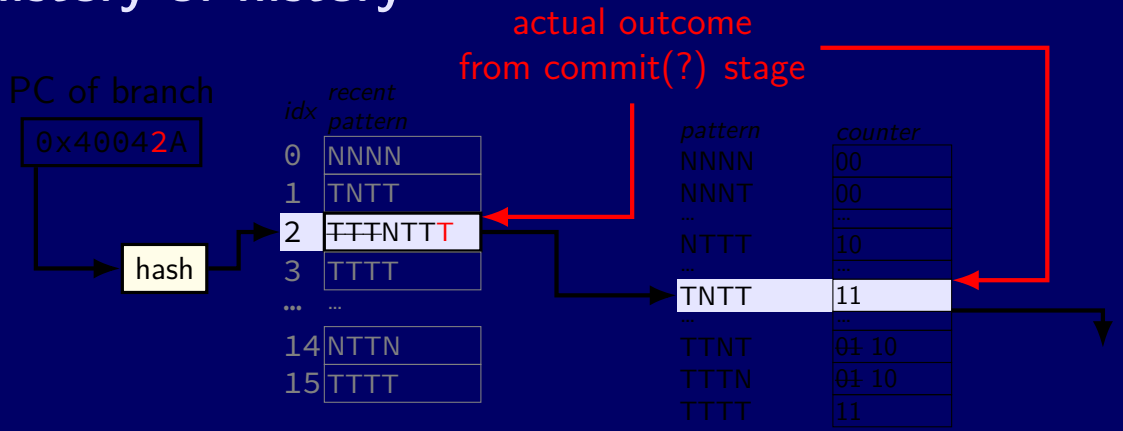
history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

prediction
to fetch sta

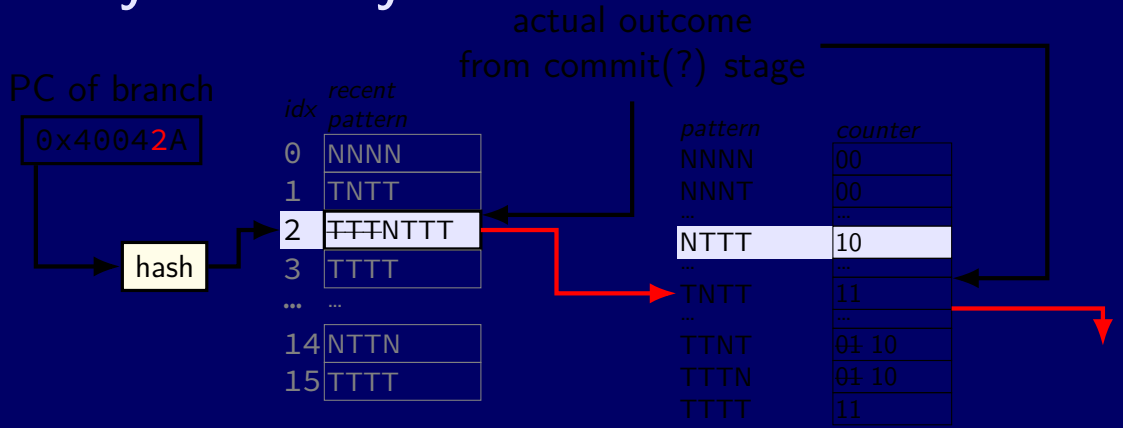
history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

