Caches IV

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

Administrivia

- Nothing due tonight!
- Sunday, 7/28
  - **Lab3 due (11:59pm)**
- Monday, 7/29
  - HW15-16 due (11:59pm)
  - **Quiz 2 due (11:59pm)**

I made some mistakes in the last lecture (I was… very tired)

- Ed slides have been fixed, and we'll go over the last problem again today

Code Analysis

- Assuming cache starts **cold** (i.e. all blocks invalid), and `sum`, `i`, and `j` are all stored in registers, calculate the **miss rate**.
  - ***m*** = 10 bits, ***C*** = 64B, ***K*** = 8B, ***E*** = 2

$S = C \div K \div E = 4$ sets

8 sets = 16B per row

```
#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 = ar[0]

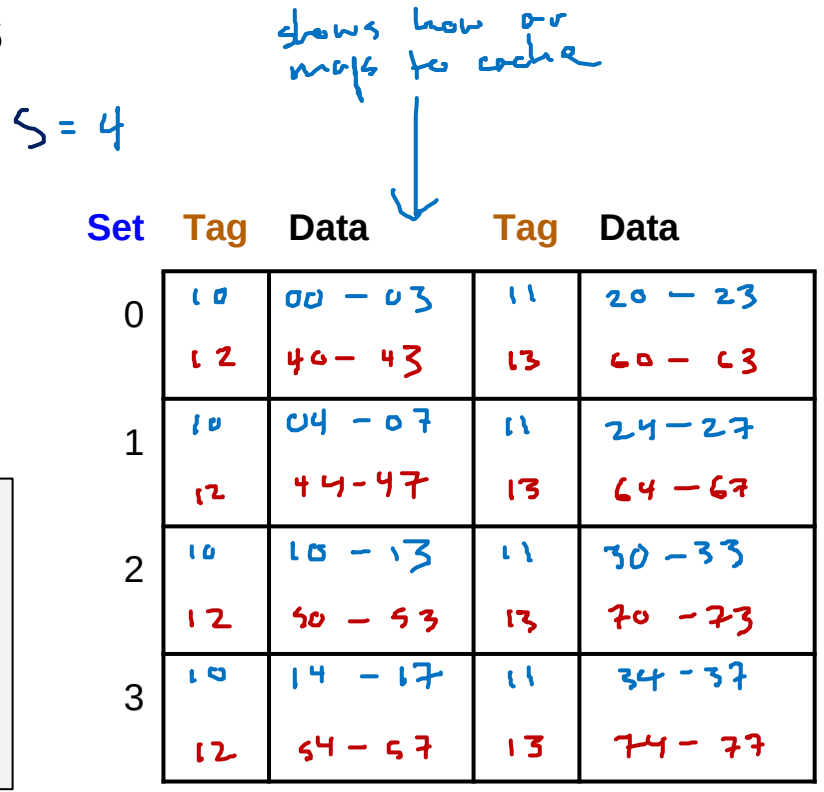
Other relevant addrs:
ar[1] = 0x210
ar[2] = 0x220
ar[3] = 0x230
...

code visits each element of array in columns

Code Analysis: relevant values

- ***m*** = 10 bits, ***C*** = 64B, ***K*** = 8B, ***E*** = 2
  - ***k*** = 3, ***s*** = 2
- 8B blocks = 4 `shorts` per block
- Starting address = `0b10000 00 000`
  - Block stored in set 0, tag = `0b10000` = `0x10`

```
#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];
}
```



Code Analysis: step 1

`i = 0, j = 0`

Misses: 1
Hits: 0

- Access `ar[0][0]`
  - `0b10000 00 000`
    - Miss! compulsory
  - Load block into set 0 with tag 0x10

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]		
1				
2				
3				

```
#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];
}
```

Code Analysis: step 2

`i = 0, j = 1`

Misses: 2
Hits: 0

- Access `ar[1][0]`
  - `0b10000 10 000`
    - Miss! Compulsory
  - Load block into set 2 with tag 0x10

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]		
1				
2	10	a[1][0] … a[1][3]		
3				

```
#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];
}
```

Code Analysis: step 3

`i = 0, j = 2`

Misses: 3
Hits: 0

- Access `ar[2][0]`
  - `0b10001 00 000`
    - Miss! Compulsory
  - Load block into set 0 with tag 0x11
    - Can store both blocks in set 0 because of associativity

```
#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];
}
```

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]		
3				

Code Analysis: step 4

`i = 0, j = 3`

Misses: 4
Hits: 0

- Access `ar[3][0]`
  - `0b10001 10 000`
    - Miss! Compulsory
  - Load block into set 1 with tag 0x11

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]	11	a[3][0] … a[3][3]
3				

```
#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];
}
```

Code Analysis: step 5

`i = 0, j = 4`

Misses: 5
Hits: 0

- Access `ar[4][0]`
  - `0b10010 00 000`
    - Miss! compulsory
  - Load block into set 0 with tag 0x12
    - Evicts least recently used block

(a[0][0] - a[0][3])

