Caches IV

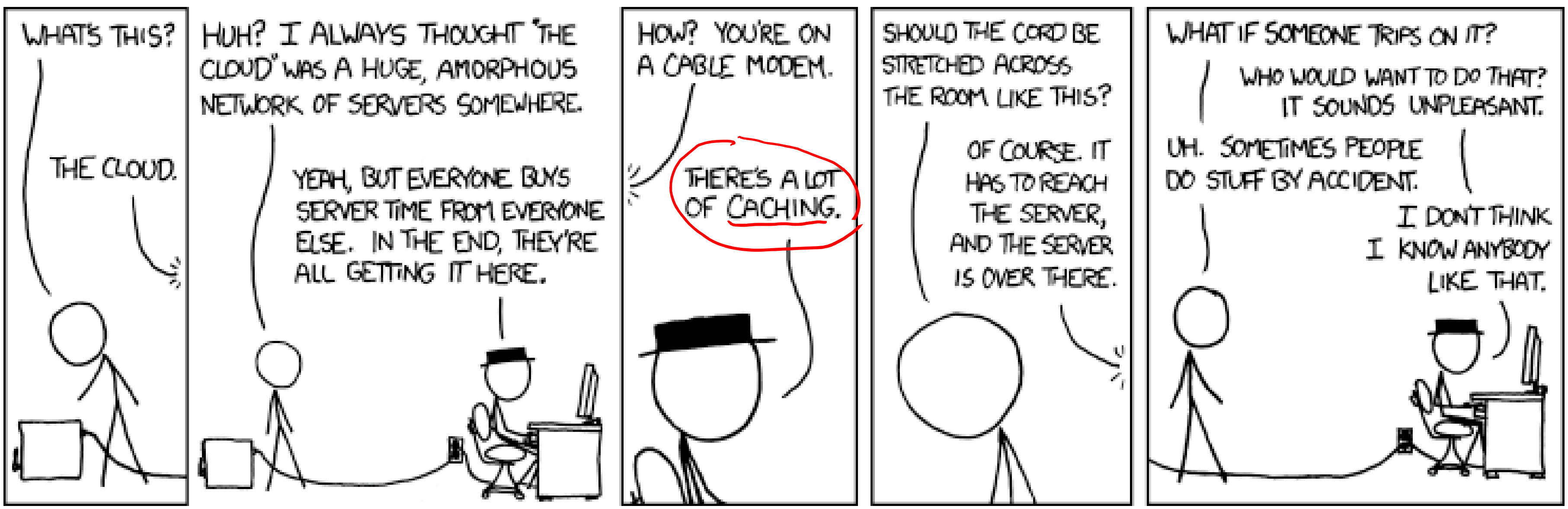
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

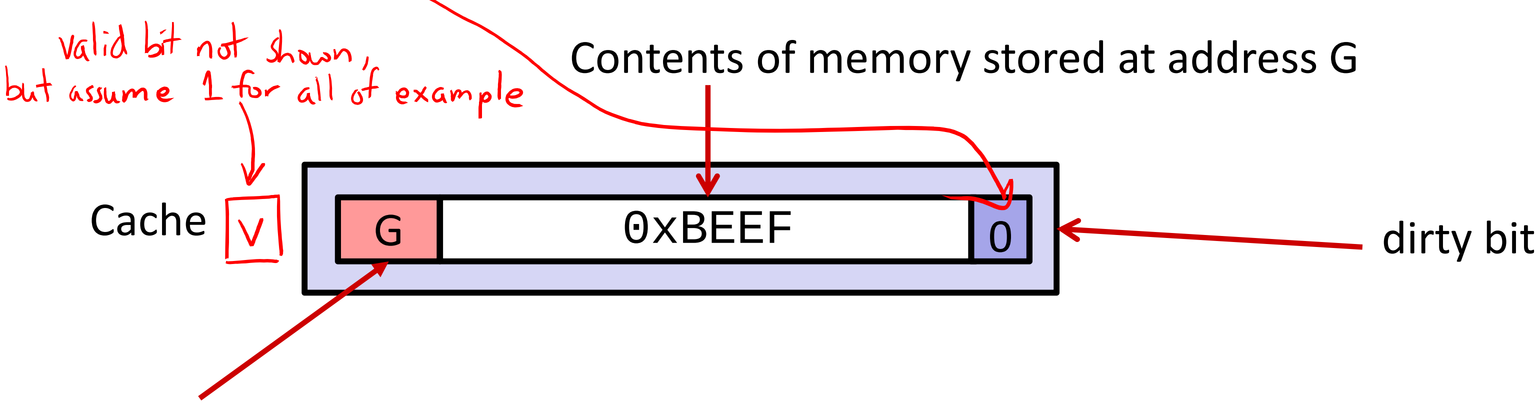


http://xkcd.com/908/

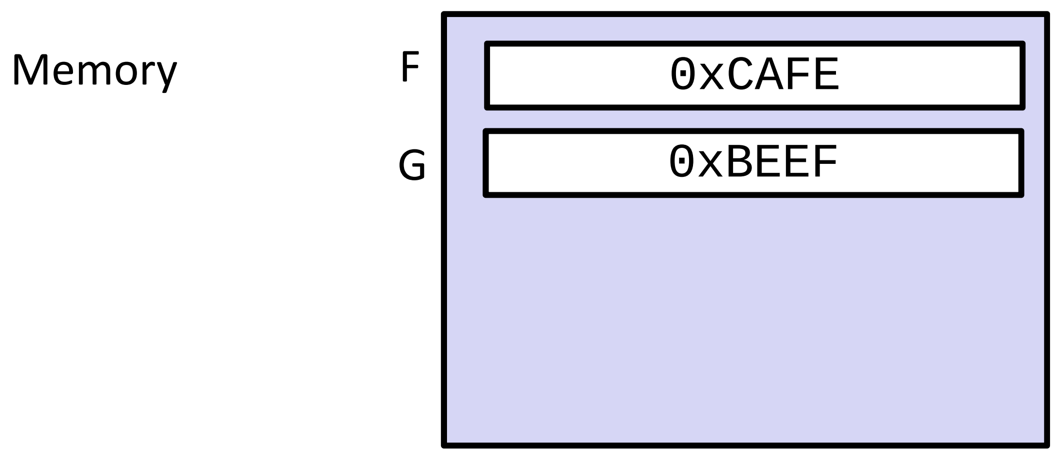
Administrivia

- Lab 3, due TONIGHT, Wednesday (5/15)
- Homework 4 , due Wed (5/22) (Structs, Caches)
- Lab 4, Coming soon!
  - Cache parameter puzzles and code optimizations

Write-back, write-allocate example



tag (there is only one set in this tiny cache, so the tag is the entire block address!)



