The Hardware/Software Interface

Memory & Caches IV

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus, Divya Ramu, Grace Zhou, Jessie Sun, Jiuyang Lyu, Kanishka Singh, Kurt Gu, Liander Rainbolt, Mendel Carroll, Ming Yan, Naama Amiel, Pollux Chen, Rose Maresh, Soham Bhosale, Violet Monserate

REFRESH TYPE	EXAMPLE SHORTCUTS	EFFECT
SOFT REFRESH	GMAIL [REFRESH] BUTTON	REQUESTS UPDATE WITHIN JAVASCRIPT
NORMAL REFRESH	F5, CTRL-R, ⌘R	REFRESHES PAGE
HARD REFRESH	CTRL-F5, CTRL-⇧, ⌘⇧R	REFRESHES PAGE INCLUDING CACHED FILES
HARDER REFRESH	CTRL-⇧-HYPER-ESC-R-F5	REMOTELY CYCLES POWER TO DATACENTER
HARDEST REFRESH	CTRL-⌘⇧#-R-F5-F-5-ESC-O-0-Ø-⏏-SCROLL LOCK	INTERNET STARTS OVER FROM ARPANET

not real

http://xkcd.com/1854/

Relevant Course Information

- Lab 4 released today, due Friday, 11/21
  - Cache parameter puzzles and code optimizations

- HW17 due Fri (11/7)
- HW19 due Fri (11/14)
  - Lab 4 preparation

- Midterm scores have been released
  - Look at the posted solutions, common issues post, and the rubric in Gradescope
  - Submit a regrade request if you see a discrepancy
    - Be kind! We are human and grading hundreds of exams. Rudeness is unacceptable.

Lecture Outline (1/5)

- **Cache Misses**
- Data Consistency and Cache Writes
- Cache-Friendly Code
- Cache Blocking
- Cache Motivation, Revisited

Types of Cache Misses: 3 C's (Review)

- Compulsory (cold) miss
  - Occurs on first access to a block
- Conflict miss
  - Conflict misses occur when the cache is large enough, but multiple blocks all map to the same set
    - *e.g.*, referencing addresses 8, 24, 8, 24, … (blocks 2 & 6) could miss every time
- Capacity miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (*i.e.*, won't fit even if cache was *fully-associative*)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses

Parameters and Misses (Review)

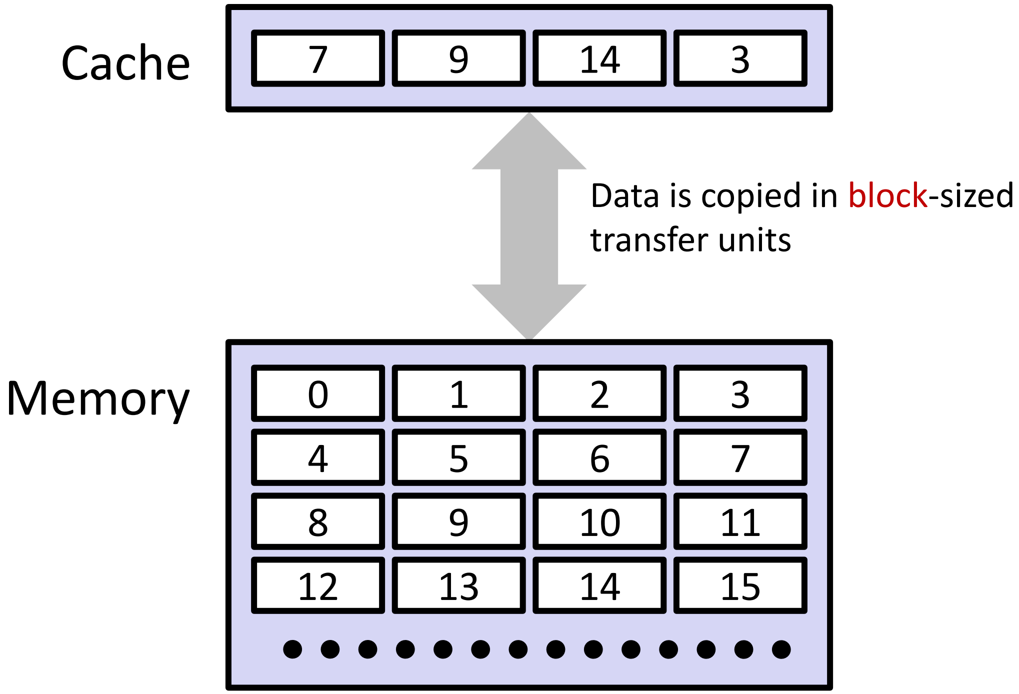
- To help with **compulsory misses**, increase the block size
  - More data is brought into the cache with each miss
- To help with **conflict misses**, increase the associativity
  - More blocks can coexist in the same set
- To help with **capacity misses**, increase the cache size or decrease the working set of data
  - Can simultaneously hold more in the cache relative to accessed data

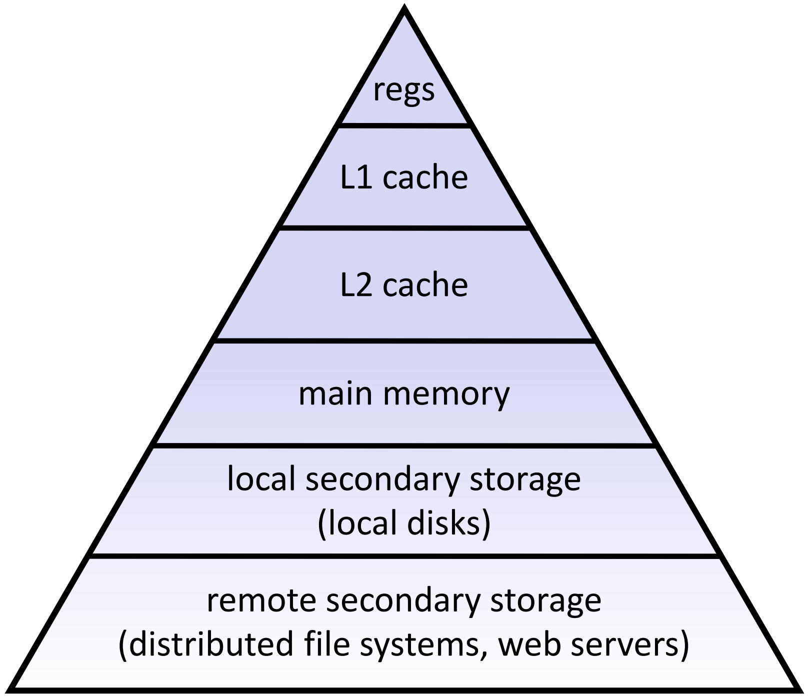
Lecture Outline (2/5)

- Cache Misses
- **Data Consistency and Cache Writes**
- Cache-Friendly Code
- Cache Blocking
- Cache Motivation, Revisited

Data Consistency

- Multiple copies of data may exist – levels of $ and main memory
  - Writes may break consistency across the multiple copies
  - Blocks are copied to, updated, and evicted from caches





Cache Write Policies (Review)

- What to do on a write-hit?
  - 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*”)

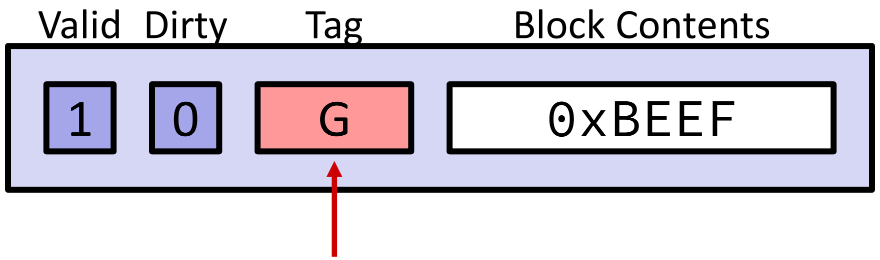
- What to do on a write-miss?
  - Write allocate: (“fetch on write”) load into cache, then execute the write-hit policy
    - Good if more writes or reads to the location follow
  - No-write allocate: (“write around”) just write immediately to next level

- Typical caches:
  - Write-back + Write allocate, usually
  - Write-through + No-write allocate, occasionally

Write-back, Write Allocate Example Setup

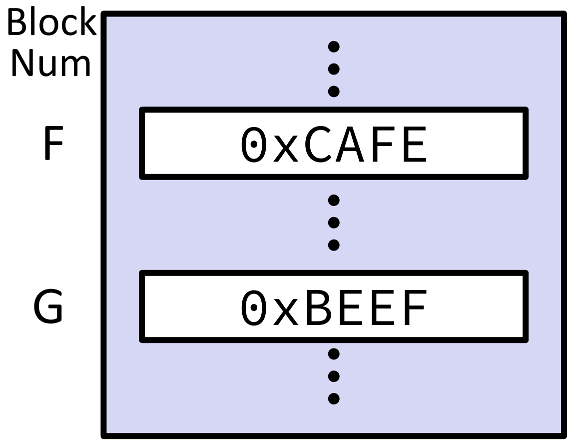
<u>Note</u>: We are making some unrealistic simplifications to keep this example simple and focus on the cache policies

Cache:



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

Memory:



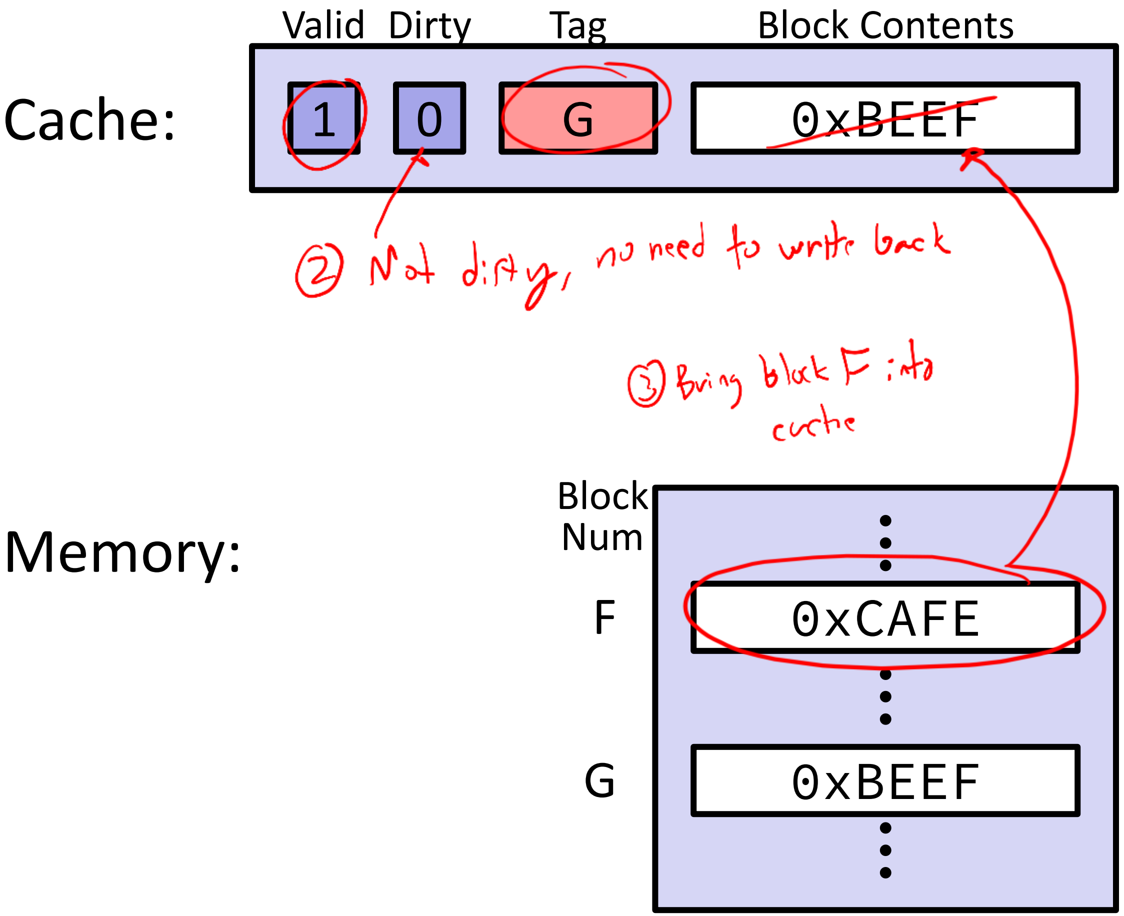
Write-back, Write Allocate Example Access #1 Step 1