```
#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];
}
```

Set	Tag	Data	Tag	Data
0	12	a[4][0] … a[4][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]	11	a[3][0] … a[3][3]
3				

Code Analysis: step 6-8

`i = 0, j = 5…`

Misses: 8
Hits: 0

- Same as step 5
  - Accesses to `a[5][0]`, `a[6][0]`, and `a[7][0]` will kick out the old blocks in the cache ↑ 3x compulsory misses
- So for i = 0:
  - 8 accesses total (j = 0…7), 8 misses

```
#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];
}
```

Set	Tag	Data	Tag	Data
0	12	a[4][0] … a[4][3]	13	a[6][0] … a[6][3]
1				
2	12	a[5][0] … a[5][3]	13	a[7][0] … a[7][3]
3				

Code Analysis: step 9

`i = 1, j = 0`

Misses: 9
Hits: 0

- Access `ar[0][1]`
  - `0b10000 00 010`
    - Same block that we loaded in in step 1, but it got evicted in step 5!
    - Miss! conflict
  - Load block back into set 0

```
#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];
}
```

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	13	a[6][0] … a[6][3]
1				
2	12	a[5][0] … a[5][3]	13	a[7][0] … a[7][3]
3				

Code Analysis: step 10+

- All future accesses will continue to follow this pattern
  - Each block is loaded in, then kicked out of the cache before it's accessed again
- **Miss rate 100%!**
- How can we fix this?

```
#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];
}
```

Improving Cache Performance

- Reduce **stride**
  - I.e. access data that's closer together

```
#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[i][j];
}
```

Everything else is the same

Access each element in rows

Improving Cache Performance Example: step 1

Misses: 1
Hits: 3

- First 4 accesses:
  - `ar[0][0]`: miss, load block into the cache
  - `ar[0][1]`: hit!
  - `ar[0][2]`: hit!
  - `ar[0][3]`: hit!

compulsory

```
#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[i][j];
}
```

Set	Tag	Data	Tag	Data
0	10	ar[0][0] … ar[0][3]		
1				
2				
3				

Improving Cache Performance Example: step 2

Misses: 2
Hits: 6

- Next 4 accesses:
  - `ar[0][4]`: miss, load block into the cache
  - `ar[0][5]`: hit!
  - `ar[0][6]`: hit!
  - `ar[0][7]`: hit!

compulsory

```
#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[i][j];
}
```

Set	Tag	Data	Tag	Data
0	10	ar[0][0] … ar[0][3]		
1	10	ar[0][4] … ar[0][7]		
2				
3				

Improving Cache Performance Example: step 3+

- All accesses follow this pattern
  - Because we use a whole block before moving on, we will miss 1 out of every 4 accesses
- **Miss rate: 25%**

