

Caches IV

CSE 351 Summer 2020

Instructor:

Porter Jones

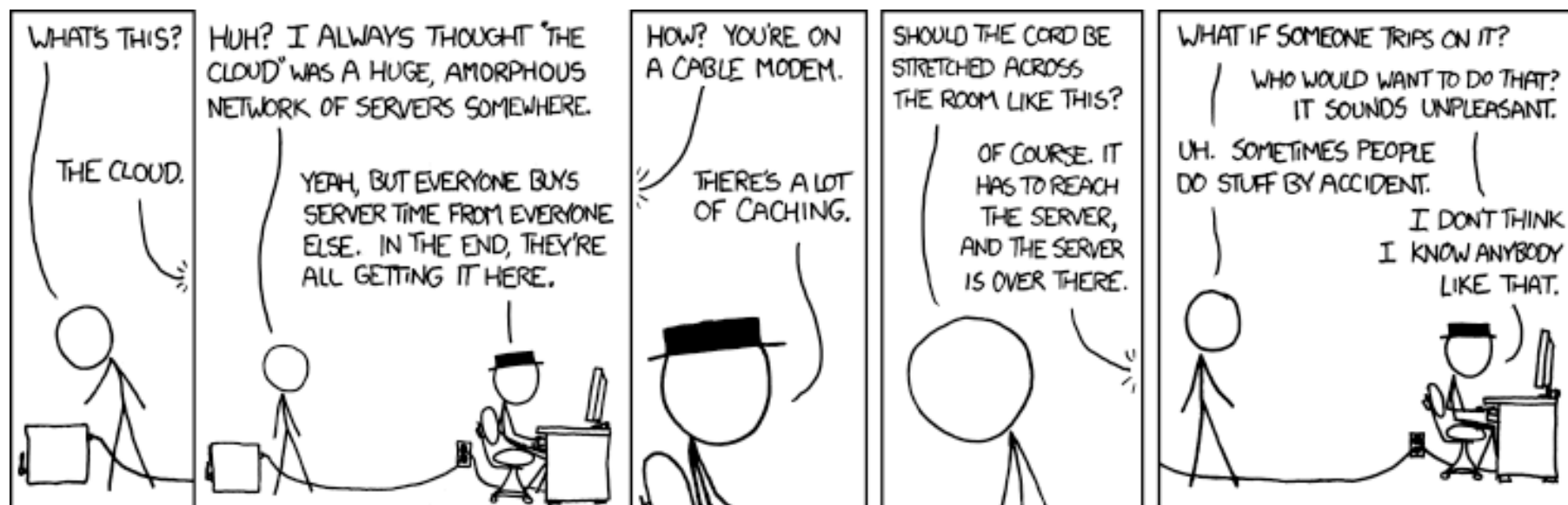
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk




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

Administrivia

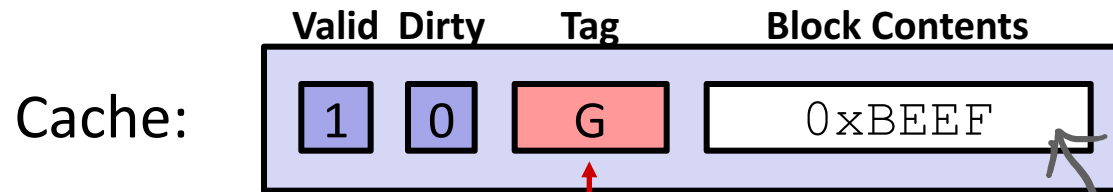
- ❖ Questions doc: <https://tinyurl.com/CSE351-8-3>
- ❖ hw16 due Wednesday (8/5) – 10:30am
- ❖ hw17 due Friday (8/7) – 10:30am
- ❖ Lab 4 due Wednesday (8/12) – 11:59pm
 - All about caches!
- ❖ Unit Summary 2 Due Wednesday (8/5) – 11:59pm

What about writes?

- ❖ Multiple copies of data may exist:
 - multiple levels of cache and main memory
- ❖ What to do on a write-hit? *(block/data already in cache)*
 - **Write-through:** 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 for write-back cache}
- ❖ What to do on a write-miss? *(block/data not already in cache)*
 - **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 
 - Write-through + No-write allocate, occasionally

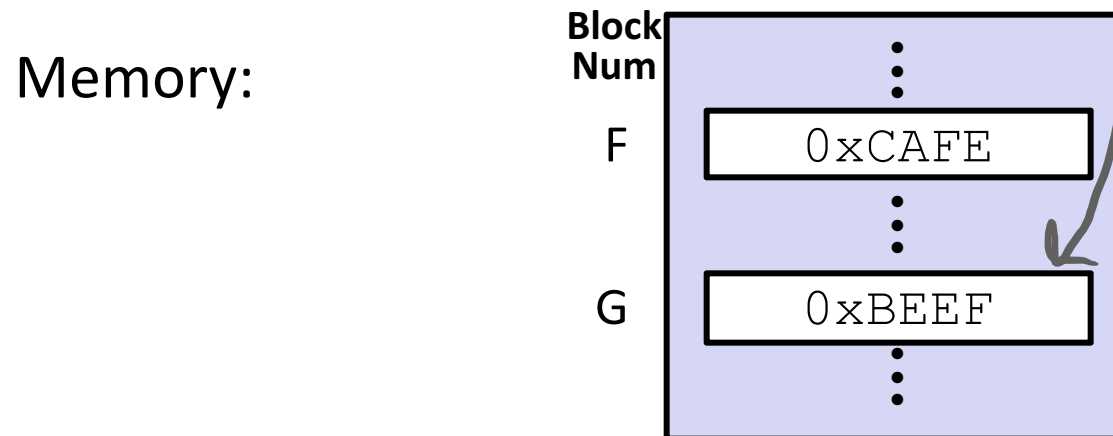
^{hit}Write-back, Write Allocate Example

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!

