

Memory & Caches IV

CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu

Assaf Vayner

David Dai

James Froelich

Paul Stevans

Arjun Narendra

Carrie Hu

Dominick Ta

Jenny Peng

Renee Ruan

Armin Magness

Clare Edmonds

Effie Zheng

Kristina Lansang

Vincent Xiao

REFRESH TYPE	EXAMPLE SHORTCUTS	EFFECT
SOFT REFRESH	EMAIL <input type="button" value="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	REMOVELY CYCLES POWER TO DATACENTER
HARDEST REFRESH	CTRL-⌘-⇧-#-R-F5-F5-ESC-O-O-Ø-▲-SCROLL LOCK	INTERNET STARTS OVER FROM ARPANET

Relevant Course Information

- ❖ Lab 4 released today, due Monday, Nov. 28
 - Cache parameter puzzles and code optimizations
- ❖ hw17 due Wed (11/16), hw19 due Fri (11/18)
 - Lab 4 preparation
- ❖ Midterm scores posted
 - See Ed post for common misconceptions
 - Regrade requests open from Nov. 15-17 (Tue-Thu)

Growth vs. Fixed Mindset



- ❖ Students can be thought of as having either a “growth” mindset or a “fixed” mindset (based on research by Prof. Carol Dweck)
 - “In a **fixed mindset** students believe their basic abilities, their intelligence, their talents, are just fixed traits. They have a certain amount and that's that, and then their goal becomes to look smart all the time and never look dumb.”
 - “In a **growth mindset** students understand that their talents and abilities can be developed through effort, good teaching and persistence. They don't necessarily think everyone's the same or anyone can be Einstein, but they believe everyone can get smarter if they work at it.”

Reading Review

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

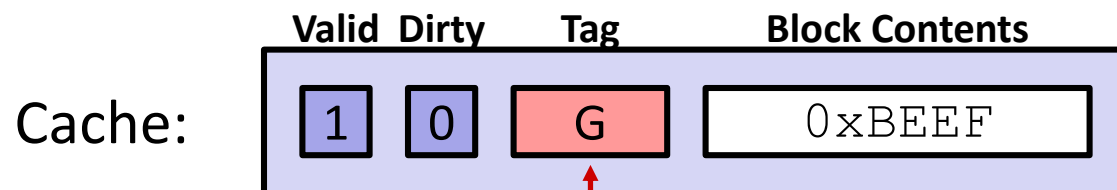
- ❖ Questions from the Reading?

What about writes? (Review)

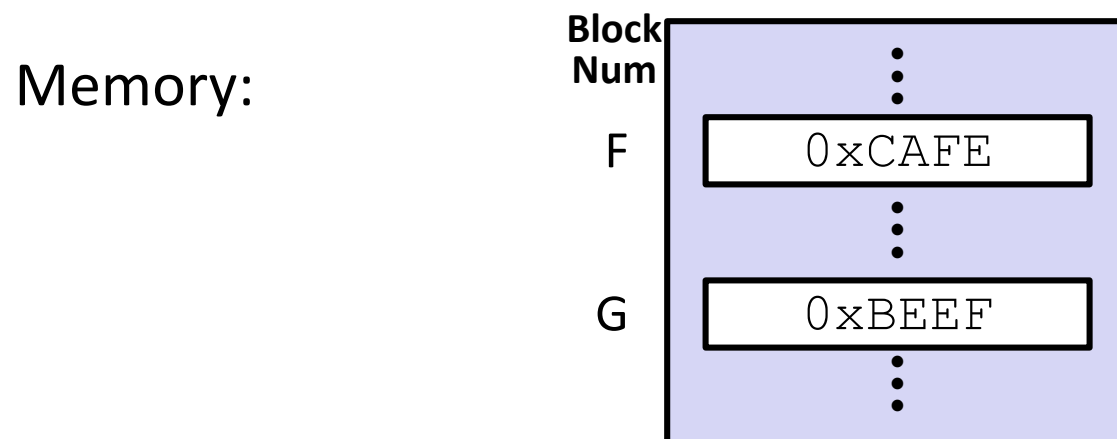
- ❖ Multiple copies of data may exist:
 - multiple levels of cache and main memory
- ❖ What to do on a write-hit?
 - **Write-through**: write immediately to next level
 - **Write-back**: defer write to next level until line is evicted (replaced)
 - Must track which cache lines have been modified (“*dirty 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

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

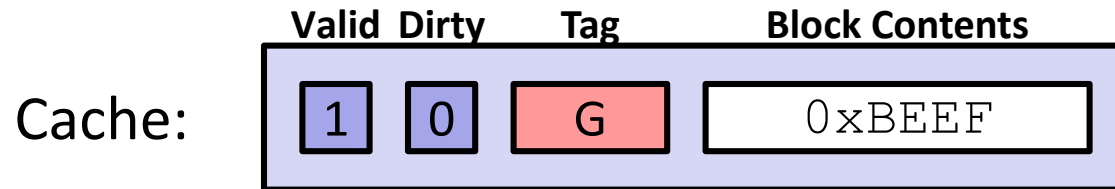


There is only one set in this tiny cache, so the tag is the entire block number!

