

Memory & Caches IV

CSE 351 Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Anirudh Kumar

Catherine Guevara

Dara Stotland

Harrison Bay

Ian Hsiao

Kevin Wang

Mara Kirdani-Ryan

Nick Durand

Sanjana Sridhar

When you have to optimize your memory accesses for caching:



When the data you want isn't already in the cache:



The cache when you ask for something that was just evicted:



Programs with good temporal locality:



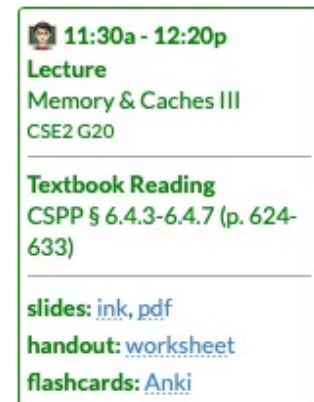
Relevant Course Information

- ❖ Lab 4 due Monday (2/28)
 - Cache parameter puzzles and code optimizations
- ❖ hw18 due Wed (2/23)
 - Lab 4 preparation
- ❖ Lab 2 grades posted
 - Regrade requests will be open for one week (until 2/21)
- ❖ Midterm grading is in progress

- ❖ Guest lecture from Mara on Friday!
 - Sam is out of town – no office hours :(

Mid-Quarter Survey

- ❖ Lecture: a little fast, more demos/examples wanted
 - We still have a live questions doc! (Tools > Lecture Questions)
 - Topic review videos (Course Info > Videos)
- ❖ Homework: nitpicky formatting is annoying 😞
 - Explanations could be better
- ❖ Flashcards: they exist!
 - <https://courses.cs.washington.edu/courses/cse351/22wi/flashcards/>
 - Posted on course calendar for each lecture
- ❖ Readings: generally a good supplement
 - Some can be too dense/confusing



Mid-Quarter Survey

- ❖ Midterm: a little too ambiguous
 - Some questions should have been more precisely worded
 - GDB practice was good, if confusing
 - Take-home format was good, 72 hours enough time
 - Design/explanation questions were challenging
- ❖ Socio-technical content: worthwhile, but imperfect
 - Needs more nuance and room for discussion
 - Can feel overly prescriptivist
 - More context would be helpful (e.g., optional readings)

Reading Review

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

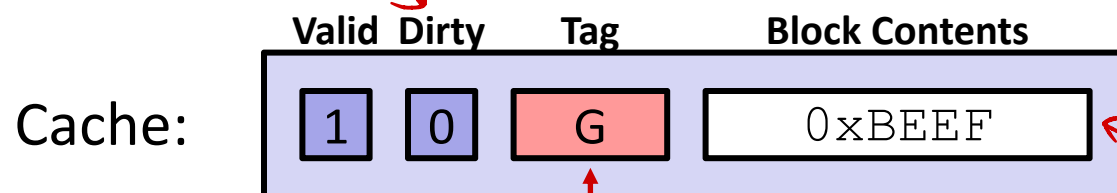
- ❖ Questions from the Reading?

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 cache)*
 - **Write-through:** write immediately to next level
 - **Write-back:** 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 caches*
- ❖ What to do on a write-miss? *(block / data not yet 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

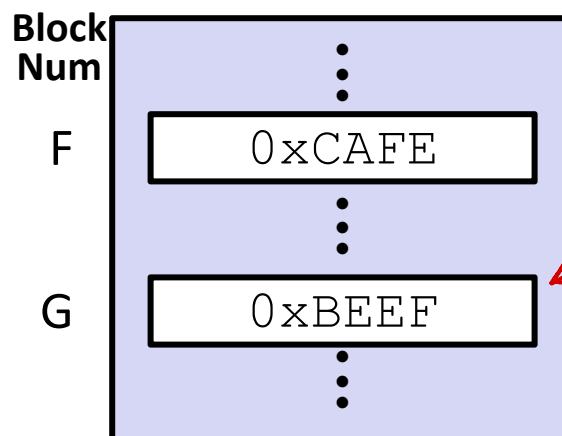
Write-back, Write Allocate Example

Note: We are making some unrealistic simplifications to keep this example simple and focus on the cache policies



