

Caches IV

CSE 351 Spring 2022

Instructor:

Ruth Anderson

Teaching Assistants:

Melissa Birchfield

Kyrie Dowling

Diya Joy

Armin Magness

Jeffery Tian

Angela Xu

Jacob Christy

Ellis Haker

Anirudh Kumar

Hamsa Shankar

Assaf Vayner

Effie Zheng

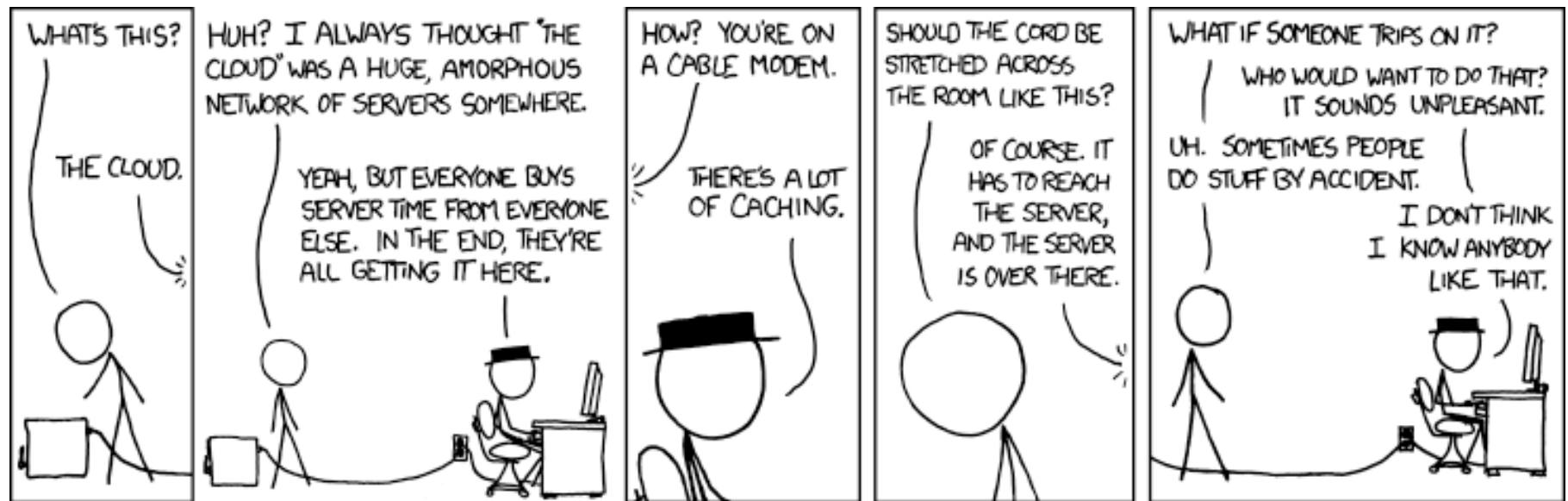
Alena Dickmann

Maggie Jiang

Jim Limprasert

Dara Stotland

Tom Wu



<http://xkcd.com/908/>

Relevant Course Information

- ❖ hw16 due Monday (5/09)
- ❖ Lab 3 due Wednesday (5/11)
- ❖ hw17 due Friday (5/13)
 - Don't wait too long, this is a BIG hw
- ❖ hw19 due Monday (5/16)
 - Lab 4 preparation
- ❖ Lab 4 coming soon!
 - Cache parameter puzzles and code optimizations

Reading Review

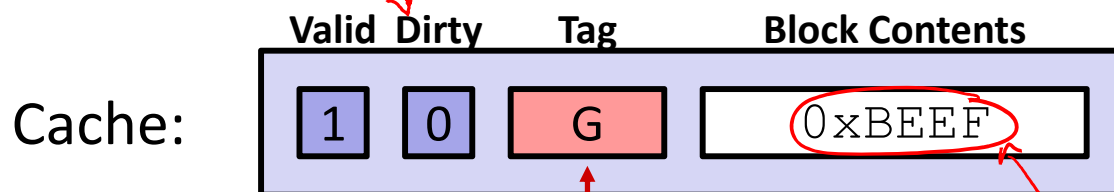
- ❖ Terminology:
 - Write-hit policies: write-back, write-through
 - Write-miss policies: write allocate, no-write allocate
 - Cache blocking

What about writes? (Review)

- ❖ Multiple copies of data may exist:
 - multiple levels of cache and main memory
- ❖ What to do on a write-hit? (block/data already in \$)
 - **Write-through:** ^{now} write immediately to next level
 - **Write-back:** ^{later} defer write to next level until line is evicted (replaced)
 - Must track which cache lines have been modified ("**dirty bit**") ← *extra management bit only for write-back cache*
- ❖ What to do on a write-miss? (block/data not currently in \$)
 - **Write allocate:** ("fetch on write") load into cache, then execute the write-hit policy
 - Good if more writes or reads to the location follow
 - **No-write allocate:** ("write around") just write immediately to next level
- ❖ Typical caches:
 - Write-back + Write allocate, usually *hit later miss*
 - Write-through + No-write allocate, occasionally

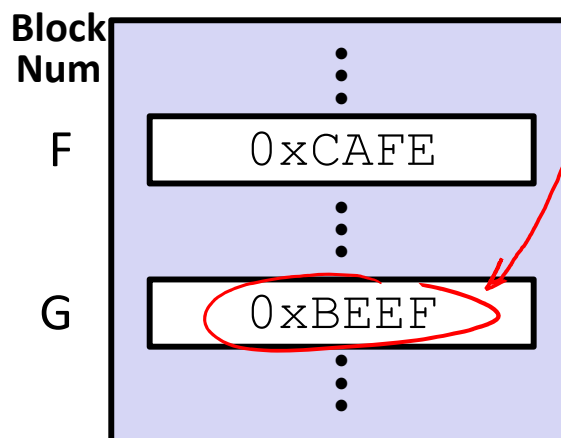
Write-back, Write Allocate Example

Note: While unrealistic, this example assumes that all requests have offset 0 and are for a block's worth of data.



There is only one set in this tiny cache, so the tag is the entire block number!