In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

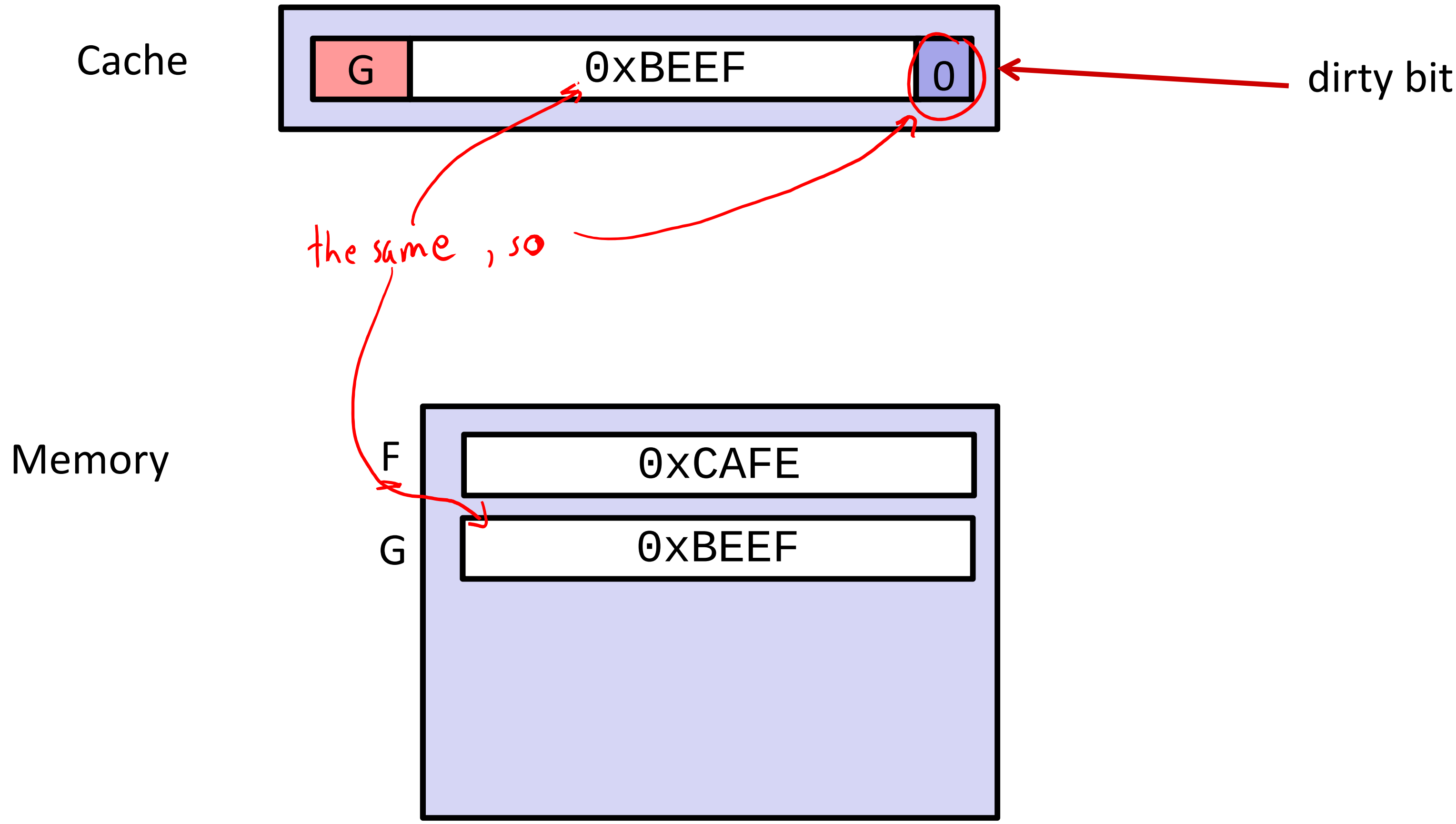
Write-back, write-allocate example

write miss

`mov 0xFACE, F`

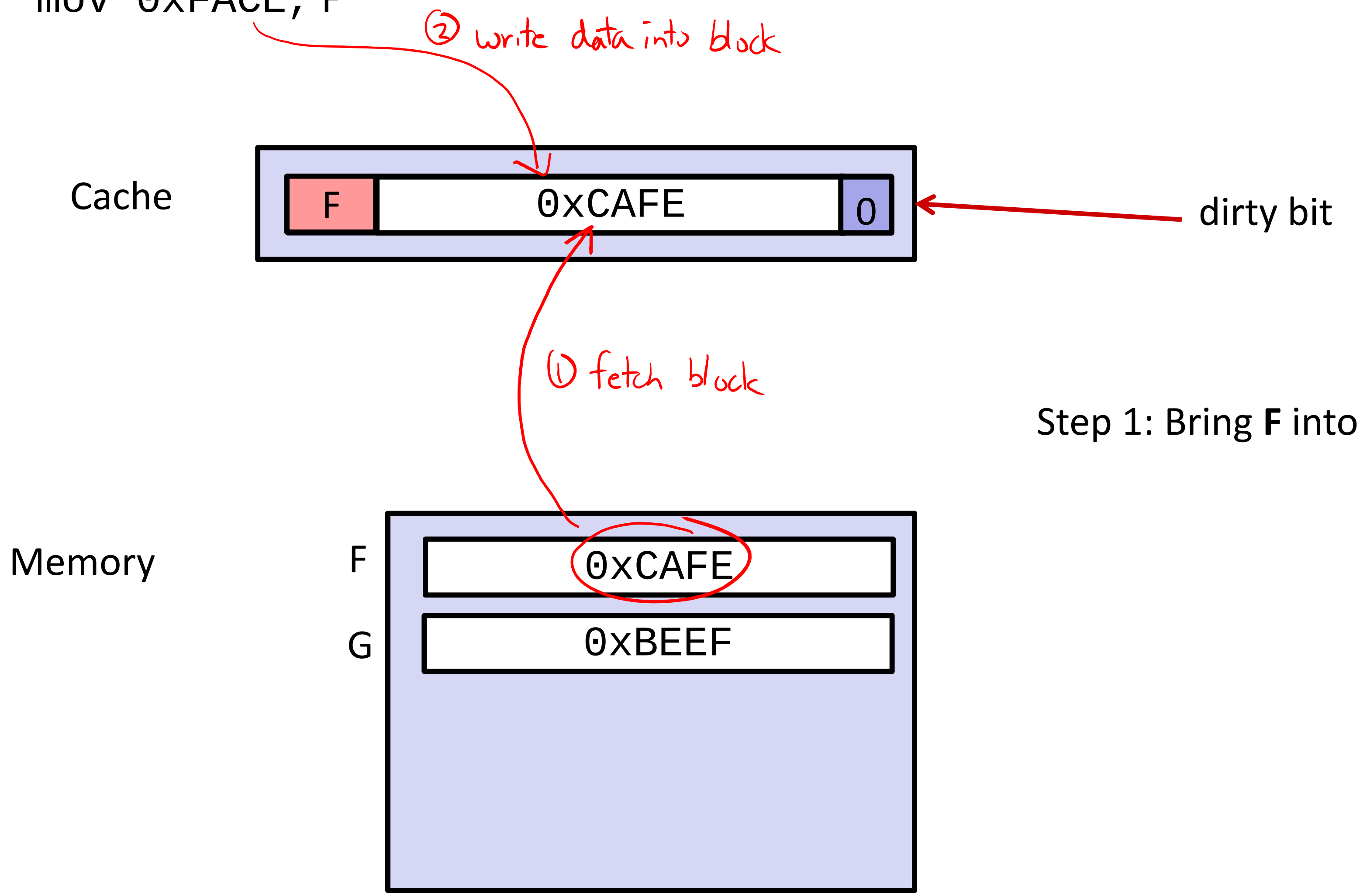
① check cache for F → miss

② pull block into $, then write



Write-back, write-allocate example