```
#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[i][j];
}
```

Caches

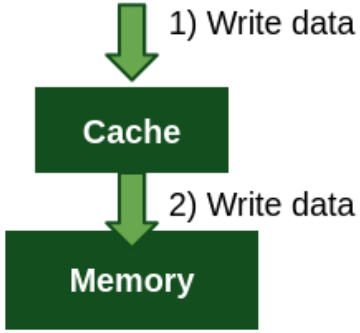
- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - **Handling writes**
- Program optimizations that consider caches

Write-Hit Policies

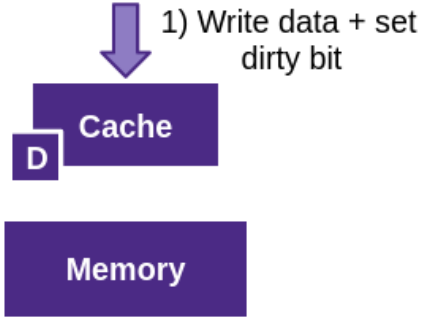
i.e. memory or another cache level

What to do if the data is already in the cache?

- **Write-through**: immediately write to the next level
- **Write-back**: don't write to next level until we have to
  - Keep track of **dirty bit** for each block
  - On eviction, if dirty bit is set, write contents back to memory



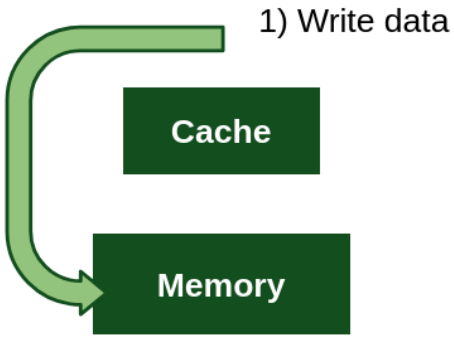
Write-through



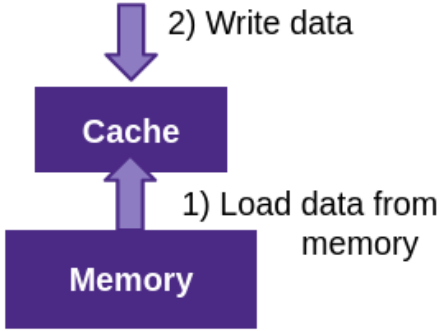
Write-back

Write-Miss Policies

- What to do if the block we want to write to is not in the cache?
- **No-write-allocate** (“write around”): don’t load into the cache, just write to the next level
- **Write-allocate** (“fetch on write”) load data into the cache before writing



No-write-allocate



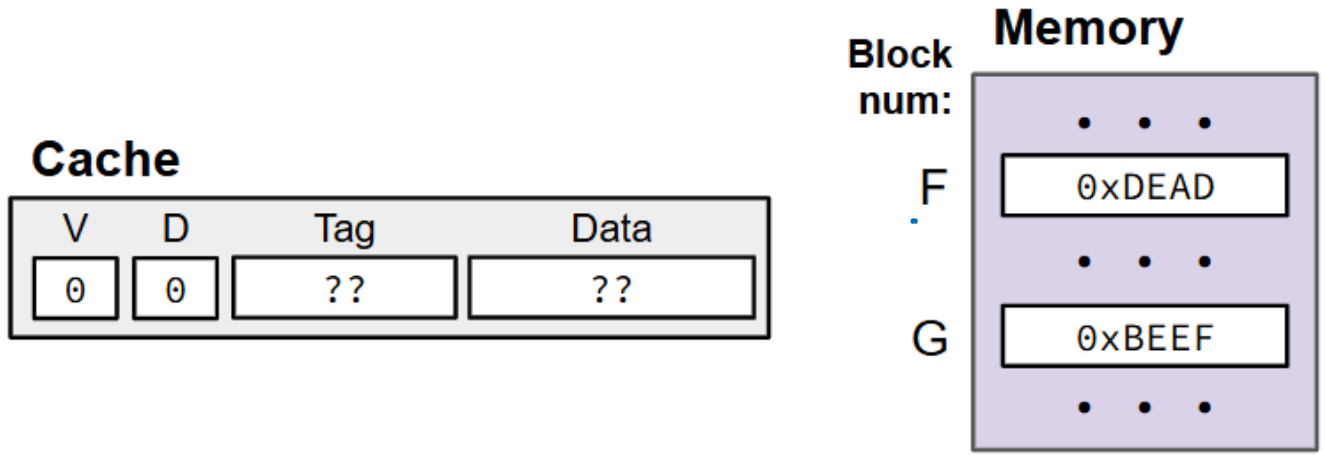
Write-allocate

Ex: Write-Back, Write-Allocate

- Single-block mini cache
  - Tag includes the entire block number