Not dirty so copies are consistent

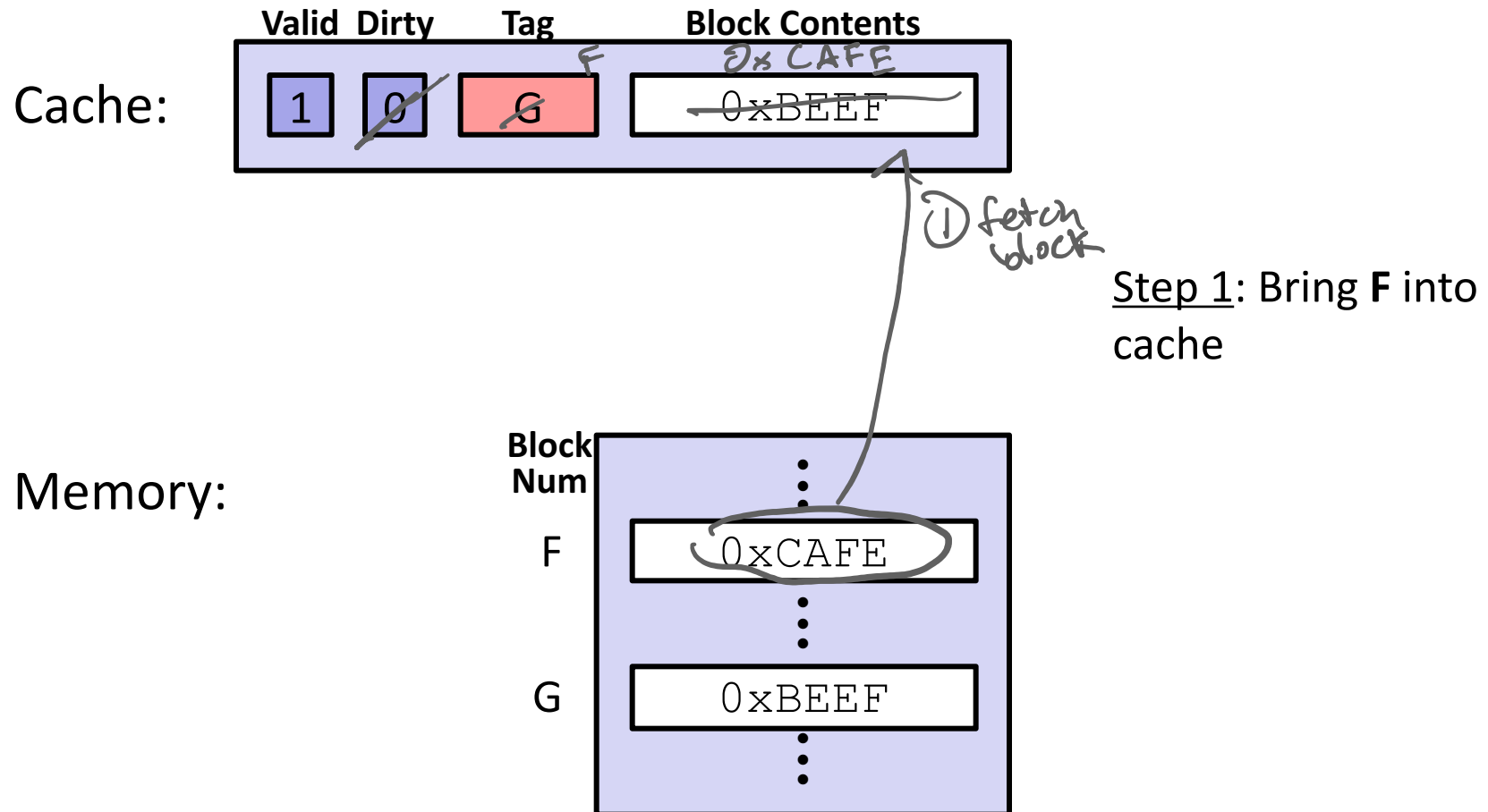


Write-back, Write Allocate Example

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

Write Miss!

Not valid x86, just using block num instead
of full byte address to keep the example simple

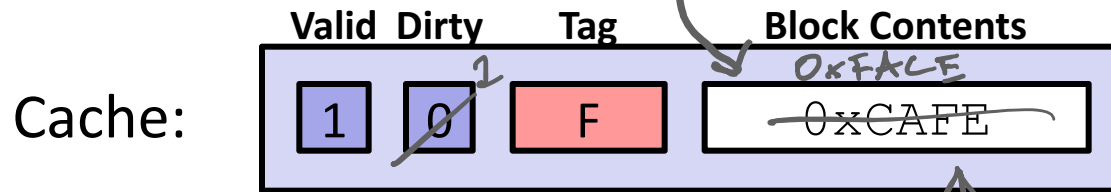


Write-back, Write Allocate Example

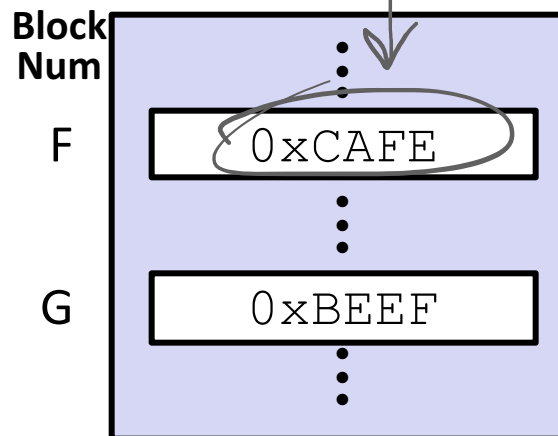
1) `mov $0xFACE, (F)`

Write Miss

② write data into cache



Memory:



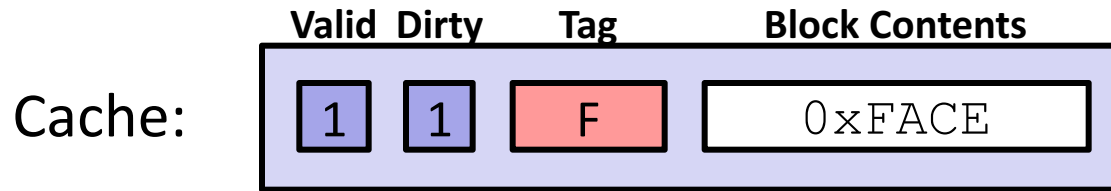
different!

Step 1: Bring F into cache

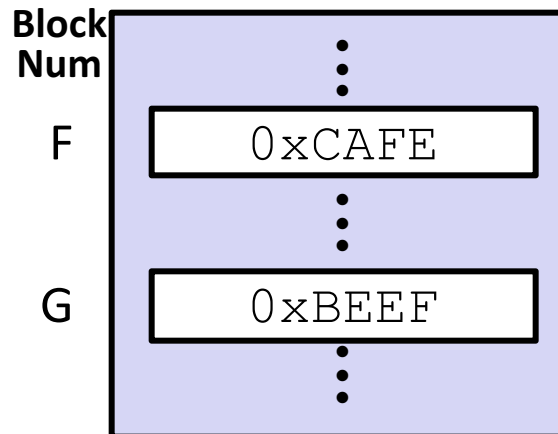
Step 2: Write 0xFACE to cache only and set the dirty bit

