CS 316: Caches-III

Kavita Bala
Fall 2007
Computer Science
Cornell University

Announcements

- HW 1 grades are out
- HW 2 is due on Friday
- PA 4 is out on Friday

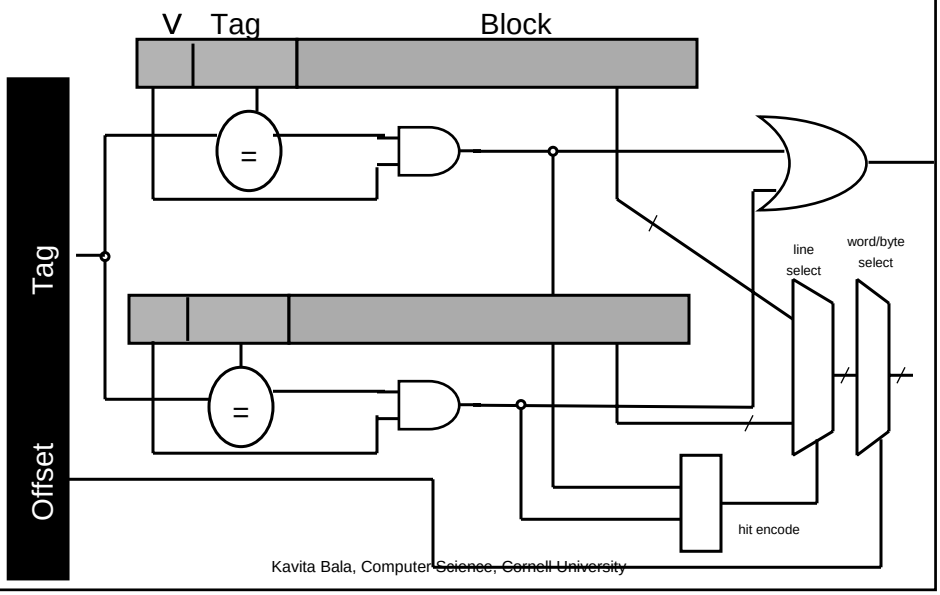
Cache Organization

- Three common designs
  - Fully associative: Block can be anywhere in the cache
  - Direct mapped: Block can only be in one line in the cache
  - Set-associative: Block can be in a few (2 to 8) places in the cache

Misses

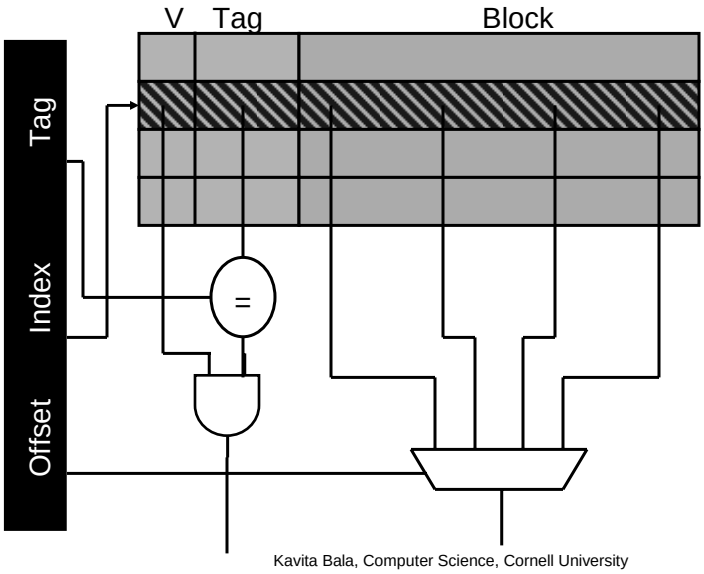
- Three types of misses
  - Cold
    - The line is being referenced for the first time
  - Capacity
    - The line was evicted because the cache was not large enough
  - Conflict
    - The line was evicted because of another access whose index conflicted

Fully Associative Cache



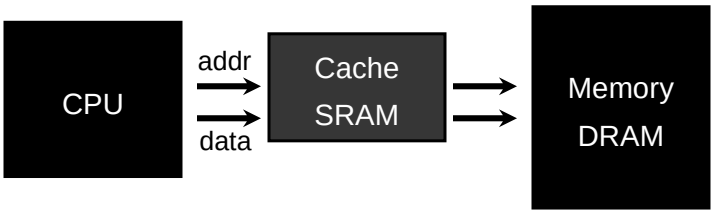
Kavita Bala, Computer Science, Cornell University

Comparison: Direct Mapped Cache



Kavita Bala, Computer Science, Cornell University

Cache Writes

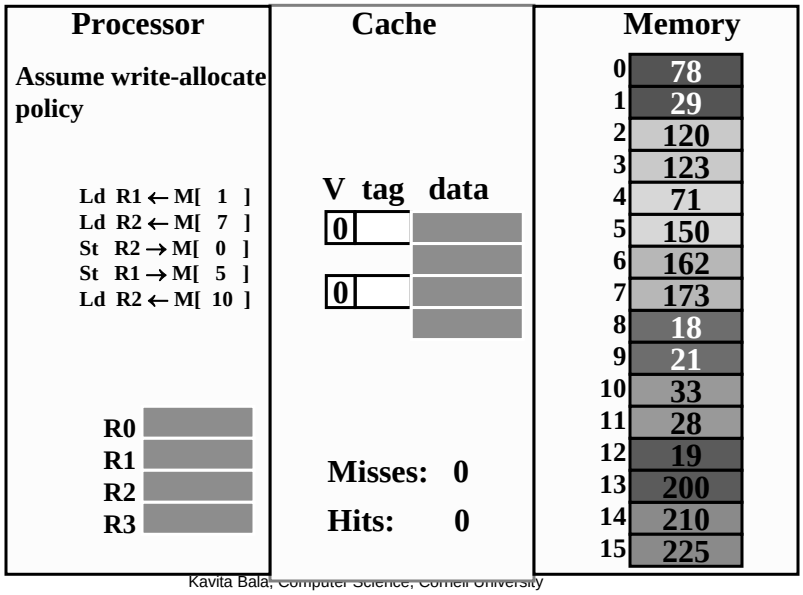


- No-Write
  - writes invalidate the cache and go to memory
- Write-Through
  - writes go to cache and to main memory
- Write-Back
  - write cache, write main memory only when block is evicted