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

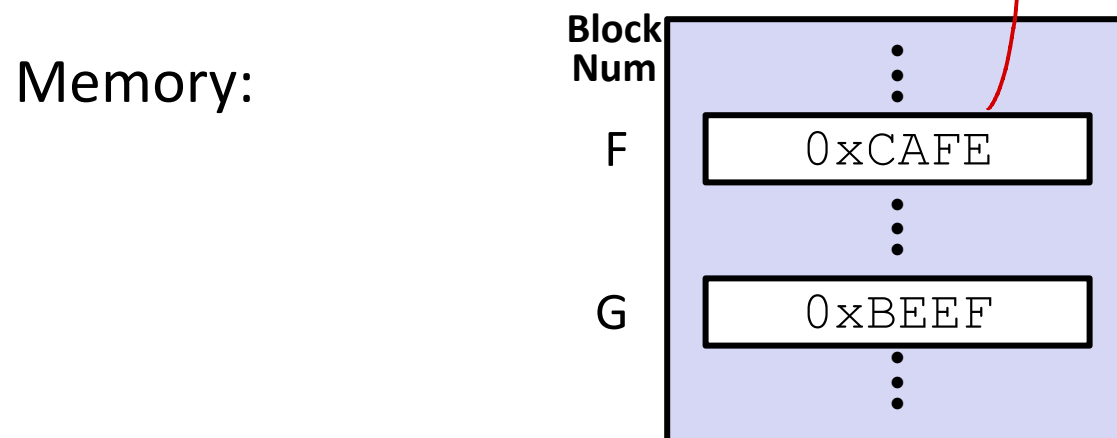
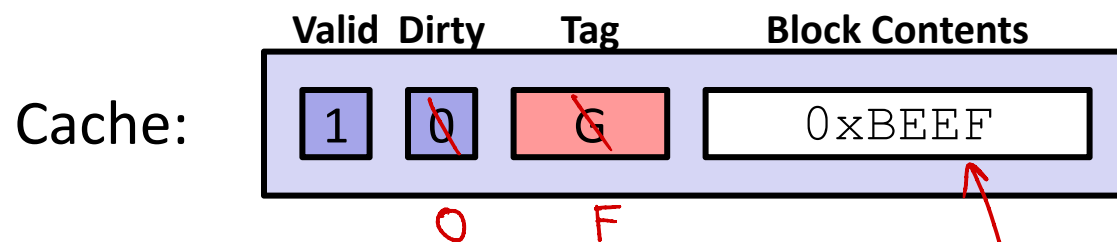
Memory:



dirty bit = 0,
so copies are
consistent

Write-back, Write Allocate Example

1) `mov $0xFACE, (F)` ↗ Not valid x86, assume we mean an address associated with this block num
Write Miss



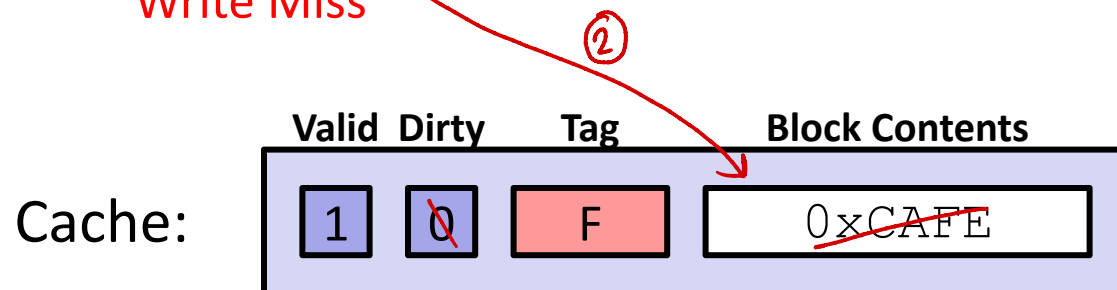
① fetch

Step 1: Bring F into cache

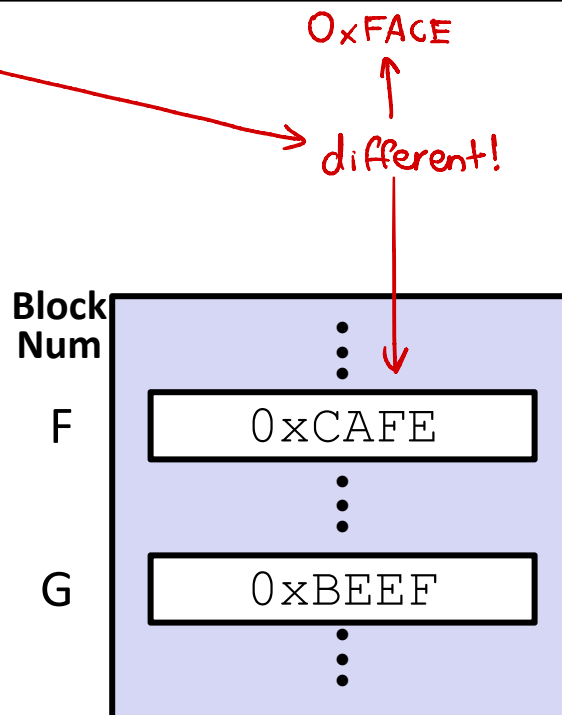
Write-back, Write Allocate Example

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

Write Miss



Memory:



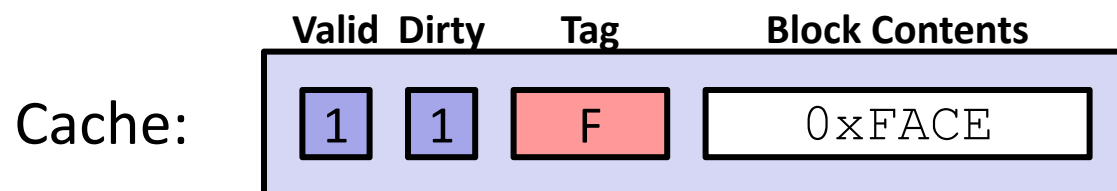
Step 1: Bring F into cache

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

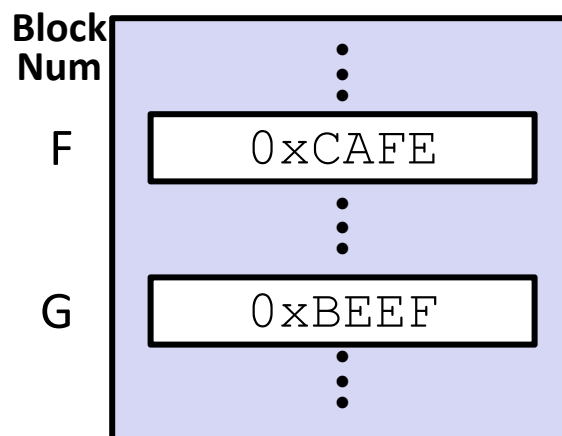
Write-back, Write Allocate Example

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

Write Miss



Memory:

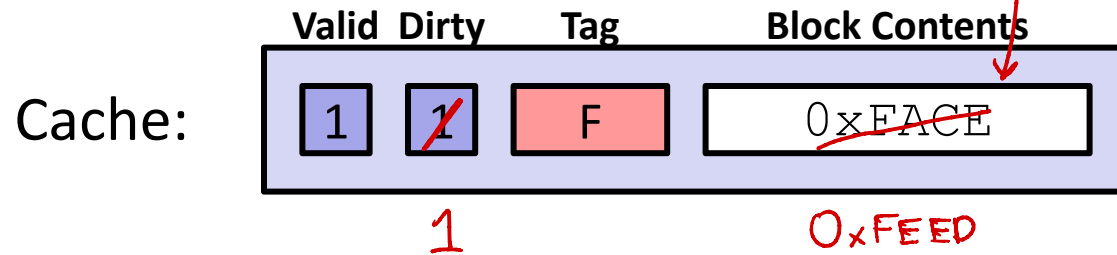


Step 1: Bring **F** into cache

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

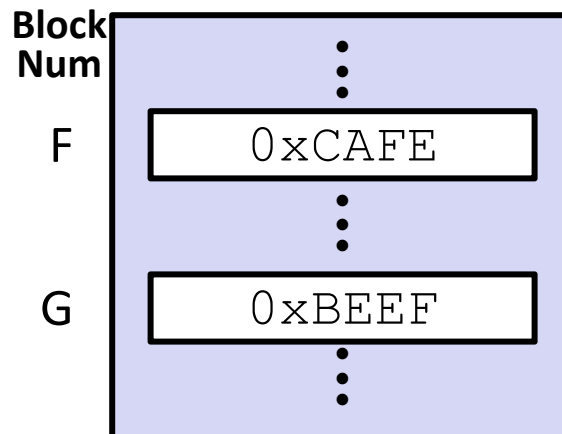
Write-back, Write Allocate Example

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



no need to update
memory!