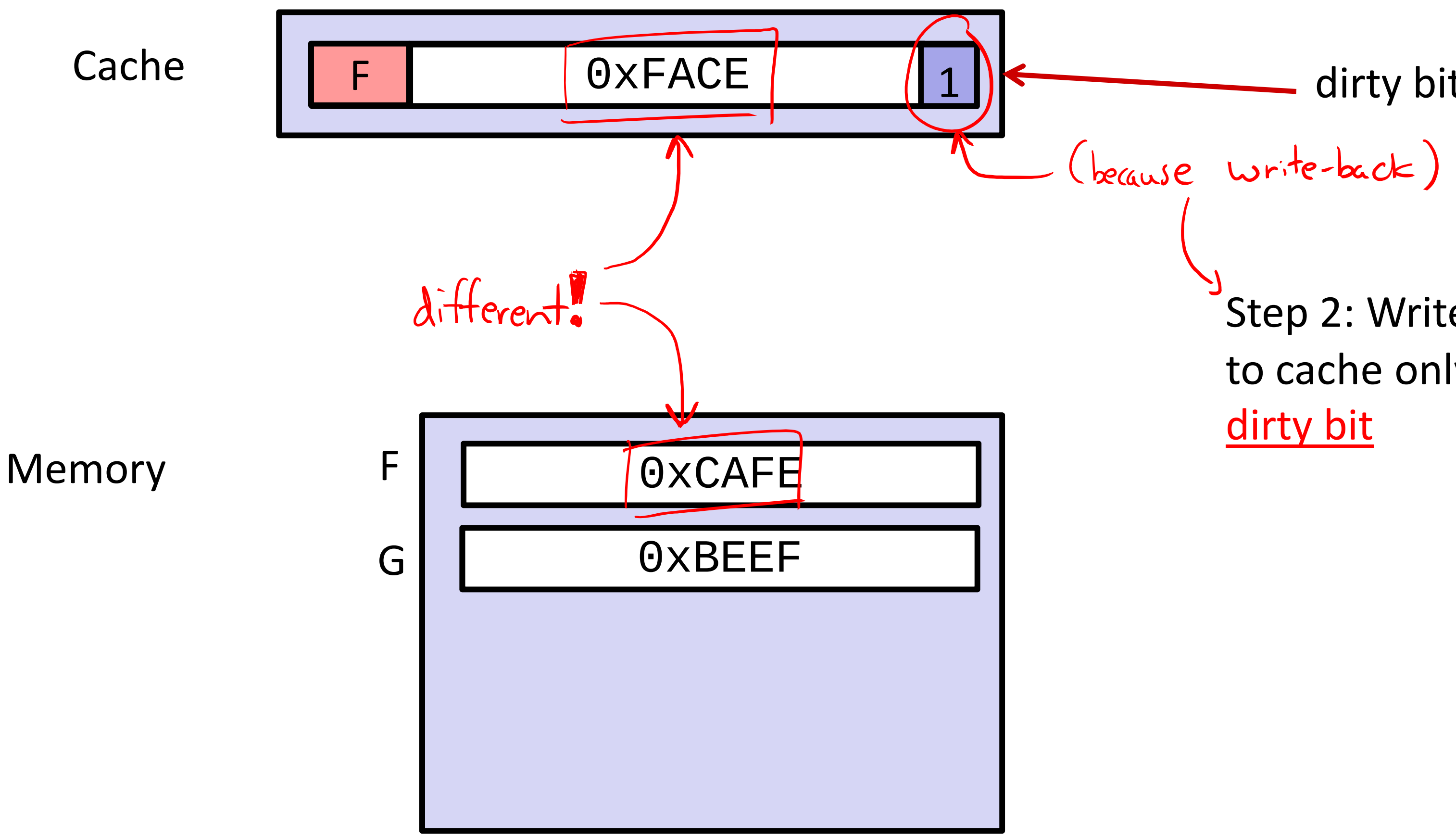
`mov 0xFACE, F`



Step 1: Bring **F** into cache

Write-back, write-allocate example

```
mov 0xFACE, F
```

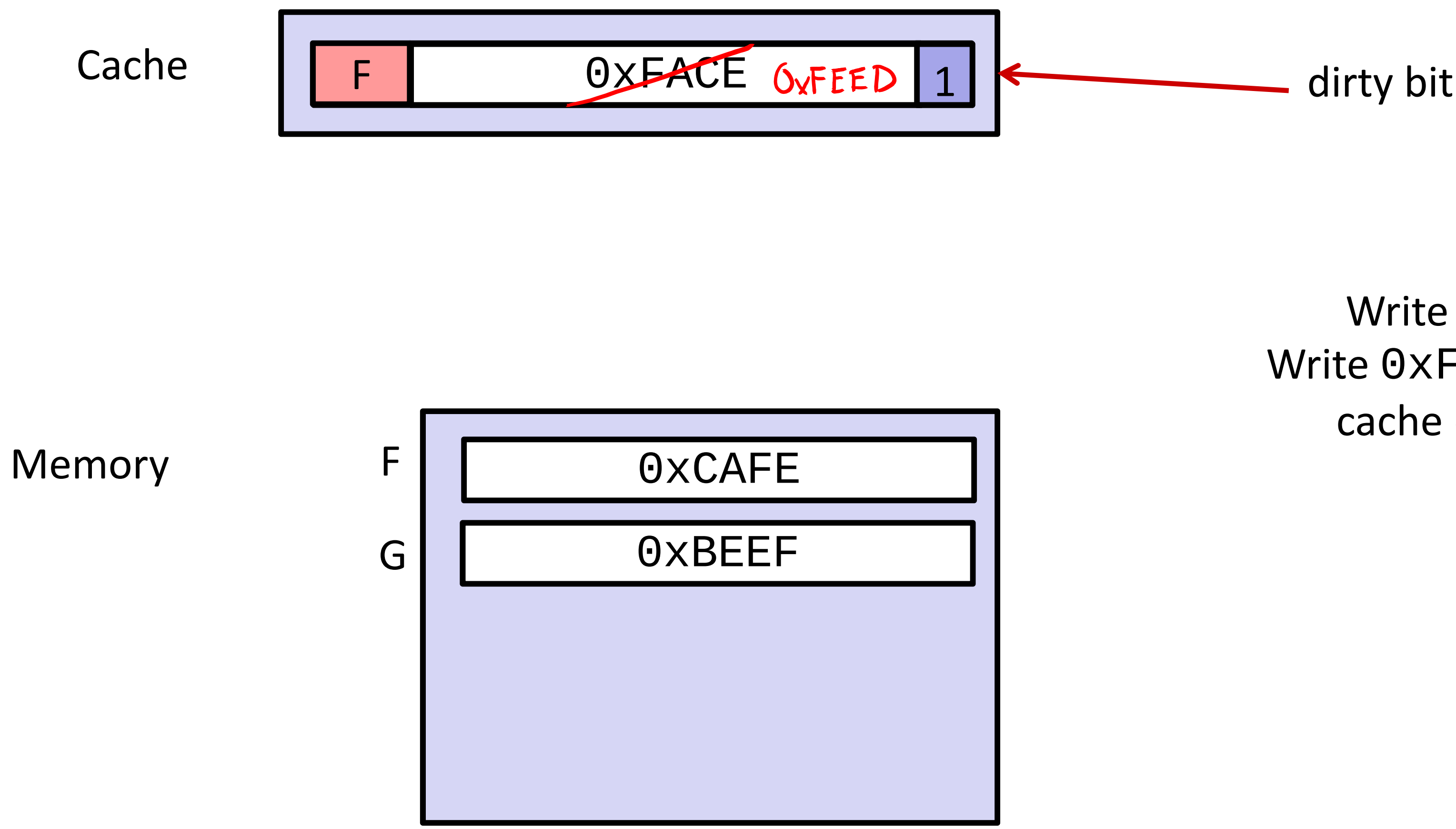


Step 2: Write `0xFACE` to cache only <u>and set dirty bit</u>