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

Not valid x86, assume we mean an address associated with this block num

Write Miss

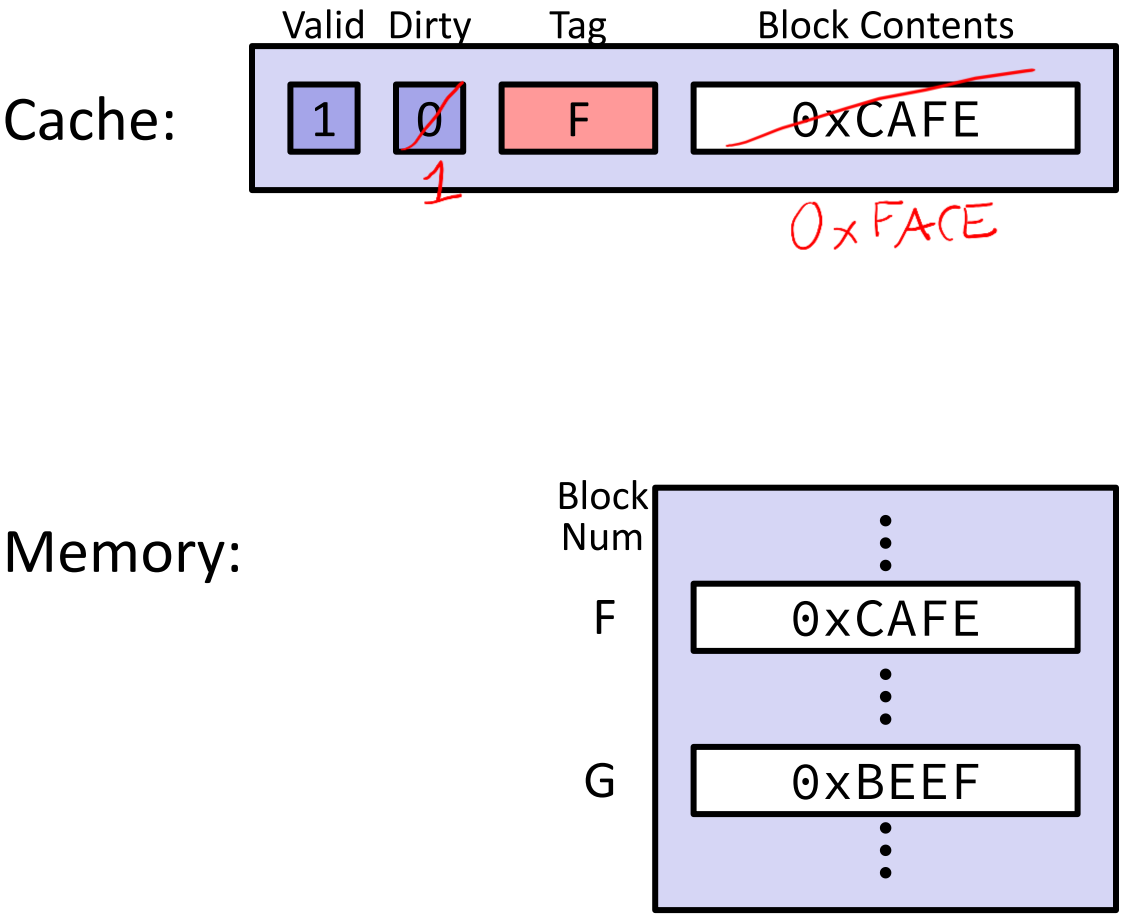
① F ≠ G → Miss



Step 1: Bring F into cache

Write-back, Write Allocate Example Access #1 Step 2

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

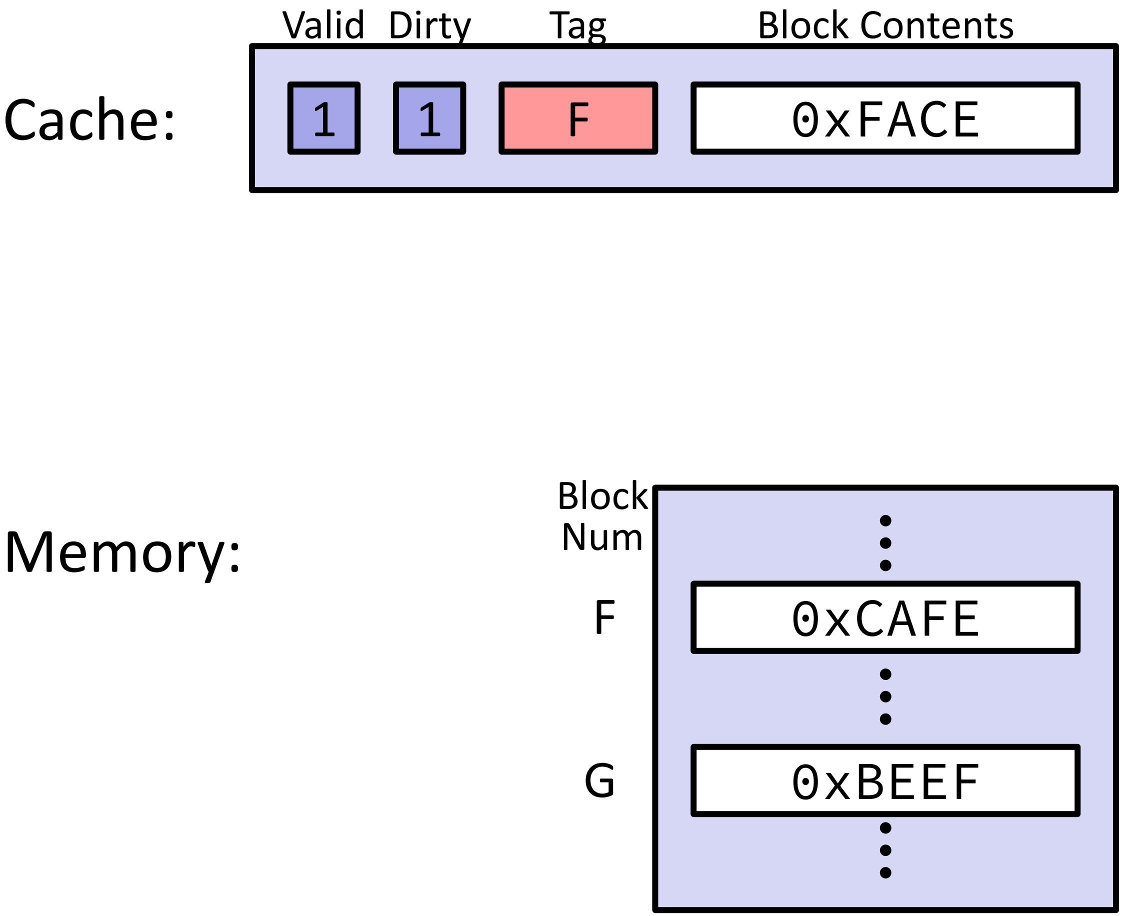


Step 1: Bring F into cache

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

Write-back, Write Allocate Example Access #1 Result

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



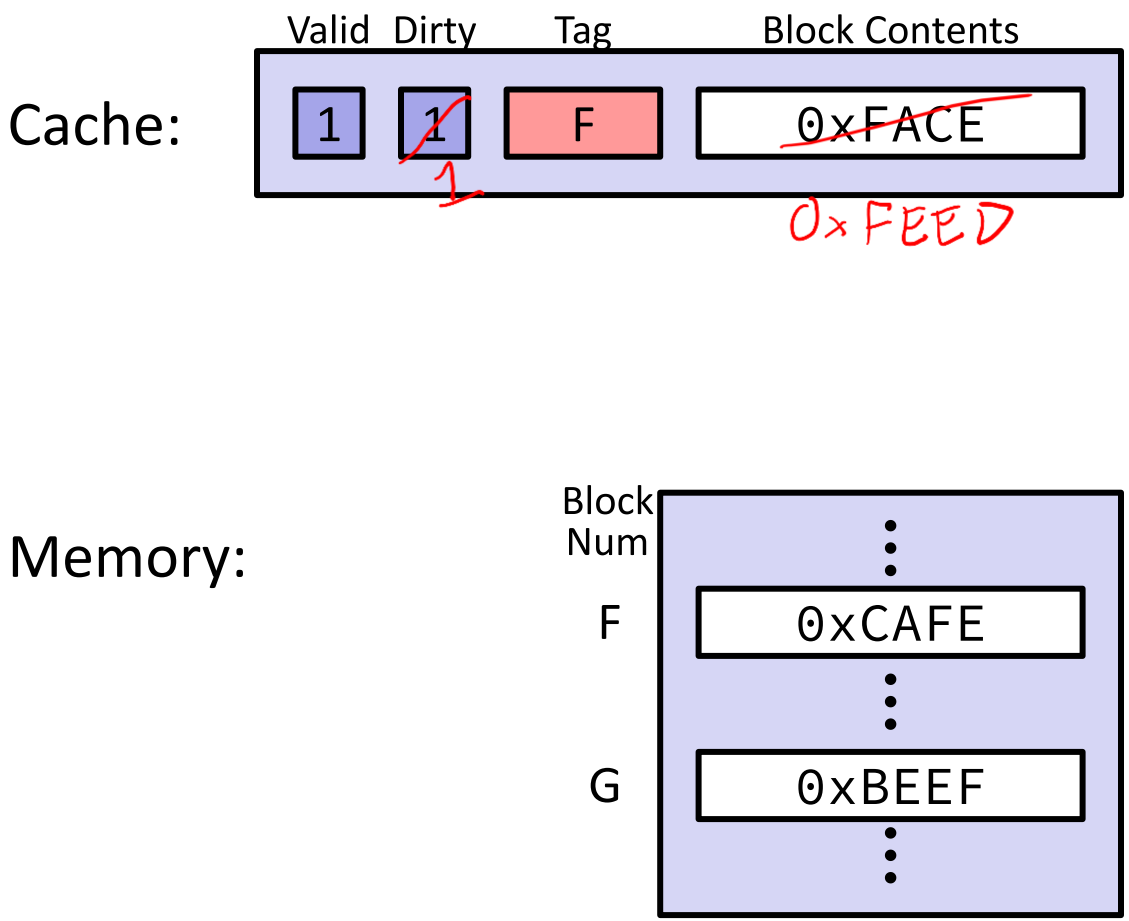
Step 1: Bring F into cache

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

Write-back, Write Allocate Example Access #2 Step 1

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

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

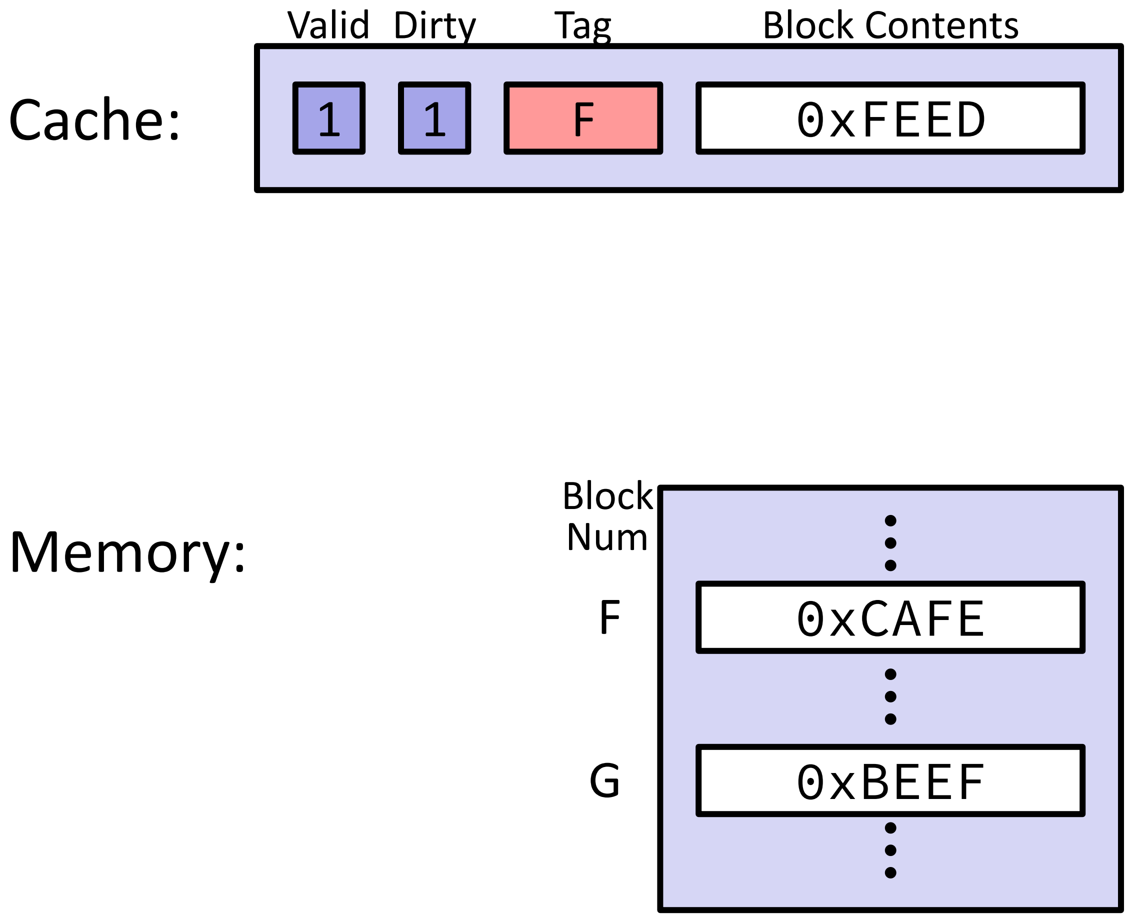


Step: Write `0xFEED` to cache only (and set the dirty bit)