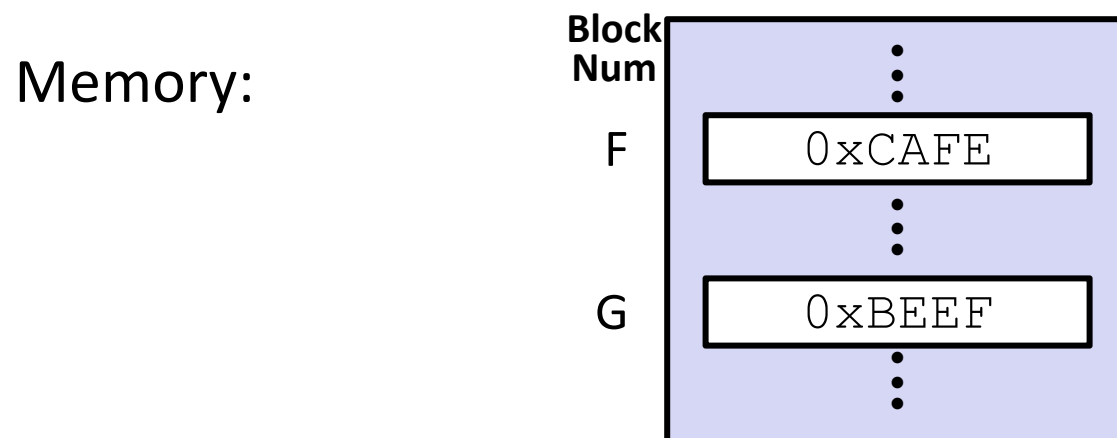
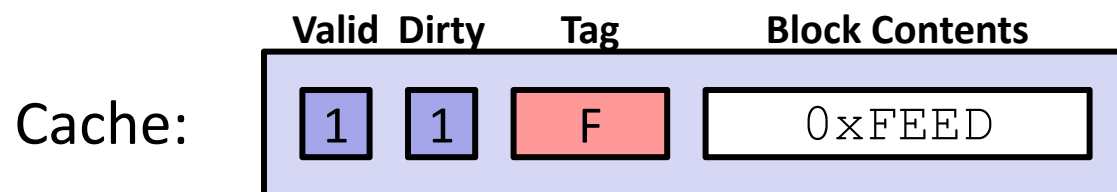
Memory:



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**

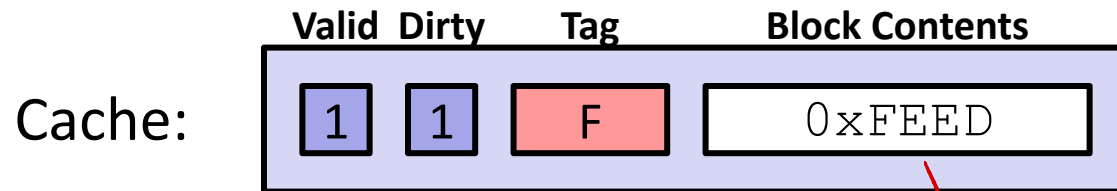


Write-back, Write Allocate Example

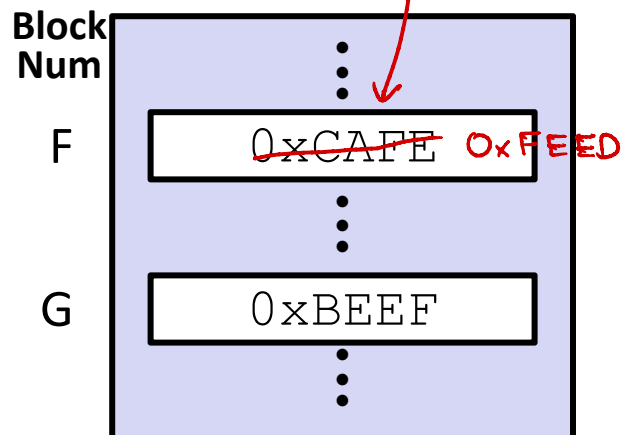
1) `mov $0xFACE, (F)`
Write Miss

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

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



Memory:



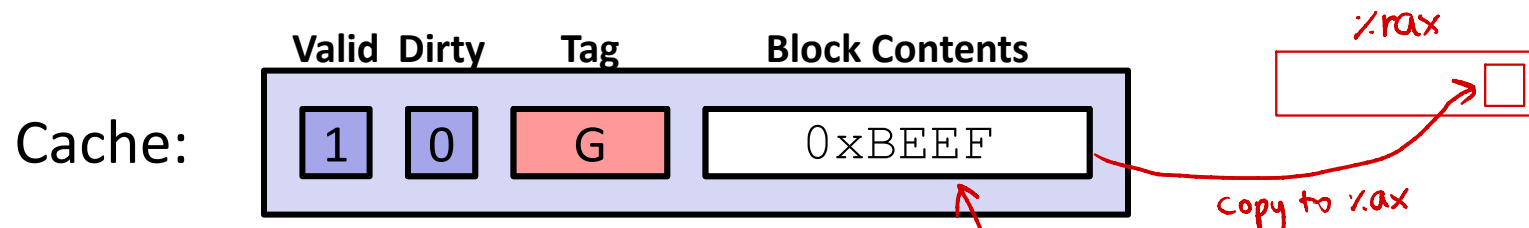
Step 1: Write **F** back to memory since it is dirty