Write-back, write-allocate example

`mov 0xFACE, F`
write miss

write hit
`mov 0xFEED, F`

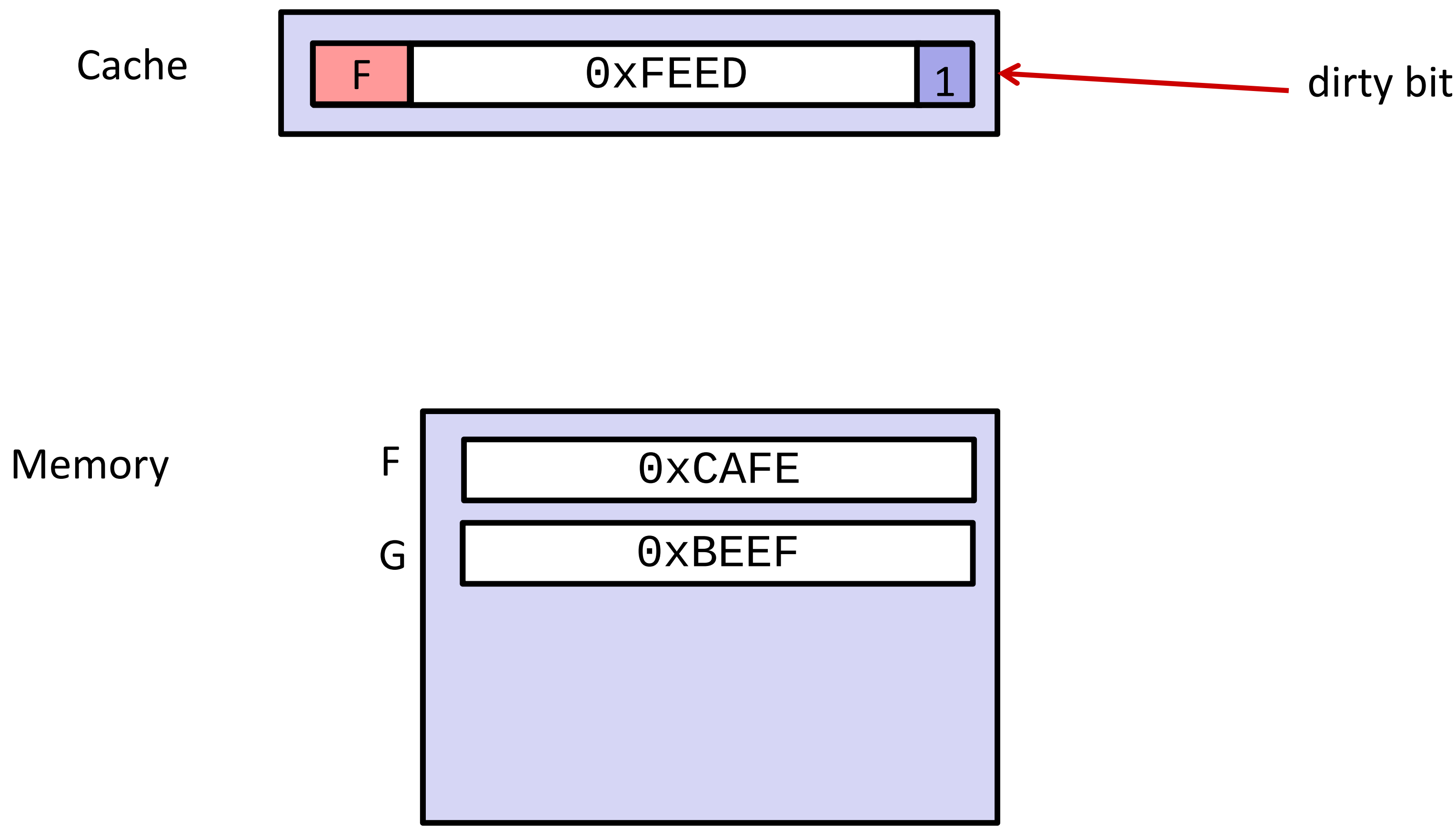


Write hit!
Write `0xFEED` to cache only

Write-back, write-allocate example

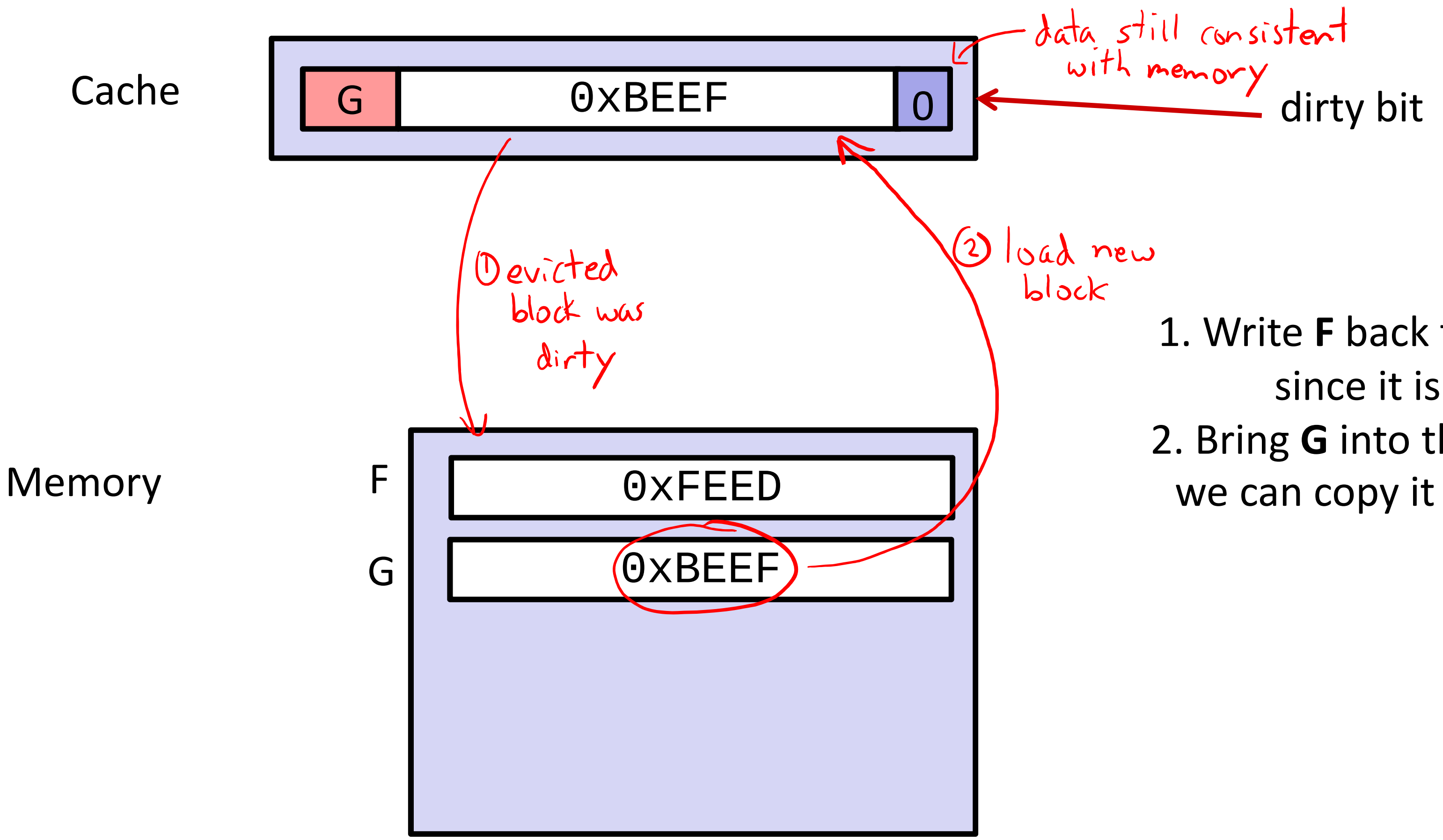
`mov 0xFACE, F` `mov 0xFEED, F` `mov G, %rax`