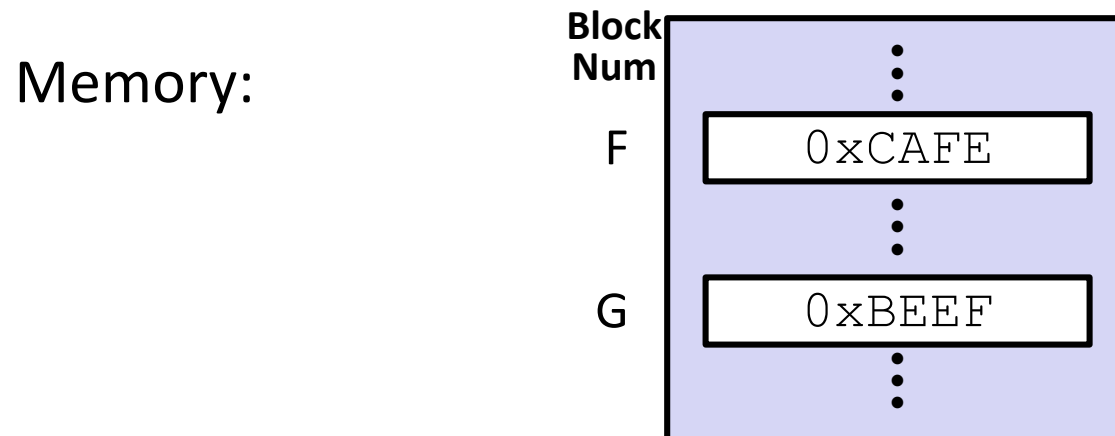


Write-back, Write Allocate Example

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



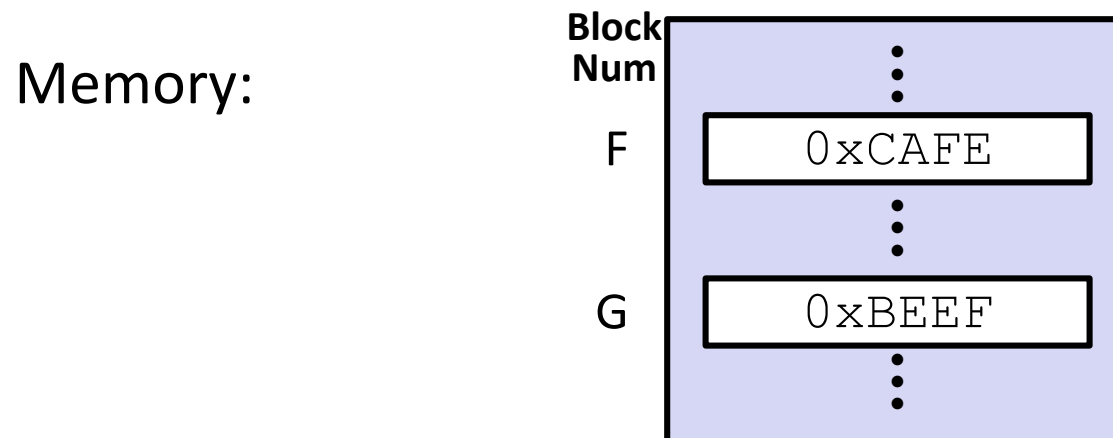
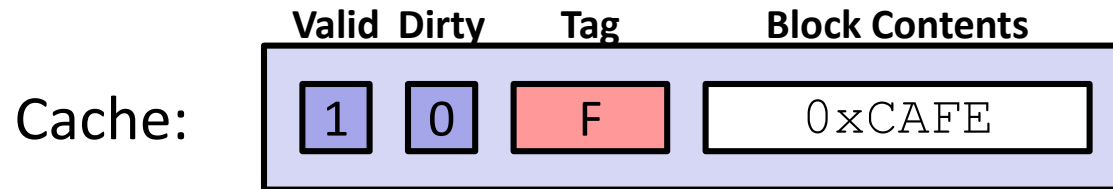
Step 1: Bring F into cache



Write-back, Write Allocate Example

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

Write Miss

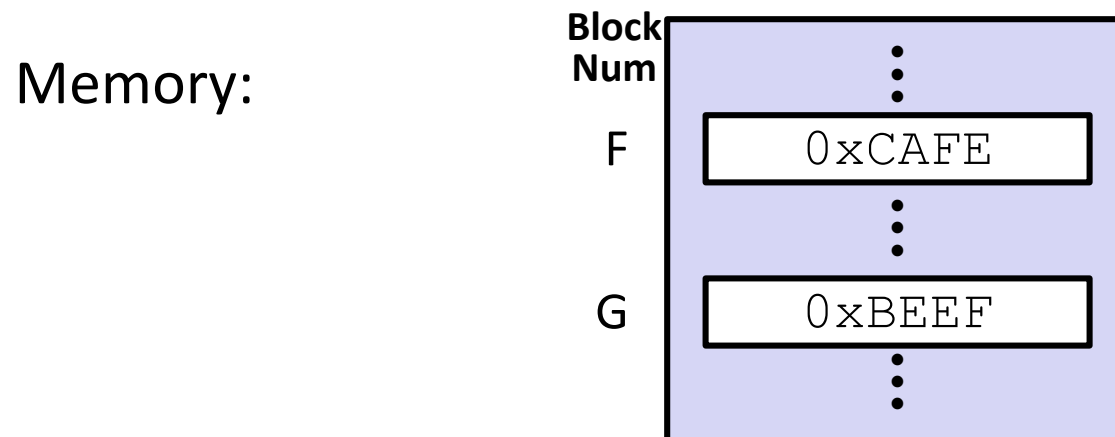
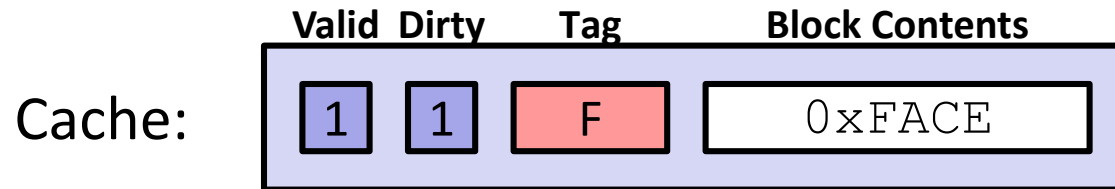