Write-back, Write Allocate Example Access #2 Result

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

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

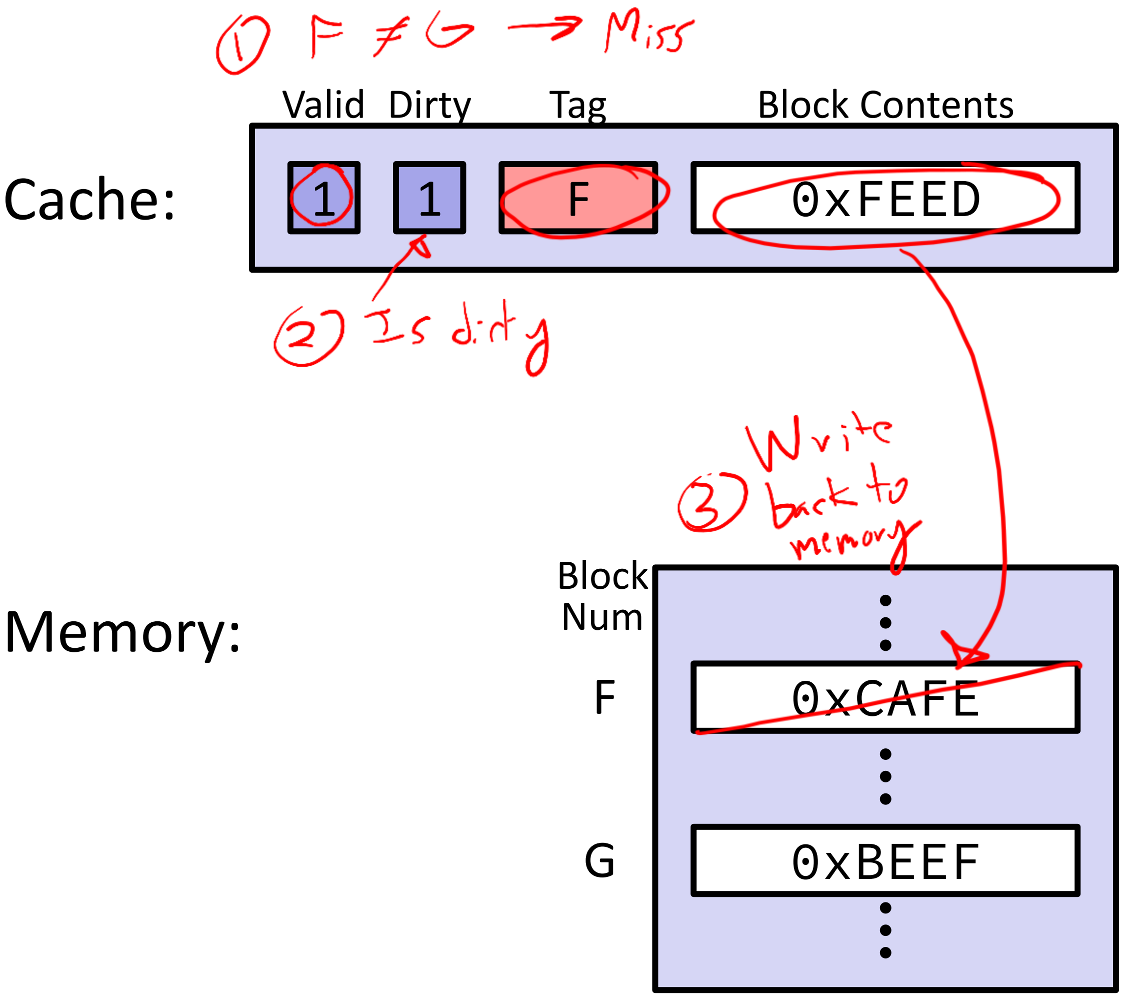


Write-back, Write Allocate Example Access #3 Step 1

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

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

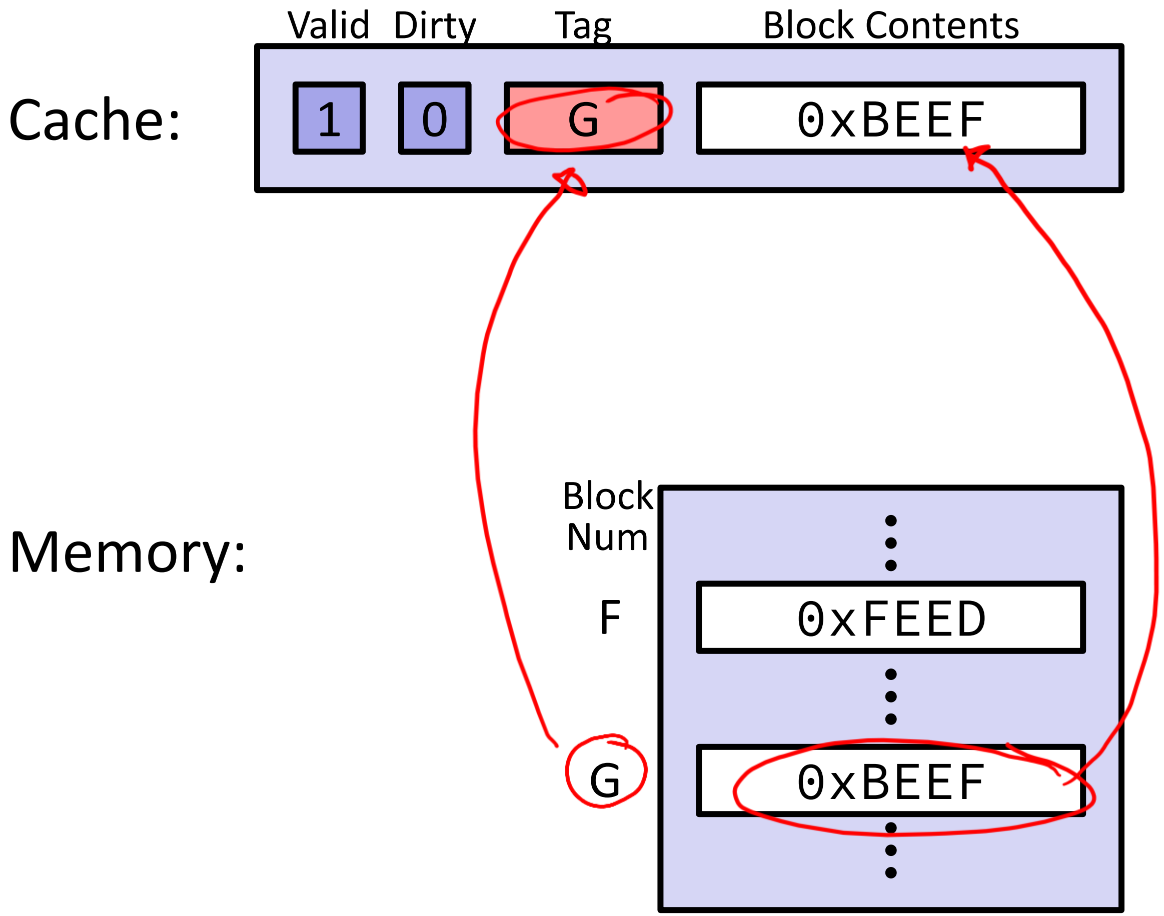
3) `mov (G), %ax` Read Miss



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

Write-back, Write Allocate Example Access #3 Step 2

1) `mov $0xFACE, (F)`	2) `mov $0xFEED, (F)`	3) `mov (G), %ax`
Write Miss	Write Hit	Read Miss



<u>Step 1</u>: Write F back to memory since it is dirty

<u>Step 2</u>: Bring G into the cache so that we can copy it into `%ax`

Polling Question

- Which of the following cache statements are FALSE? (Select all)

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…**

Lecture Outline (3/5)

- Cache Misses
- Data Consistency and Cache Writes
- **Cache-Friendly Code**
- Cache Blocking
- Cache Motivation, Revisited

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

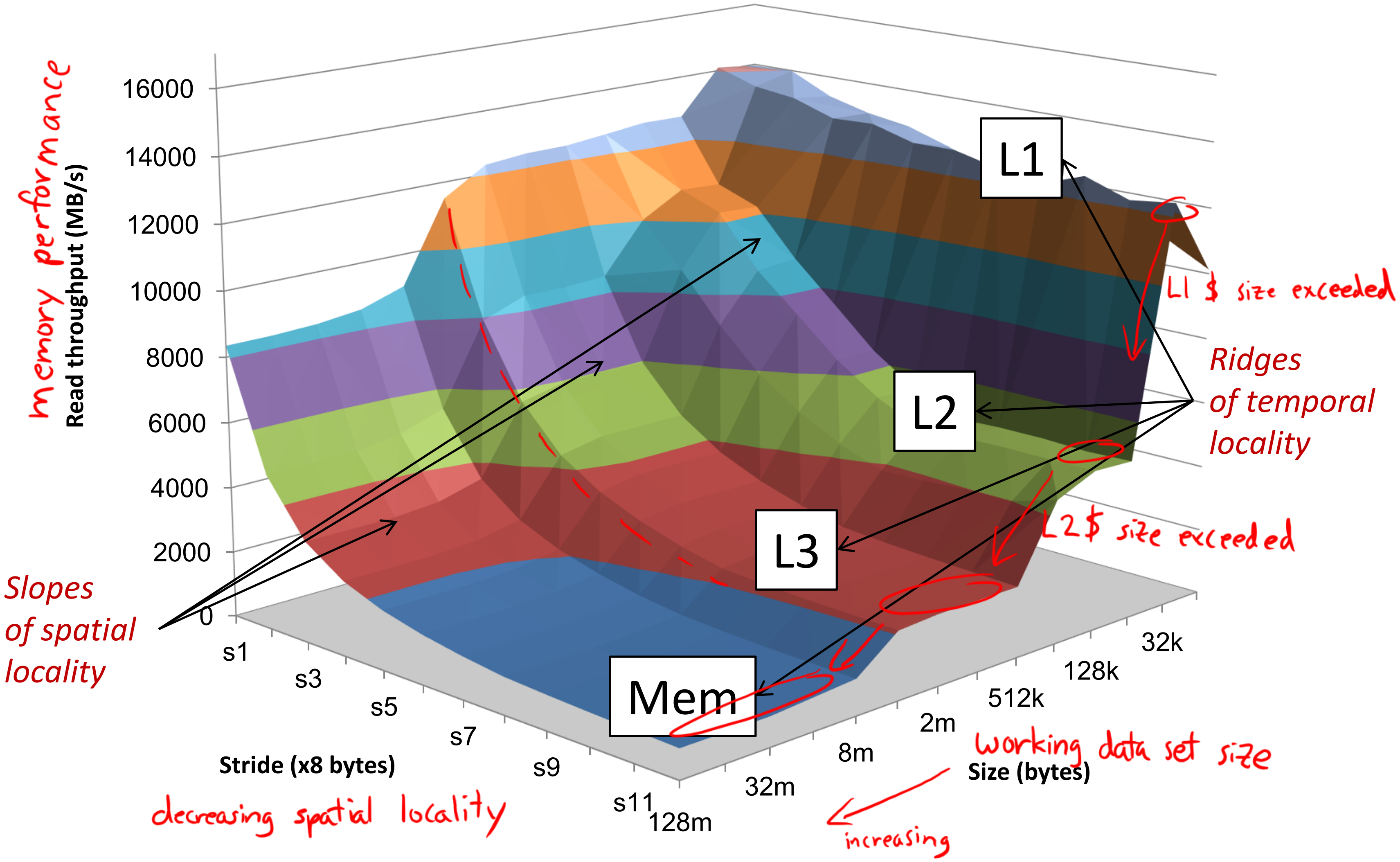
- How can you achieve locality?
  - Proper choice of algorithm
  - Loop transformations
  - Adjust memory accesses in *code* (software) to improve miss rate (MR)
    - Requires knowledge of *both* how caches work as well as your system's parameters

