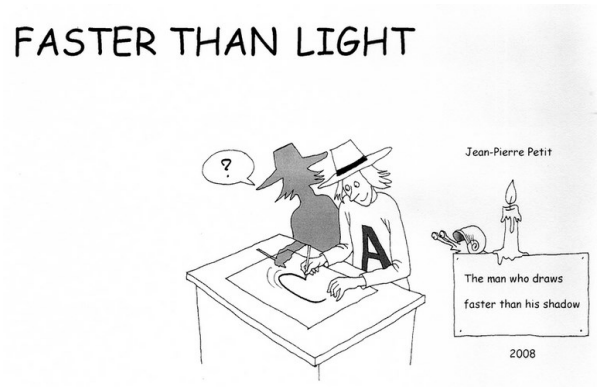
Caches III

CSE 351 Winter 2017



Administrivia

- Luis is sad he can't be here this morning ☹

  Giving a lecture on DNA Storage!

Example Placement

block size:	16 B
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

	T = A–I–O	I = $\log_2$(C/K/N)	O = $\log_2$(K)
A-bit address:	Tag (T)	Index (I)	Offset (O)

I = 3
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

I = 2
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

I = 1
4-way set associative

Set	Tag	Data
0		
1		

3

UNIVERSITY of WASHINGTON L17: Caches III CSE351, Winter 2017

Example Placement

block size:	16 B
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

	T = A–I–O	I = $\log_2$(C/K/N)	O = $\log_2$(K)
A-bit address:	Tag (T)	Index (I)	Offset (O)

I = 3
Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

I = 2
2-way set associative

Set	Tag	Data
0		
1		
2		
3		

I = 1
4-way set associative

Set	Tag	Data
0		
1		

4

Block Replacement

- *Any* empty block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches
  - Caches typically use something close to ***least recently used (LRU)*** (hardware usually implements "*not most recently used*")

LRU takes advantage of temporal locality

Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

4-way set associative

Set	Tag	Data
0		
1		

Cache Puzzle #2

- What can you infer from the following behavior?
  - Cache starts *empty*, also known as a *cold cache*
  - Access (addr: hit/miss) stream:
    - (10: miss), (12: miss), (10: miss)

Since the 10 block was kicked out, we know N = 1

- Associativity?

Since 12 was a miss, 10 and 12 are not in the same block! ∴ K = 1 or 2

K = 1 byte

K = 1 → O = 0 bits

10 = 1010

12 = 1100

S could be 1 or 2 → same set

K = 2 bytes

10 = 1010 (offset)

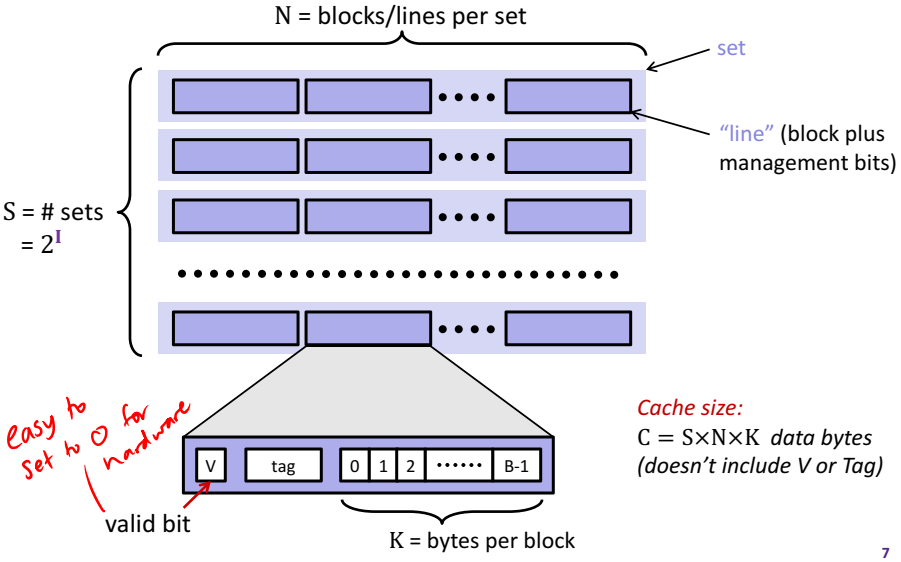
12 = 1100

K = 2 → O = 1 bit

S = 1 → since there's no overlap in any remaining bits, I = 0

- Number of sets?