Write-back, Write Allocate Example

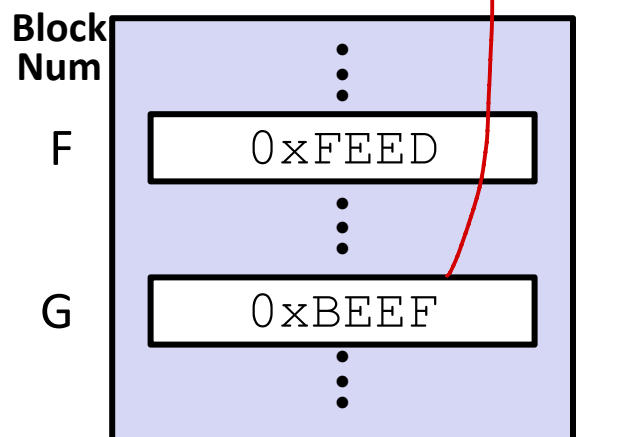
1) `mov $0xFACE, (F)`
Write Miss

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

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



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 hw18 – Lab 4 Preparation

Polling Question

❖ Which of the following cache statements is FALSE?

■ Vote at: <https://PollEv.com/wolfson>

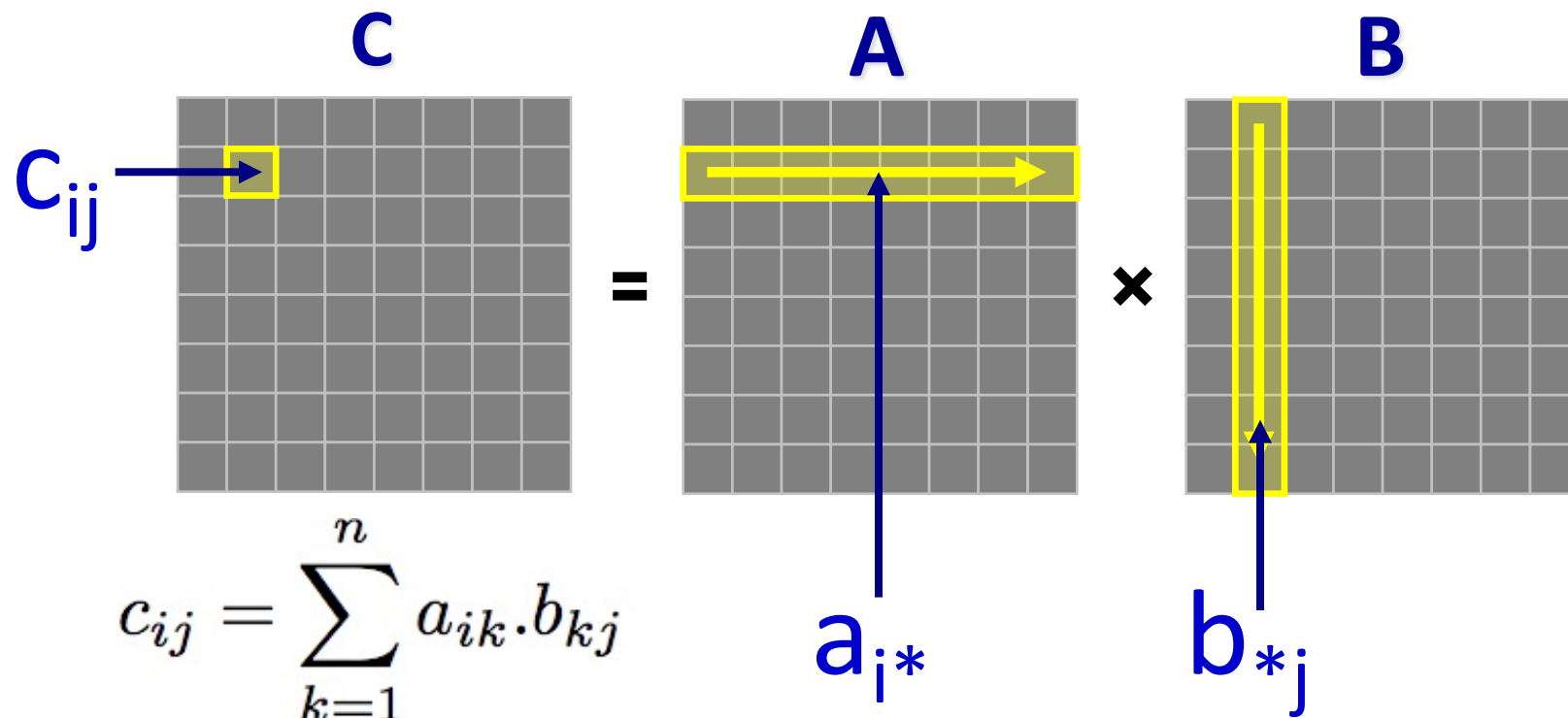
- A. We can reduce compulsory misses by decreasing our block size** *smaller block size pulls in fewer bytes on miss*
- B. We can reduce conflict misses by increasing associativity** *more placement options before evictions*
- C. A write-back cache will save time for code with good temporal locality on writes** *frequently-used blocks rarely evicted, so fewer write backs*
- D. A write-through cache will always match data with the memory hierarchy level below it** *yes—main goal is 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