prediction
to fetch sta

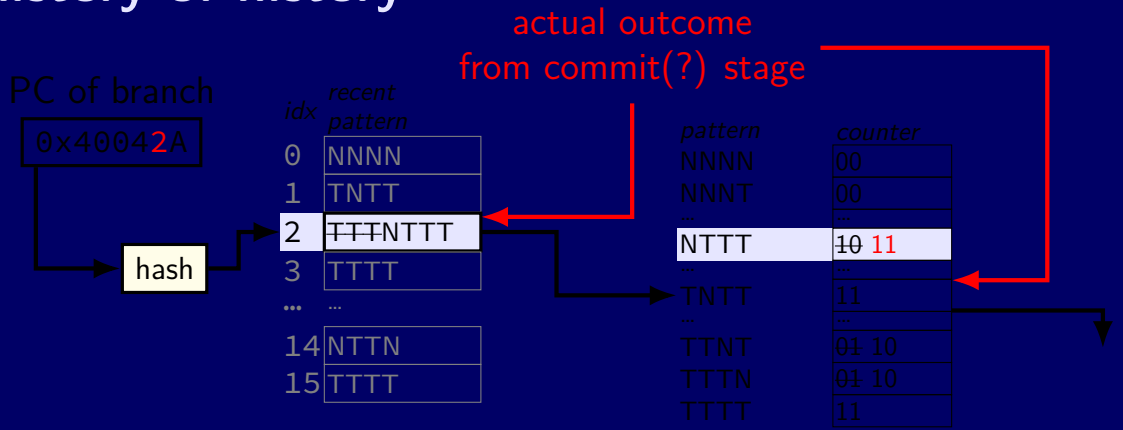
history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

prediction
to fetch sta

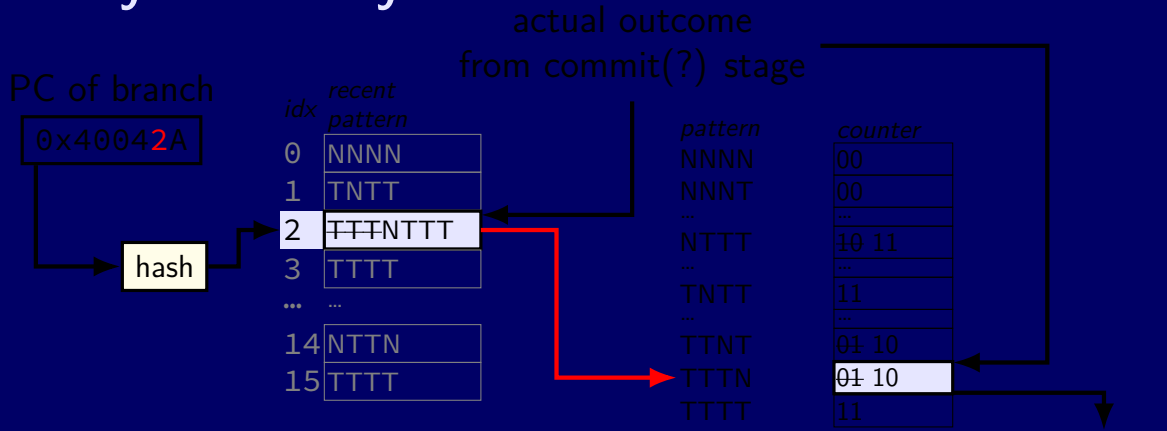
history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