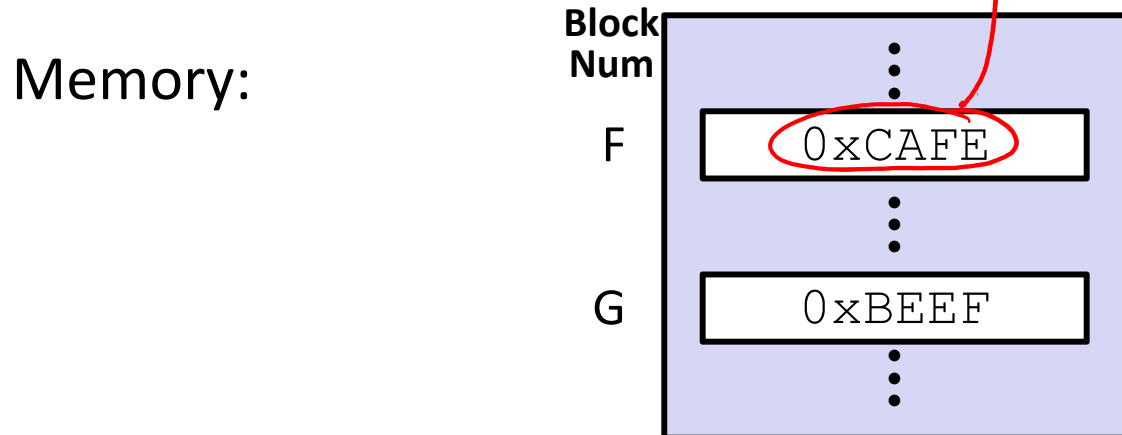
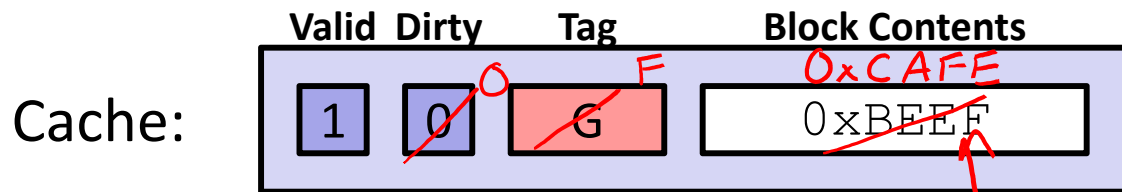
Memory:



not dirty, so these copies are consistent

Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`
on a hit *on a miss*
 Write Miss! Not valid x86, just using block num instead of full byte address to keep the example simple



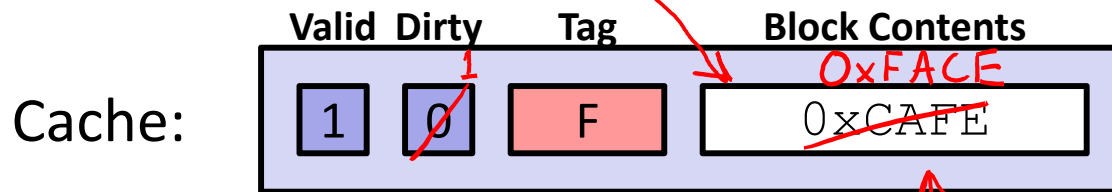
Step 1: Bring F into cache

Write-back, Write Allocate Example

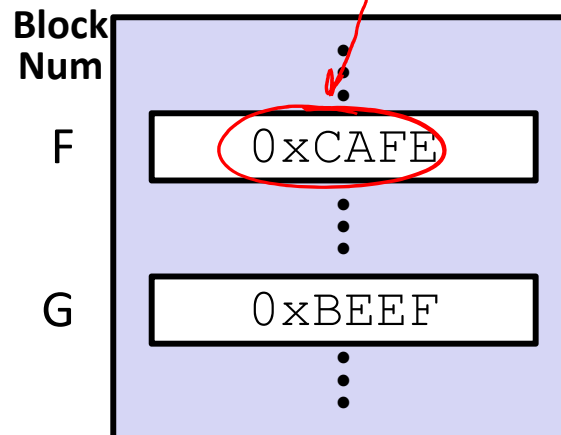
1) `mov $0xFACE, (F)`

Write Miss

② write data into block



Memory:



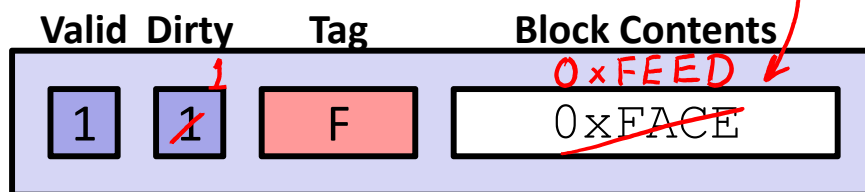
Step 1: Bring F into cache

Step 2: Write 0xFACE to cache only and set the dirty bit

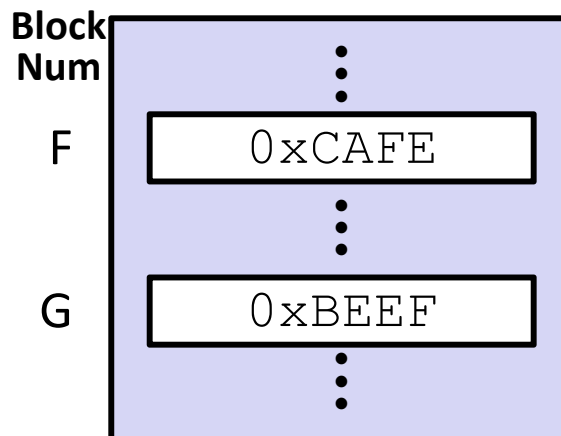
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`

Write Miss



Memory:



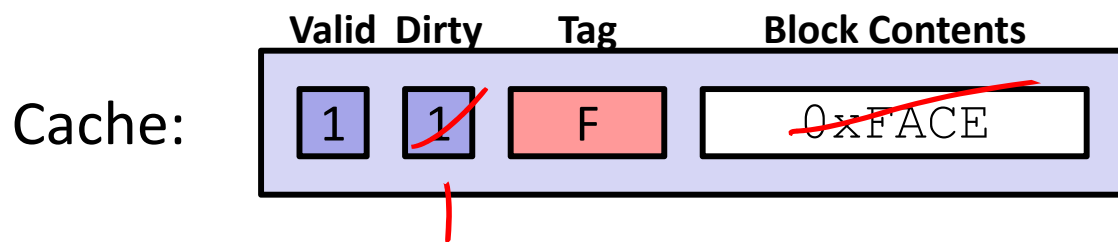
Step 1: Bring F into cache

Step 2: Write 0xFACE to cache only and set the dirty bit

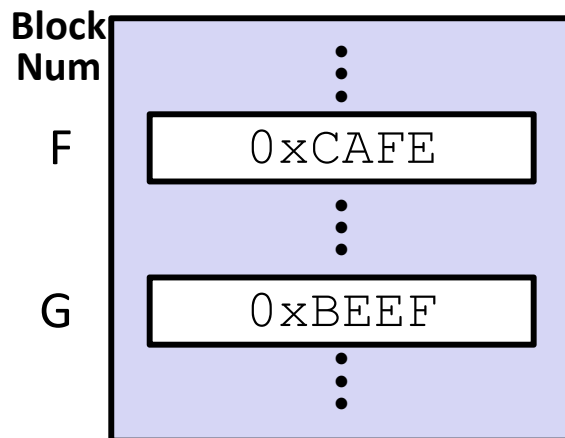
Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`
Write Miss