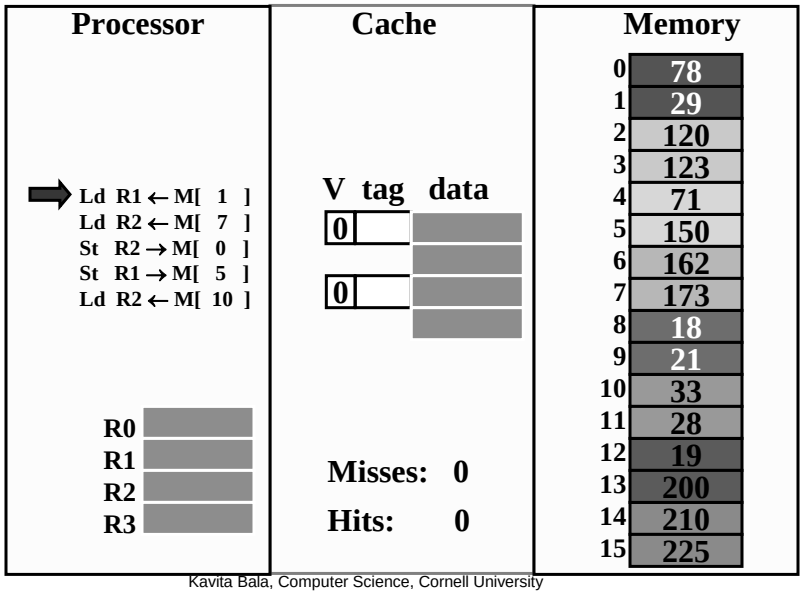
What about Stores?

- Where should you write the result of a store?
  - If that memory location is in the cache?
    - Send it to the cache
    - Should we also send it to memory right away? (write-through policy)
    - Wait until we kick the block out (write-back policy)
  - If it is not in the cache?
    - Allocate the line (put it in the cache)? (write allocate policy)
    - Write it directly to memory without allocation? (no write allocate policy)

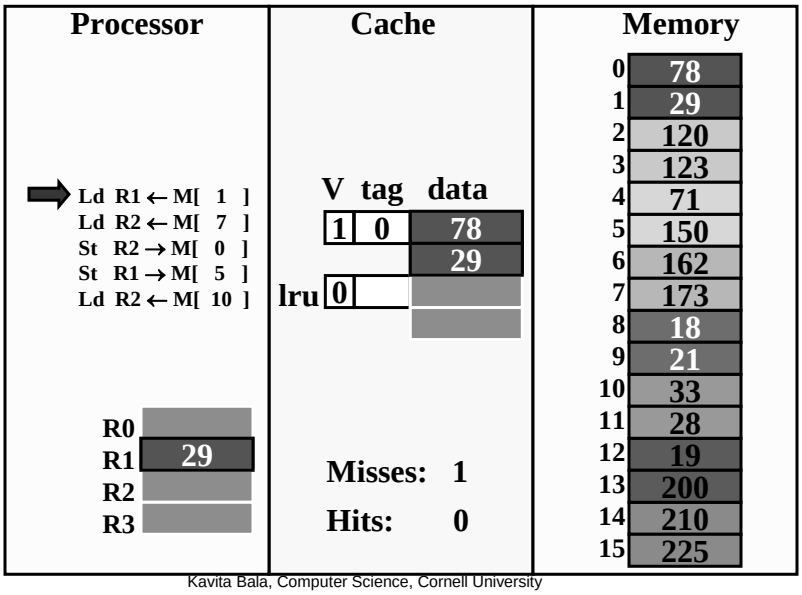
Handling Stores (Write-Through)



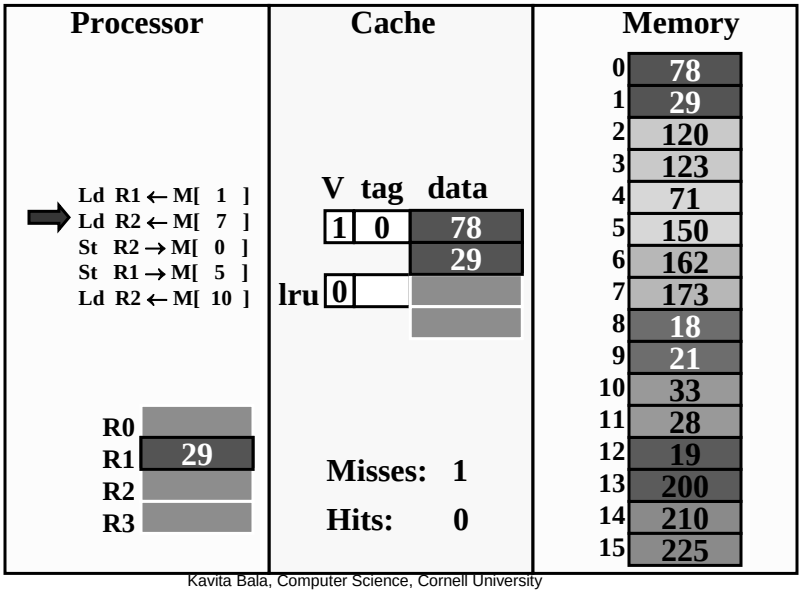
Write-Through (REF 1)



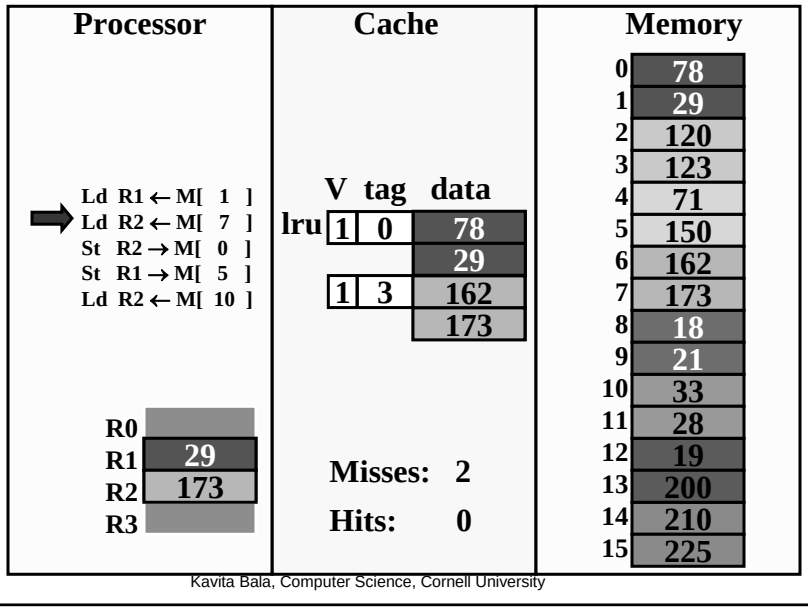
Write-Through (REF 1)