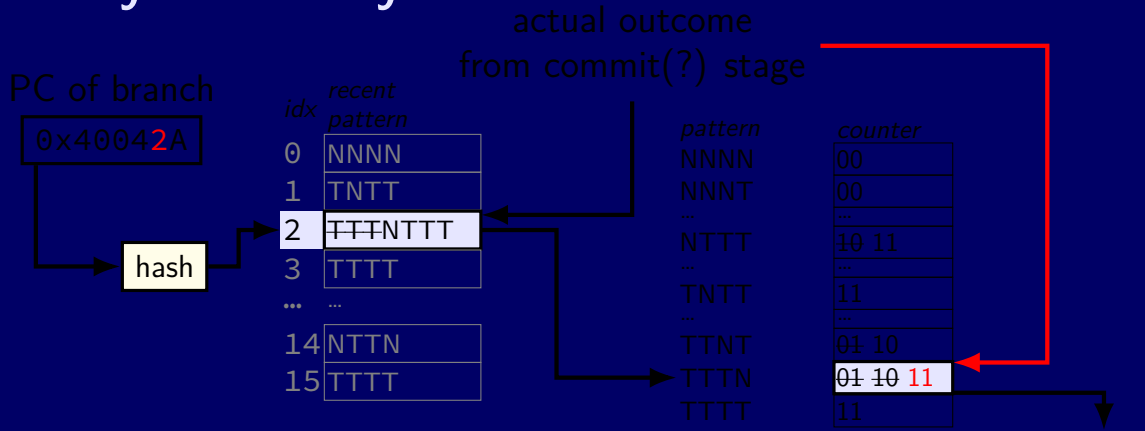
prediction
to fetch sta

history of history



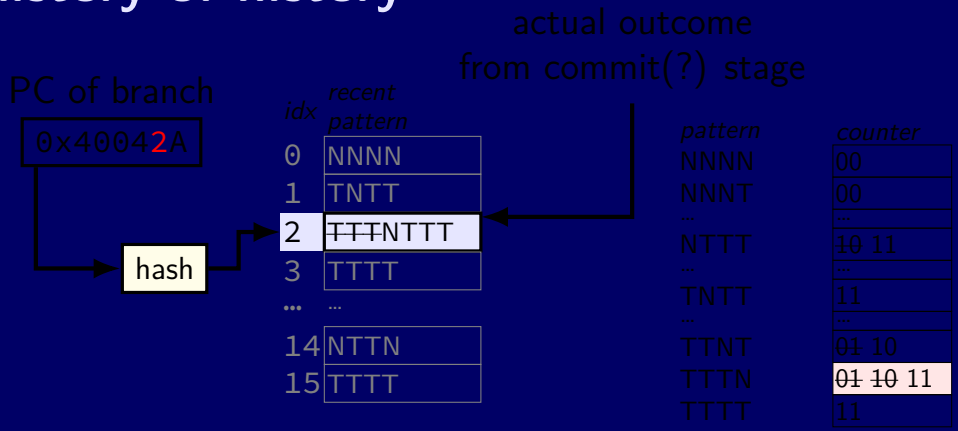
iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

history of history



iter.	branch to pat. tbl before	pat. to counter before	predict	actual	pat. to counter after	branch to pat. tbl after
1	TTTN	01	not taken	taken	10	TTNT
2	TTNT	01	not taken	taken	10	TNTT
3	TNTT	11	taken	taken	11	NTTT

predictio
to fetch sta

local patterns and collisions (1)

```
i = 10000;
do {
    p = malloc(...);
    if (p == NULL) goto error;  // BRANCH 1
    ...
} while (i-- != 0); // BRANCH 2
```

what if branch 1 and branch 2 hash to same table entry?

local patterns and collisions (1)

```
i = 10000;
do {
    p = malloc(...);
    if (p == NULL) goto error;  // BRANCH 1
    ...
} while (i-- != 0); // BRANCH 2
```

what if branch 1 and branch 2 hash to same table entry?

pattern: TNTNTNTNTNTNTNTNT...

actually no problem to predict!

local patterns and collisions (2)

```
i = 10000;
do {
    if (i % 2 == 0) goto skip; // BRANCH 1
    ...
    p = malloc(...);
    if (p == NULL) goto error;  // BRANCH 2
skip: ...
} while (i-- != 0); // BRANCH 3
```

what if branch 1 and branch 2 and branch 3 hash to same table entry?

local patterns and collisions (2)

```
i = 10000;
do {
    if (i % 2 == 0) goto skip; // BRANCH 1
    ...
    p = malloc(...);
    if (p == NULL) goto error;  // BRANCH 2
skip: ...
} while (i-- != 0); // BRANCH 3
```

what if branch 1 and branch 2 and branch 3 hash to same table entry?

pattern: TTNNTTNNTTNNTTNNTT

also no problem to predict!

local patterns and collisions (3)

```
i = 10000;
do {
    if (A) goto one  // BRANCH 1
    ...
one:
    if (B) goto two  // BRANCH 2
    ...
two:
    if (A or B) goto three // BRANCH 3
    ...
    if (A and B) goto three // BRANCH 4
    ...
three:
    ... // changes A, B
} while (i-- != 0);
```

what if branch 1-4 hash to same table entry?