2) `mov $0xFEED, (F)`
Write Hit!



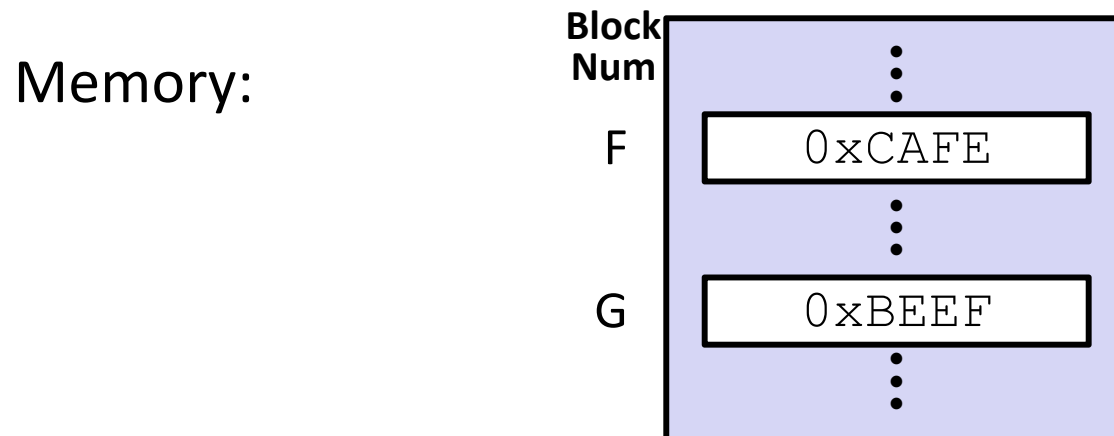
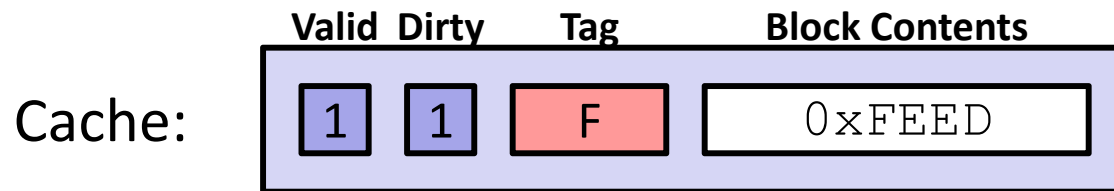
Memory:



Step: Write
0xFEED to cache
only (and set the
dirty bit)

Write-back, Write Allocate Example

1) `mov $0xFACE, (F)` 2) `mov $0xFEED, (F)`
 Write Miss **Write Hit**

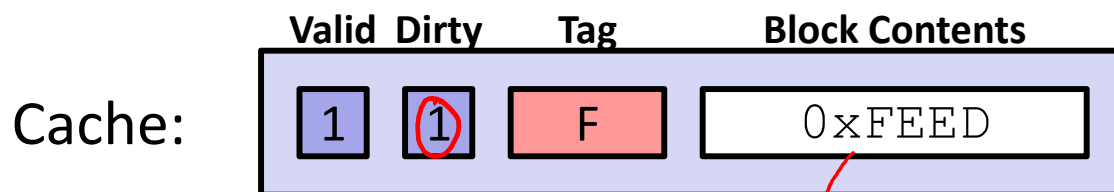


Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`
Write Miss

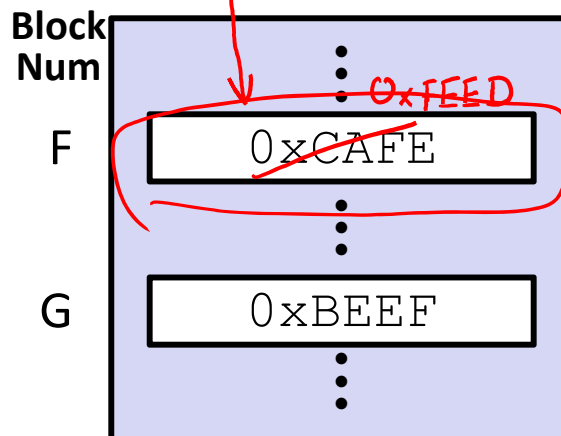
2) `mov $0xFEED, (F)`
Write Hit

3) `mov (G), %ax`
Read Miss!



0 evicted block was dirty

Memory:



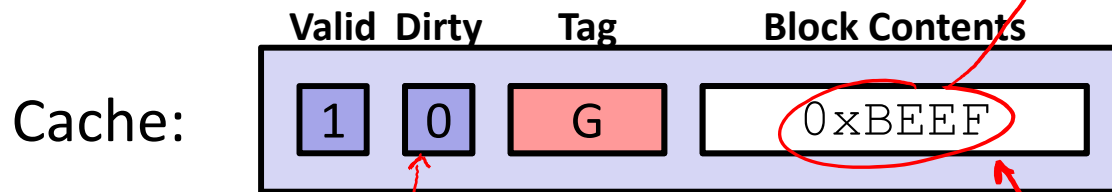
Step 1: Write F back to memory since it is dirty

Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`
Write Miss

2) `mov $0xFEEF, (F)`
Write Hit

3) `mov (G), %ax`
Read Miss



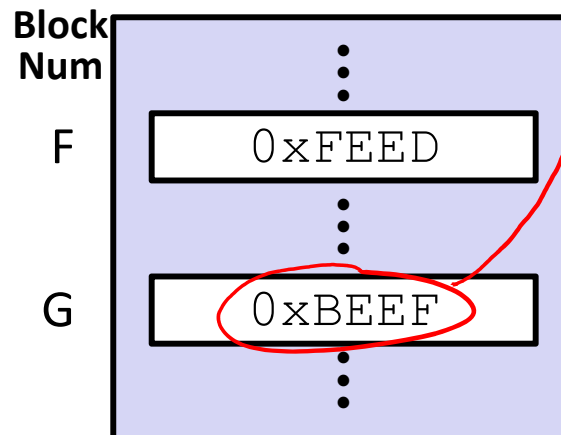
new block is consistent with memory

② load new block

③ copy into %ax

%rax

Memory:



Step 1: Write **F** back to memory since it is dirty

Step 2: Bring **G** into the cache so that we can copy it into `%ax`

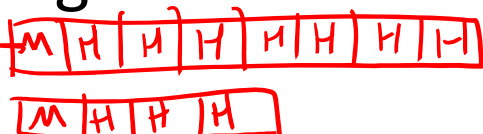
Cache Simulator

- ❖ Want to play around with cache parameters and policies? Check out our cache simulator!
 - <https://courses.cs.washington.edu/courses/cse351/cachesim/>
- ❖ Way to use:
 - Take advantage of “explain mode” and navigable history to test your own hypotheses and answer your own questions
 - Self-guided Cache Sim Demo posted along with Section 7
 - Will be used in hw19 – Lab 4 Preparation

Polling Question

❖ Which of the following cache statements is **FALSE?**

▪ Vote in Ed Lessons



False **A.**

A. We can reduce compulsory misses by decreasing our block size

*smaller block size pulls fewer bytes into \$
on a miss*

B. We can reduce conflict misses by increasing associativity

*more options to place blocks before
evictions occur*

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*

D. A write-through cache will always match data with the memory hierarchy level below it

*yes, its main
goal is data
consistency*

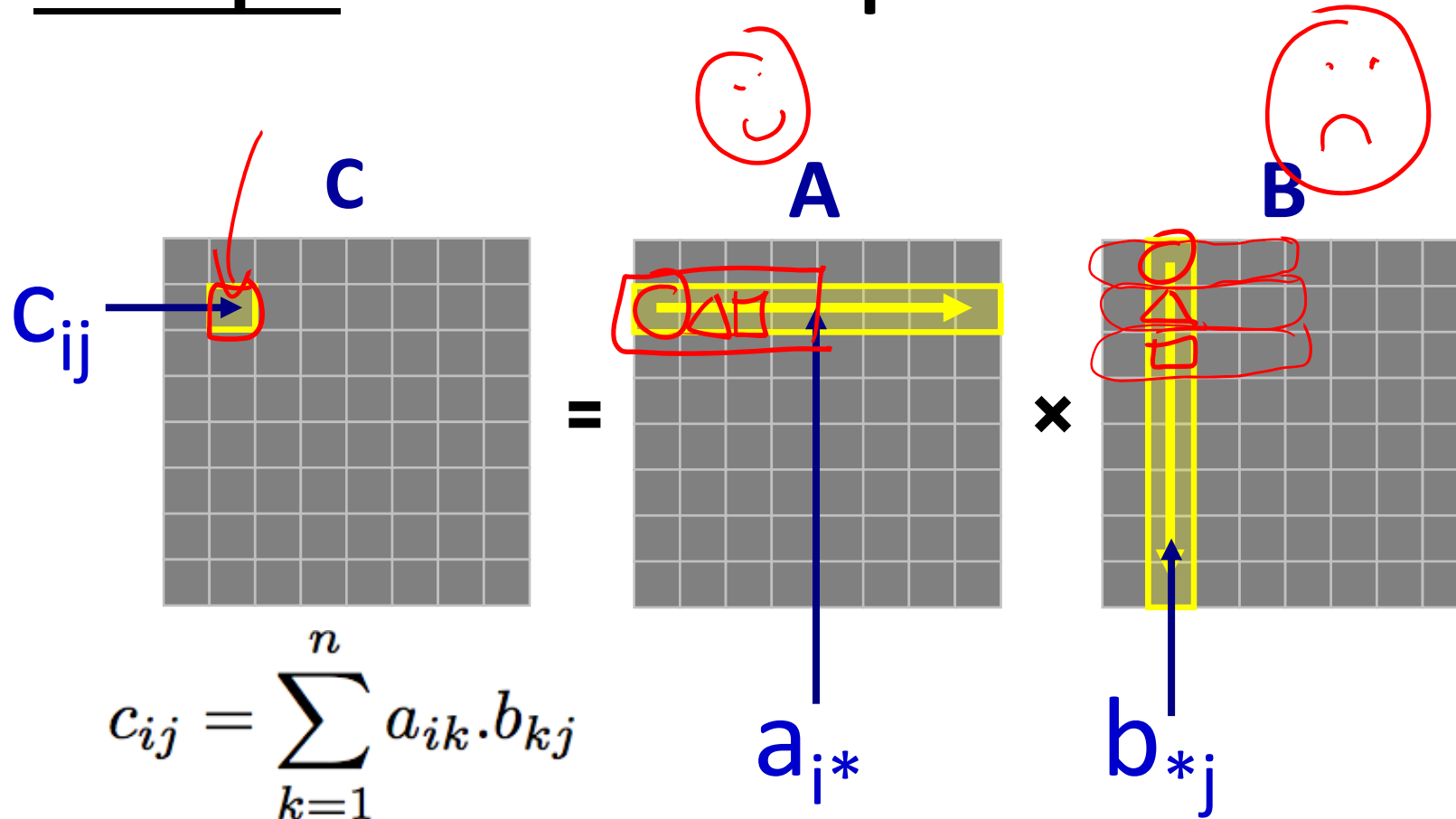
E. We're lost...

