# Lecture 21: Review

Wednesday, February 27, 2019 10:28 AM

## AMAT + CPI example

L1 cache hit: 1 cycle, miss: 20 cycles, 95% hit ratio. 40% memory instructions.

penalty

assume no hazards

What is CPI without a cache? What is the speedup of a "perfect" cache?

$CPI = 0.6 \times 1 + 0.4 \times 20 = .6 + 8 = 8.6$

ratio of non-mem insts

CPI non-mem

ratio of memory

CPI for mem

speedup of perfect $= \frac{8.6}{1} = 8.6x$

What is CPU with the cache? How close to the "perfect" cache?

$AMAT = 1 + 0.05 \cdot 20 = 2$ cycles

hit time for cache

miss ratio

miss penalty

$CPI$ w/ cache $= 0.6 \times 1 + 0.4 \times 2 = 1.4$

speedup of perfect compared to cache $= \frac{1.4}{1} = 1.4x$

Branches have 3 cycle penalty (20% branches)

not branch or mem

$CPI = 0.4 \times 1 + 0.2 \times 3 + 0.4 \times 2$

Branch resolved when you know the next PC

12. Assume a traditional 5-stage pipeline. Answer the following questions using the code block below.
    i. Assume that, when we start executing this code block, `x7 = 72` and `x8 = ~~228~~` 200. Can we move the `sw` before the `lw`? Why or why not?
    ii. Assume we do not have a forwarding unit. Can we move instruction 8 between instructions 0 and 4 to eliminate some of the NOPs? Why or why not?

```
0:    add  x3, x1, x2
4:    bne  x3, x4, 16
8:    lw   x5, 124(x7)
12:   sw   x6, -32(x8)
16:   add  x4, x1, x3
20:   ...
```

→ addr 196 (72 + 124)

→ addr 196 (228 - 32)

Safe to reorder if independent

↳ don't share registers

↳ don't share memory addresses

Since same addr cannot reorder

What information does proc need from branch predictor to fetch next address

1. taken or not (prediction)
2. where to go (next PC)

Question 10:

expose Static ILP

increase ILP through rescheduling

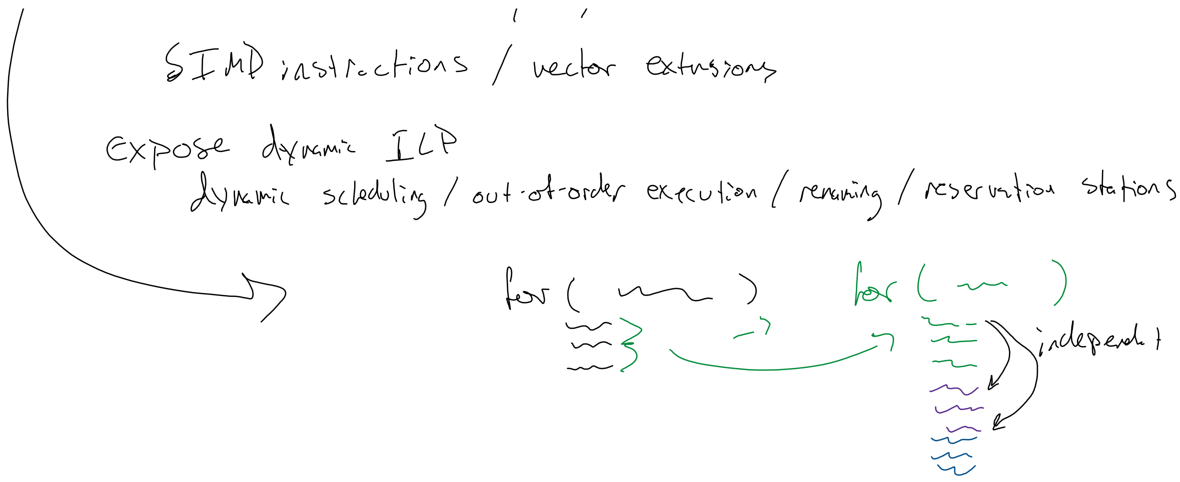
loop unrolling

new ISAs like VLIW or EPIC → explicitly parallel

↳ very long instruction word

SIMD instructions / vector extensions

Expose dynamic ILP
dynamic scheduling / out-of-order execution / renaming / reservation stations

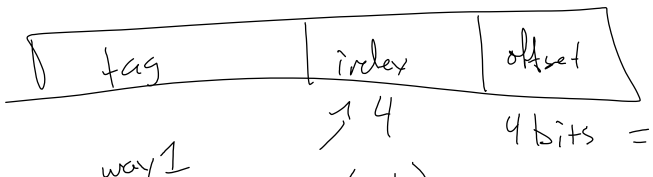


For each access, mark if it is a hit or a miss.

| | |
|---|---|
| M | 0x408ac |
| M | 0x408bc |
| M | 0x508bc |
| M | 0x608bc |
| M | 0x00000 |
| H | 0x408a0 |
| M | 0x00010 |
| H | 0x608b4 |

offset (c)
index (a)

1) breakdown of address



16 byte blocks

4 bits = log(16 bytes)

| Index | way 0 | way 1 |
|---|---|---|
| 0 | 0x000 | |
| 1 | 0x000 | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| a | 0x408 | |
| b | ~~0x408~~ 0x608 | 0x508 |
| c | | |
| d | | |
| e | | |
| f | | |

$\#\text{sets} = \frac{\#\text{lines}}{\#\text{of ways}}$

$16 = \frac{32}{2}$

Hit rate = $2/8 = 0.25$

Why always taken is better than always ~~not~~ not taken?
loops take lots of branches

for ( ) {
}
top:
blt
top

100
taken 100 times
not taken 1 time

taken = $\frac{100}{101}$ = 98.---%

not taken = $\frac{1}{101}$ = 1%

In-order Read after write deps.

[illegible]

```
    0:    lw     x2, 0(x1)
 -  4:    addi   x7, x2, -5
 -  8:    sw     x4, 12(x3)
 -> 12:   add    x10, x7, x4
 -> 16:   lw     x8, 100(x10)
 -> 20:   sub    x10, x9, x8
```

sw reads x4

Out of order

Write after write ←

Write after read ←