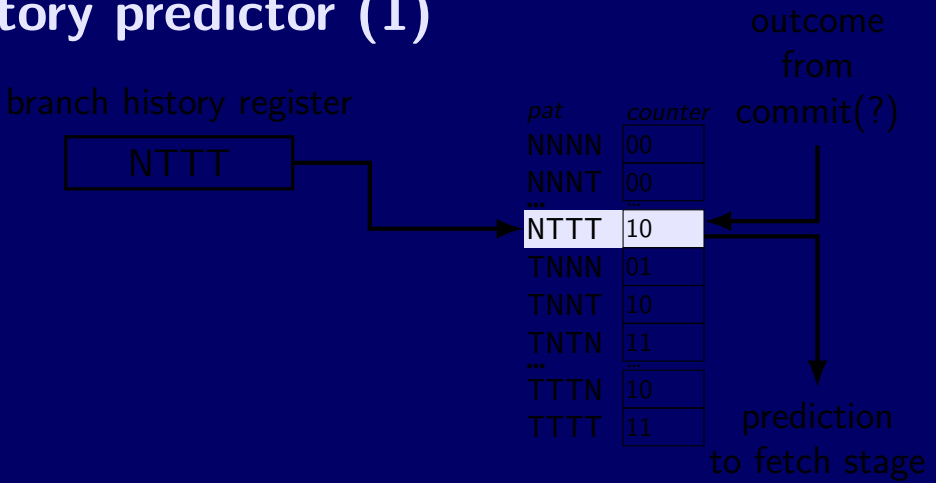
global history predictor: idea

one predictor idea: ignore the PC

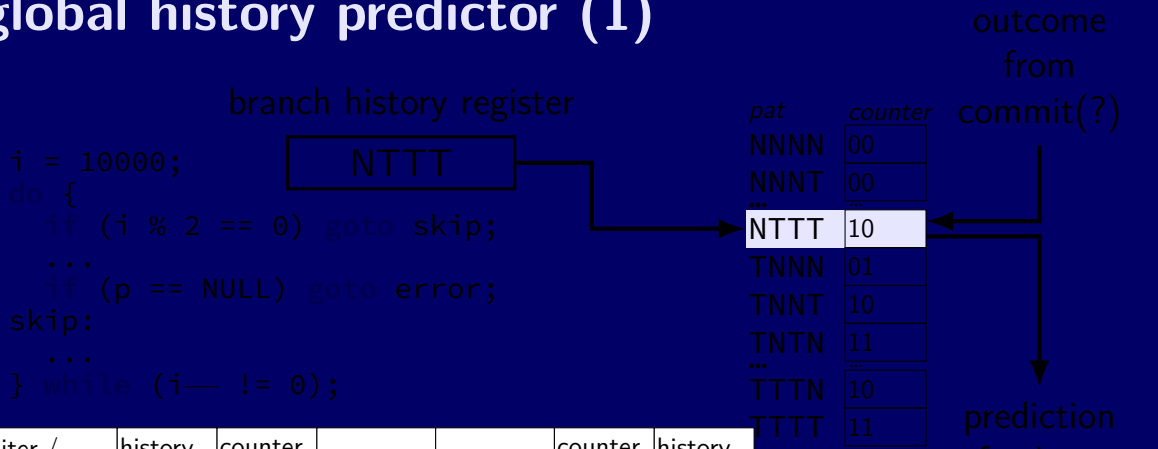
just record taken/not-taken pattern for all branches

lookup in big table like for local patterns

global history predictor (1)



global history predictor (1)