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

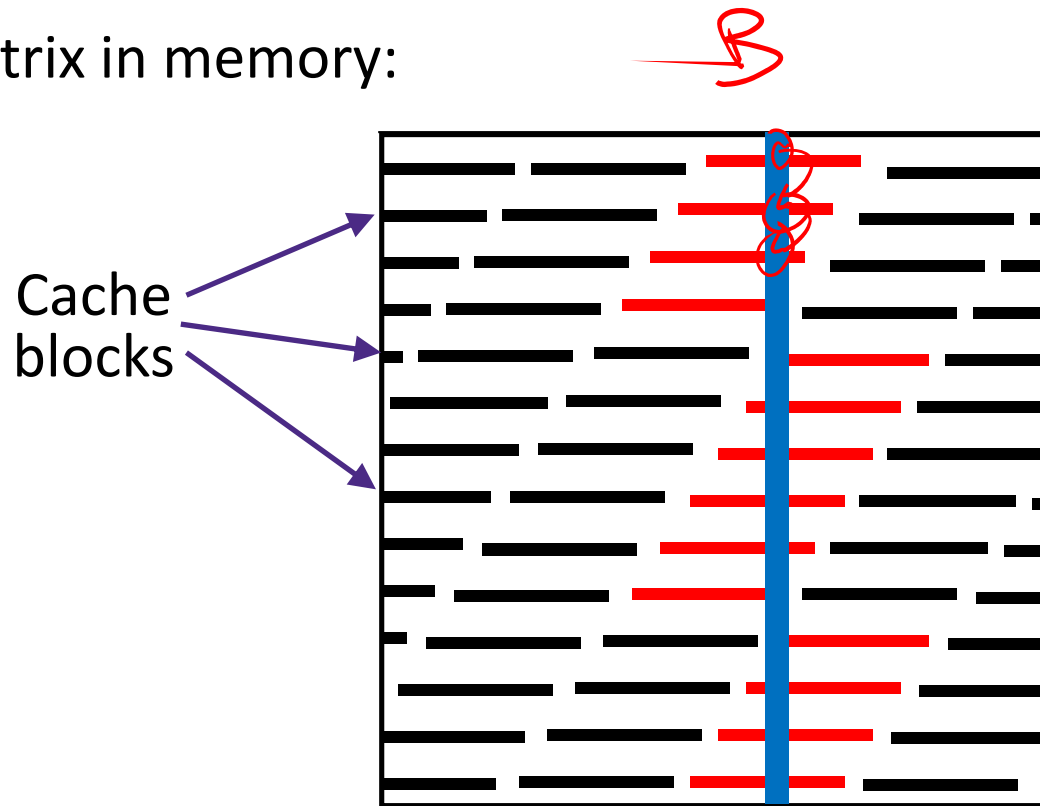
- ❖ How can you achieve locality?
 - Adjust memory accesses in *code* (software) to improve miss rate (MR)
 - Requires knowledge of *both* how caches work as well as your system's parameters
 - Proper choice of algorithm
 - Loop transformations

Example: Matrix Multiplication



Matrices in Memory

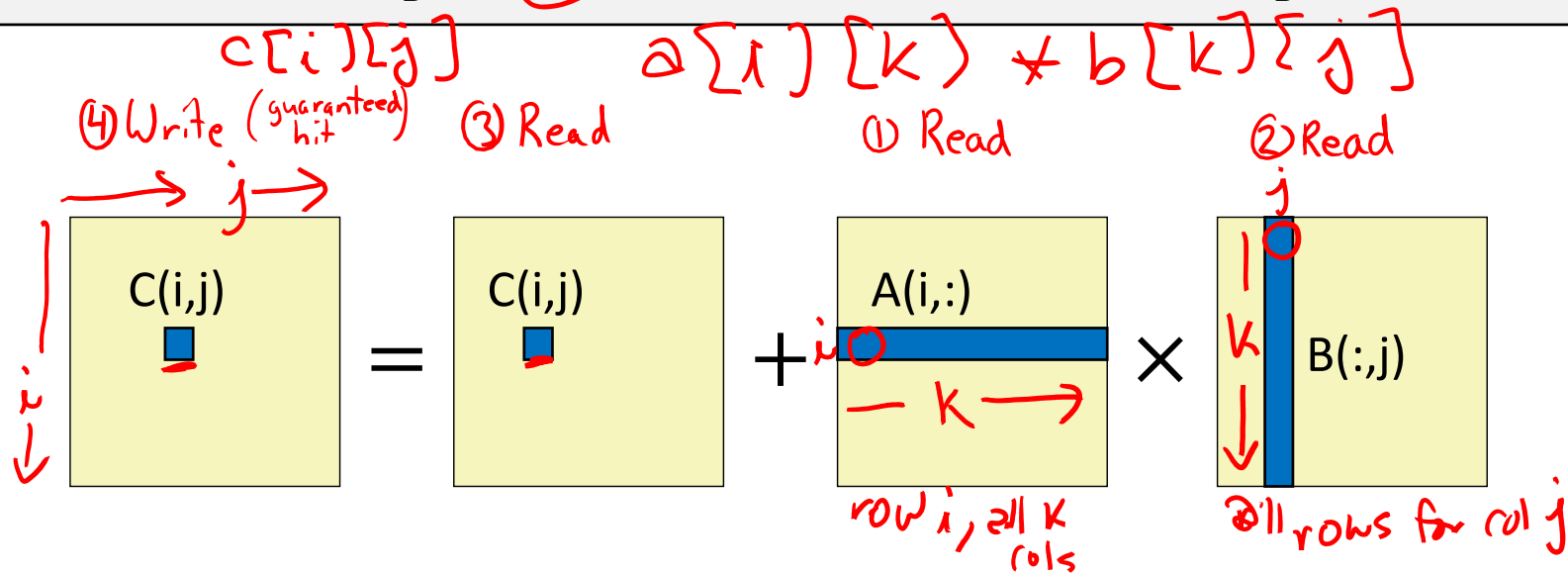
- ❖ How do cache blocks fit into this scheme?
 - Row major matrix in memory:



COLUMN of matrix (blue) is spread among cache blocks shown in red

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];
```



Cache Miss Analysis (Naïve)

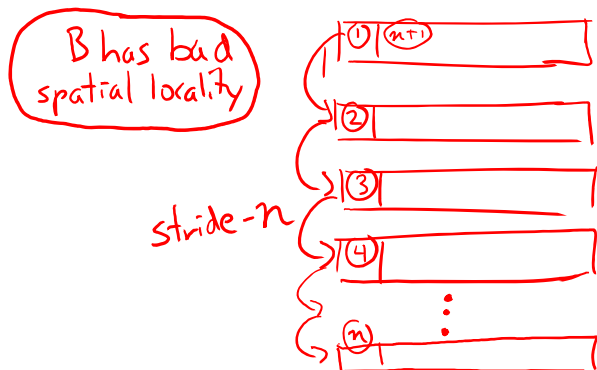
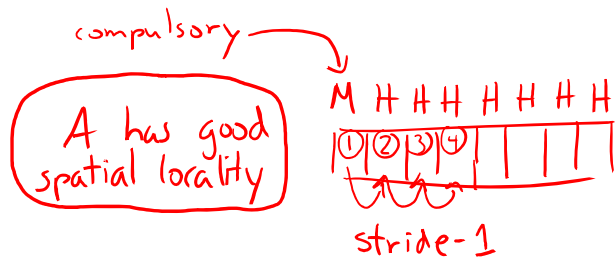
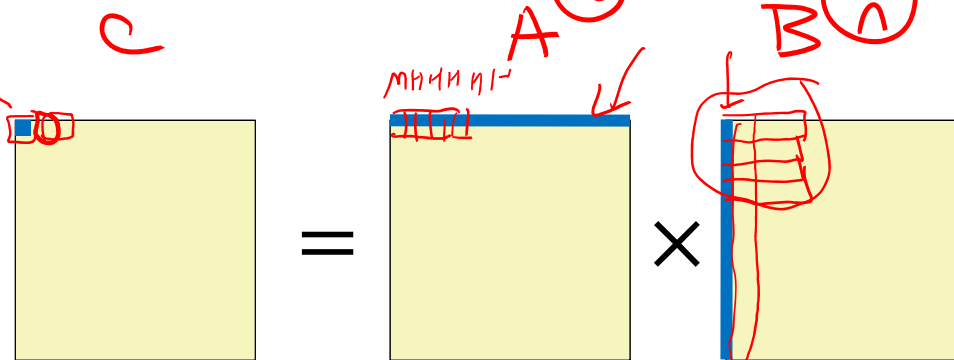
Ignoring matrix C

❖ 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:

$\frac{n}{8} + \underline{n} = \frac{9n}{8}$ misses



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

Cache Miss Analysis (Naïve)

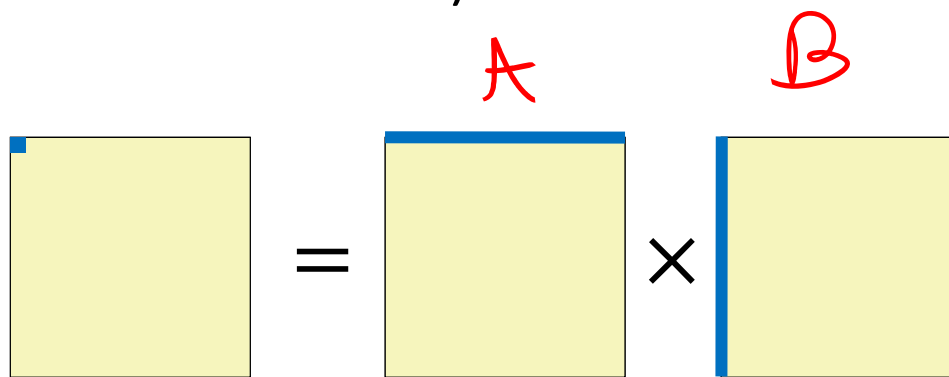
Ignoring matrix C

❖ 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)

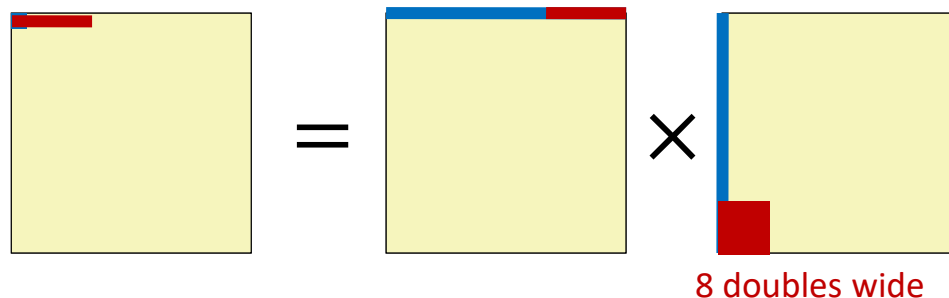
❖ Each iteration:

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



- Afterwards **in cache:** (schematic)

red showing blocks remaining in the \$



Cache Miss Analysis (Naïve)

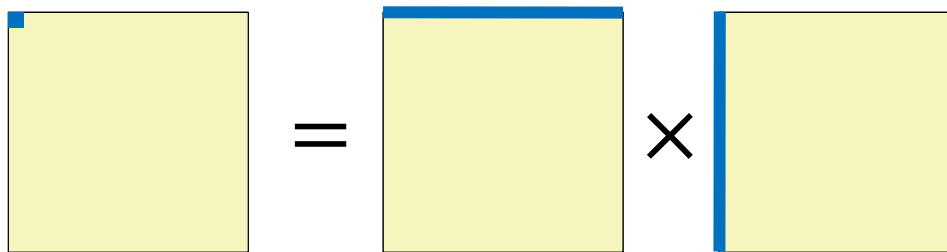
Ignoring
matrix C

❖ Scenario Parameters:

- Square matrix ($n \times n$), elements are doubles
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- 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 product matrix element

Linear Algebra to the Rescue (1)

This is extra
(non-testable)
material

- ❖ 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:

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

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

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}

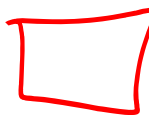
B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

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

blocks r
 r 

- ❖ Blocked version of the naïve algorithm:

```
# move by r x r 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];
```

loop over block matrices

loop within block matrices

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

Cache Miss Analysis (Blocked)

Ignoring matrix C

❖ Scenario Parameters:

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

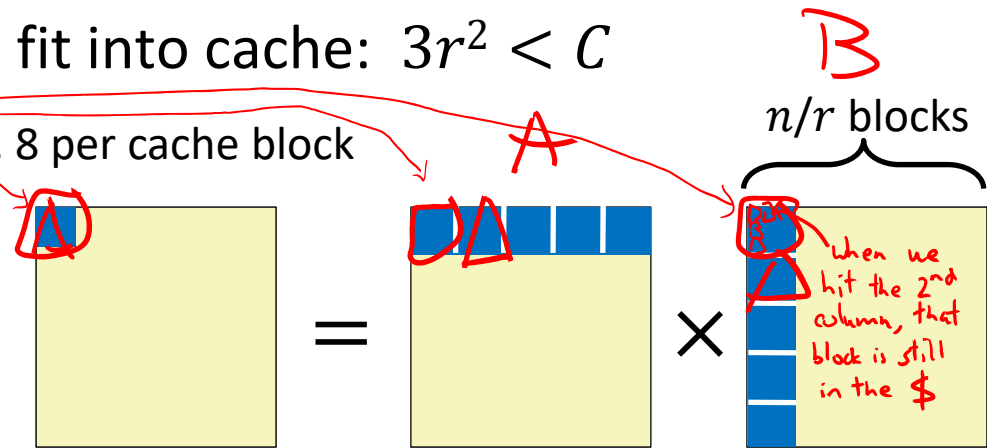
r^2 elements per block, 8 per cache block

❖ Each block iteration:

- $\frac{r^2}{8}$ misses per block

- $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$

n/r blocks in row and column



Cache Miss Analysis (Blocked)

Ignoring matrix C

❖ Scenario Parameters:

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

❖ Each block iteration:

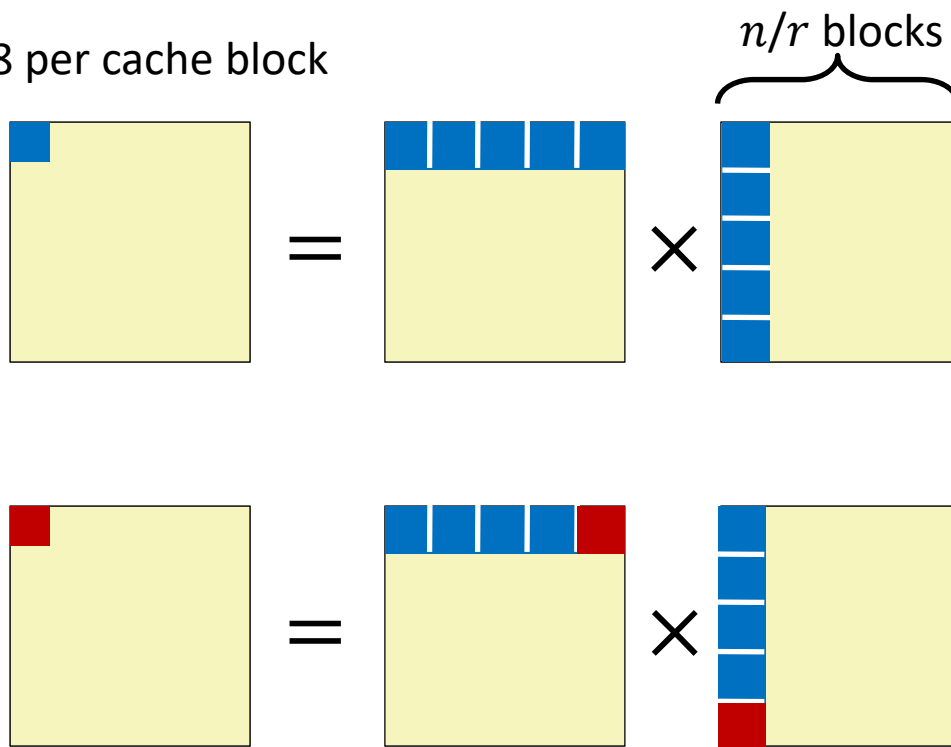
- $\frac{r^2}{8}$ misses per block

- $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$

r^2 elements per block, 8 per cache block

n/r blocks in row and column

- Afterwards in cache (schematic)



Cache Miss Analysis (Blocked)

Ignoring matrix C