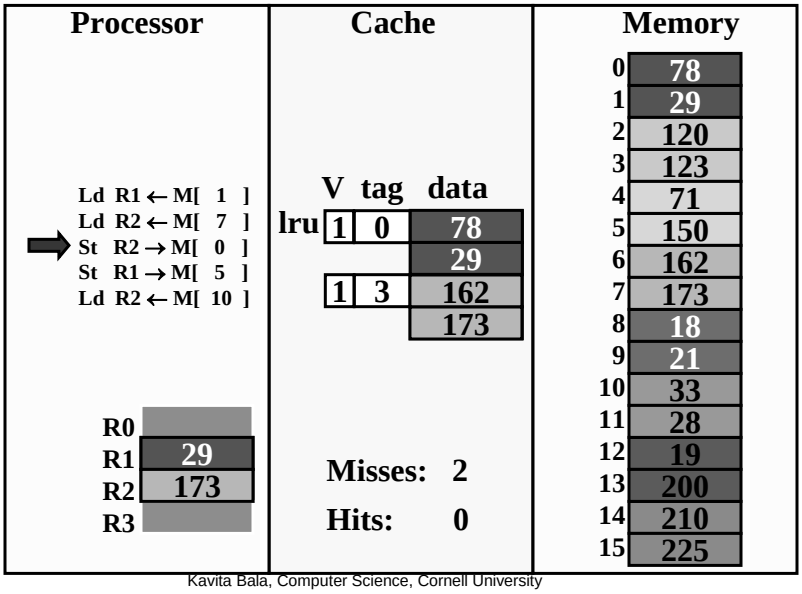
Write-Through (REF 2)



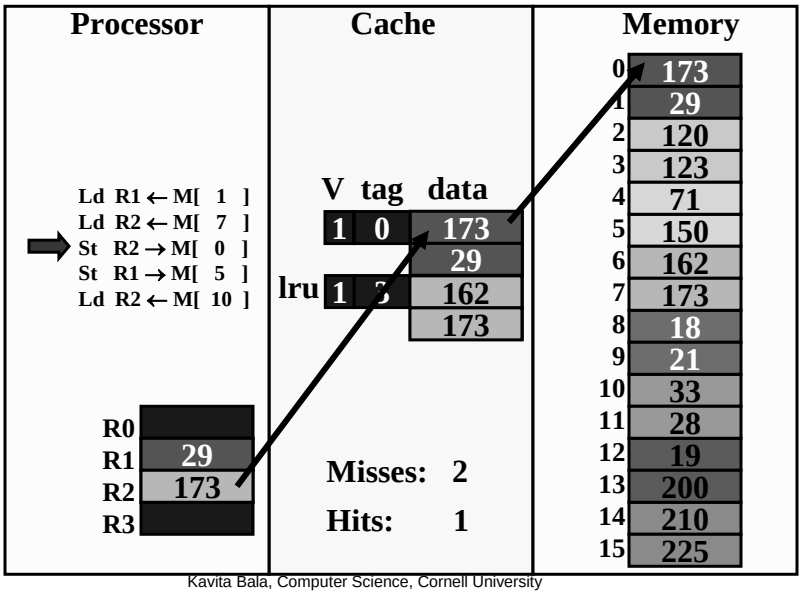
Write-Through (REF 2)



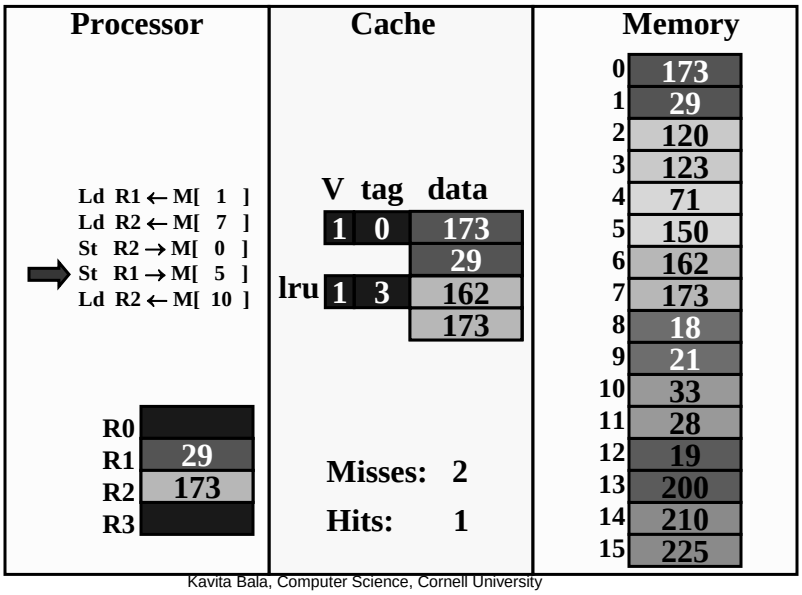
Write-Through (REF 3)



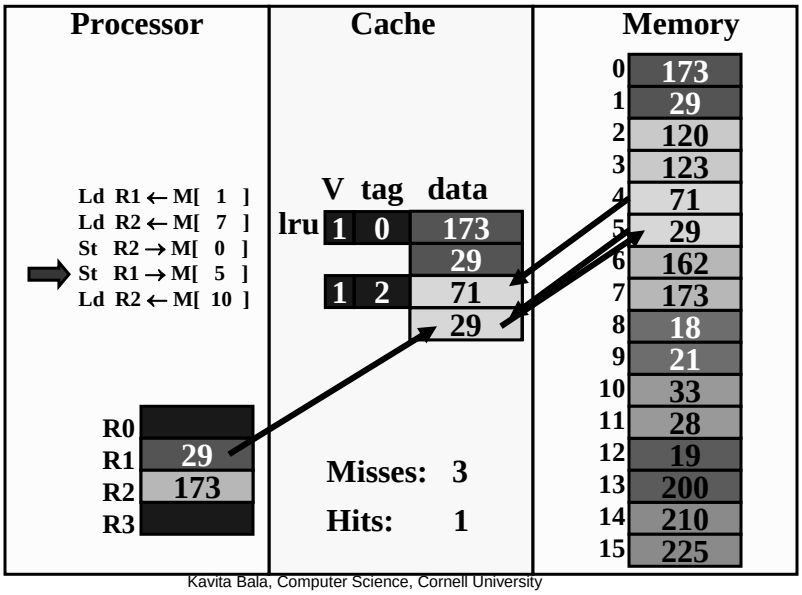
Write-Through (REF 3)