  - *Not a realistic example*

hit policy → **Write-back:** defer write to next level until line is evicted

miss policy → **Write-allocate:** on a miss, bring the data into cache



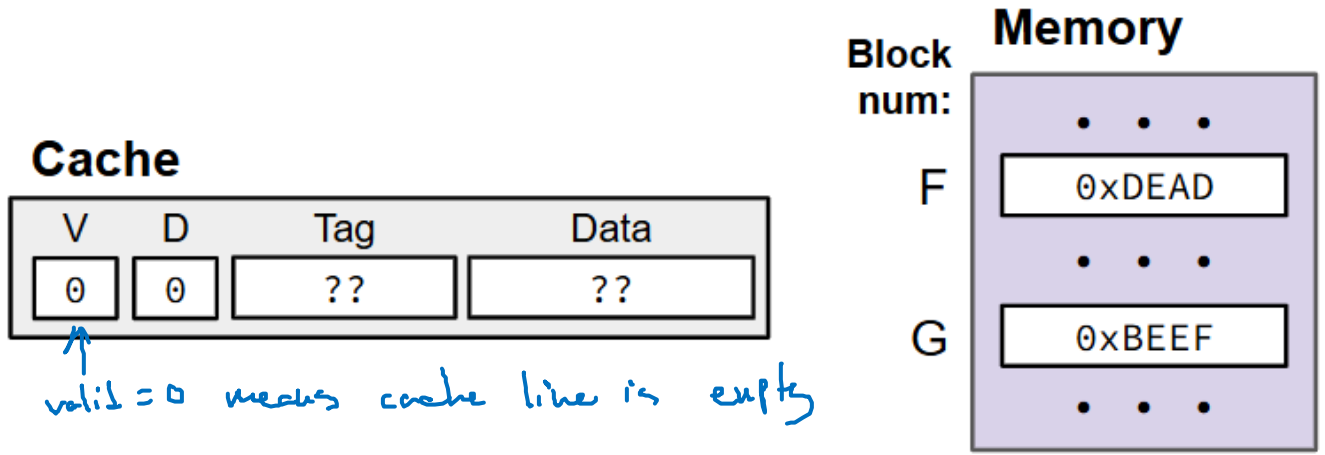
Ex: Write-Back, Write-Allocate (pt 2)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1. `mov $0xB0BA, (F)`

Write miss

Not valid x86. Assume we mean an address within block F.



Ex: Write-Back, Write-Allocate (pt 3)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1. `mov $0xB0BA, (F)`

Write miss

because of write-allocate policy

1. Bring F into cache

Cache

V	D	Tag	Data
1	0	F	0xDEAD

Memory

Block num:	
	• • •
F	0xDEAD
	• • •
G	0xBEEF
	• • •

Ex: Write-Back, Write-Allocate (pt 4)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1. `mov $0xB0BA, (F)`

Write miss

1. Bring F into cache
2. Write `0xB0BA` into cache *only*
   a. Set dirty bit

because of write-back policy

Cache

V	D	Tag	Data
1	1	F	0xB0BA

memory keeps old value

Memory

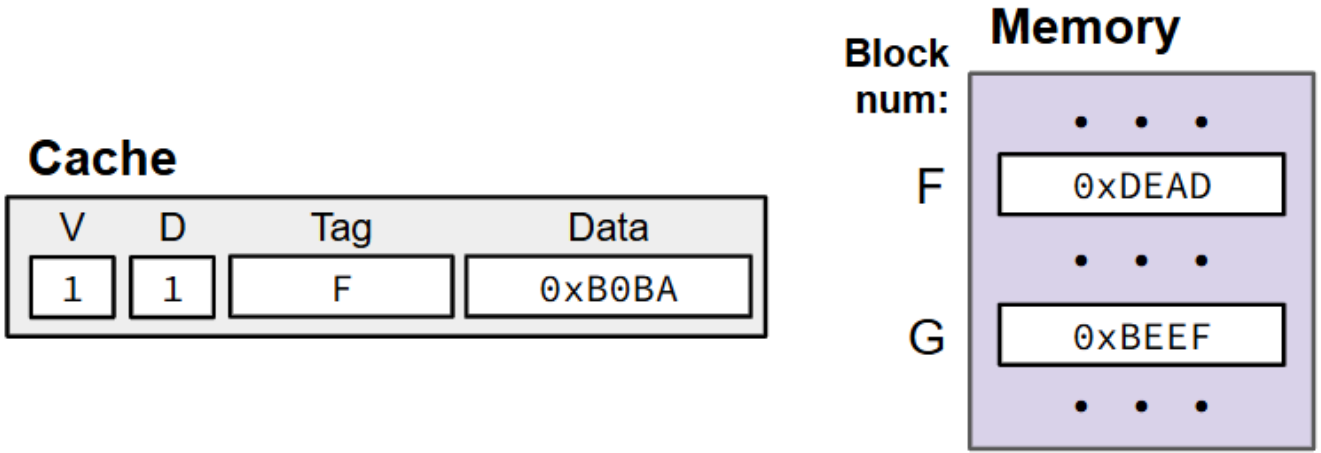
Block num:	
F	0xDEAD
G	0xBEEF

Ex: Write-Back, Write-Allocate (pt 5)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

2. `mov $0xF00D, (F)`

Write hit