read miss



Write-back, write-allocate example

`mov 0xFACE, F`   `mov 0xFEED, F`   `mov G, %rax`



1. Write **F** back to memory since it is dirty
2. Bring **G** into the cache so we can copy it into `%rax`

Peer Instruction Question

- Which of the following cache statements is FALSE?
  - Vote at http://pollev.com/rea

False A. We can reduce compulsory misses by decreasing our block size
smaller block size pulls fewer bytes into $ on a miss

True B. We can reduce conflict misses by increasing associativity
more options to place blocks before evictions occur

True C. A write-back cache will save time for code with good temporal locality on writes
frequently-used blocks rarely get evicted, so fewer write-backs

True D. A write-through cache will always match data with the memory hierarchy level below it
yes, its main goal is data consistency

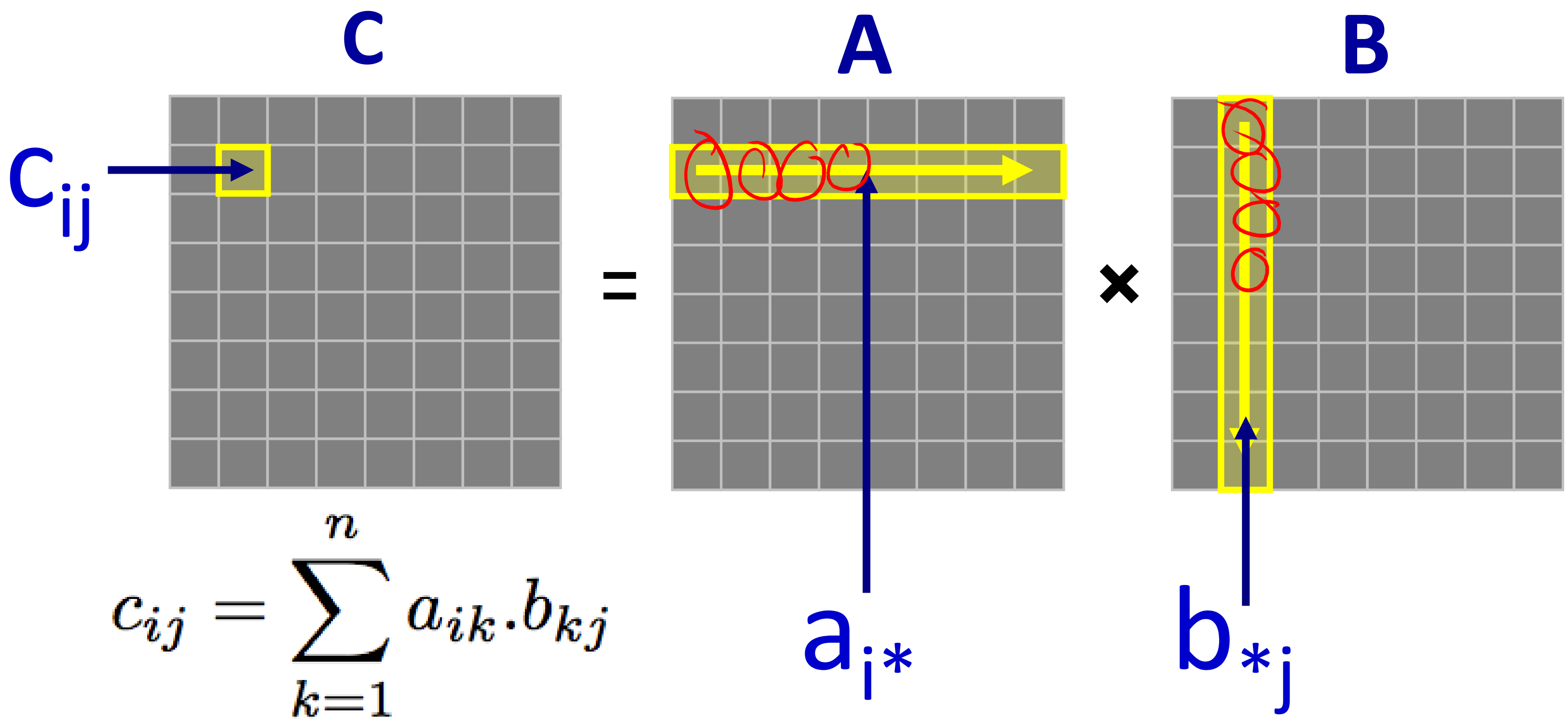
E. We're lost…

Optimizations for the Memory Hierarchy

- Write code that has locality!
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How can you achieve locality?
  - Adjust memory accesses in *code* (software) to improve miss rate (MR)
    - Requires knowledge of *both* how caches work as well as your system's parameters
  - Proper choice of algorithm
  - Loop transformations

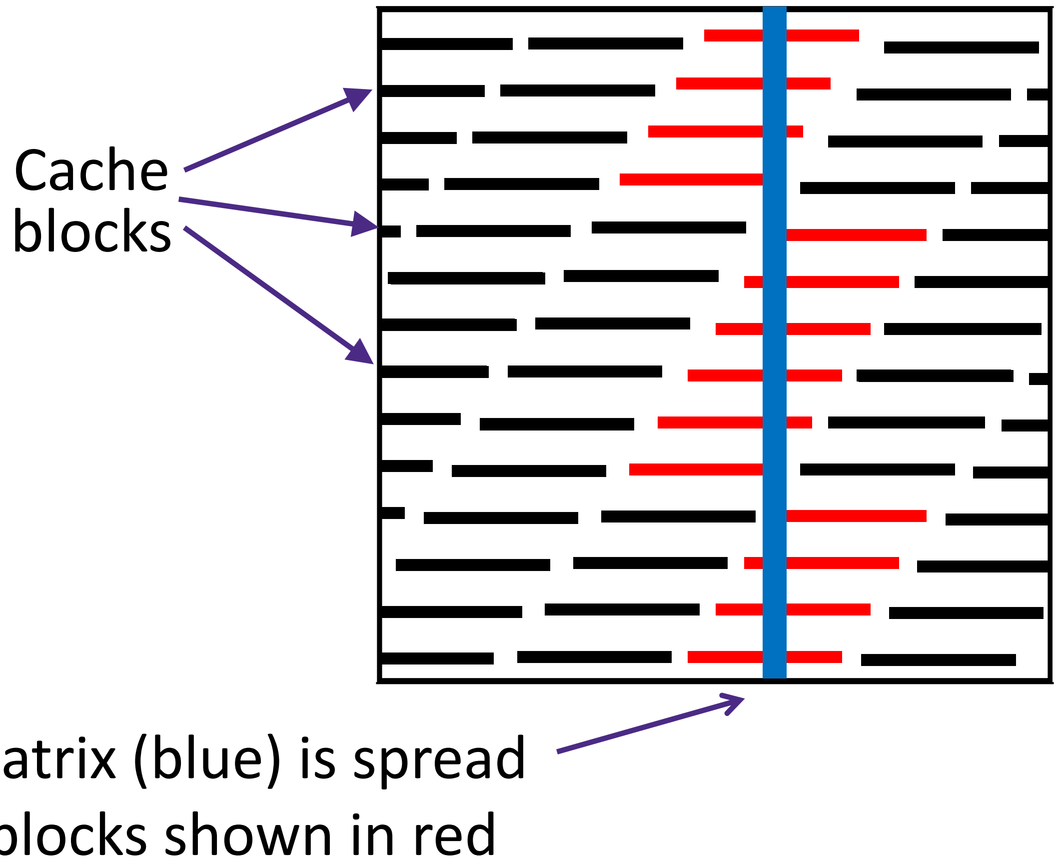
Example: Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