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**

Hit

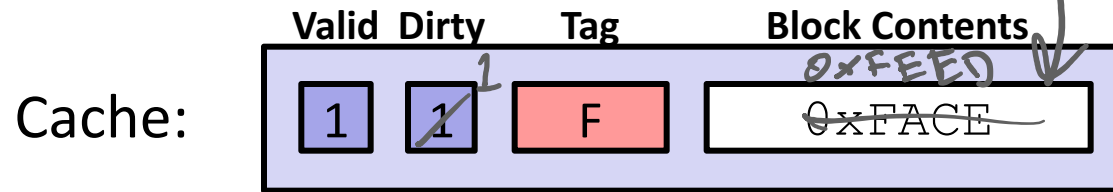
Miss

Write-back, Write Allocate Example

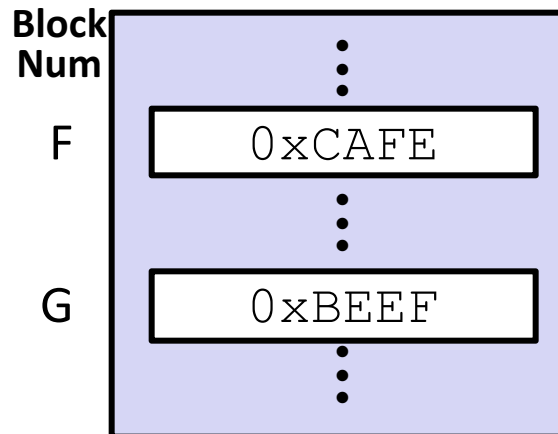
1) `mov $0xFACE, (F)`
Write Miss

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

write
data
into
block



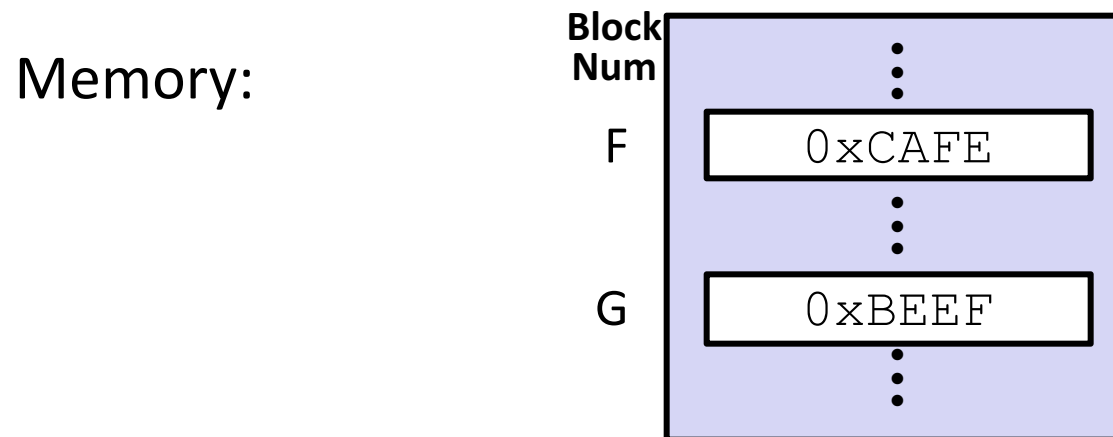
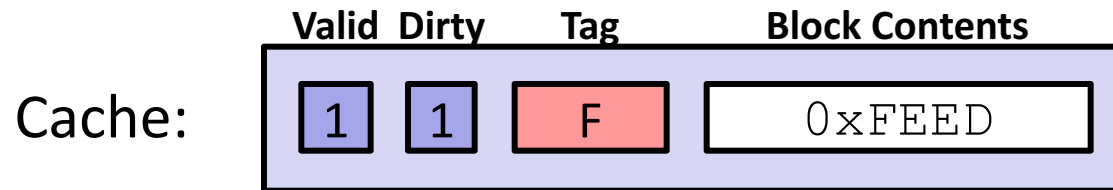
Memory:



Step: Write
0xFEEF 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**

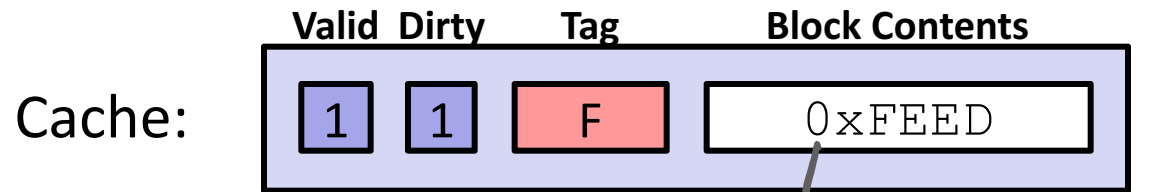


Write-back, Write Allocate Example

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

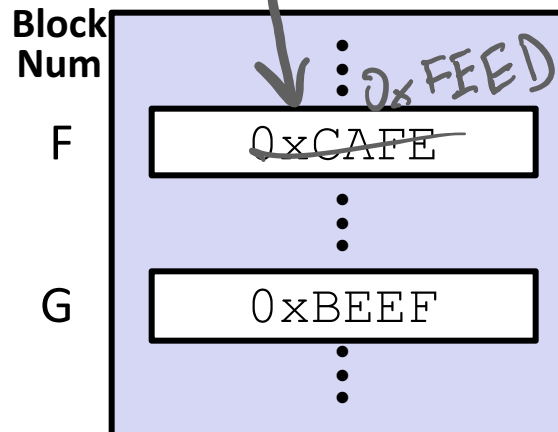
2) `mov $0xFEED, (F)`
Write Hit

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



*Dirty block is dirty
so update
memory*

Memory:



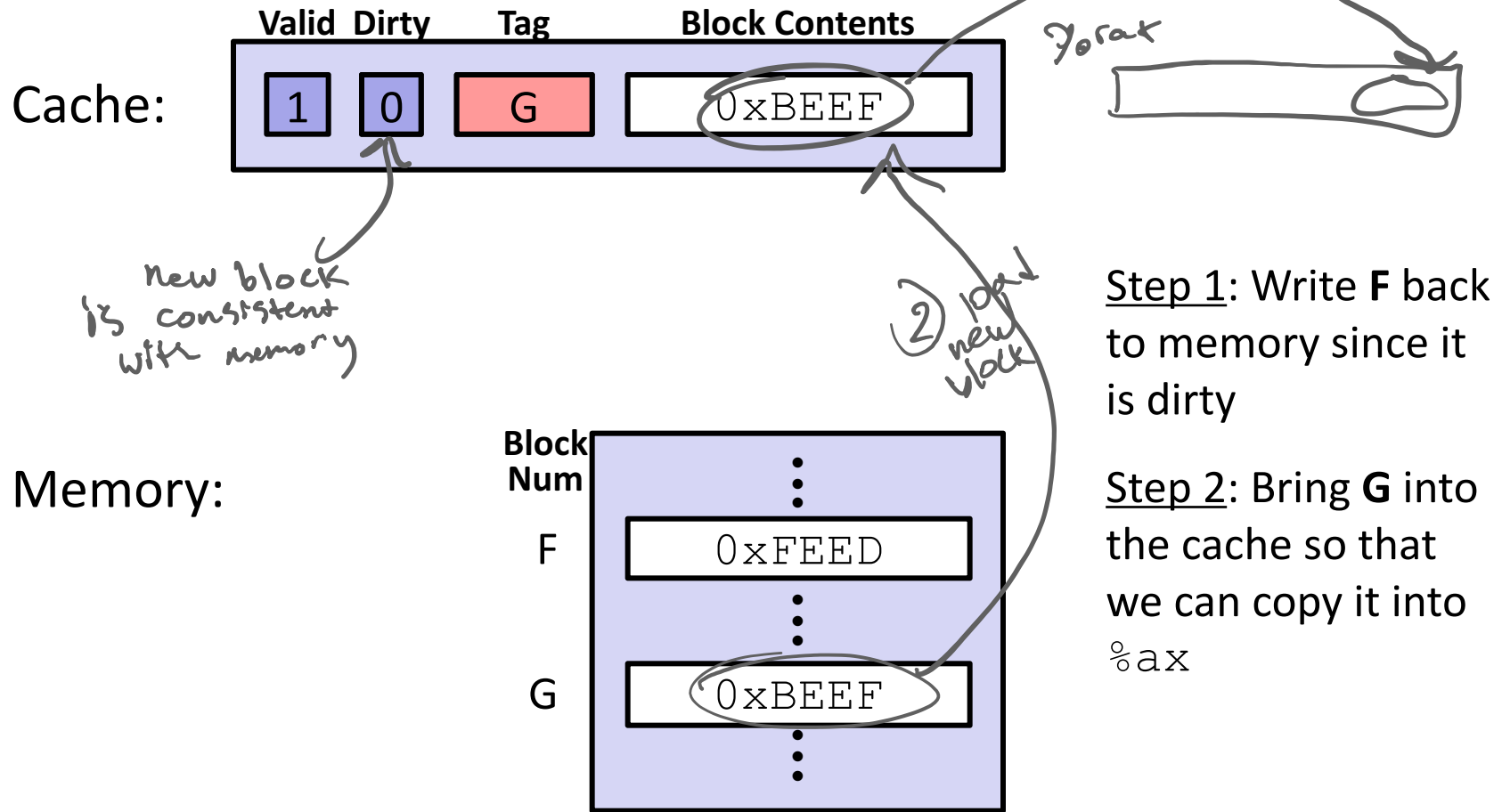
Step 1: Write **F** back
to memory since it
is dirty