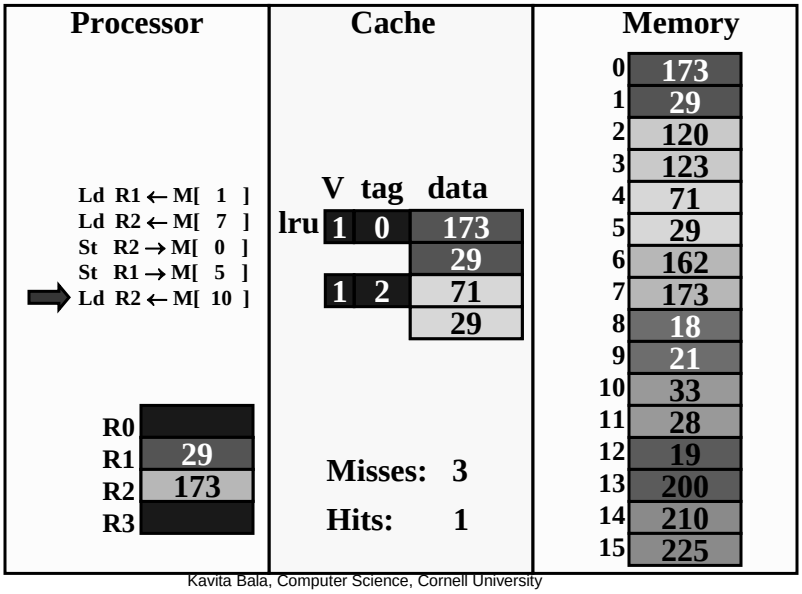
Write-Through (REF 4)



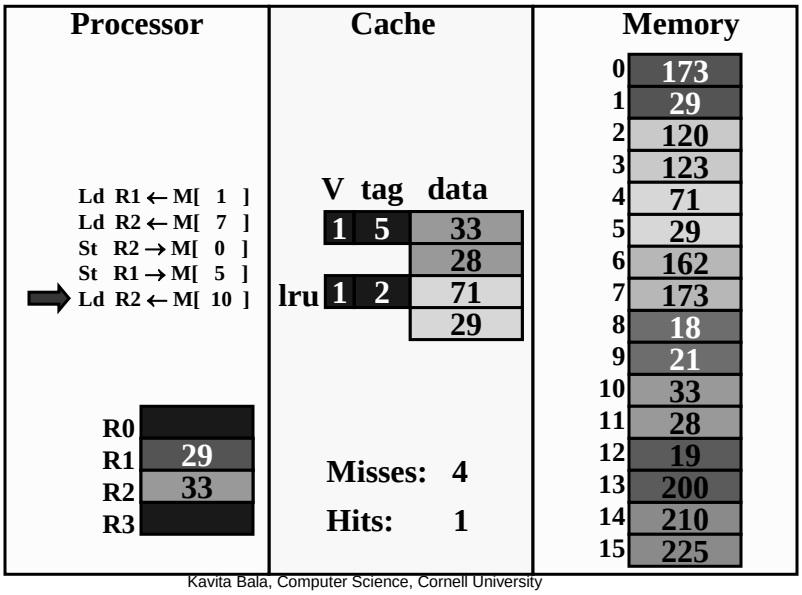
Write-Through (REF 4)



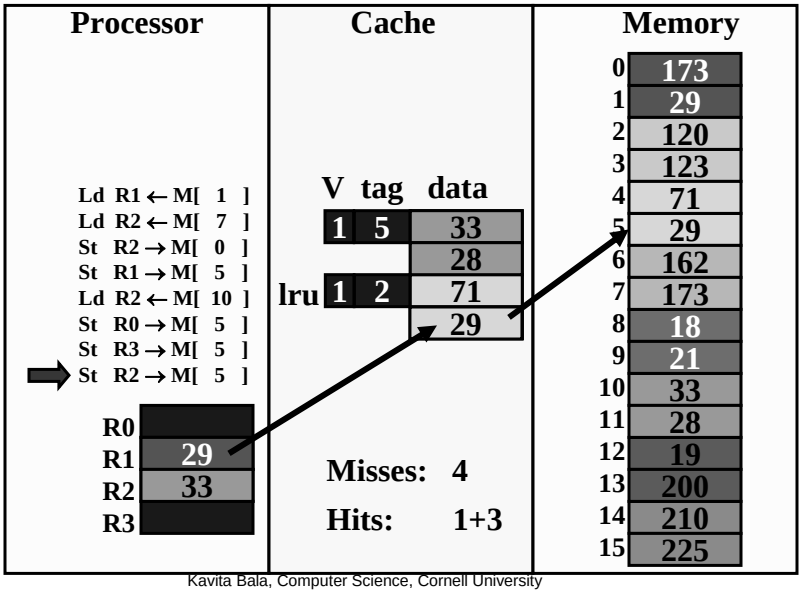
Write-Through (REF 5)



Write-Through (REF 5)



Write-Through (REF 6,7,8)



How Many Memory References?

- Each miss reads a block (only two words in this cache)
- Each store writes a word (or a block, depends)
- Total reads: eight words
- Total writes:
  - Before last 3 stores: two words
  - After last 3 stores: five

Write-Through vs. Write-Back

Can we also design the cache NOT to write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is evicted (write-back policy)
- Do we need to write-back all evicted lines?
- No, only blocks that have been stored into (written)
- Keep a “dirty bit”, reset when the line is allocated, set when the block is written
- If a block is “dirty” when evicted, write its data back into memory