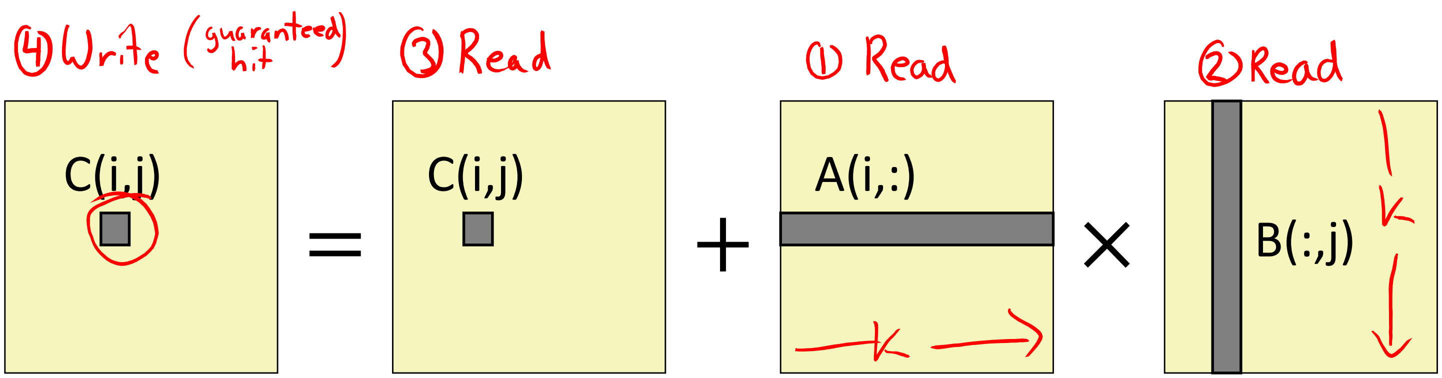
Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:



Naïve Matrix Multiply

```
# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for (k = 0; k < n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];
```

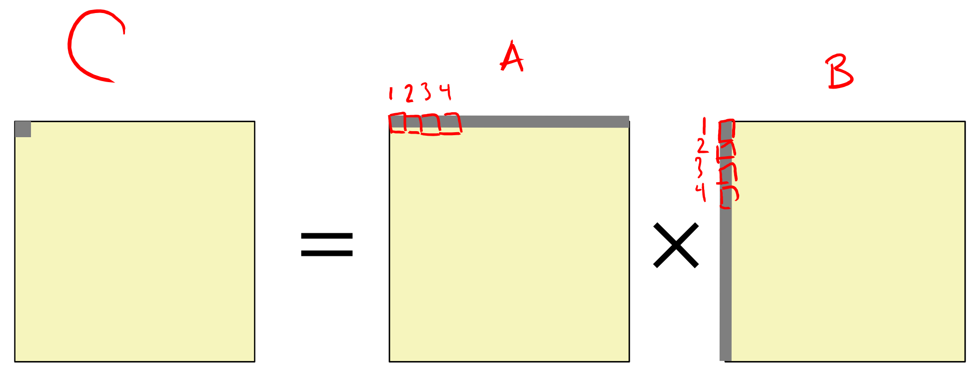


c[i][j] += a[i][k] * b[k][j]

Cache Miss Analysis (Naïve)

Ignoring matrix c

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles ← 8 matrix elements per cache block
  - Cache size $C \ll n$ (much smaller than n)
    key assumption!



- Each iteration:
  - $\overset{A}{\frac{n}{8}} + \overset{B}{n} = \frac{9n}{8}$ misses

compulsory

A has good spatial locality

M H H H H H H H

① ② ③ ④

stride-1

B has bad spatial locality

① (n+1)

②

③

④

⋮

ⓝ

stride-n

by the time we get to n+1, block has been kicked out of $

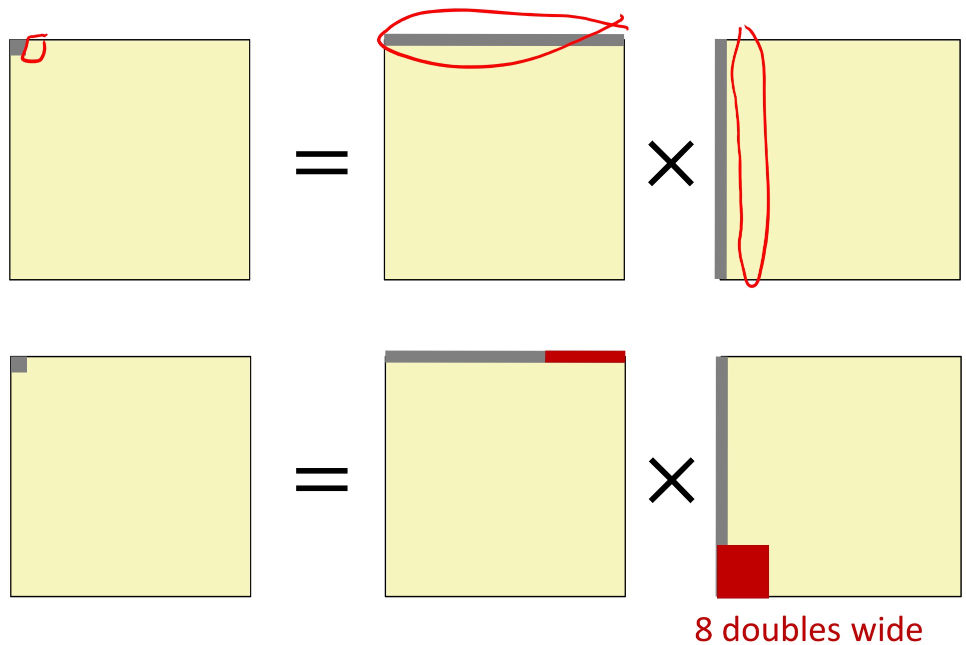
Cache Miss Analysis (Naïve)

Ignoring matrix c

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses
  - Afterwards in cache: (schematic)

red showing blocks remaining in the $

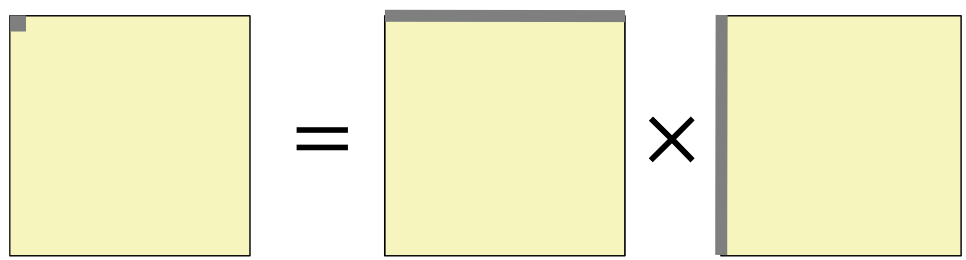


Cache Miss Analysis (Naïve)