Write-back, Write Allocate Example

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

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

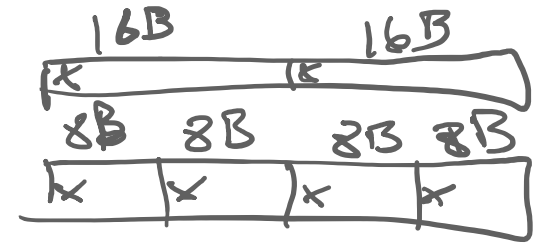
3) `mov (G), %ax`
Read Miss



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 6
 - Will be used in hw17 – Lab 4 Preparation

Polling Question [Cache IV]



❖ Which of the following cache statements is FALSE?

■ Vote at <http://pollev.com/pbjones>

A. We can reduce compulsory misses by decreasing our block size *Smaller block size pulls fewer bytes into cache on miss*

B. We can reduce conflict misses by increasing associativity *True! More options to place blocks before eviction occurs*

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

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

E. We're lost...

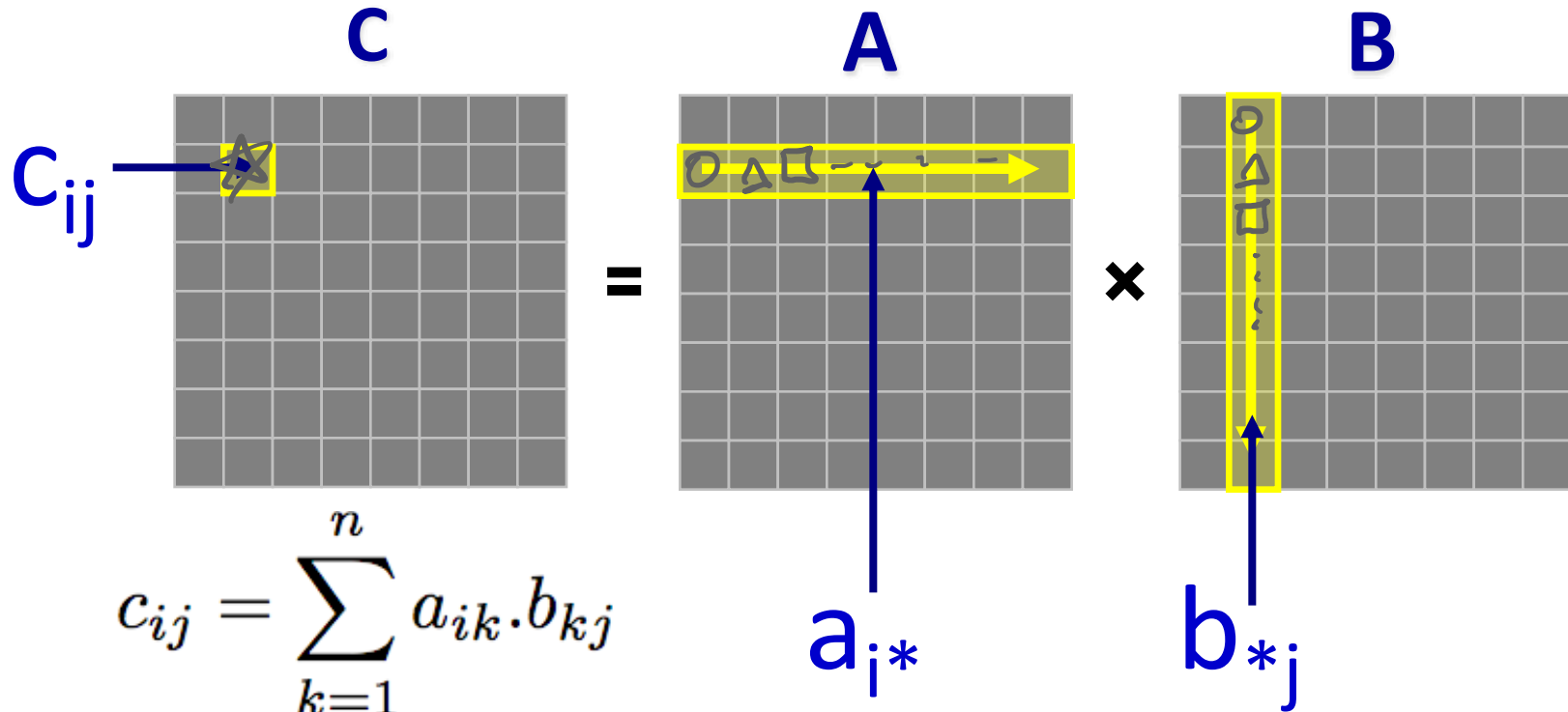
Optimizations for the Memory Hierarchy

- ❖ 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

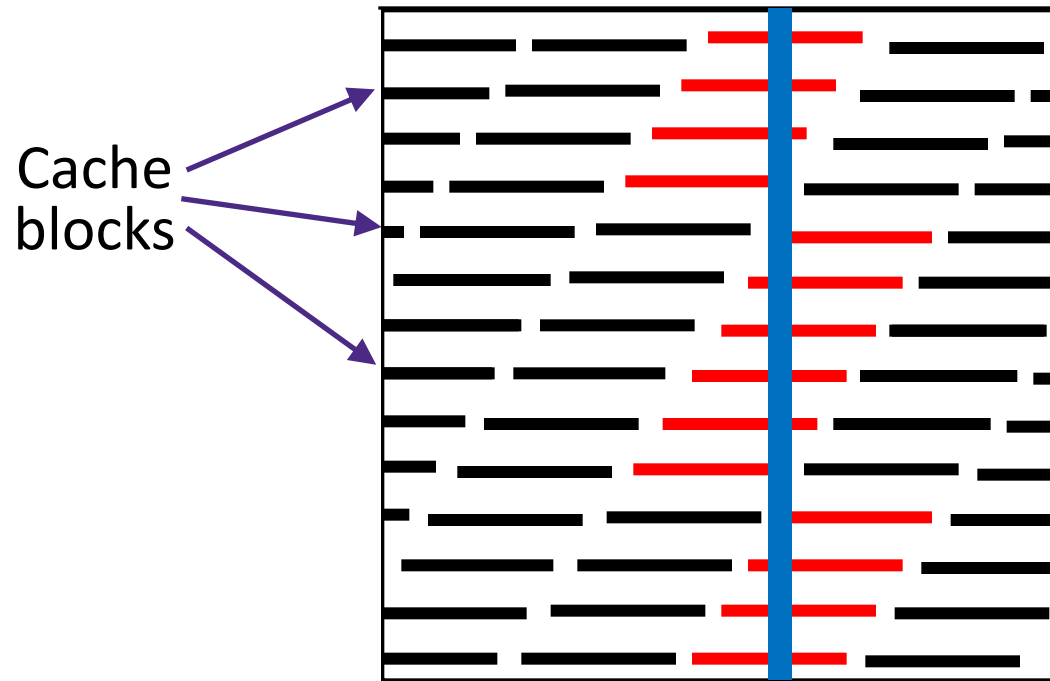
Example: Matrix Multiplication

$$\star = 0 \times 0 + \Delta \times \Delta + \square \times \square \dots$$