Step 1: Bring F into cache

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

Write-back, Write Allocate Example

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

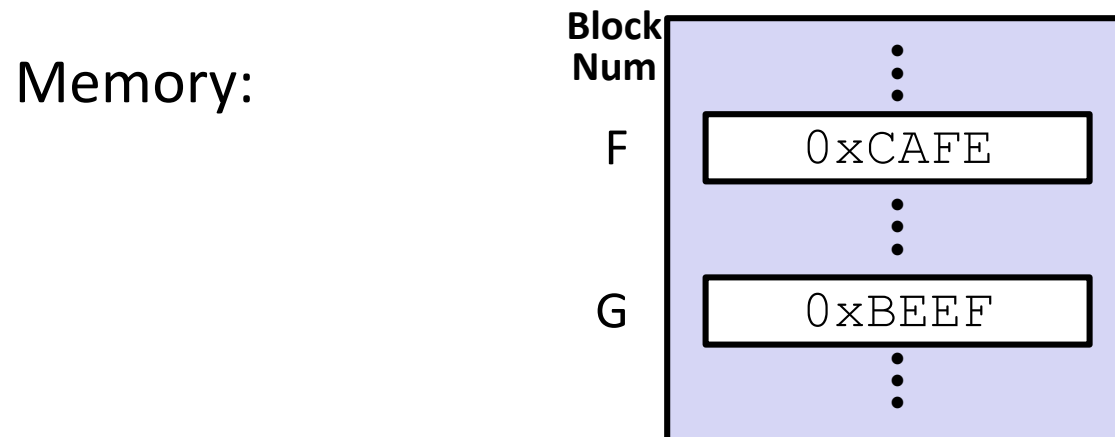
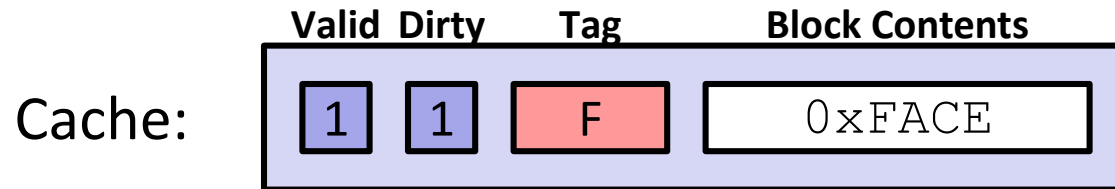


Step 1: Bring F into cache

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

Write-back, Write Allocate Example

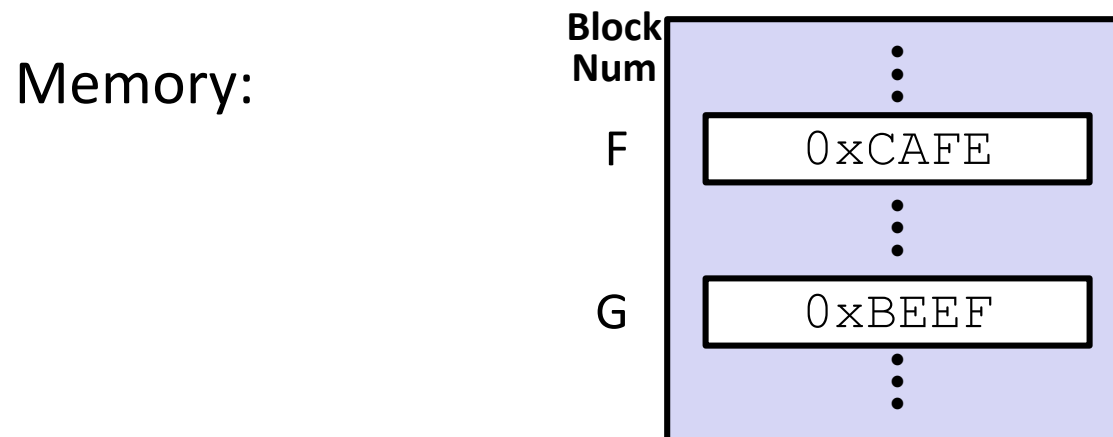
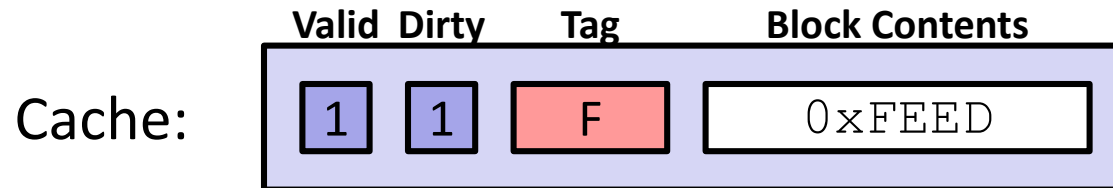
1) `mov $0xFACE, (F)` 2) `mov $0xFEED, (F)`
 Write Miss **Write Hit**