Ignoring matrix c

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$

once per element

This is extra (non-testable) material

Linear Algebra to the Rescue (1)

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix "blocks")

- For example, multiply two 4×4 matrices:

A_{11} A_{12}

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ with } B \text{ defined similarly.}$$

A_{21} A_{22}

$$\boxed{AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}}$$

Linear Algebra to the Rescue (2)

This is extra (non-testable) material

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

Matrices of size $n \times n$, split into 4 blocks of size r (n=$4r$)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k}{*}B_{k2}$$

- Multiplication operates on small "block" matrices
  - Choose size so that they fit in the cache!
  - This technique called "*cache blocking*" ☆

Blocked Matrix Multiply

6 nested loops may seem less efficient, but leads to much faster code!

- Blocked version of the naïve algorithm:

```
# move by rxr BLOCKS now
for (i = 0; i < n; i += r)
  for (j = 0; j < n; j += r)
    for (k = 0; k < n; k += r)
      # block matrix multiplication
      for (ib = i; ib < i+r; ib++)
        for (jb = j; jb < j+r; jb++)
          for (kb = k; kb < k+r; kb++)
            c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

walk thru entire matrix a block at a time

walk thru one block

  - r = block matrix size (assume r divides n evenly)

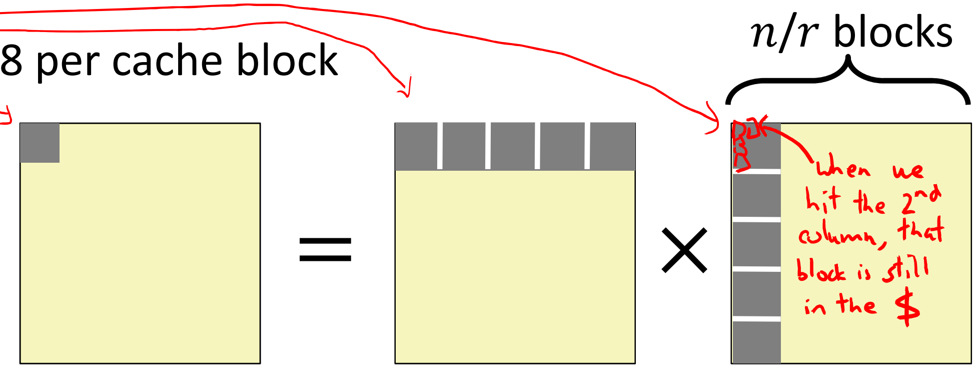
Cache Miss Analysis (Blocked)

Ignoring matrix c

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ $(r \times r)$ fit into cache: $3r^2 < C$