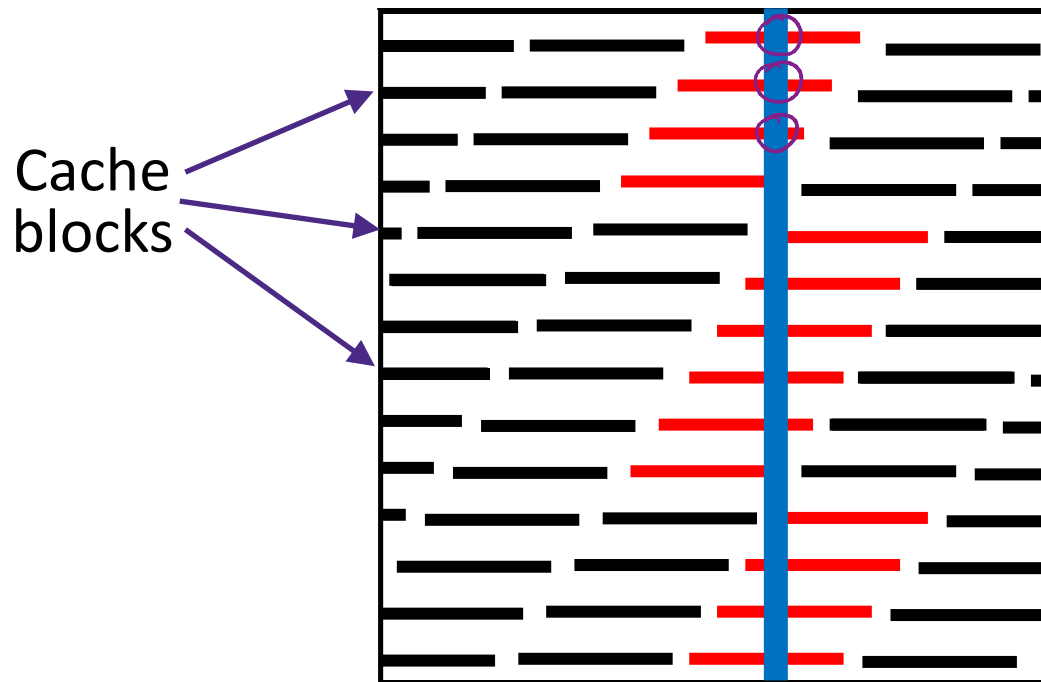
- ❖ How can you achieve locality?
 - Adjust memory accesses in *code* (software) to improve miss rate (MR)
 - Requires knowledge of *both* how caches work as well as your system's parameters
 - Proper choice of algorithm
 - Loop transformations