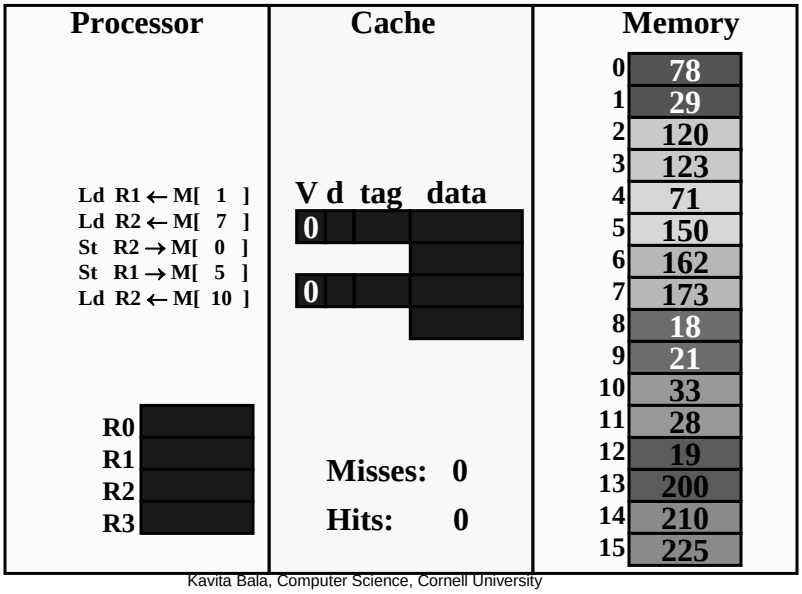
Dirty Bits and Write-Back Buffers

V	D	Tag	Data Byte 0, Byte 1 … Byte N
1	0		
1	1		
1	0		

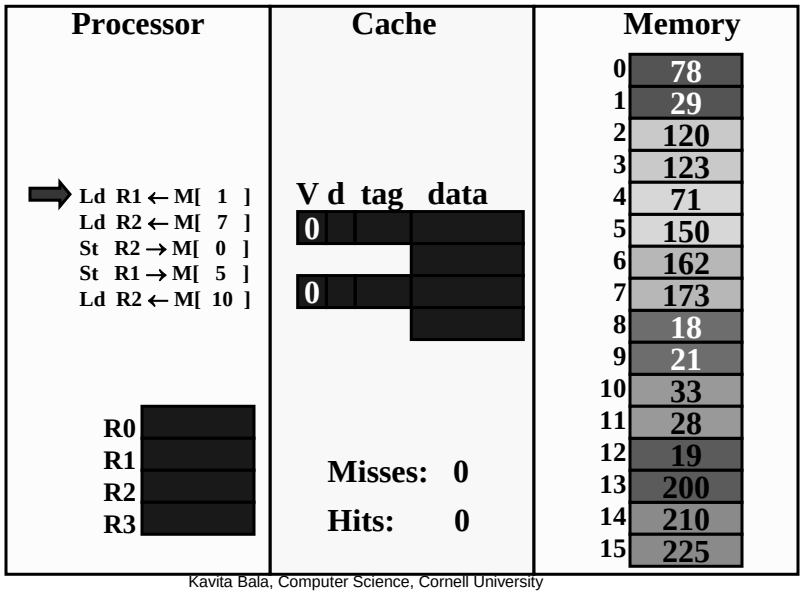
Line

- Dirty bits indicate which lines have been written
- Dirty bits enable the cache to handle multiple writes to the same cache line without having to go to memory
- Write-back buffer
  - A queue where dirty lines are placed

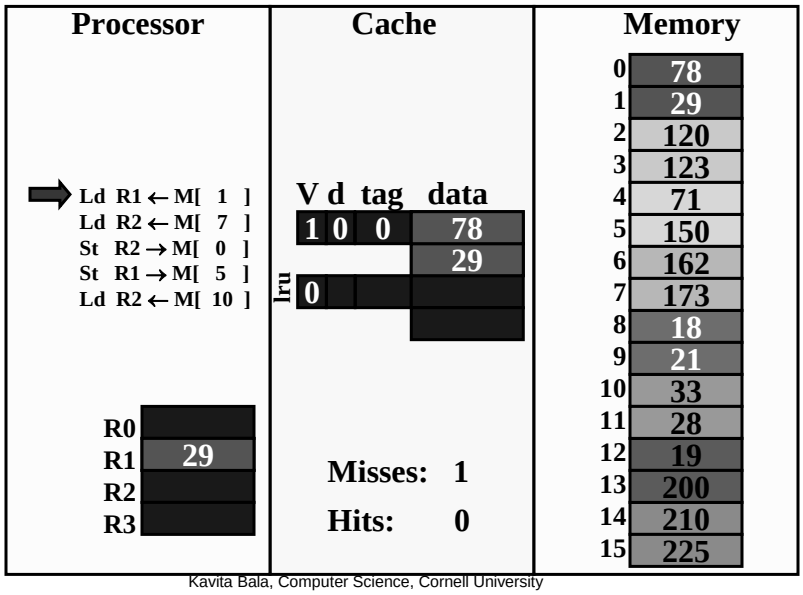
Handling Stores (Write-Back)



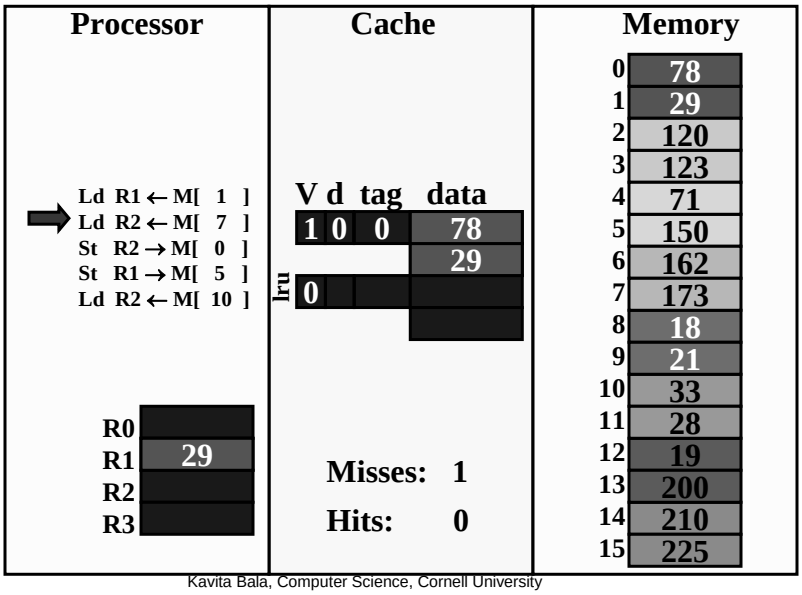
Write-Back (REF 1)