General Cache Organization (S, N, K)



Notation Changes

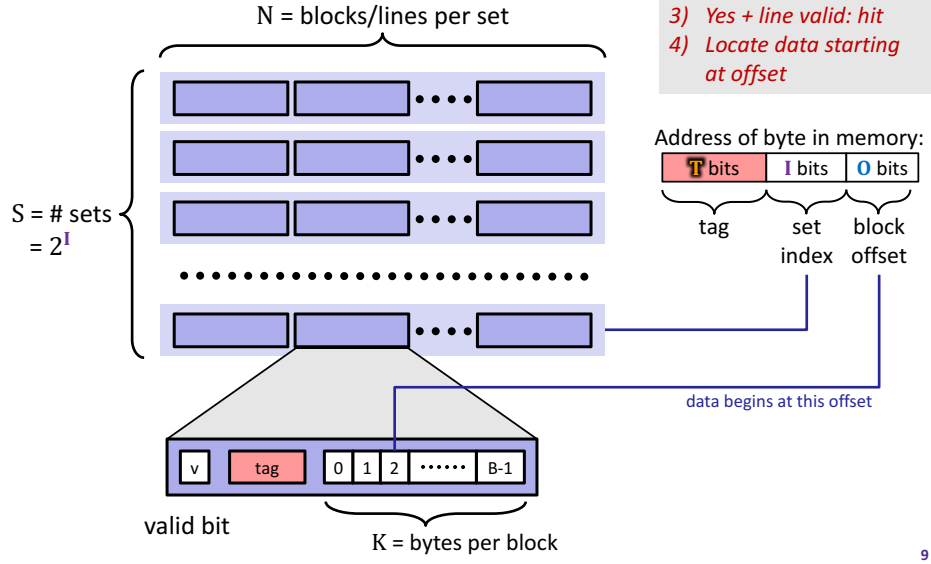
- We are using different variable names from previous quarters
  - Please be aware of this when you look at past exam questions or are watching videos

Variable	This Quarter	Previous Quarters
Block size	K	B
Cache size	C	---
Associativity	N	E
Address width	A	m
Tag field width	T	t
Index field width	I	k, s
Offset field width	O	n, b
Number of Sets	S	S

Cache Read

Set → line → byte

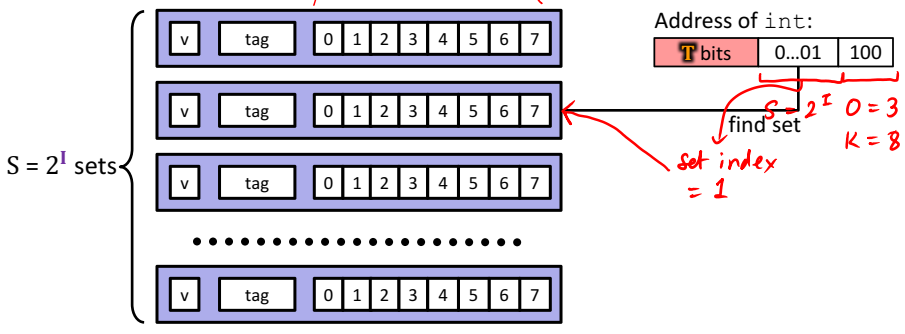
1) *Locate set*
2) *Check if any line in set has matching tag*
3) *Yes + line valid: hit*
4) *Locate data starting at offset*



Example: Direct-Mapped Cache (N = 1)