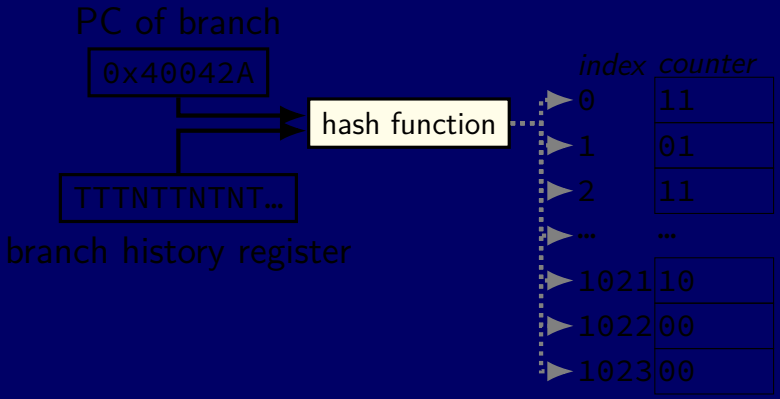


iter./ branch	history before	counter before	predict	outcome	counter after	history after
0/mod 2	NTTT	10	taken	taken	11	TTTT
0/loop	TTTT			taken		TTTT
1/mod 2	TTTT			not taken		TTTN
1/error	TTTN			not taken		TTNN
1/loop	TNNT			taken		NNTT

correlating predictor

global history *and* local info good together

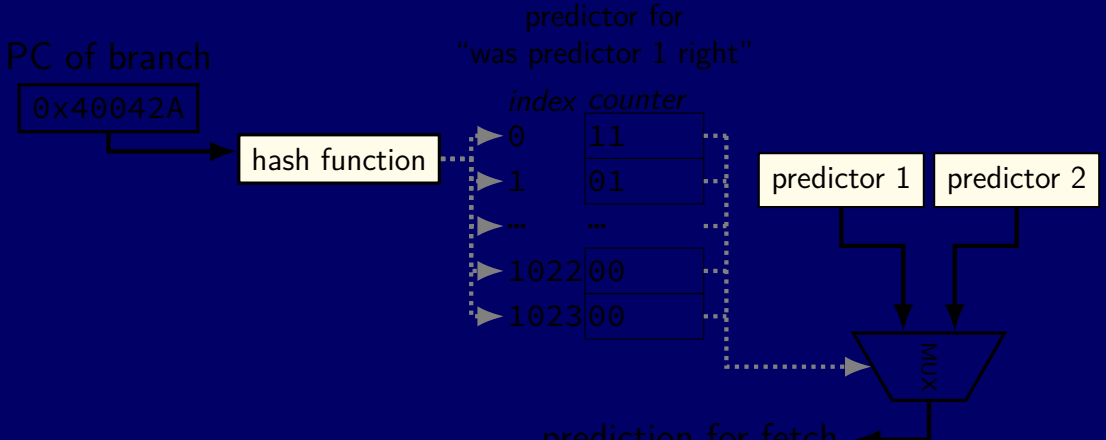
one idea: combine history register + PC ("gshare")



mixing predictors

different predictors good at different times

one idea: have two predictors, + predictor to predict which is right



loop count predictors (1)

```
for (int i = 0; i < 64; ++i)
    ...
```

can we predict this perfectly with predictors we've seen

yes — local or global history with 64 entries

but this is very important — more efficient way?

loop count predictors (2)

loop count predictor idea: look for NNNNNNT+repeat (or TTTTTTN+repeat)

track for each possible loop branch:

- how many repeated Ns (or Ts) so far
- how many repeated Ns (or Ts) last time before one T (or N)
- something to indicate this pattern is useful?

known to be used on Intel

benchmark results

from 1993 paper

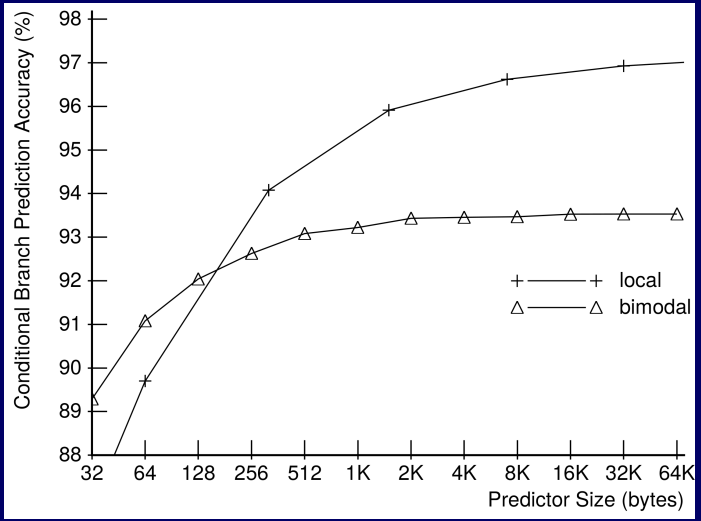
(not representative of modern workloads?)