❖ Scenario Parameters:

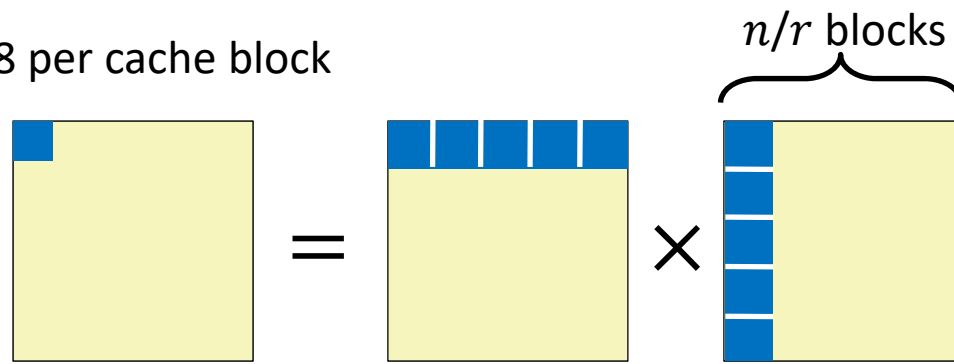
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ Each block iteration:

- $\frac{r^2}{8}$ misses per block

- $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$

n/r blocks in row and column



❖ Total misses:

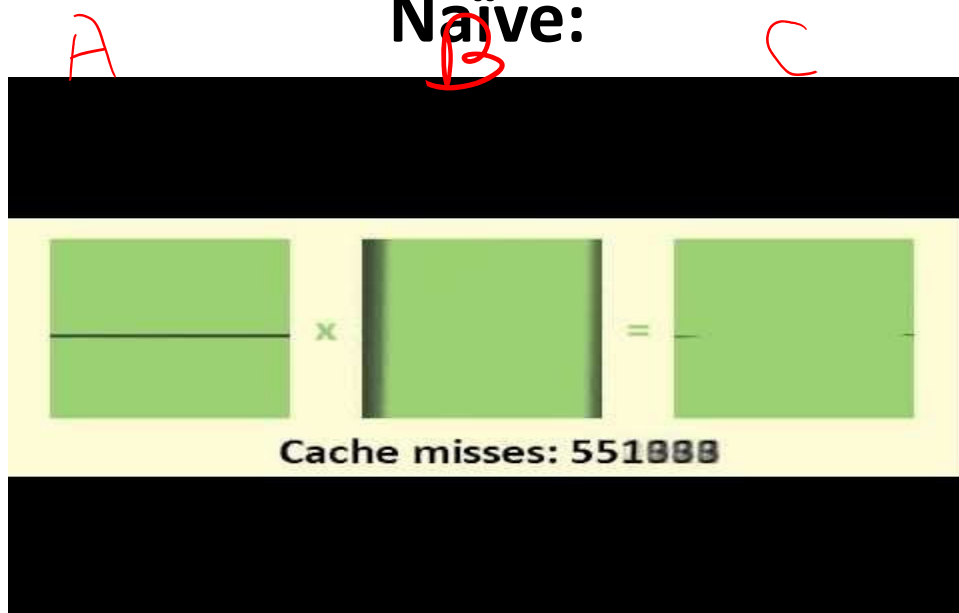
- $\left(\frac{nr}{4}\right) \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{4r}$ vs. $9n^3/8$

Matrix Multiply Visualization

❖ Here $n = 100$, $C = 32$ KiB, $r = 30$

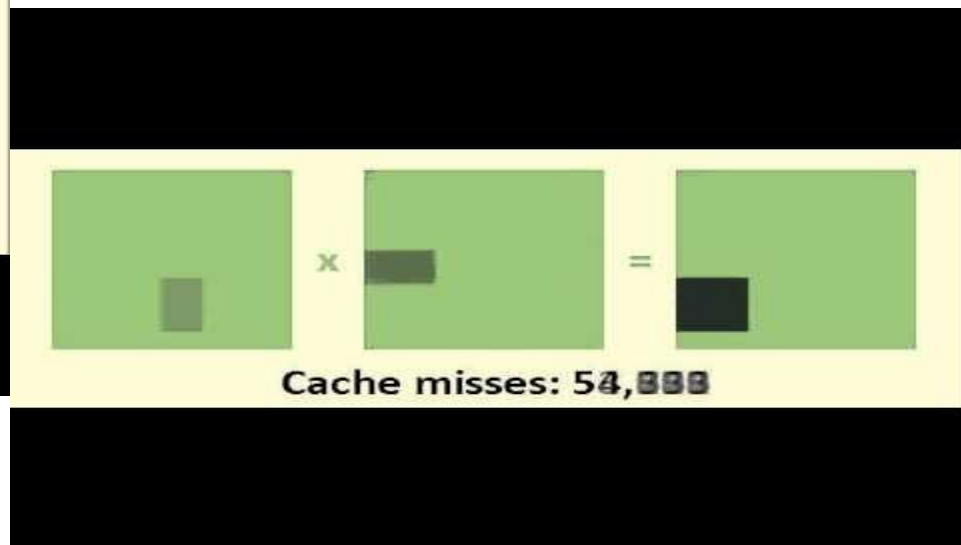
shaded areas show blocks stored in the cache

Naïve:



≈ 1,020,000
cache misses

Blocked:



≈ 90,000
cache misses

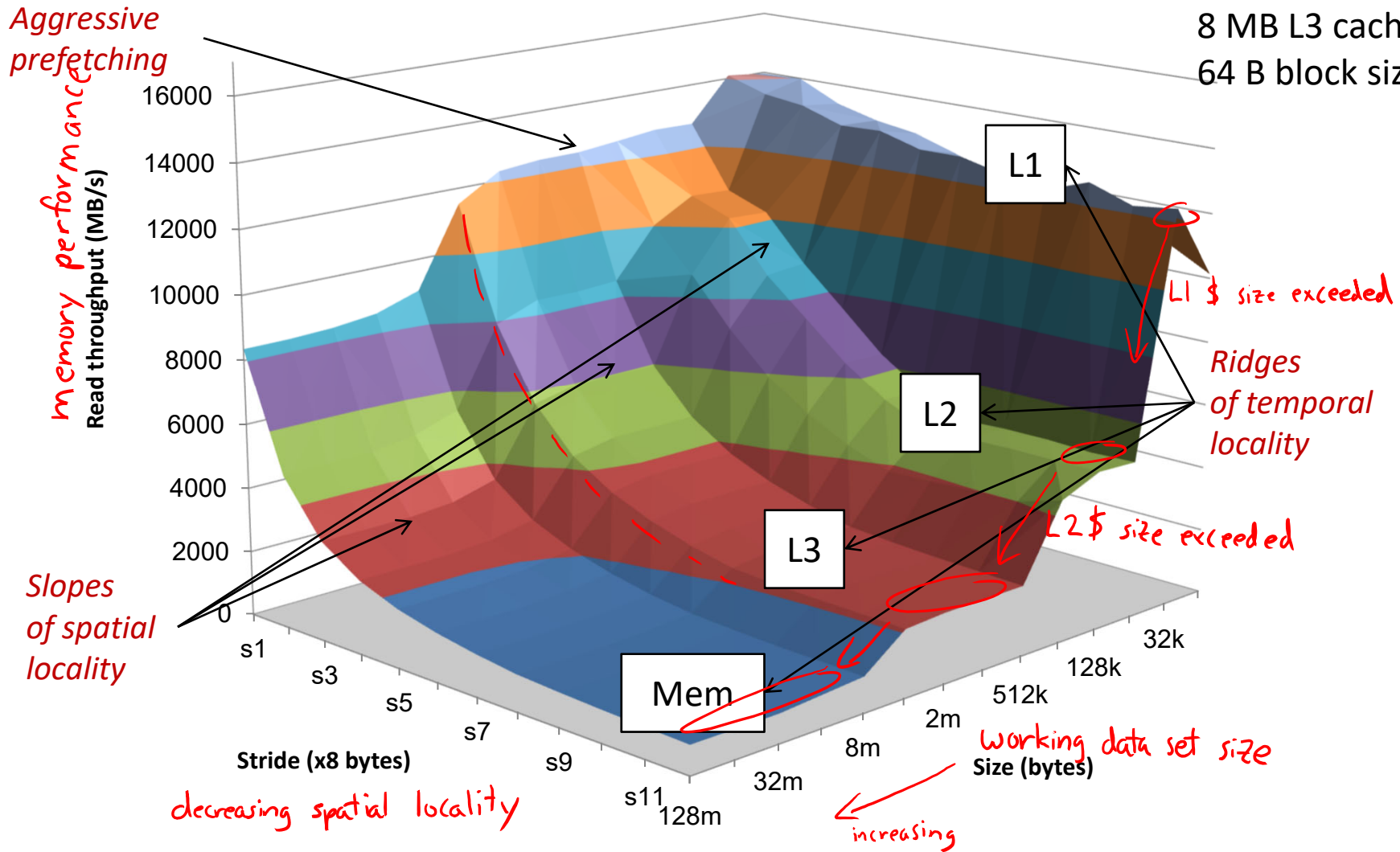
Cache-Friendly Code

- ❖ 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/`
 - Example: `cat /sys/devices/system/cpu/cpu0/cache/index*/size`

❖ Windows:

- `wmic memcache get <query>` (all values in KB)
- Example: `wmic memcache get MaxCacheSize`

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