Example: Matrix Multiplication



Matrices in Memory

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

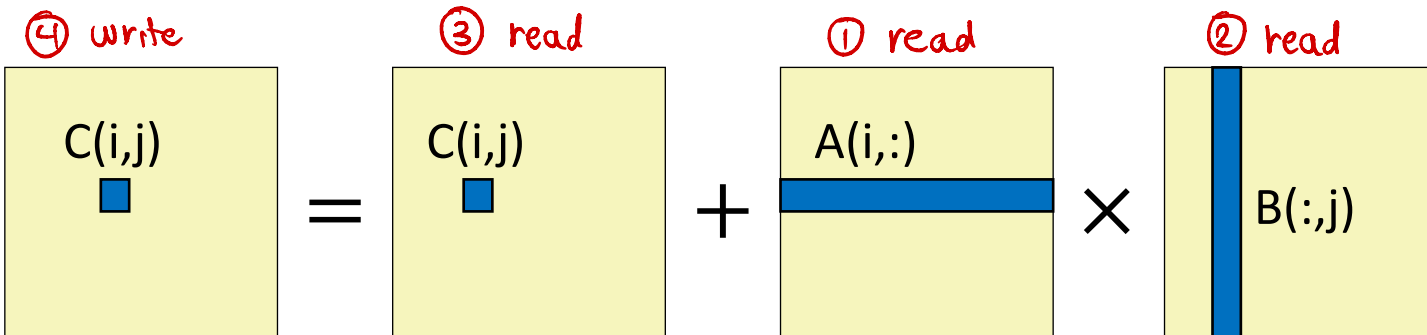


COLUMN of matrix (blue) is spread
among cache blocks shown in red

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++)
      how many accesses? → c[i*n+j] += a[i*n+k] * b[k*n+j];
  
```



Cache Miss Analysis (Naïve)

Ignoring
matrix C

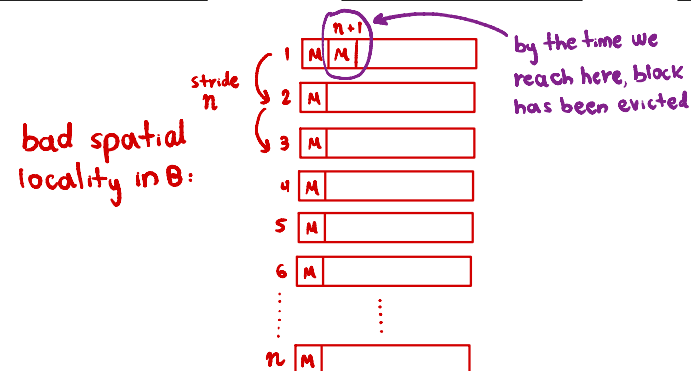
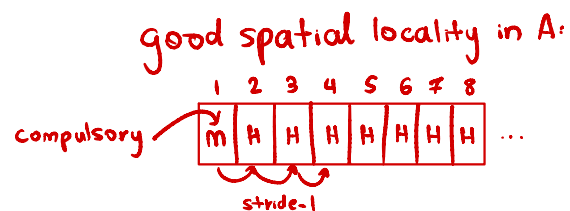
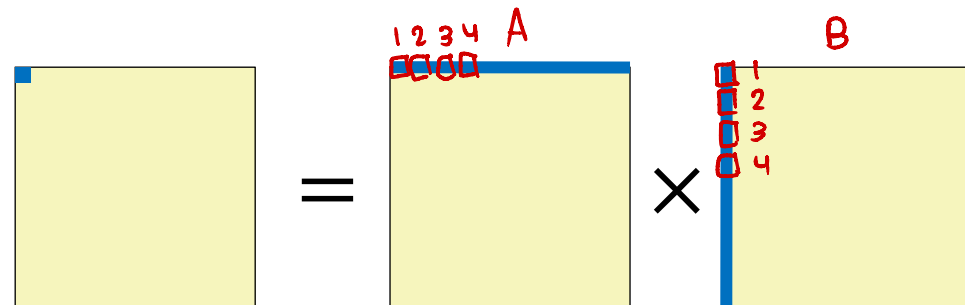
❖ Scenario Parameters:

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

8 elements
per block

❖ Each iteration:

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



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)

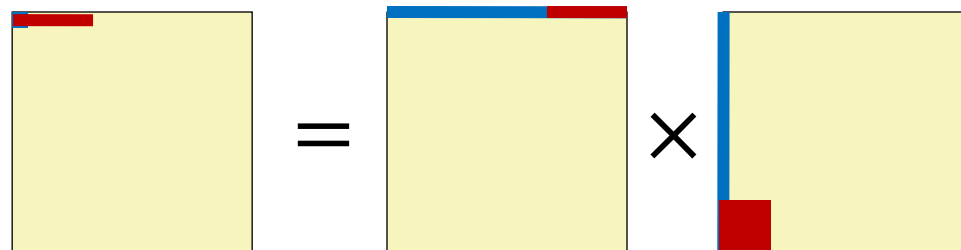
❖ Each iteration:

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



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

red shows
blocks in
cache



8 doubles wide

Cache Miss Analysis (Naïve)

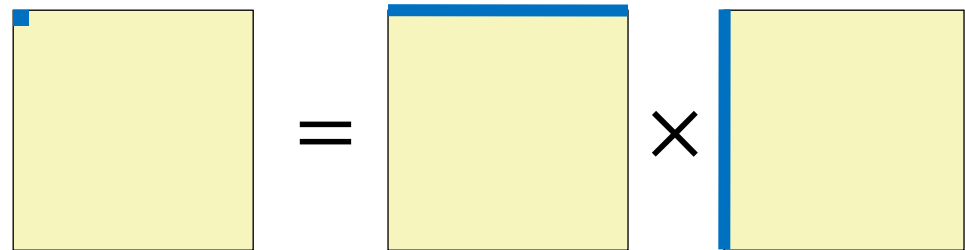
Ignoring
matrix C

❖ 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: $n \left(\frac{n}{8} + n \right) \stackrel{=}{\neq} \frac{9n}{8} \times n^2 = \frac{9}{8} n^3$ not so great :)

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:

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

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

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_{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

❖ 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];
```

6 nested loops look scary, but actually more efficient! ☺

loop over blocks

loop within blocks

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

Cache Miss Analysis (Blocked)