rate for conditional branches on benchmark

variable table sizes

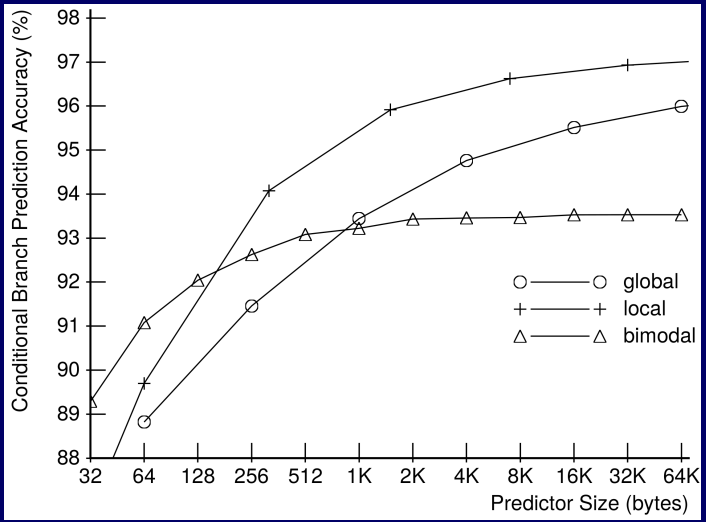
2-bit ctr + local history

from McFarling, "Combining Branch Predictors" (1993)



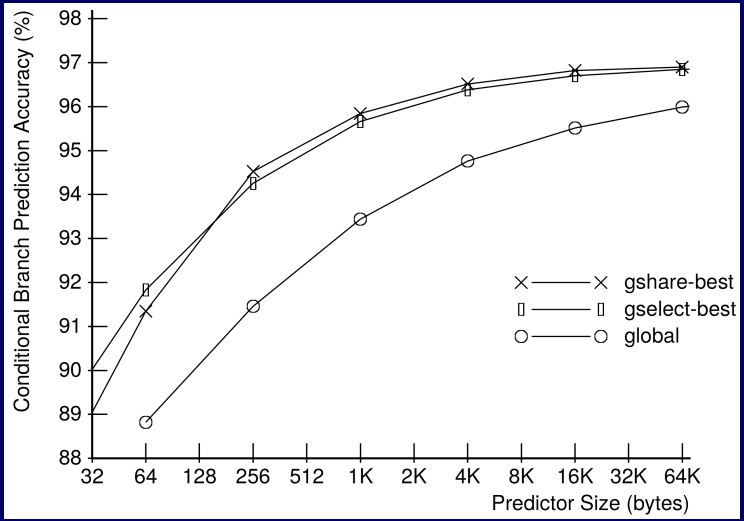
2-bit (bimodal) + local + global hist

from McFarling, "Combining Branch Predictors" (1993)



global + hash(global+PC) (gshare/gselect)

from McFarling, "Combining Branch Predictors" (1993)



real BP?

details of modern CPU's branch predictors often not public

but...

Google Project Zero blog post with reverse engineered details

https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

for RE'd BTB size:

https://xania.org/201602/haswell-and-ivy-btb

reverse engineering Haswell BPs

branch target buffer

- 4-way, 4096 entries
- ignores bottom 4 bits of PC?
- hashes PC to index by shifting + XOR
- seems to store 32 bit offset from PC (not all 48+ bits of virtual addr)

indirect branch predictor

- like the global history + PC predictor we showed, but...
- uses history of recent branch addresses instead of taken/not taken
- keeps some info about last 29 branches

what about conditional branches??? loops???

- couldn't find a reasonable source