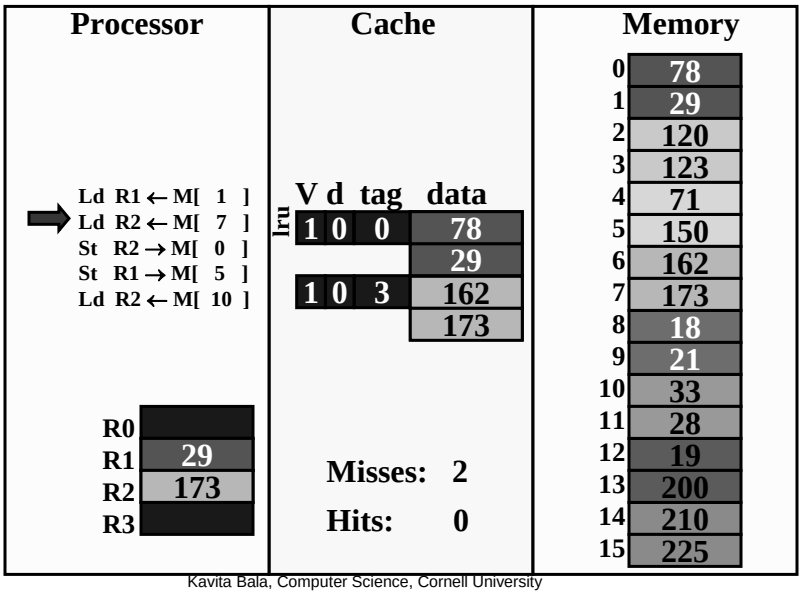
Write-Back (REF 1)



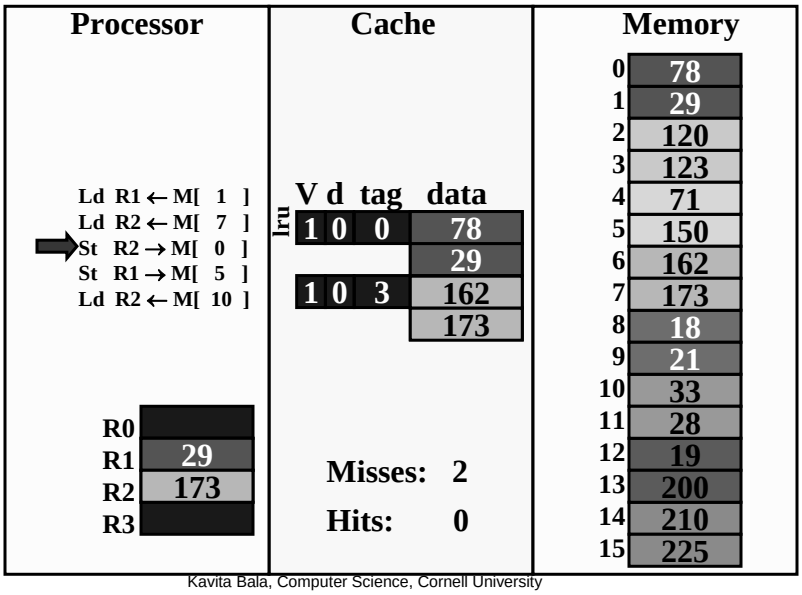
Write-Back (REF 2)



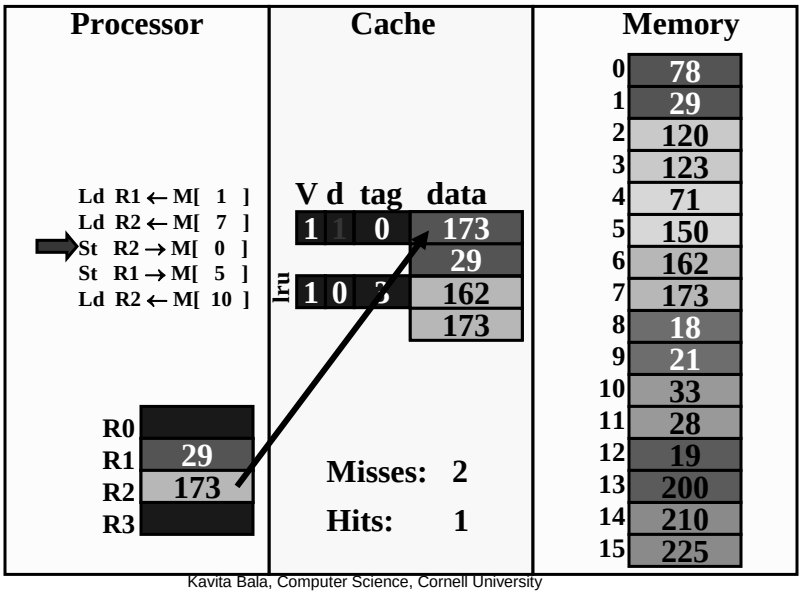
Write-Back (REF 2)



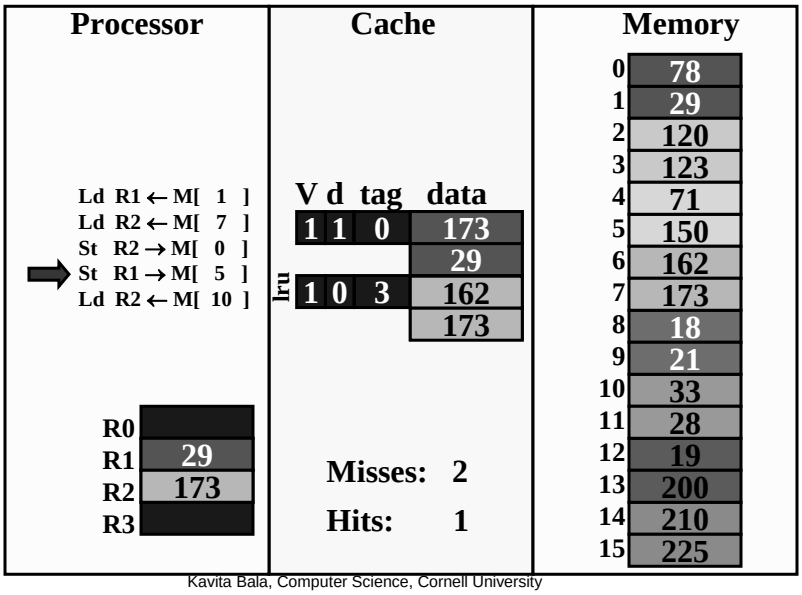
Write-Back (REF 3)