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

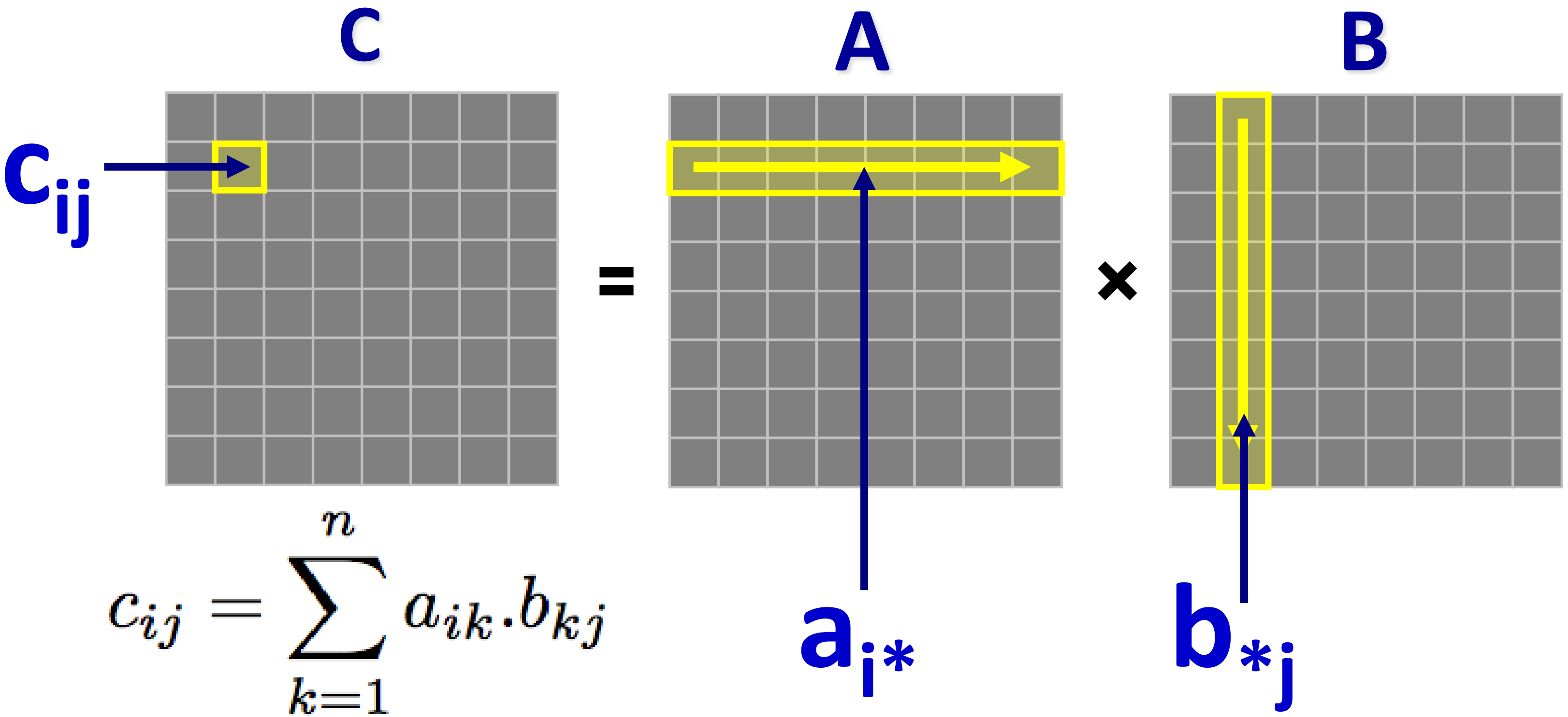
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Lecture Outline (4/5)

- Cache Misses
- Data Consistency and Cache Writes
- Cache-Friendly Code
- **Cache Blocking**
- Cache Motivation, Revisited