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

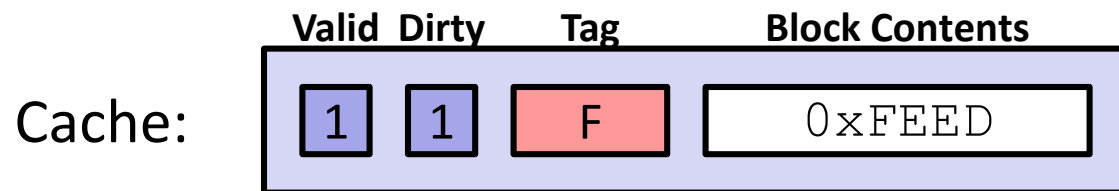


Write-back, Write Allocate Example

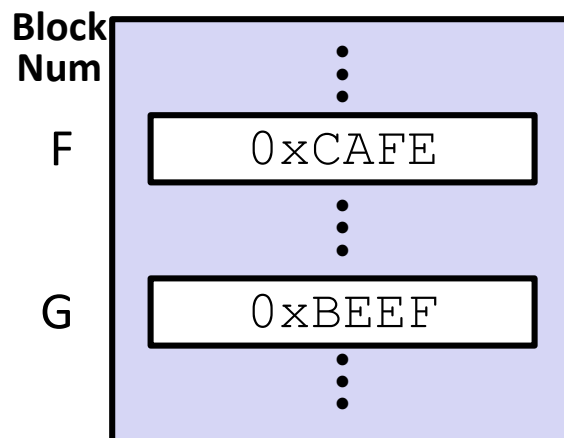
1) `mov $0xFACE, (F)`
Write Miss

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

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



Memory:



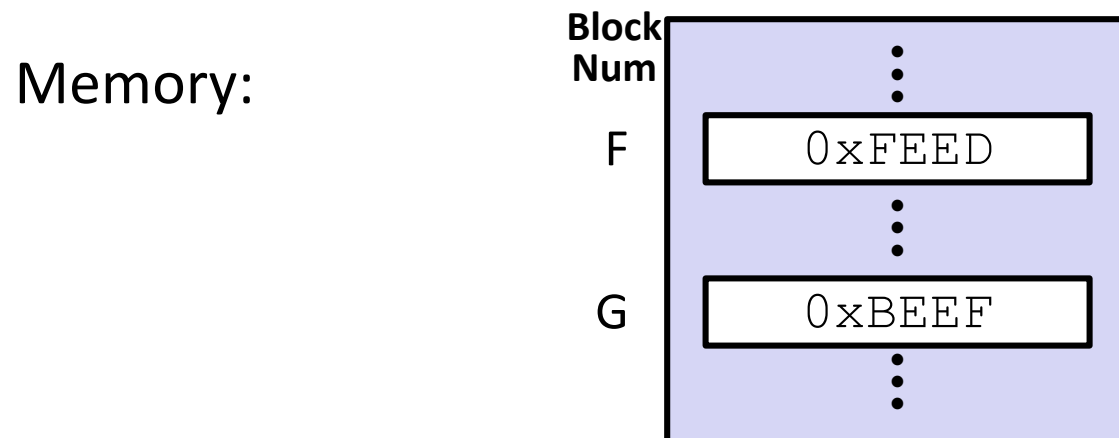
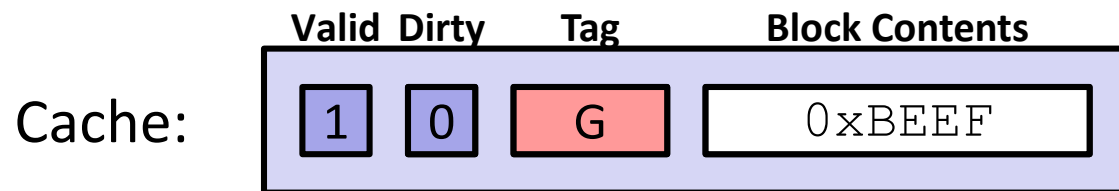
Step 1: Write **F** back to memory since it is dirty