Ex: Write-Back, Write-Allocate (pt 6)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

2. `mov $0xF00D, (F)`

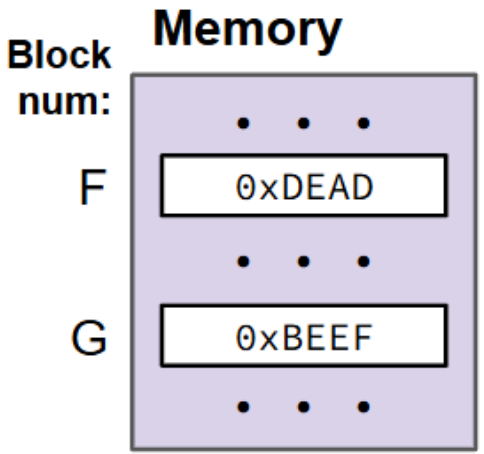
Write hit

because of write-back

1. Write `0xF00D` into cache *only*
   a. Set dirty bit

Cache

V	D	Tag	Data
1	1	F	0xF00D

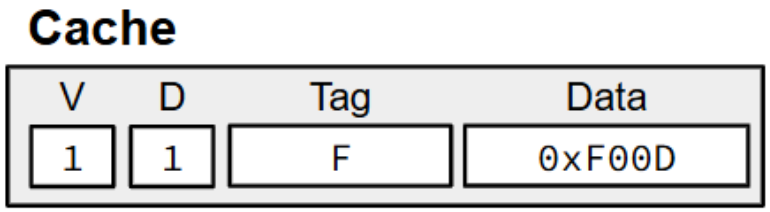


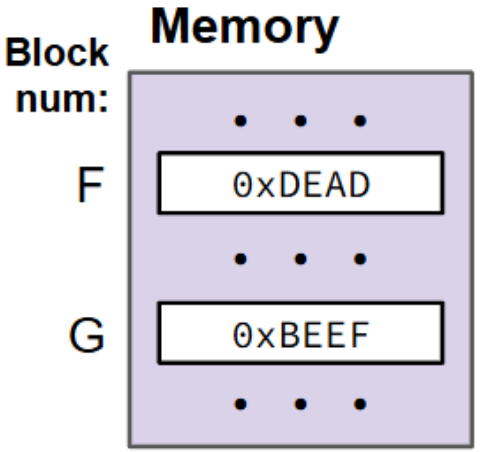
Ex: Write-Back, Write-Allocate (pt 7)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

3. `mov (G), %ax`

Read miss





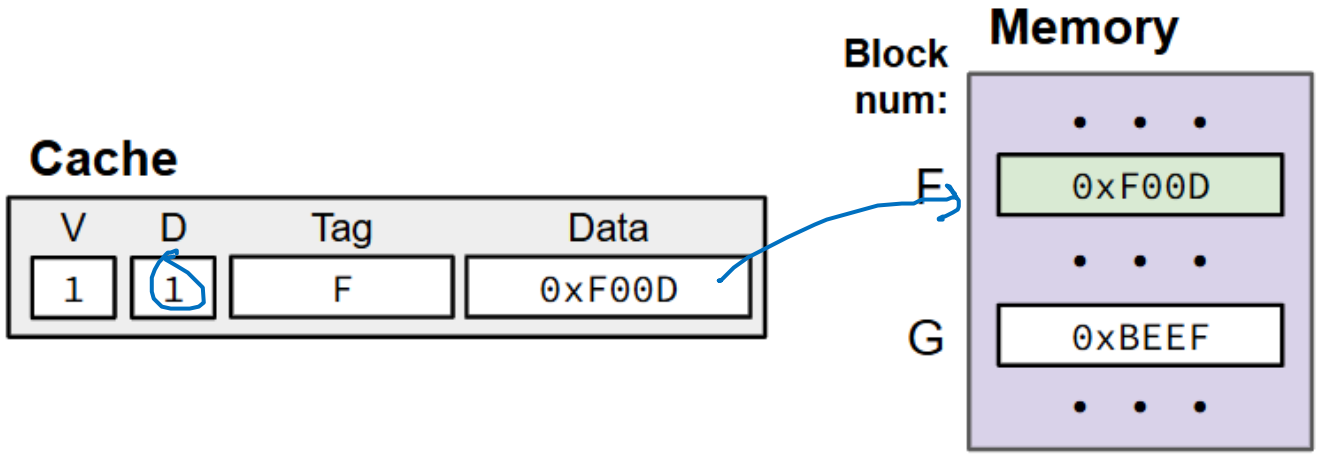
Ex: Write-Back, Write-Allocate (pt 8)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

3. `mov (G), %ax`

Read miss

1. Write F back to memory since it is dirty



Ex: Write-Back, Write-Allocate (pt 9)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

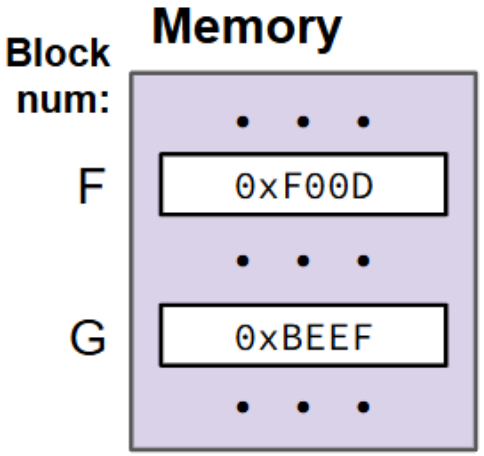
3. `mov (G), %ax`

Read miss

1. Write F back to memory since it is dirty
2. Bring G into cache

Cache

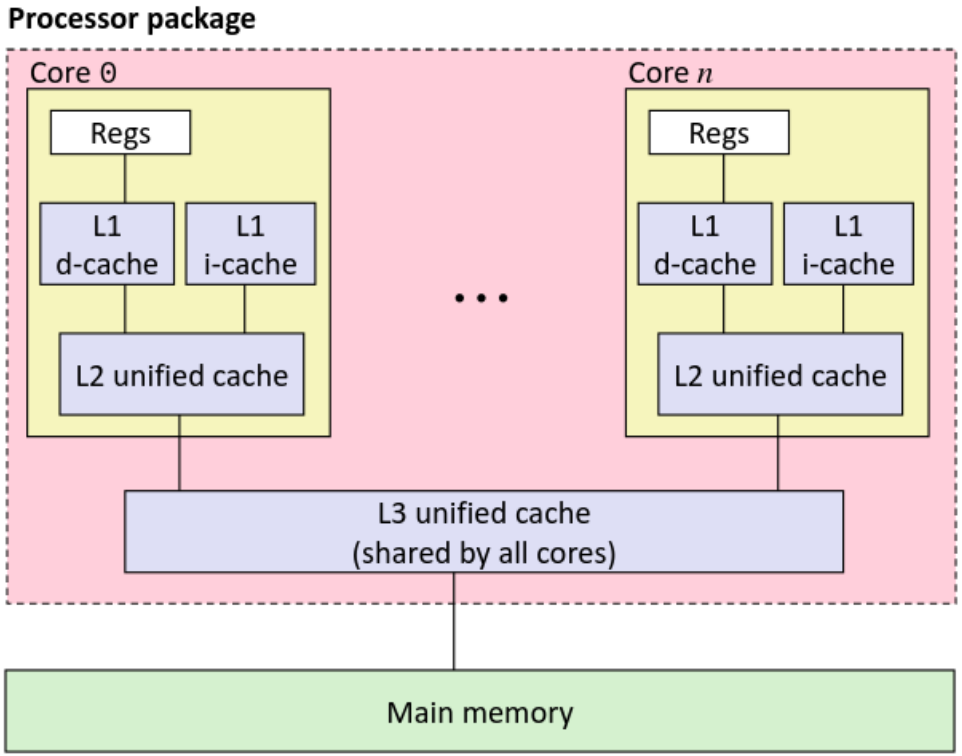
V	D	Tag	Data
1	0	G	0xBEEF



Common Policies

- **Write-back + Write-allocate**
  - (most common)
- Write-through + No-write-allocate
  - When would this be used?

Ex: L2 cache may be write-through so that other cores can access data through shared L3