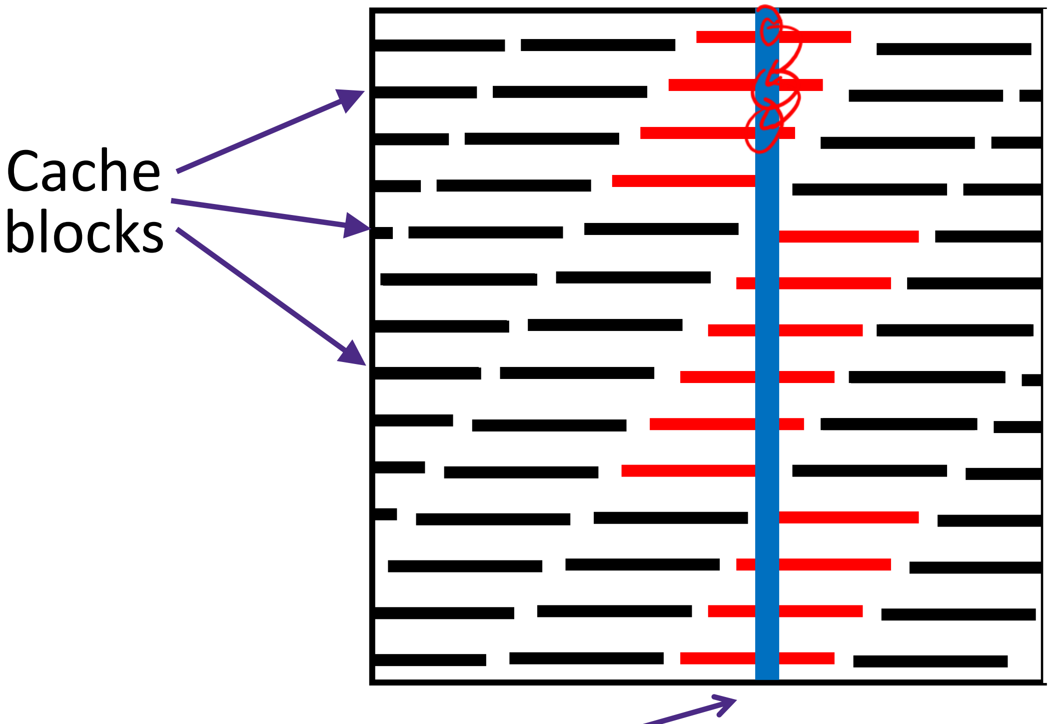
Example: Matrix Multiplication



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

Matrices in Memory

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

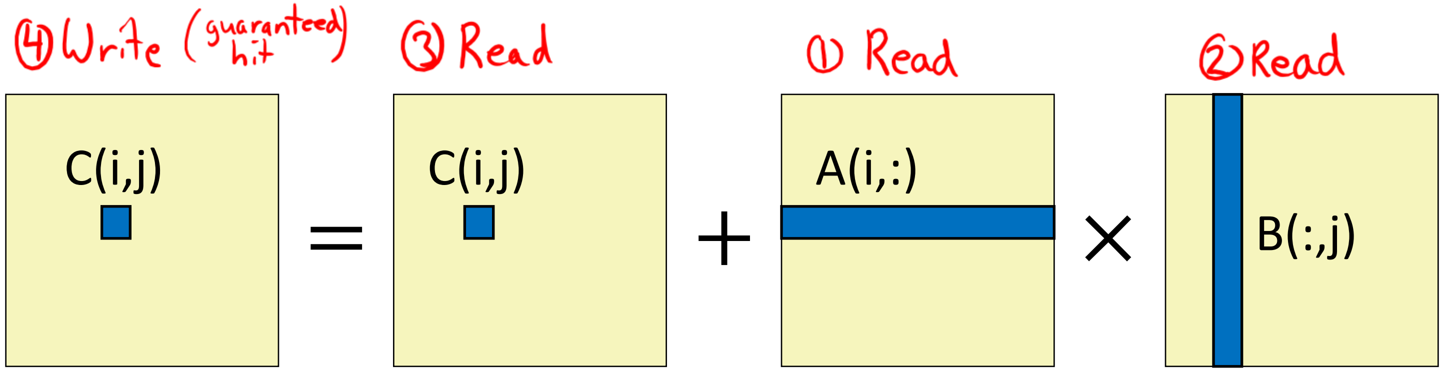


COLUMN of matrix (blue) is spread among cache blocks shown in red

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][j] += A[i][k] * B[k][j];
```

check mem access pattern

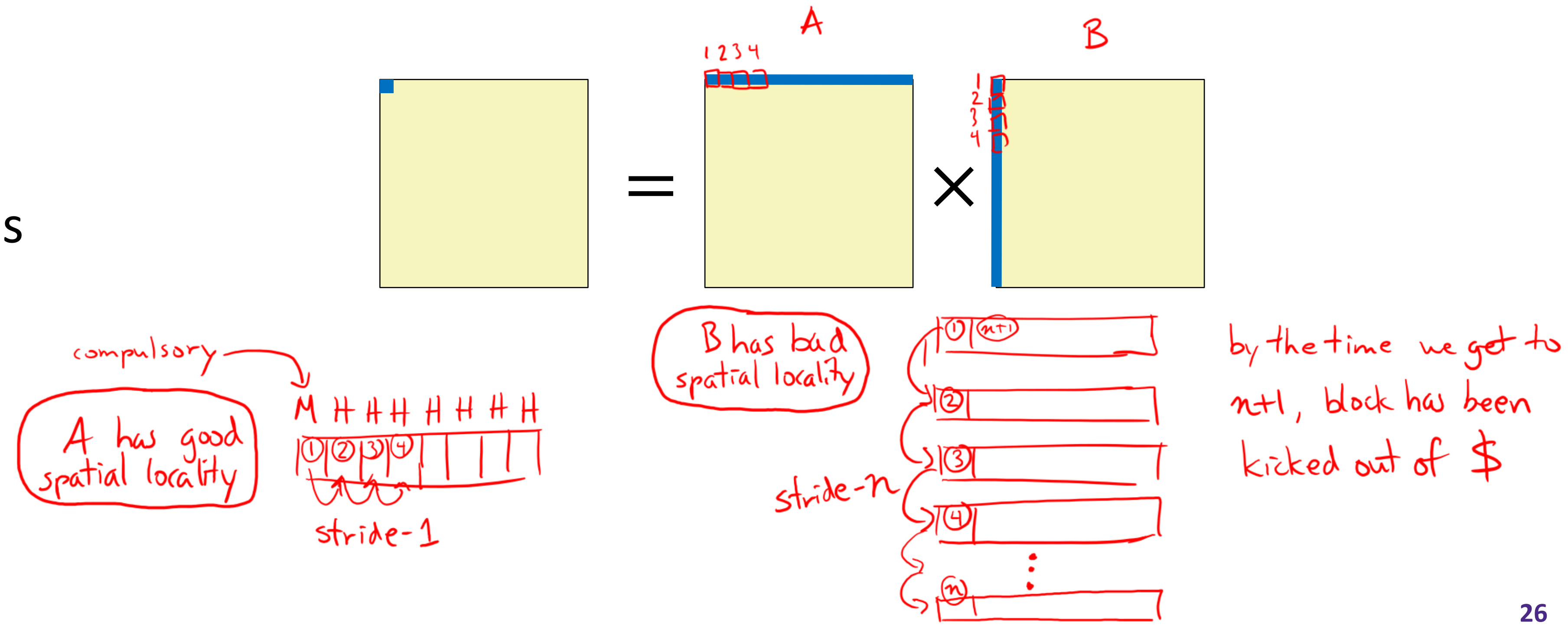


Cache Miss Analysis (Naïve)

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 is much smaller than n — key assumption!

- First iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses (A: $\frac{n}{8}$, B: n)

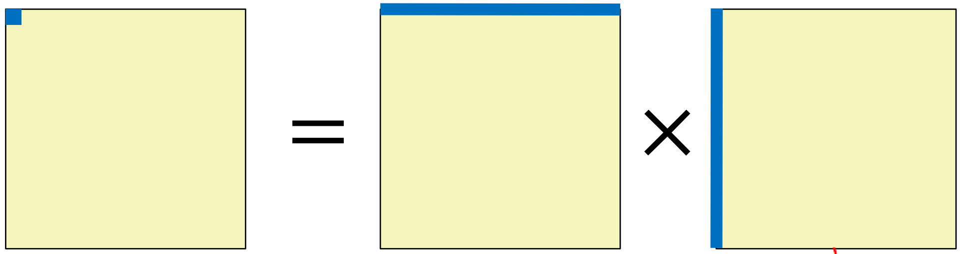


Cache Miss Analysis (Naïve)

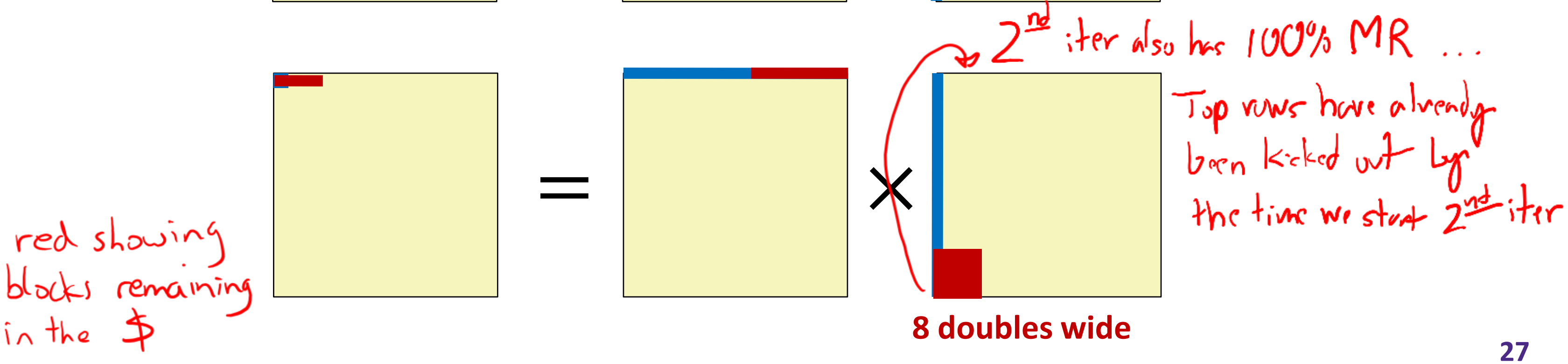
Ignoring matrix C

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

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



  - Afterwards in cache: (schematic)

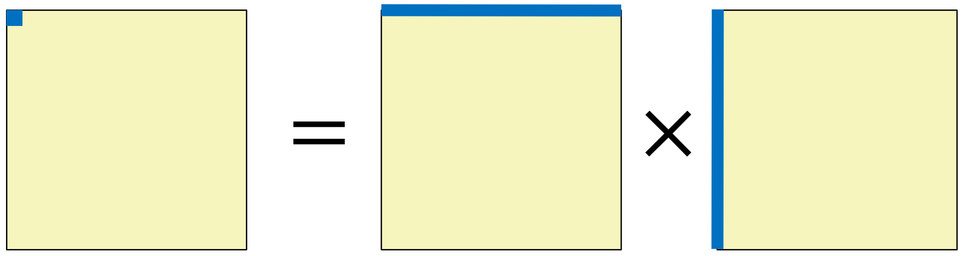


Cache Miss Analysis (Naïve)

Ignoring matrix C

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 `doubles`
  - Cache size is 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, multiplying two 4×4 matrices:

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

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

This is extra (non-testable) material

Linear Algebra to the Rescue (2)

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 ($n = 4r$)
- Multiplication operates on small "block" matrices
  - $C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} \times B_{k2}$
  - 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][jb] += A[ib][kb] * B[kb][jb];
```

loop over block matrices (the i, j, k loops)

loop within block matrices (the ib, jb, kb loops)

Don't overwrite previous subresults, add to them (the += operator)

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

Cache Miss Analysis (Blocked)

Ignoring matrix C

How do I choose r?

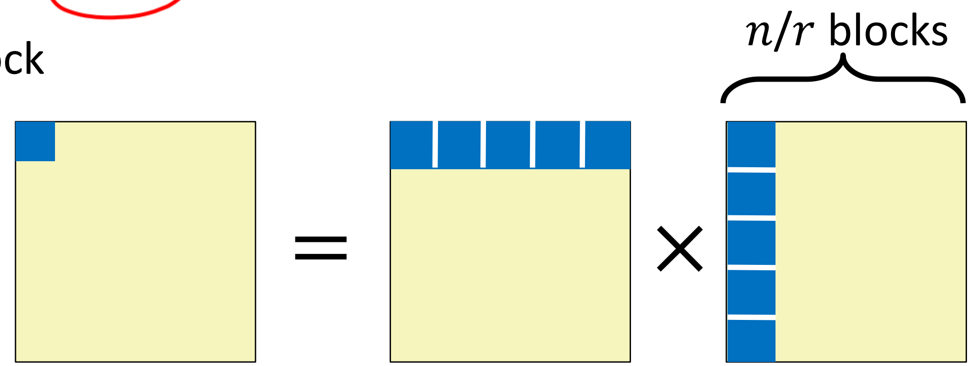
Working set should fit in cache

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

r^2 elements per block, 8 per cache block

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

n/r blocks in row and column



Cache Miss Analysis (Blocked)

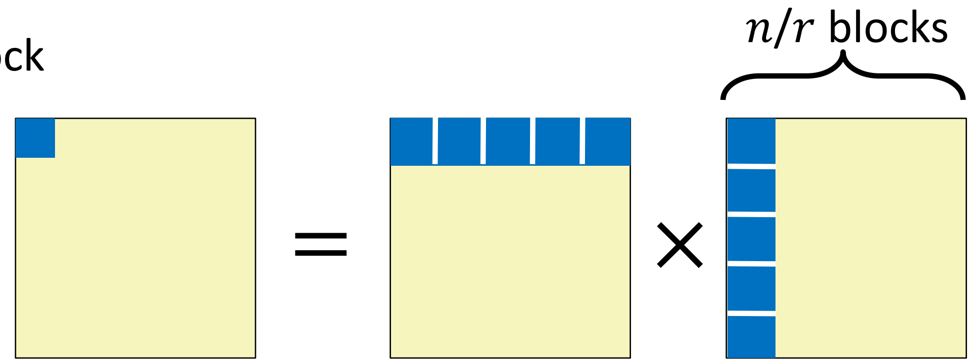
Ignoring matrix C

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

r^2 elements per block, 8 per cache block

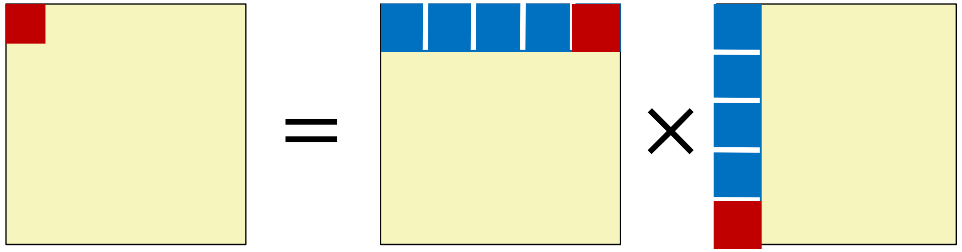
- Each block iteration:
  - $r^2/8$ misses per block
  - $n/r \times r^2/8 \times 2 = nr/4$

n/r blocks in row and column



  - Afterwards in cache (schematic)

red showing blocks remaining in the $



Cache Miss Analysis (Blocked)