Write-back, Write Allocate Example

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

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

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



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

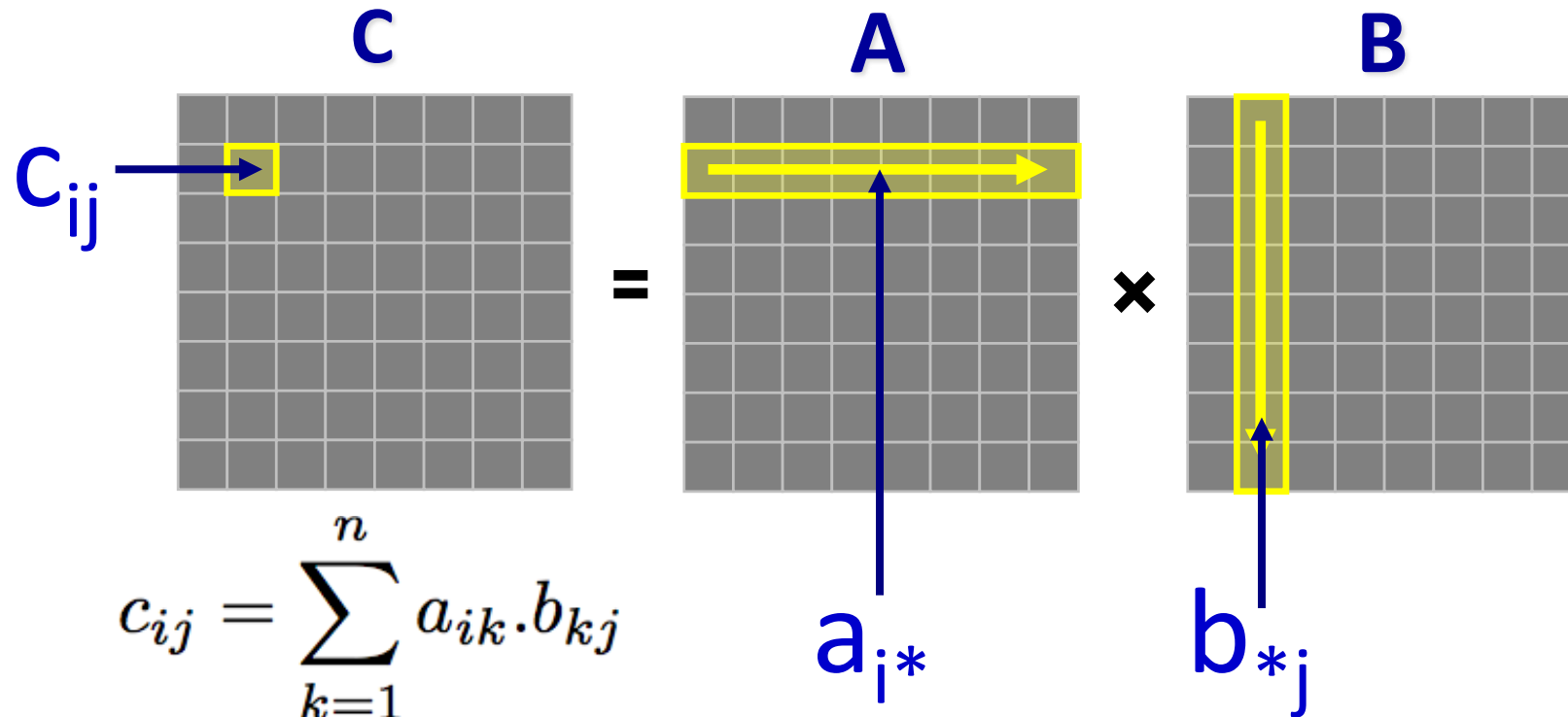
- ❖ Which of the following cache statements is FALSE?
 - Vote in Ed Lessons
 - A. **We can reduce compulsory misses by decreasing our block size**
 - B. **We can reduce conflict misses by increasing associativity**
 - C. **A write-back cache will save time for code with good temporal locality on writes**
 - D. **A write-through cache will always match data with the memory hierarchy level below it**
 - 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

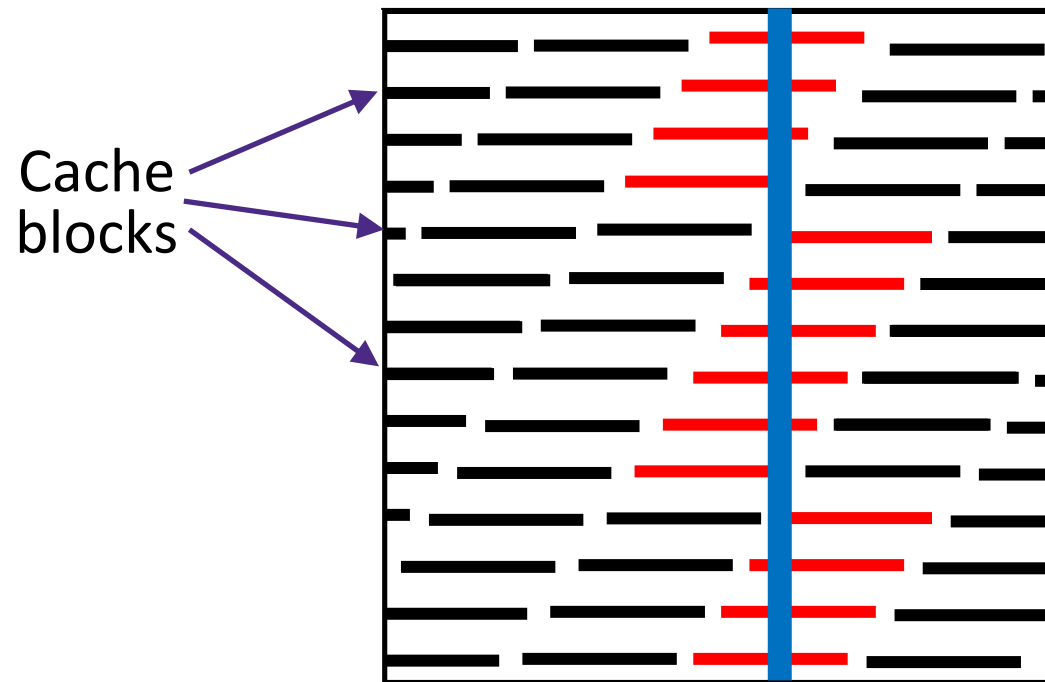
- ❖ How can you achieve locality?
 - Adjust memory accesses in *code* (software) to improve miss rate (MR)
 - Requires knowledge of *both* how caches work as well as your system's parameters
 - Proper choice of algorithm
 - Loop transformations