Matrices in Memory

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

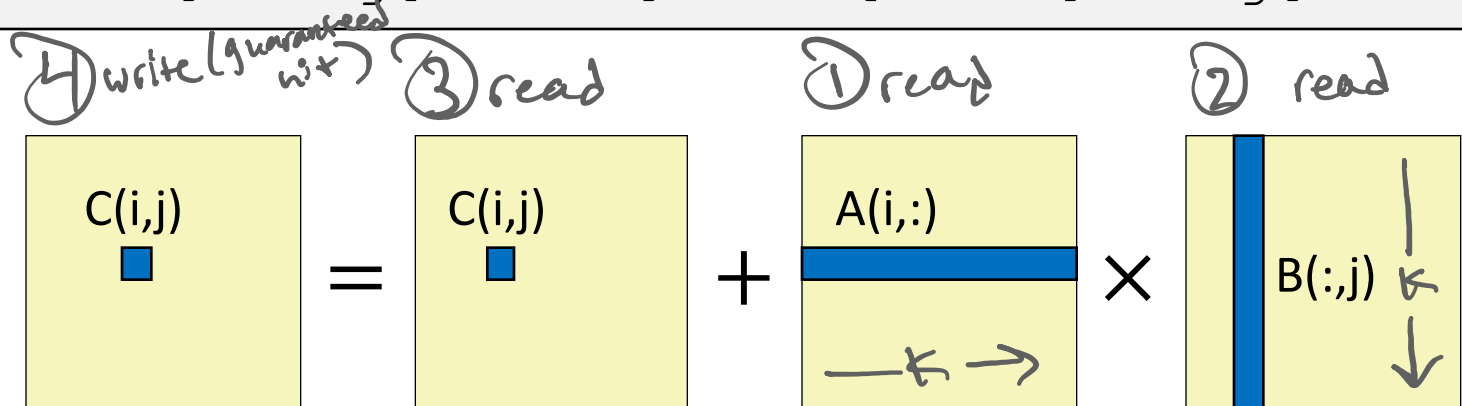


COLUMN of matrix (blue) is spread
among cache blocks shown in red

Naïve Matrix Multiply *matrix $n \times n$*

```

# 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

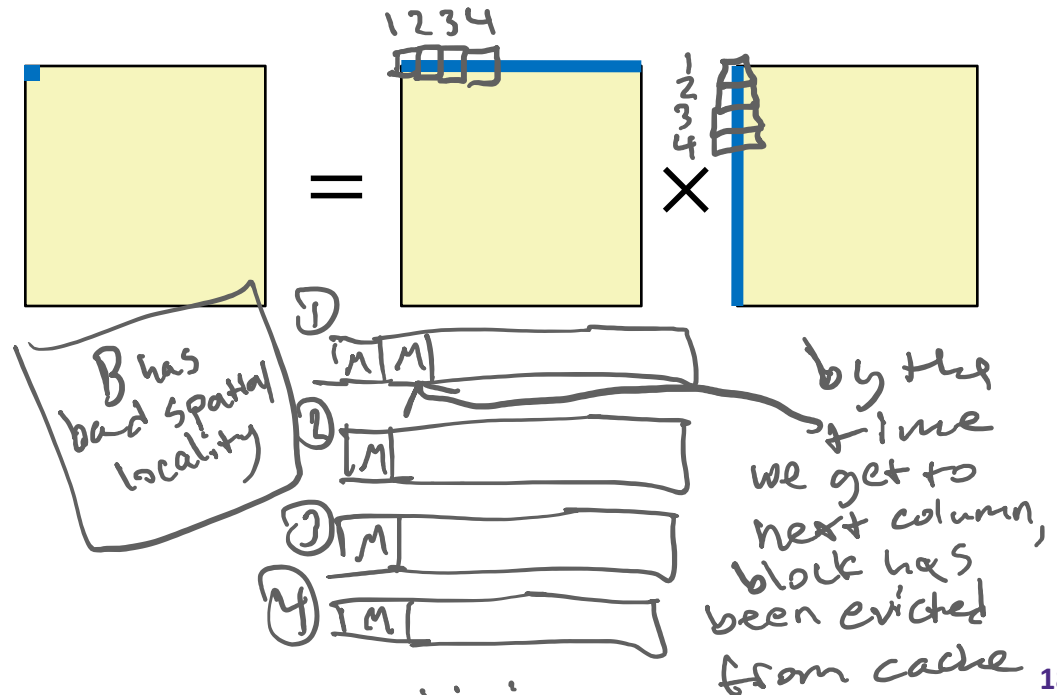
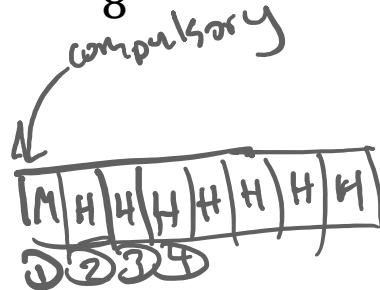
❖ Scenario Parameters:

- Square matrix ($n \times n$), elements are doubles *8 elements per cache block*
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- ❖ Cache size $C \ll n$ (much smaller than n)
key assumption!

❖ Each iteration:

$$\text{■ } \frac{n}{8} + n = \frac{9n}{8} \text{ misses}$$

A
Has good
spatial
locality!



Cache Miss Analysis (Naïve)

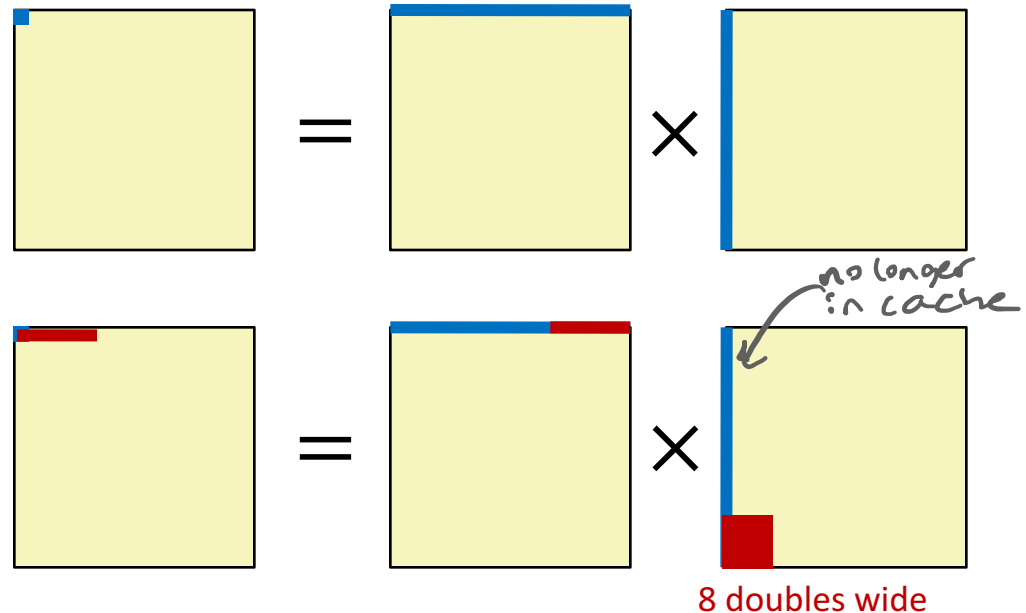
Ignoring
matrix C

❖ 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



Cache Miss Analysis (Naïve)

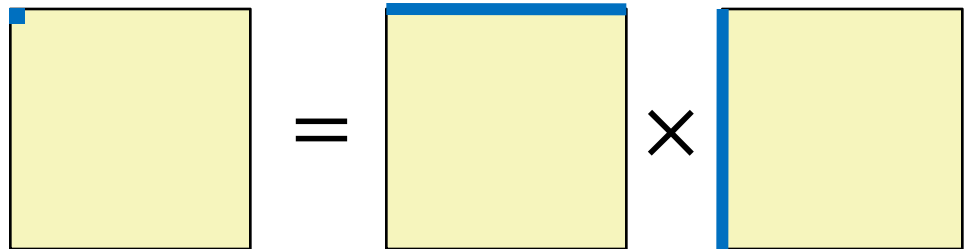
Ignoring
matrix C

❖ 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:

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