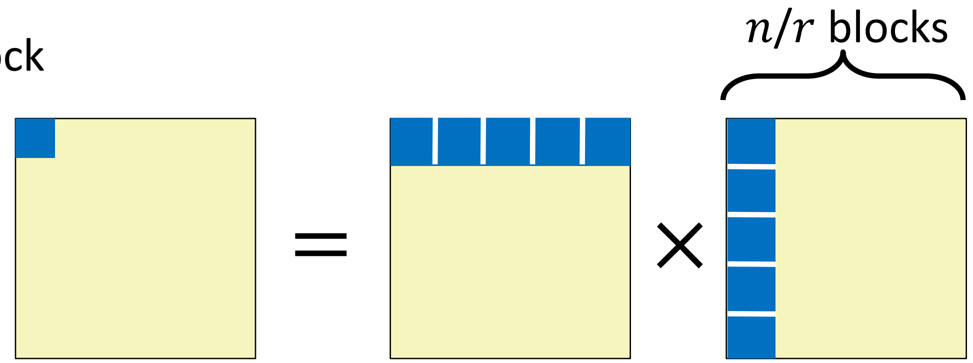
Ignoring matrix C

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

r^2 elements per block, 8 per cache block

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

n/r blocks in row and column



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

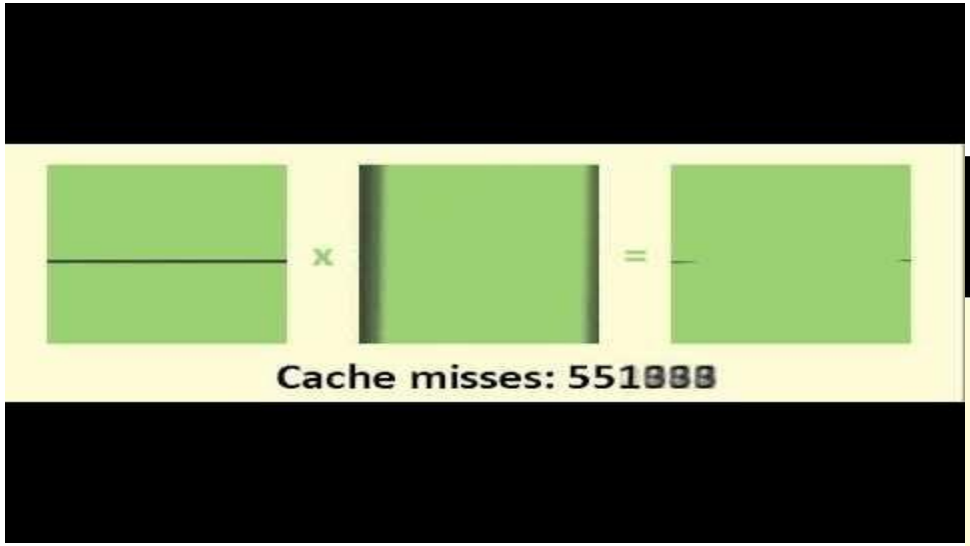
vs. $9n^3/8$

Matrix Multiply Visualization

shaded areas show blocks stored in the $

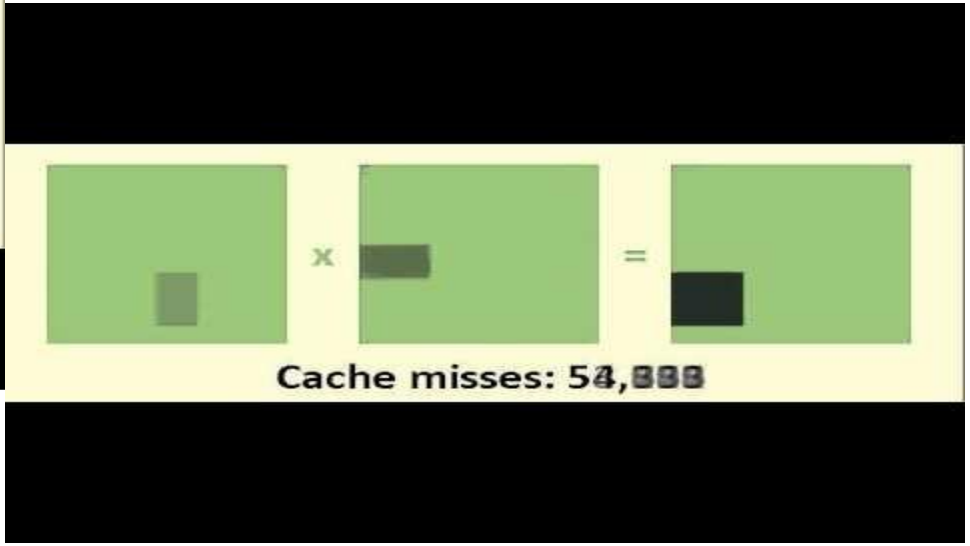
- Here n = 100, C = 32 KiB, r = 30

Naïve:



≈ 1,020,000
cache misses

Blocked:



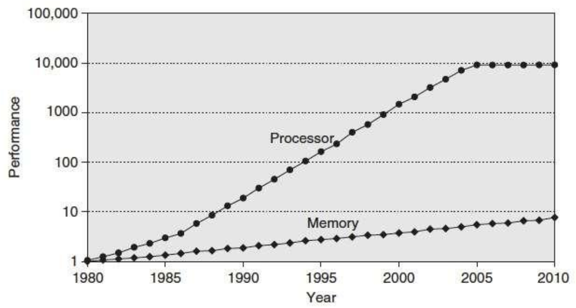
≈ 90,000
cache misses

Lecture Outline (5/5)

- Cache Misses
- Data Consistency and Cache Writes
- Cache-Friendly Code
- Cache Blocking
- **Cache Motivation, Revisited**

Cache Motivation, Revisited

- Memory accesses are expensive!
  - Massive speedups to processors without similar speedups in memory only made the problem worse
  - "Processor-Memory Bottleneck":



- We defined "locality", based on observations about existing programs, written by an extremely small subset of the population
  - We built hardware that utilizes locality to improve performance (*e.g.*, AMAT)

Cache "Conclusions"

- All systems favor "cache-friendly code"
  - Can get most of the advantage with generic coding rules

- ⚠ We implicitly made value judgments about "good" and "bad" code
  - "Good" code exhibits "good" locality
  - "Good" code might be considered the (desired) *common case*

Common Case Optimizations

- Optimizing for the common case is a classic (arguably foundational) CS technique!
  - *e.g.*, algorithms analysis often uses worse case or average case performance
  - *e.g.*, caches optimize for an *average program* ("most programs") that exhibits locality
- Natural conclusion is to make the common case as performant as possible at the expense of edge-cases
  - Generally, bigger performance impact with common case than edge case optimizations
  - What's the danger here?

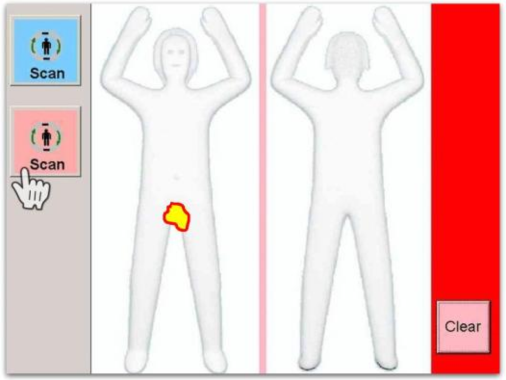
The Common Case and Normativity

- "**Normativity** is the phenomenon in human societies of designating some actions or outcomes as good or desirable or permissible and others as bad or undesirable or impermissible."
  - https://en.wikipedia.org/wiki/Normativity

- **Norms** are what are considered "usual" or "expected"
  - These often get conflated with the common case:
    norm gets "common case" treatment, *abnormal* gets "edge case" treatment
  - Who determines the norms?

Example: TSA Body Scanners

- TSA used machine learning to determine predictable variation among “average” bodies
  - Built two models: one for “men” and one for “women”
- TSA agent chooses model to use based on assumptions about traveler’s presentation:



- Who are the “edge cases?”
- What is the “edge case performance?”

Design Considerations

- Make sure you account for non-normative cases
  - Is this (change to) edge-case behavior okay/acceptable?

- Be careful of implicit normative assumptions
  - Can erase people's experiences and diversity, even labeling/categorizing them as threats
  - Caches aren't neutral, either – they assume that the underlying data doesn't change
    - Changes can come from above (the CPU), but not from below
    - *e.g.*, changing your name in Google Drive "breaks" the browser cache

Cannot change UW NetID without legal name change!

Discussion Questions

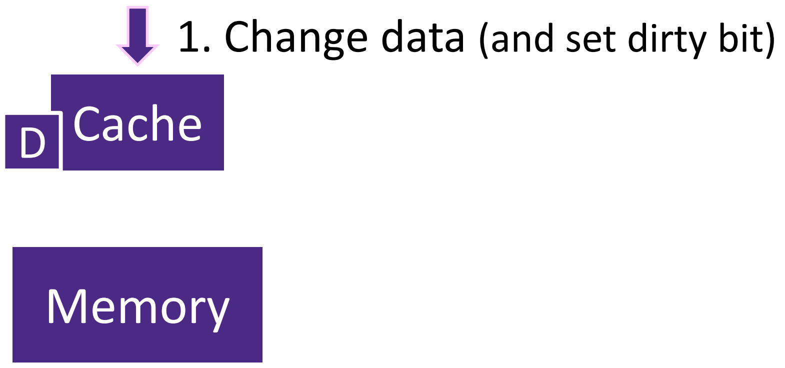
- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

- Where else do you see normative assumptions made in tech or CS? What are the consequences of the "edge case" behaviors in these situations?