Polling Question

definition of write-through

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

actually the opposite! ↓ block size = need to load in more blocks

can hold more blocks per set = fewer evictions

only need to write to memory once, instead of for each write

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 in section
  - Will be used in HW17 – Lab 4 Preparation

Caches

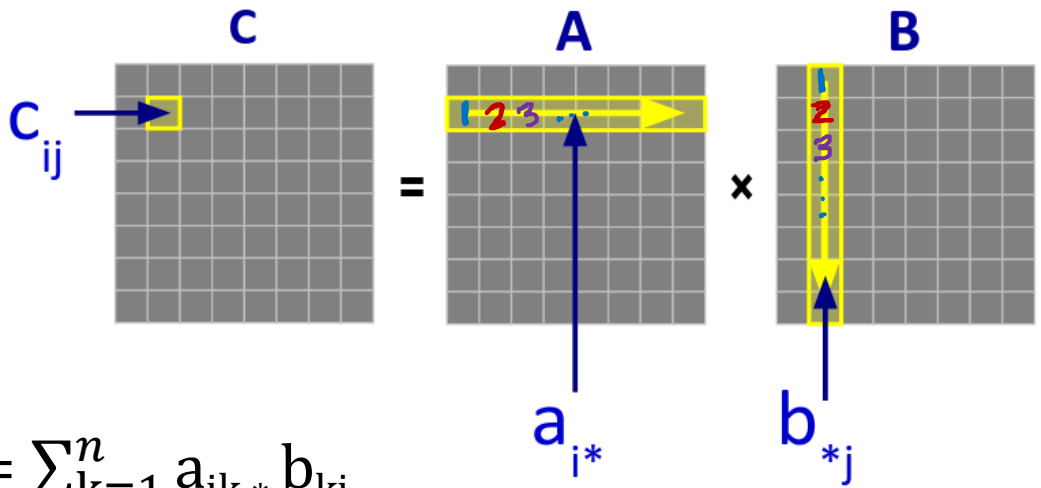
- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- **Program optimizations that consider caches**

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 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

C[i][j] = A[i][1] · B[1][j] + A[i][2] · B[2][j] + A[i][3] · B[3][j] + ...

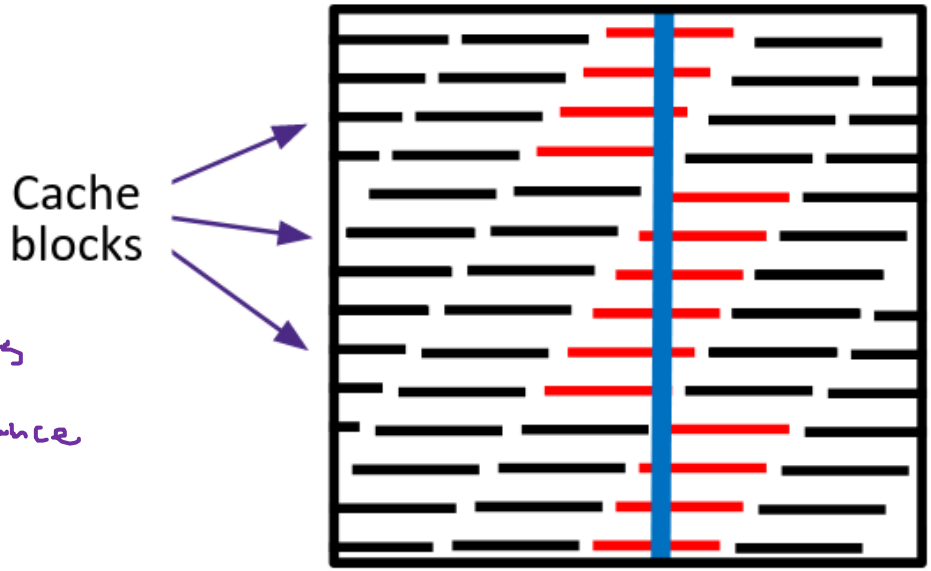


Go through row of A and col of B:
- multiply values
- sum of all results = 1 element of resulting matrix

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

Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row-major in memory
  - Column of matrix is spread across multiple cache blocks

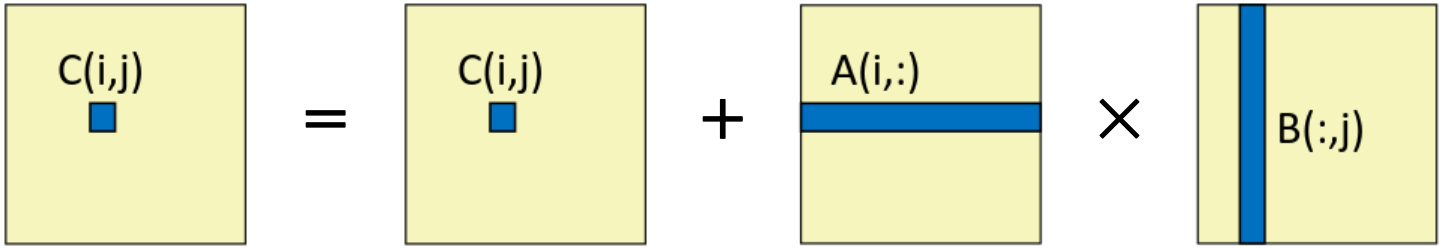


i.e. – moving by columns gets worse cache performance

Naive Matrix Multiply