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

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_{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 is actually much faster due to access pattern!

❖ 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 (jib = j; jib < j+r; jib++)
                    for (kib = k; kib < k+r; kib++)
                        c[ib*n+jib] += a[ib*n+kib]*b[kib*n+jib];
```

loop over block matrices

choose block

loop within block matrices

work for one block

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

Cache Miss Analysis (Blocked)

Ignoring
matrix C

❖ Scenario Parameters:

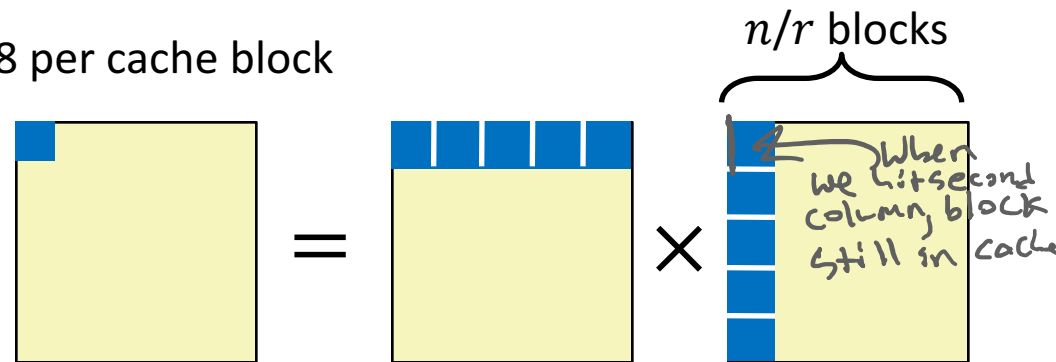
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ Each block iteration:

- $r^2/8$ 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:

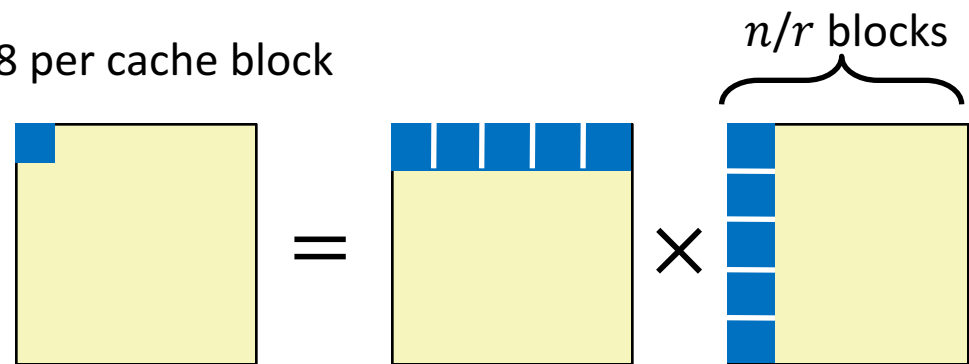
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ Each block iteration:

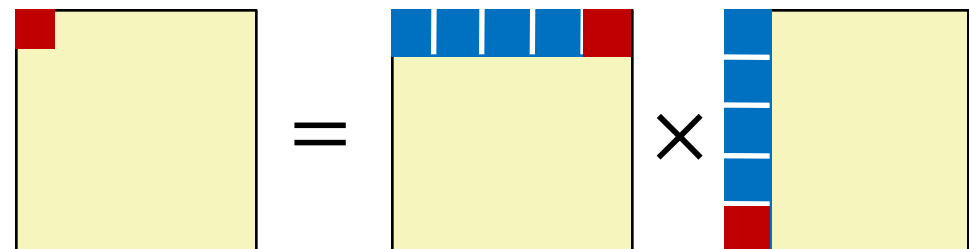
- $r^2/8$ misses per block

$$2n/r \times r^2/8 = nr/4$$

n/r blocks in row and column



- Afterwards in cache (schematic)