Example: Matrix Multiplication



Matrices in Memory

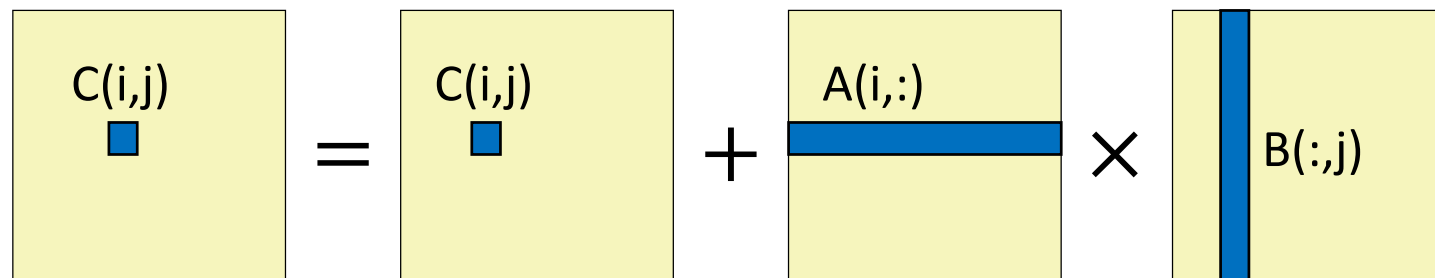
- ❖ How do cache blocks fit into this scheme?
 - Row major matrix in memory:



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



Cache Miss Analysis (Naïve)

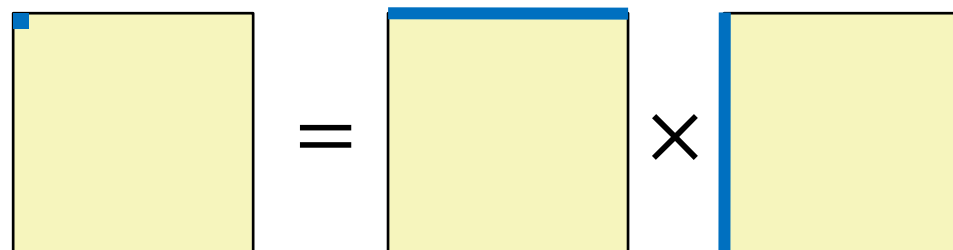
Ignoring
matrix C

❖ Scenario Parameters:

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

❖ Each iteration:

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



Cache Miss Analysis (Naïve)

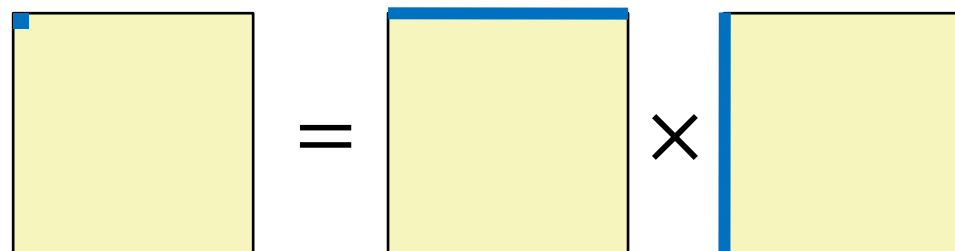
Ignoring matrix C

❖ Scenario Parameters:

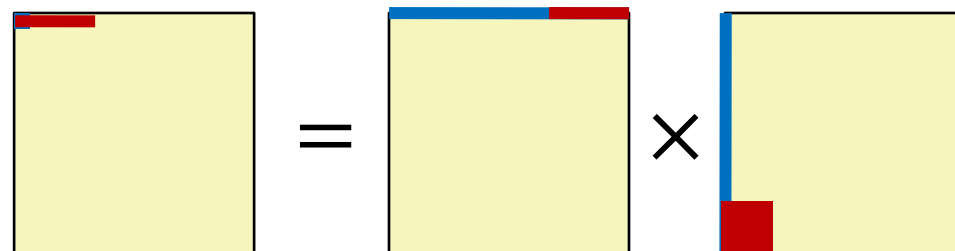
- Square matrix ($n \times n$), elements are doubles
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size is much smaller than n

❖ Each iteration:

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



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



8 doubles wide

Cache Miss Analysis (Naïve)

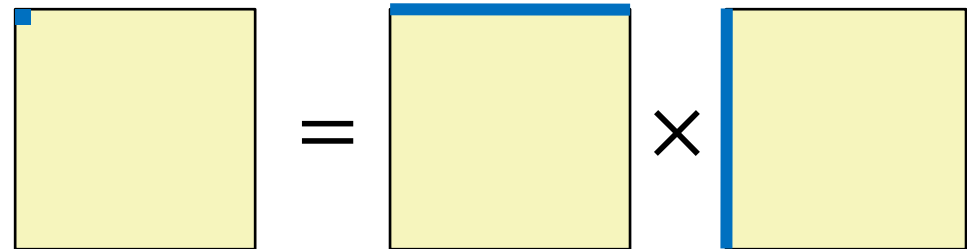
Ignoring
matrix C

❖ Scenario Parameters:

- Square matrix ($n \times n$), elements are doubles
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size 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, 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_{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][jb] += A[ib][kb] * B[kb][jb];
```

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

Cache Miss Analysis (Blocked)

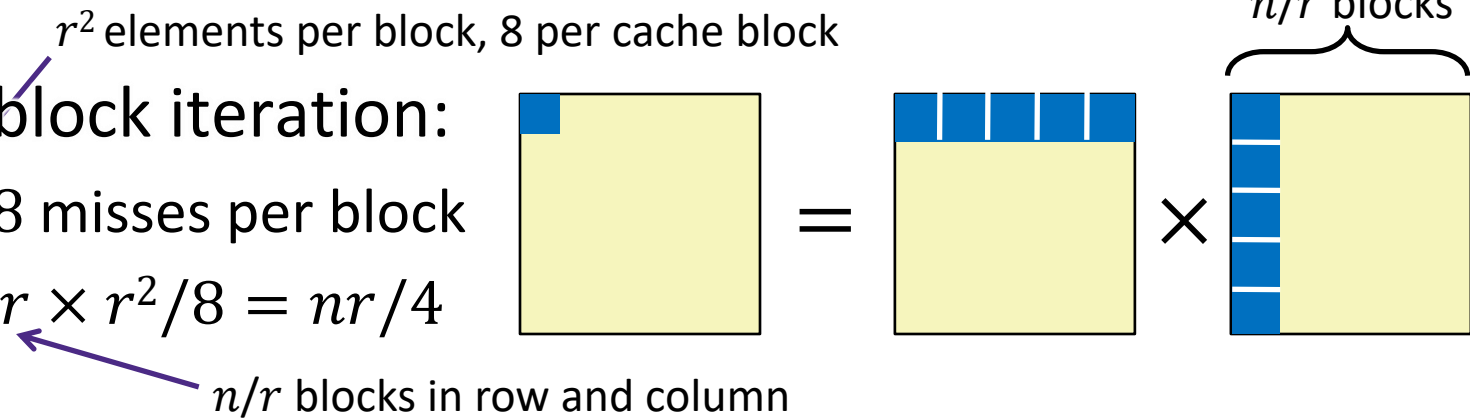
Ignoring matrix C

❖ Scenario Parameters:

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

❖ Each block iteration:

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



Cache Miss Analysis (Blocked)

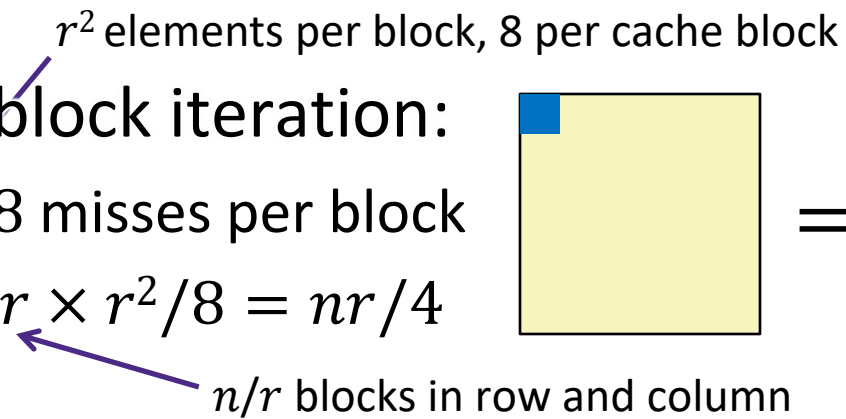
Ignoring matrix C

❖ Scenario Parameters:

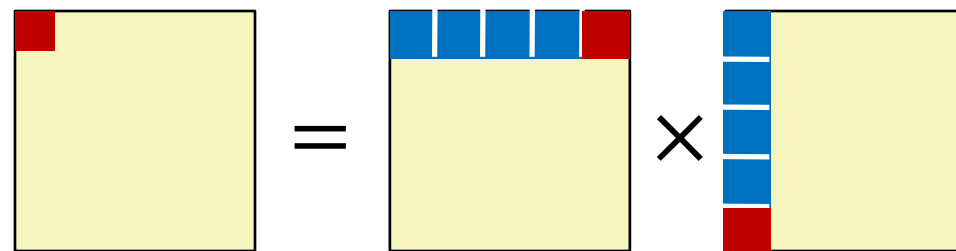
- Cache block size $K = 64 \text{ B} = 8 \text{ doubles}$
- Cache size is much smaller than n
- Three blocks \blacksquare ($r \times r$) fit into cache: $3r^2 < \text{cache size}$

❖ Each block iteration:

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