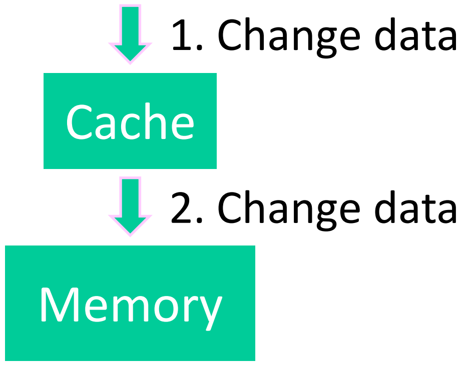
Summary (1/3)

- The 3 C's of cache misses: **compulsory**, **conflict**, and **capacity**
  - There are both parameter and code changes that can help with each kind

- Write-**hit** policies:
  - Write back + write allocate
    - Each line of cache has a *dirty bit*



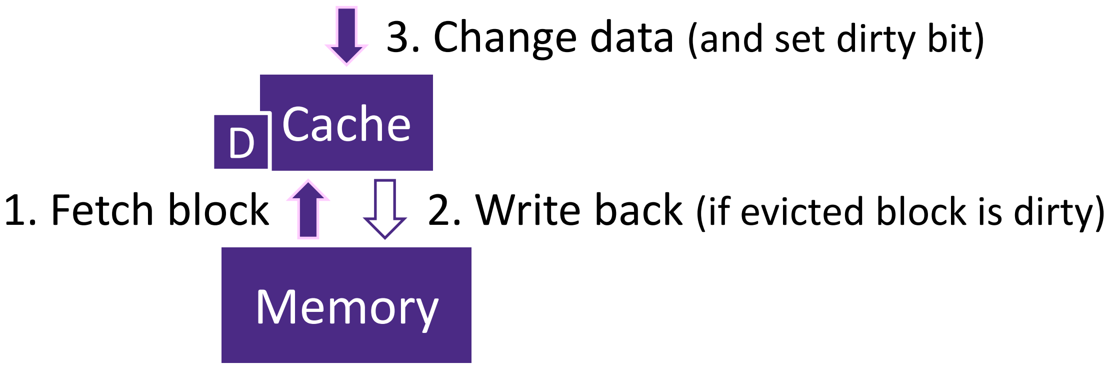
  - Write through + no write allocate



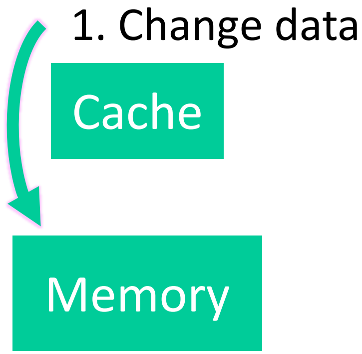
Summary (2/3)

- The 3 C's of cache misses: **compulsory**, **conflict**, and **capacity**
  - There are both parameter and code changes that can help with each kind

- Write-**miss** policies:
  - Write back + write allocate
    - Each line of cache has a *dirty bit*

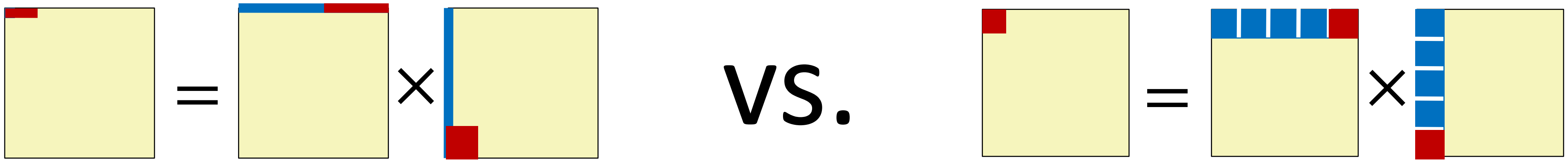


  - Write through + no write allocate



Summary (3/3)

- ***Cache blocking*** is a cache optimization technique that reorders memory accesses to maximize the use of cache blocks while they are in the cache
  - Use data in cache block as much as possible before evicting that block
  - Subdivide larger problem (*e.g.*, matrix multiplication) into smaller ones where *working set* can fit in the cache



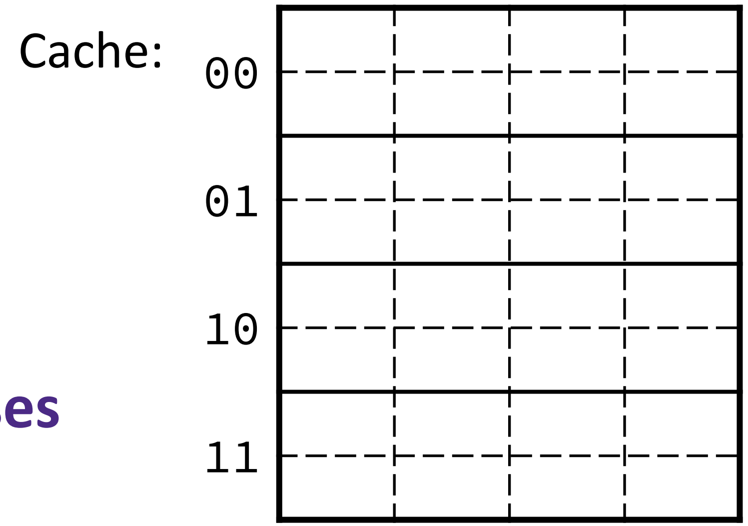
- **Cache-friendly code:**
  - Work with a reasonably small amount of data at any given time
  - Use small strides whenever possible in terms of loop and index ordering
  - Focus your time and energy on optimizing the inner loop code

Bonus Slides!

Cache Miss Examples: Compulsory, Conflict

- Code analysis from last lecture:

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0;  // &ar=0x200
for (int i = 0; i < SIZE; i++)
   for (int j = 0; j < SIZE; j++)
      sum += ar[j][i];
```



  - `ar[0][0]` and `ar[1][0]` are **compulsory misses**
    - Never accessed those blocks before
  - `ar[0][1]` is a **conflict miss**
    - Its block got kicked out earlier by access to `ar[4][0]`
    - Two other sets were unused

Cache Miss Examples: Capacity

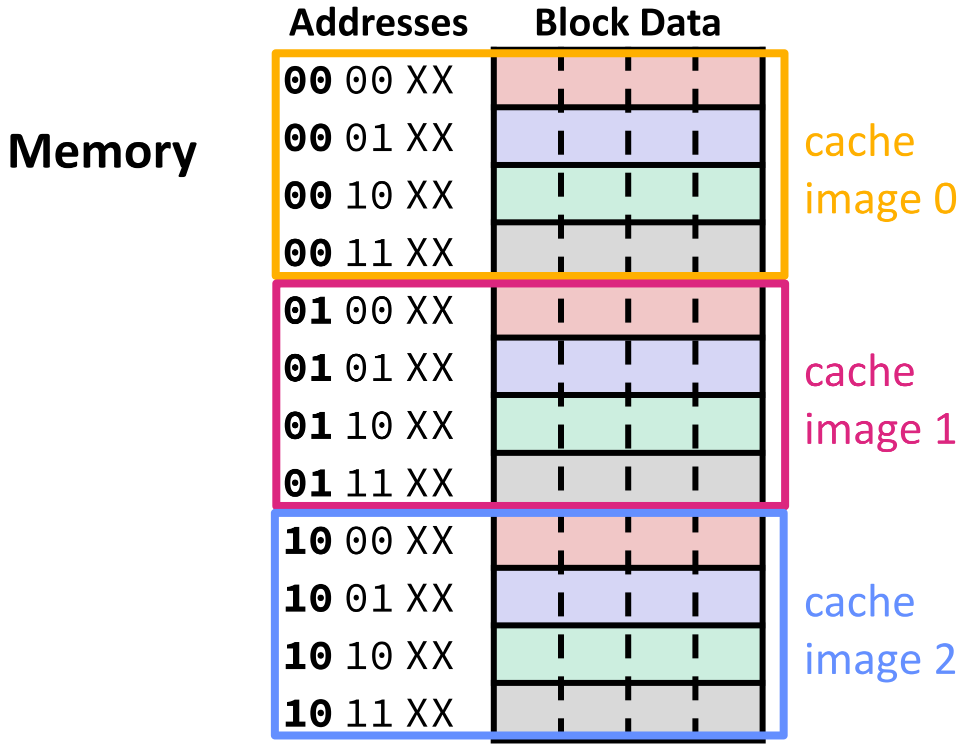
- Assume array `ar_cap` is twice as big as cache:

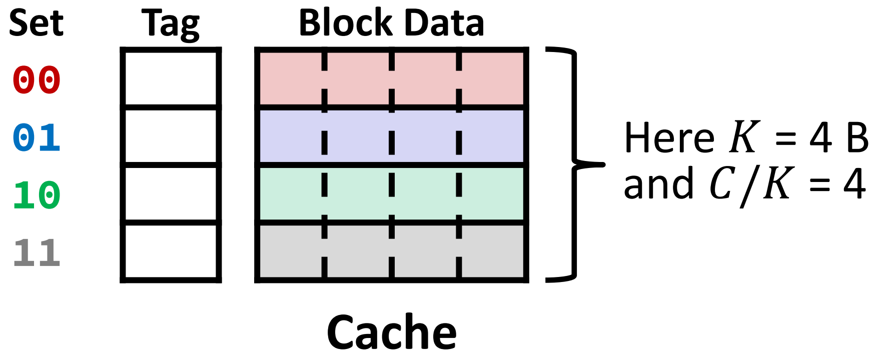
```
for (int i = 0; i < SIZE; i++)
   sum = ar_cap[i];
for (int i = 0; i < SIZE; i++)
   ar_cap[i] = 0;
```

  - All misses in the second loop are **capacity misses**
    - First loop accesses consecutive blocks, so cache is guaranteed to fill up
    - First half of first loop fills cache, second half of first loop completely replaces the first half in cache
    - Second loop revisits block from array that are no longer in the cache

Homework Setup (1/2)

- Homework 19 explores the idea of a ***cache image*** – a view of memory chunking by cache size instead of block size
  - Each cache image maps entirely onto (*i.e.*, exactly fills) the cache
  - Each cache image has a unique *tag* (instead of block number)





Homework Setup (2/2)

e.g., B[0] = A[0];
B[1] = A[1];

e.g., char temp = A[0];
B[1] = A[1];
B[0] = temp;

- Assume our code currently does: ① R 0x00, ② W 0x20, ③ R 0x01, ④ W 0x21
  - What is the current miss rate? 100%
    all accesses go in set 0, alternating between cache images 0 and 2
  - How could we rearrange these accesses to improve our miss rate?
    e.g., R 0x00, R 0x01, W 0x21, W 0x20 ⇒ 50% miss rate

M = miss
H = hit