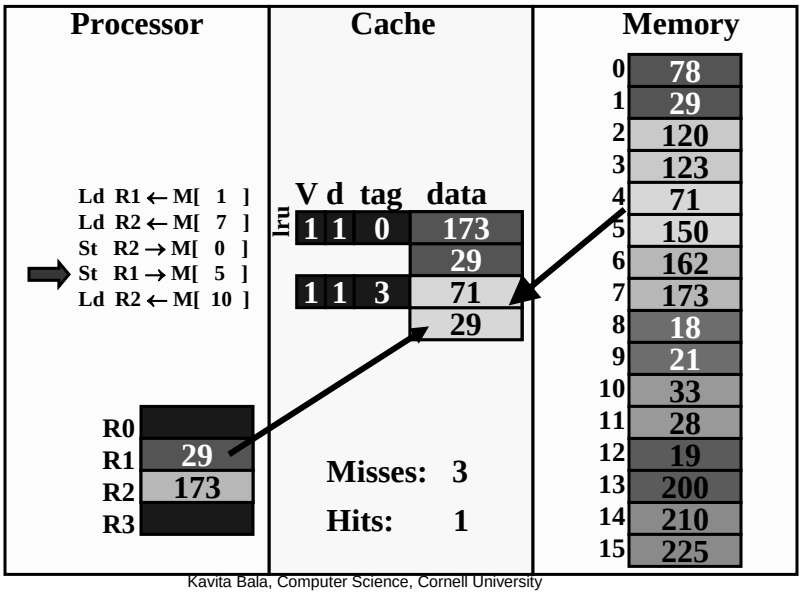
Write-Back (REF 3)



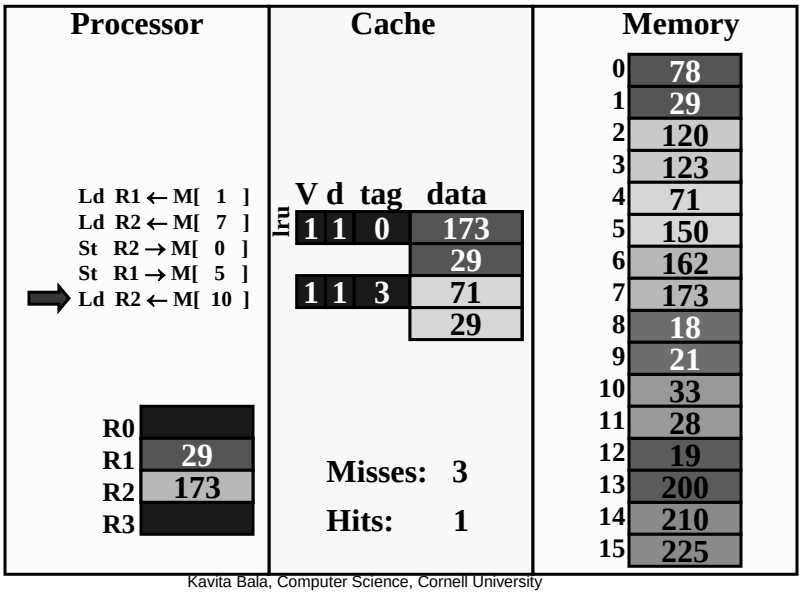
Write-Back (REF 4)



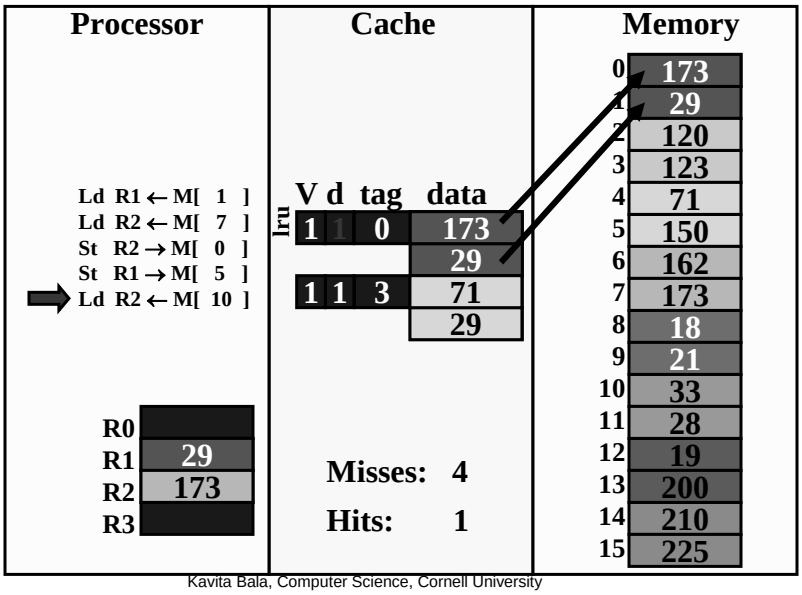
Write-Back (REF 4)



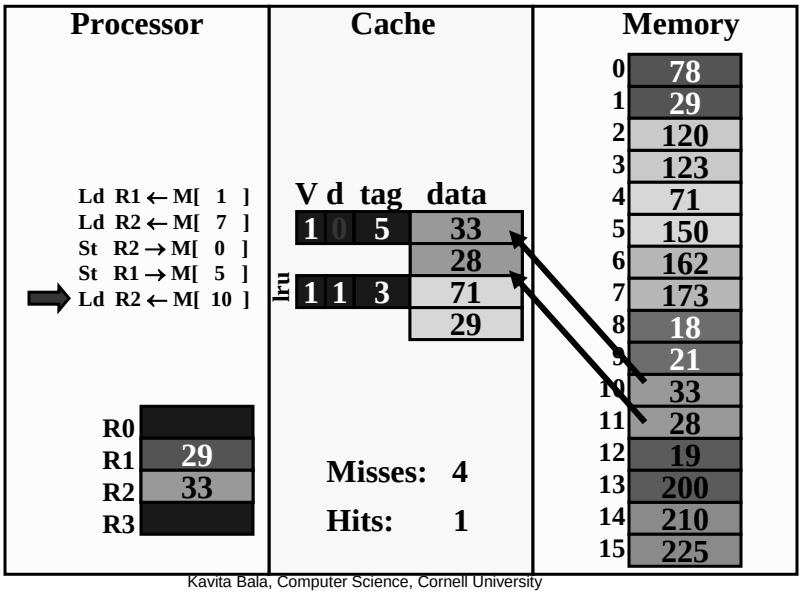
Write-Back (REF 5)



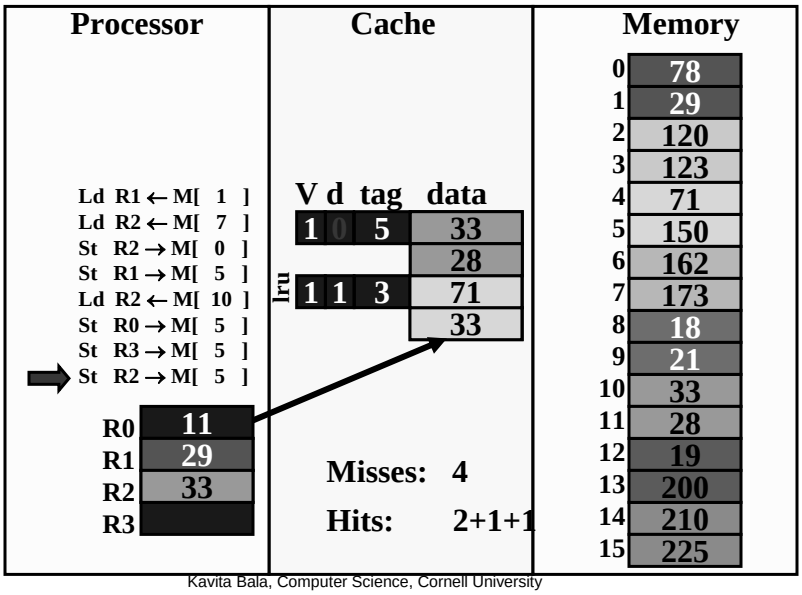
Write-Back (REF 5)



Write-Back (REF 5)



Write-Back (REF 6, 7, 8)



How many memory references?