- Afterwards in cache (schematic)



Cache Miss Analysis (Blocked)

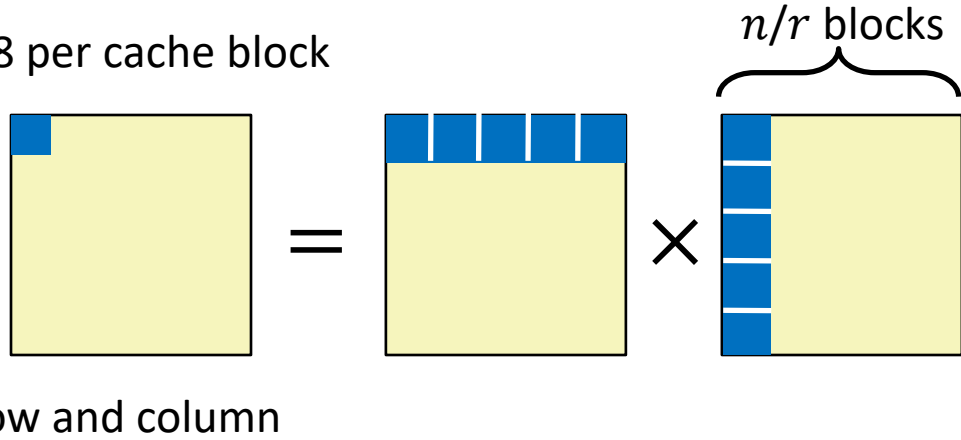
Ignoring matrix C

❖ Scenario Parameters:

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

❖ Each block iteration:

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



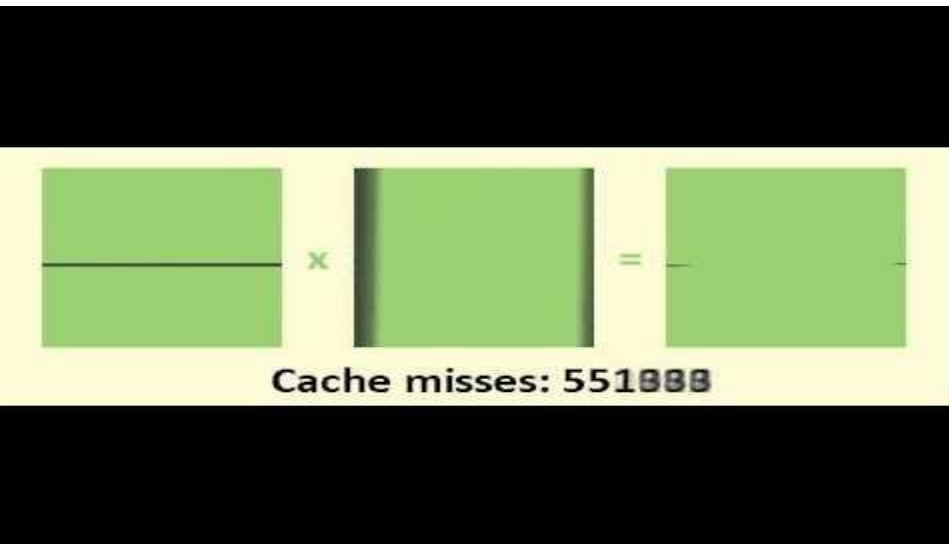
❖ Total misses:

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

Matrix Multiply Visualization

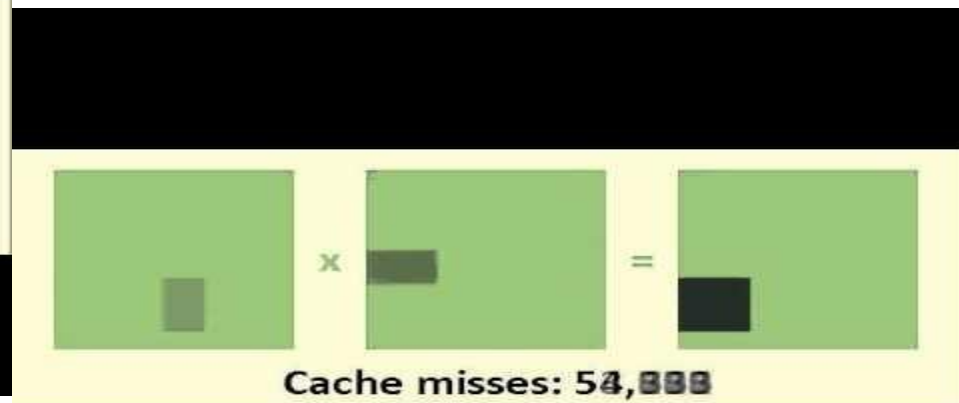
❖ Here $n = 100$, $C = 32$ KiB, $r = 30$

Naïve:



$\approx 1,020,000$
cache misses

Blocked:



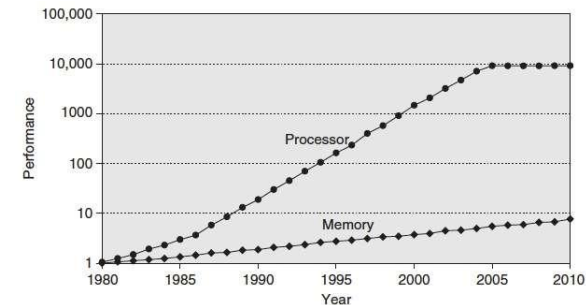
$\approx 90,000$
cache misses

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


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?**

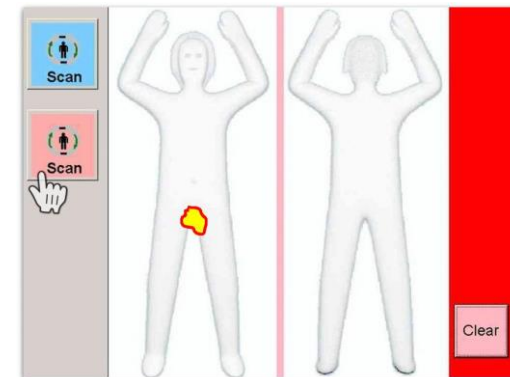
The Common Case and Normativity

- ❖ “**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 how the traveler is presenting:*



- ❖ 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
- ❖ Discuss: Where else do you see normative assumptions made in tech or CS?