Cache Miss Analysis (Blocked)

Ignoring
matrix C

❖ Scenario Parameters:

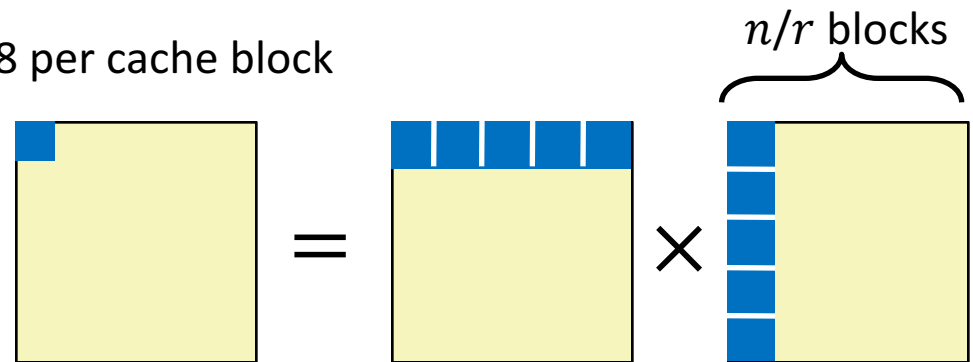
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ Each block iteration:

- $r^2/8$ misses per block

$$2n/r \times r^2/8 = nr/4$$

n/r blocks in row and column



❖ Total misses:

$$nr/4 \times (n/r)2 = \mathbf{n^3/(4r)}$$

before: $9/8 n^3$

Matrix Multiply Visualization

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

Naïve:



Cache misses: 551,000

$\approx 1,020,000$
cache misses

Blocked:



Cache misses: 54,000

$\approx 90,000$
cache misses

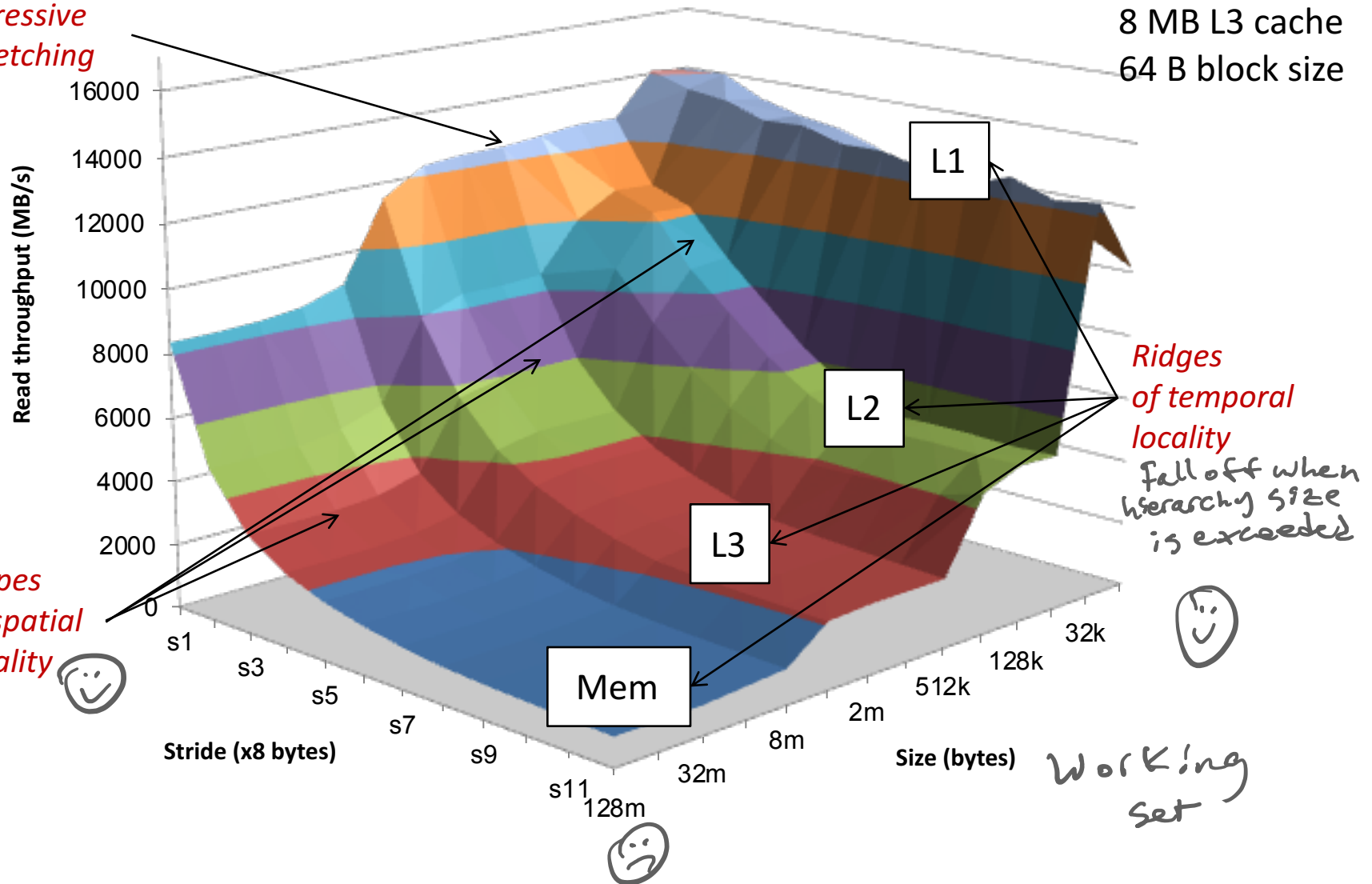
Cache-Friendly Code

- ❖ 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

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

*Aggressive
prefetching*



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`

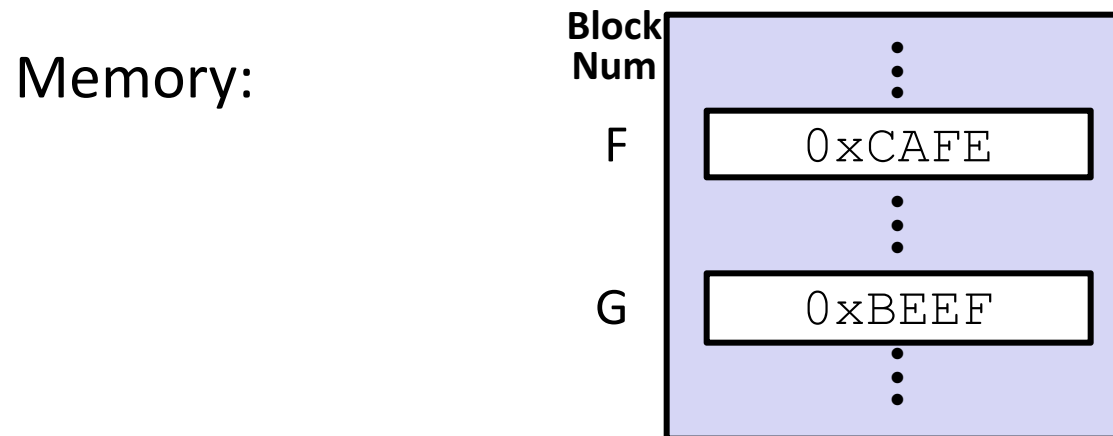
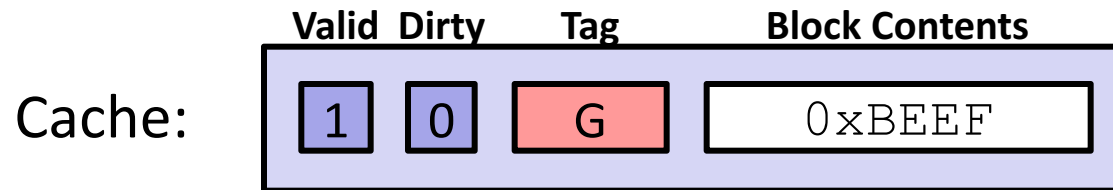
- ❖ Modern processor specs: <http://www.7-cpu.com/>

Write-back, Write Allocate Example

1) `mov 0xFACE, F`

2) `mov 0xFEEB, F`

3) `mov G, %ax`



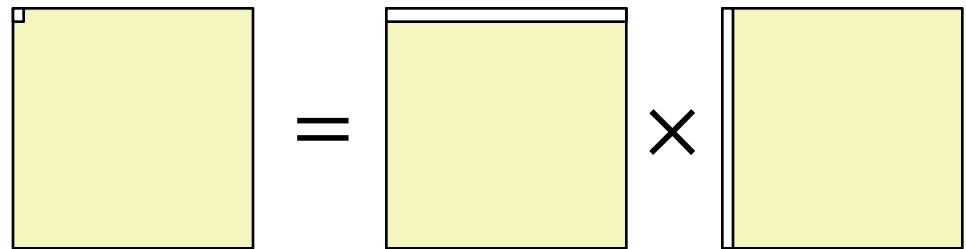
Cache Miss Analysis Comparison

Ignoring
matrix C

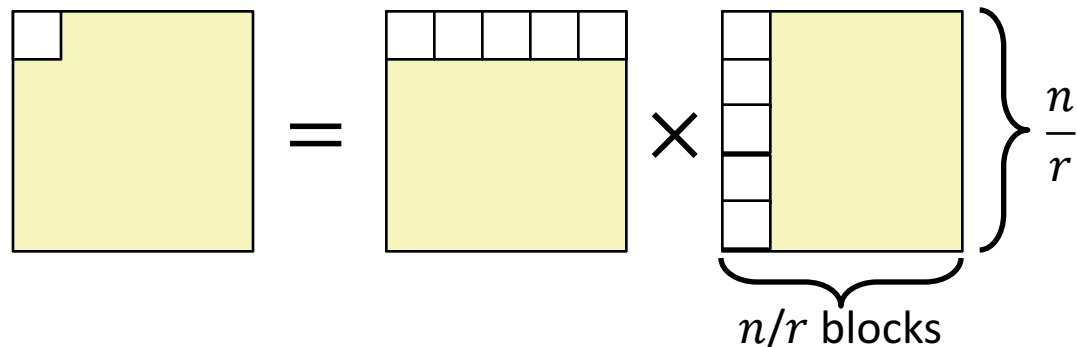
❖ Scenario Parameters:

- Square matrix ($n \times n$) of doubles
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ and three blocks ($r \times r$) fit into cache: $3r^2 < C$

❖ Naïve:

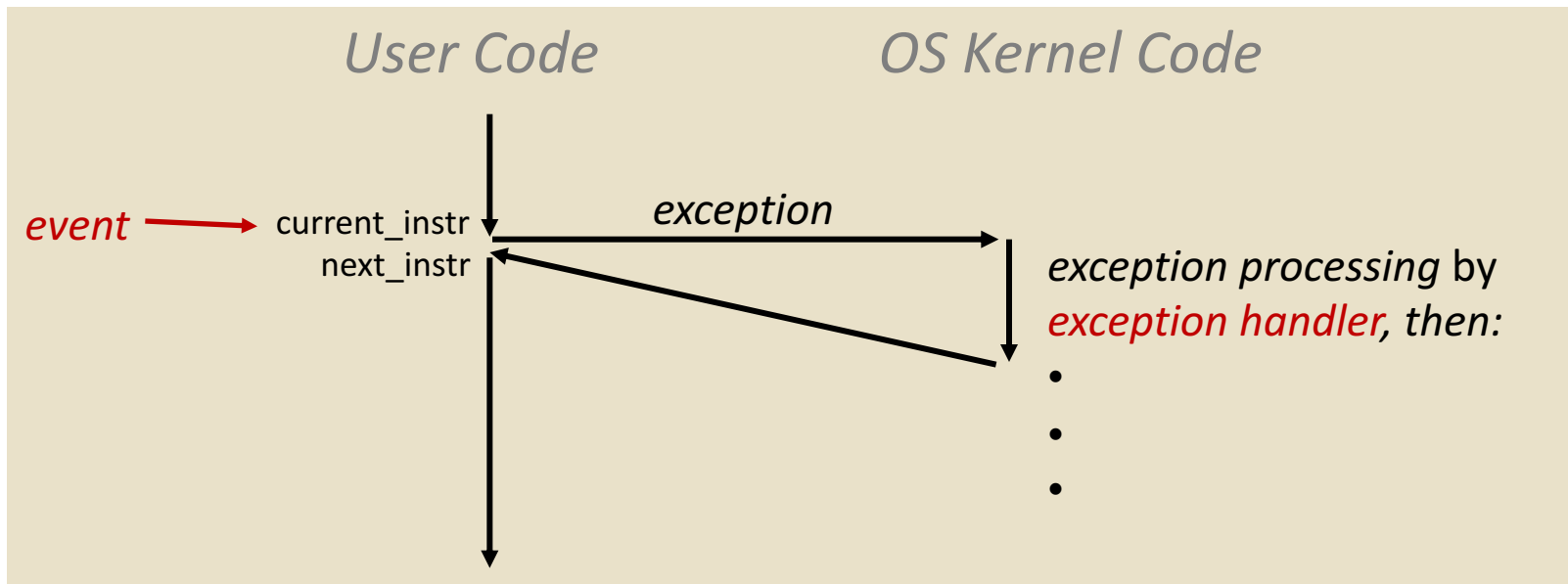


❖ Blocked:



Exceptions - Handout

- ❖ An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (i.e. change in processor state)
 - Kernel is the memory-resident part of the OS
 - Examples: division by 0, page fault, I/O request completes, Ctrl-C



- ❖ *How does the system know where to jump to in the OS?*

Notes Diagrams