r^2 elements per block, 8 per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and column

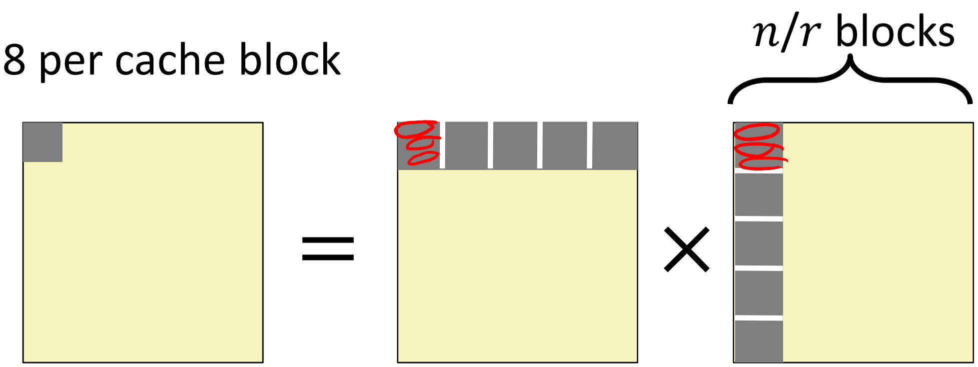
Cache Miss Analysis (Blocked)

Ignoring matrix c

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ $(r \times r)$ fit into cache: $3r^2 < C$

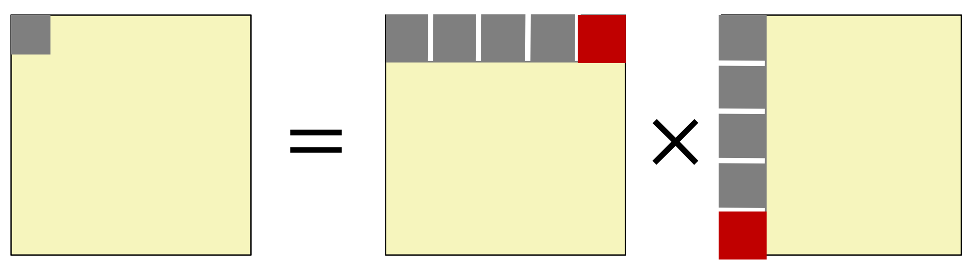
r^2 elements per block, 8 per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and column

  - Afterwards in cache (schematic)



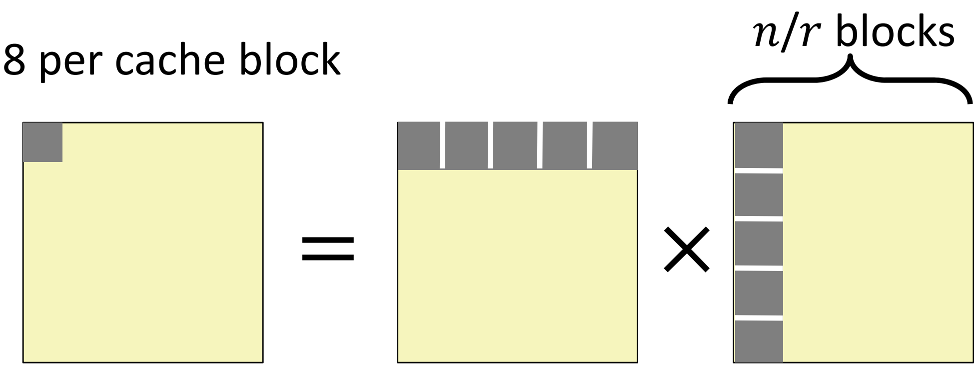
Cache Miss Analysis (Blocked)

Ignoring matrix c

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ $(r \times r)$ fit into cache: $3r^2 < C$

r^2 elements per block, 8 per cache block

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and column

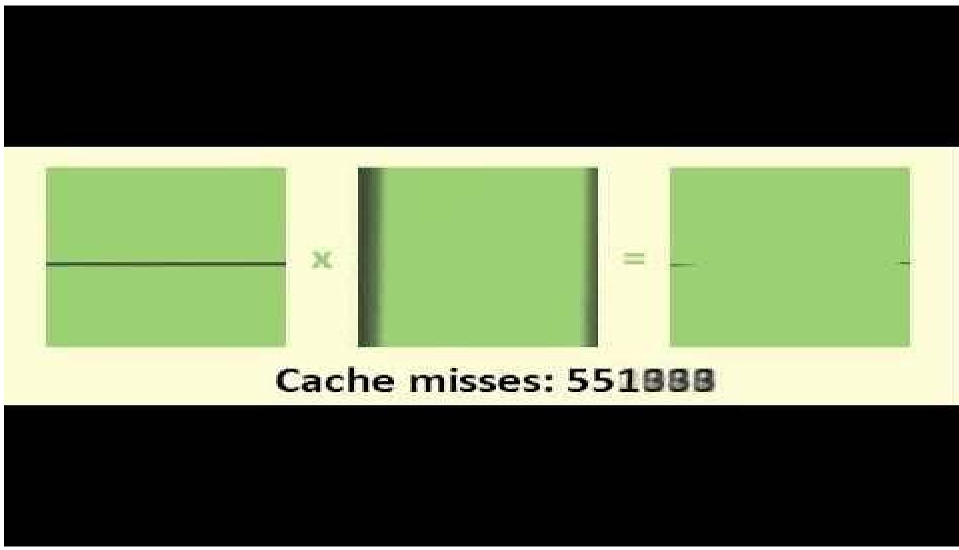
- Total misses:
  - $nr/4 \times (n/r)2 = n^3/(4r)$ vs. $9n^3/8$

Matrix Multiply Visualization

- Here n = 100, C = 32 KiB, r = 30

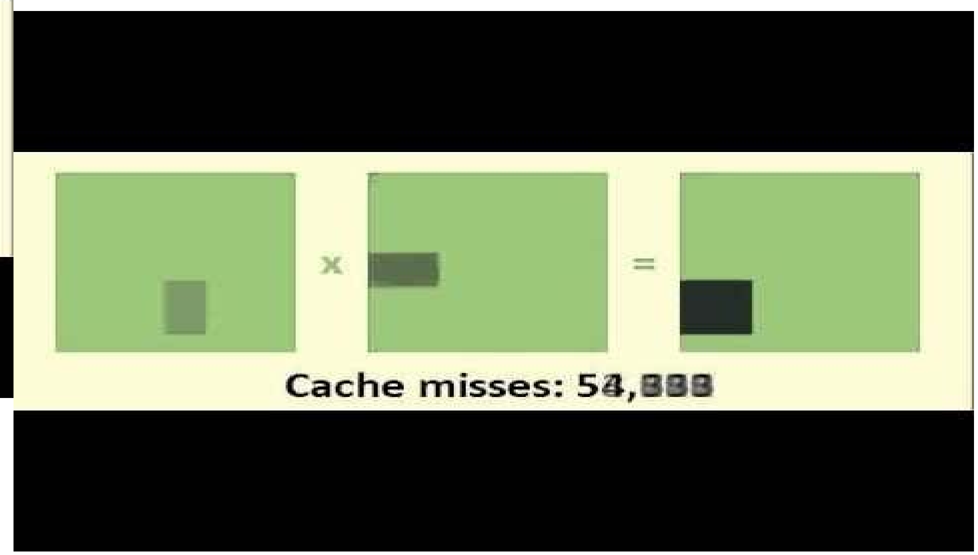
shaded areas show blocks stored in the $

Naïve:



≈ 1,020,000
cache misses

Blocked:



≈ 90,000
cache misses

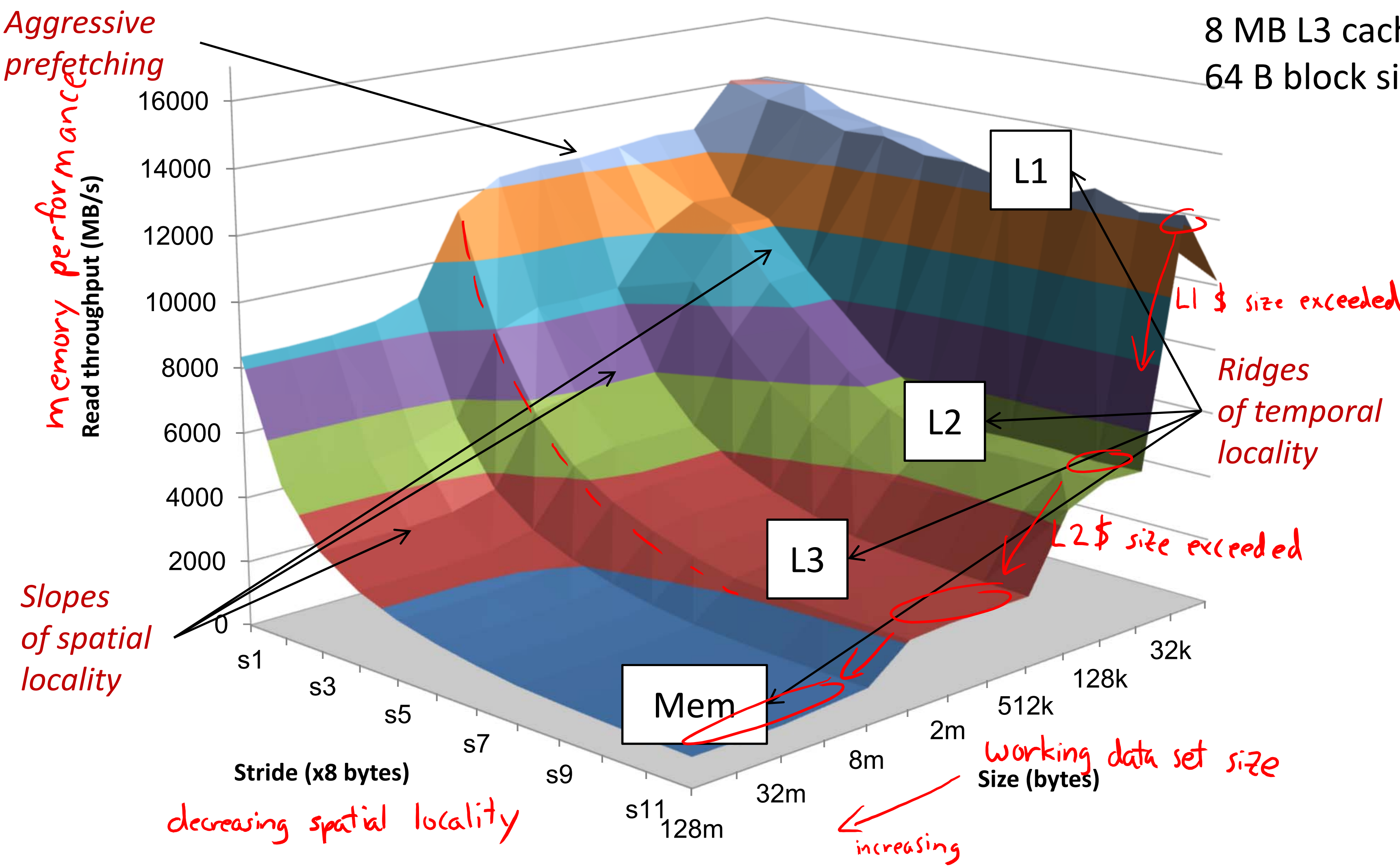
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor "cache-friendly code"
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code

great general rules of thumb!

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Learning About Your Machine

- **Linux:**
  - `lscpu`
  - ls /sys/devices/system/cpu/cpu0/cache/index0/
    - Ex: cat /sys/devices/system/cpu/cpu0/cache/index*/size
- **Windows:**
  - `wmic memcache get <query>` (all values in KB)
  - Ex: `wmic memcache get MaxCacheSize`

- Modern processor specs: http://www.7-cpu.com/