```
// move along rows of A
for (i = 0; i < n; i++) {
  // move along columns of B
  for (j = 0; j < n; j++) {
    // Reads row of A, col of B
    // 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];
  }
}
```

How many memory accesses are in this line?



Cache Miss Analysis (Naive)

Ignoring misses in matrix C

- Example parameters:
  - A and B are square ($n \times n$) matrices of `doubles`
  - Cache block size ***K*** = 64B
    - 8 elements per block
  - Cache size **C** << *n* (much smaller)
- For each element of C
  - *n*/8 misses per row of A
  - *n* misses per column B
  - 9*n*/8 misses per iteration
- n^2 elements in total
  - **$9n^3/8$ misses**

going by rows gets optimal miss rate

every access misses

$= n + \frac{n}{8}$

$= \frac{9n}{8} \cdot n^2$

Linear Algebra To The Rescue

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (“blocks”)
- Example: multiply 2 4×4 matrices

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

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

Don't need to know the details, what's important is that we can split the matrix into smaller chunks

Linear Algebra To The Rescue (pt 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}

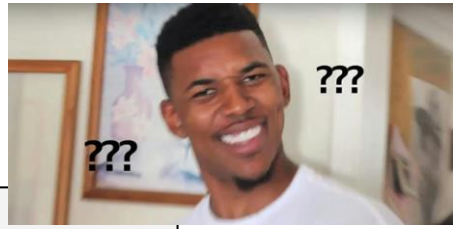
B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

- Split up each matrix into smaller blocks
  - Choose a block size that will fit in the cache!
  - This technique is called **cache blocking**
- Perform multiplication block-by-block (rather than row-by-row)

similar alg as earlier, but now each of these is a mini-matrix

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_{k=1}^{n} A_{2k} * Bk_2$$

Blocked Matrix Multiply



- r = length of a block (assume r divides n evenly)

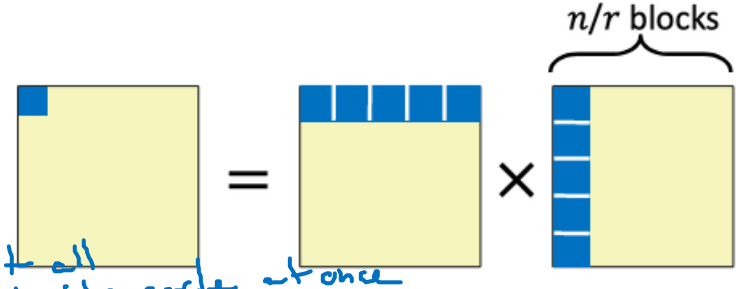
```
// Go through each rxr block of C
for (i = 0; i < n; i += r)
    for (j = 0; j < n; j += r)
        // Go through each block A(i, k) and B(k, j)
        for (k = 0; k < n; k += r)
            // Multiply the two blocks (same as naive algorithm)