Ignoring
matrix C

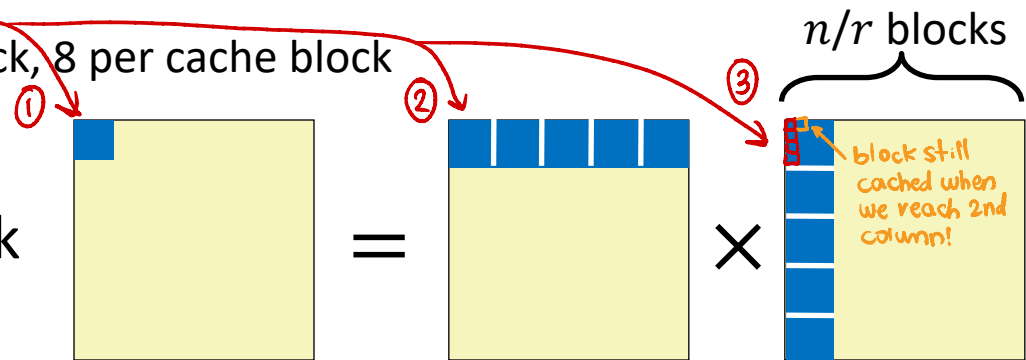
❖ Scenario Parameters:

- 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:

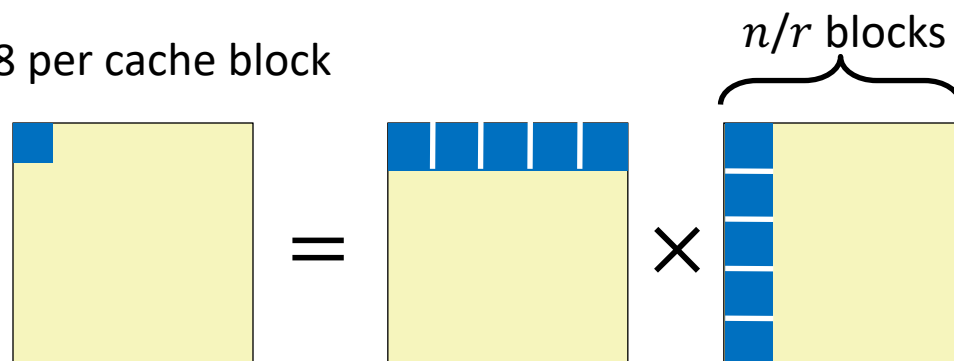
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ Each block iteration:

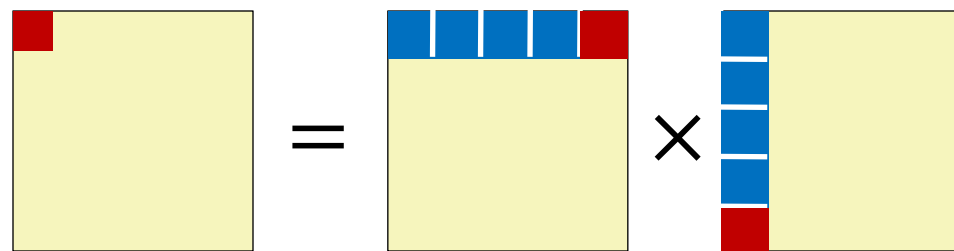
- $r^2/8$ misses per block

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

n/r blocks in row and column



- Afterwards in cache (schematic)



Cache Miss Analysis (Blocked)

Ignoring
matrix C

❖ Scenario Parameters:

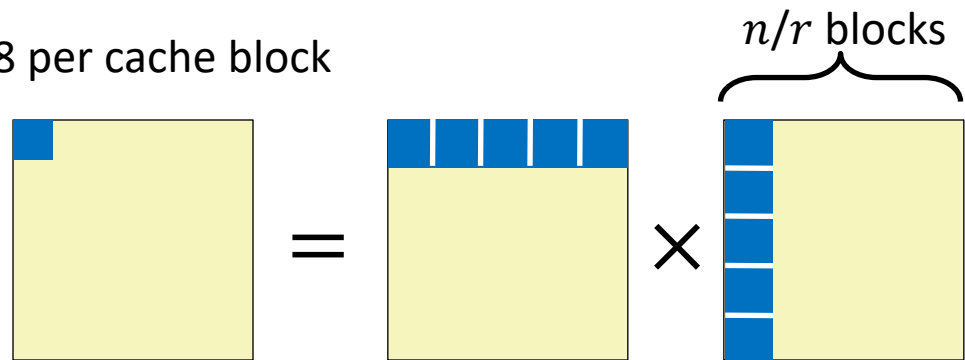
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < C$

❖ 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:

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

still $\Theta(n^3)$, but
we can significantly
reduce coefficient
as r grows

Matrix Multiply Visualization

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

Naïve:



Cache misses: 551,000

$\approx 1,020,000$
cache misses

Shaded areas are
blocks stored in cache

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 coding rules
 - 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*