- Each miss reads a block
  Two words in this cache
- Each evicted dirty cache line writes a block
- Total reads: eight words
- Total writes: four after final eviction

Choose write-back or write-through?

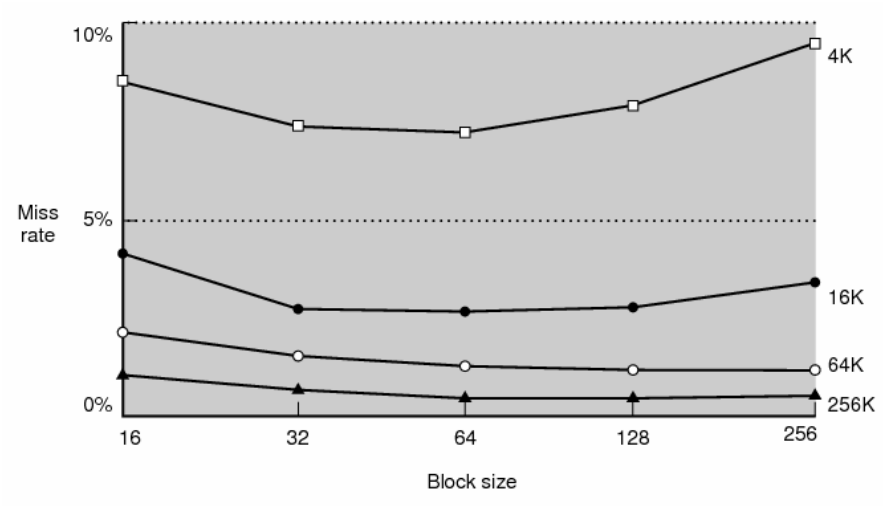
Cache Design

- Need to determine parameters
  - Block size
  - Number of ways
  - Eviction policy
  - Write policy
  - Separate I-cache from D-cache

Basic Cache Organization

Decide on the block size

- How? Simulate lots of different block sizes and see which one gives the best performance
- Most systems use a block size between 32 bytes and 128 bytes



Tradeoff

- Larger sizes reduce the overhead by
  - Reducing the number of tags
  - Reducing the size of each tag
- But
  - Have fewer blocks available
  - And the time to fetch the block on a miss is longer

Short Performance Discussion

- Complicated
  - Time from start-to-end (wall-clock time)
  - System time, user time
  - CPI (Cycles per instruction)
- Ideal CPI?

Cache Performance

- Consider hit (H) and miss ratio (M)
- H x AT_{cache} + M x AT_{memory}
- Hit rate = 1 – Miss rate
- Access Time is given in cycles
- Ratio of Access times, 1:50

- 90% : .90 + .1 x 50 = 5.9
- 95% : .95 + .05 x 50 = .95+2.5=3.45
- 99% : .99 + .01 x 50 = 1.49
- 99.9%: .999 + .001 x 50 = 0.999 + 0.05 = 1.049

Cache Hit/Miss Rate

- Consider processor that is 2x times faster
  - But memory is same speed

- Since AT is access time in terms of cycle time: it doubles 2x
- H x AT_{cache} + M x AT_{memory}
- Ratio of Access times, 1:100
- 99% : .99 + .01 x 100 = 1.99

Cache Hit/Miss Rate

- Original is 1GHz, 1ns is cycle time
- CPI (cycles per instruction): 1.49
- Therefore, 1.49 ns for each instruction

- New is 2GHz, 0.5 ns is cycle time.
- CPI: 1.99, 0.5ns. 0.995 ns for each instruction.

- So it doesn't go to 0.745 ns for each instruction.
- Speedup is 1.5x (not 2x)

Misses

- Three types of misses
  - Cold
    - The line is being referenced for the first time
  - Capacity
    - The line was evicted because the cache was not large enough
  - Conflict
    - The line was evicted because of another access whose index conflicted

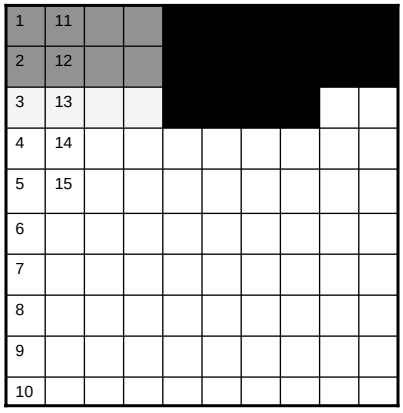
Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(j = 0; j < NCOL; ++j)
   for(i = 0; i < NROW; ++i)
        sum += a[j][i];
```

- Speed up this program!

Cache Conscious Programming

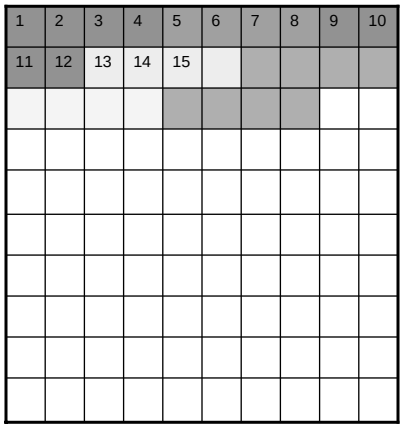
```
int a[NCOL][NROW];
int sum = 0;

for(j = 0; j < NCOL; ++j)
        for(i = 0; i < NROW; ++i)
                sum += a[j][i];
```



- Every access is a cache miss!

Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(i = 0; i < NROW; ++i)
        for(j = 0; j < NCOL; ++j)
                sum += a[j][i];
```



- Same program, trivial transformation, 3 out of four accesses hit in the cache