            for (ib = i; ib < i+r; ib++)
                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];
```

same as before, but because matrices are smaller, better cache performance

Cache Miss Analysis (Blocked)

Ignoring misses in matrix C

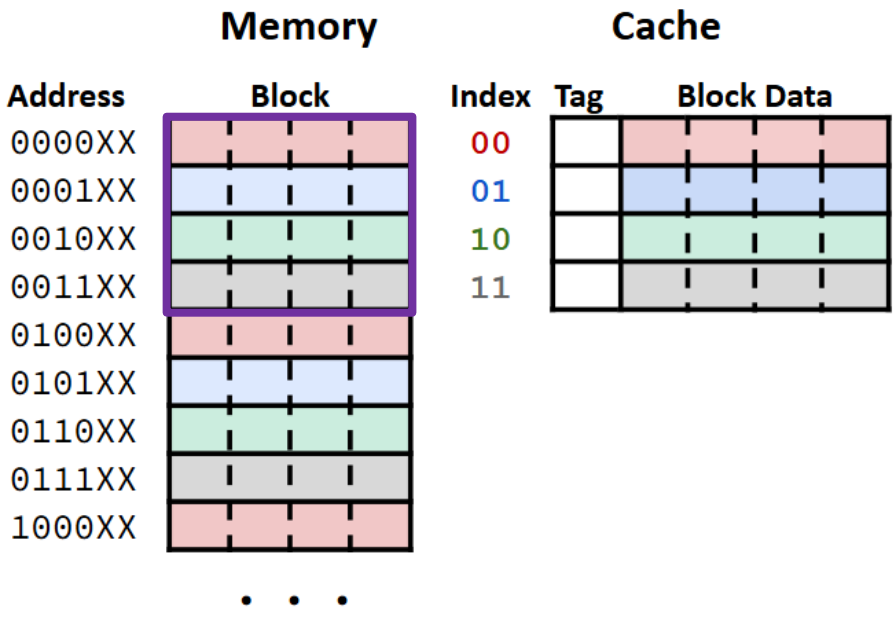
- Example parameters:
  - Cache block size ***K*** = 64B
    - 8 elements per block
  - Cache size **C** << *n* (much smaller)
  - Cache can hold 3 blocks ($r \times r$): $3r^2 < C$
- For each block of C
  - $r^2/8$ misses per block – optimal miss rate because it all fits in the cache at once
    - 8 elements per cache block
  - $2n/r$ blocks accessed (row and col) -> $2n/r * r^2/8 = nr/4$ misses
- $(n/r)^2$ blocks in total
  - $nr/4 * (n/r)^2 = \mathbf{n^3/4r}$ **misses**



Compare to $9n^3/8$ from before

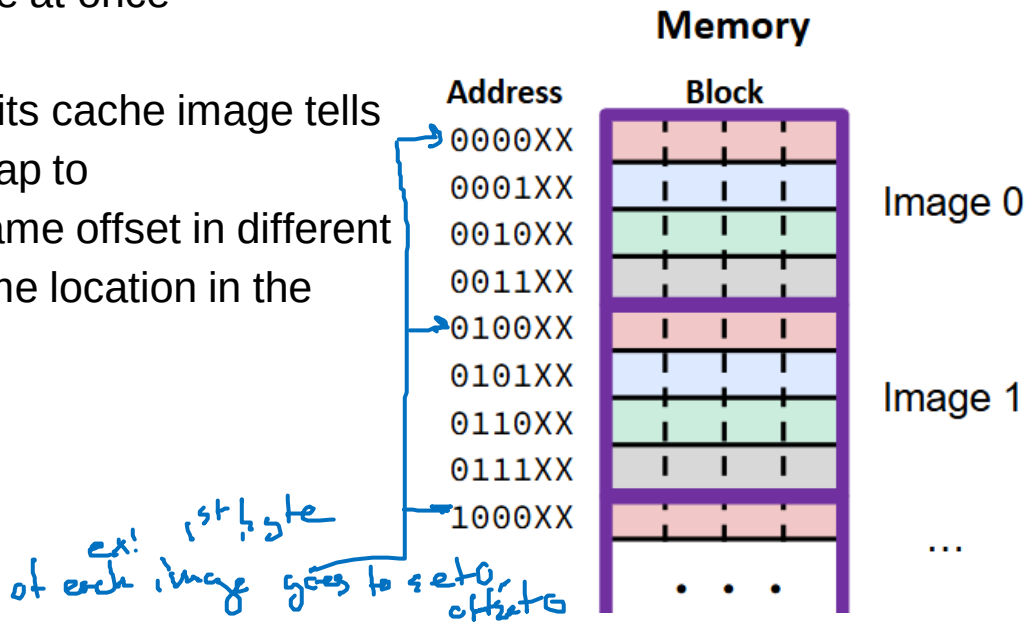
Cache Images

- Recap:
  - Blocks are contiguous chunks of memory
  - Contiguous blocks in memory map to contiguous cache sets
- Conclusion:
  - Let ***B*** = the number of blocks the cache can hold (***C*** / ***K***)
  - A contiguous, aligned chunk of ***B*** blocks (***C*** bytes) can all fit into the cache at once!



Cache Images (pt 2)

- **Cache image**: an aligned chunk of memory the same size as the cache
  - Guaranteed to all fit in the cache at once
    - Locality!
  - The offset of an address within its cache image tells you where in the cache it will map to
    - Two addresses with the same offset in different images will map to the same location in the cache



Summary

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data is accessed
    - **Cache blocking** for 2D arrays
- Getting absolute optimum performance is very platform specific
  - Depends on cache size, block size, associativity, etc.
  - Generic guidelines:
    - Keep **working set** reasonably small (temporal locality)
    - Use small **strides** (spatial locality)
    - Focus on inner loop code (for nested loops)
- **Cache images** can help you quickly figure out where data will go in the cache