Direct-mapped: One line per set
Block Size K = 8 B

8B

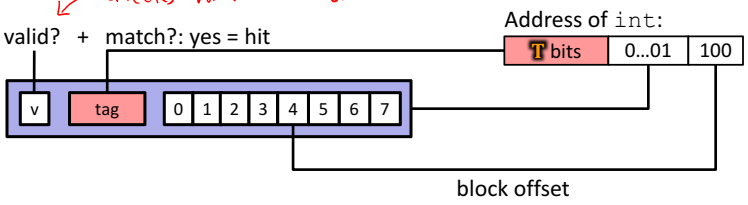


Example: Direct-Mapped Cache (N = 1)

Direct-mapped: One line per set
Block Size K = 8 B

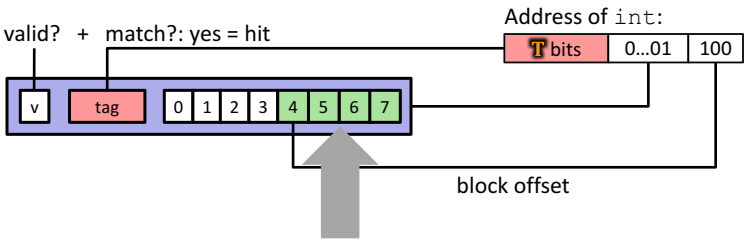
optimize checking for match

checks valid bit first



Example: Direct-Mapped Cache (N = 1)

Direct-mapped: One line per set
Block Size K = 8 B



int (4 B) is here

an int can only have offset 0 or 4

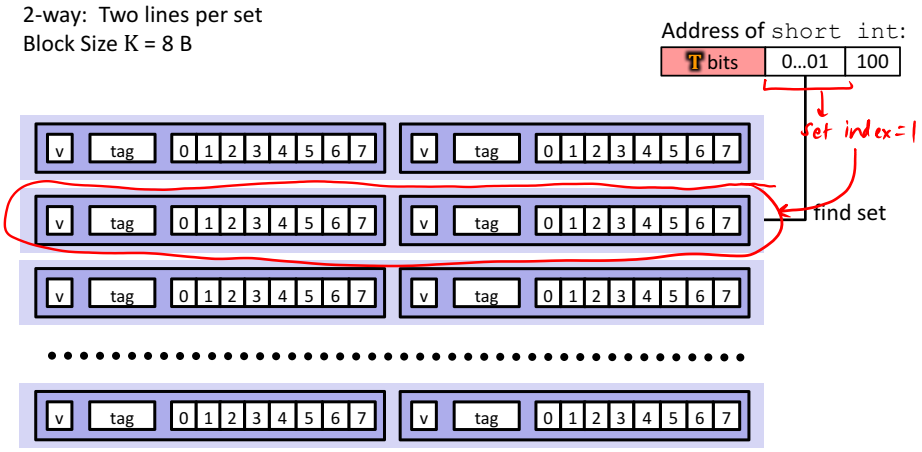
No match? Then old line gets evicted and replaced

This is why we want alignment!

no unnecessary extra cache accesses across block boundaries

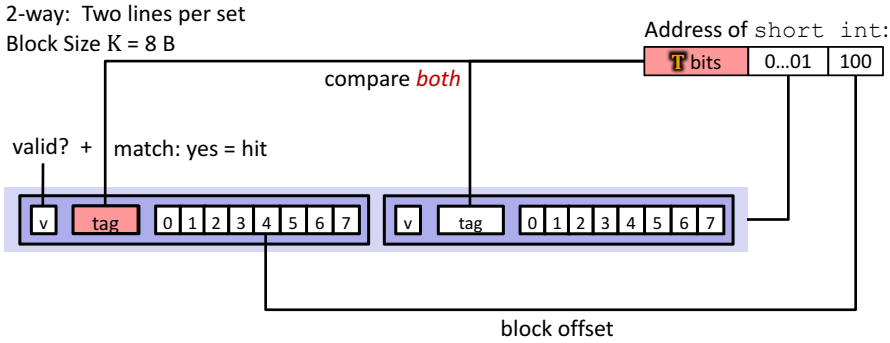
Example: Set-Associative Cache (N = 2)

2-way: Two lines per set
Block Size K = 8 B

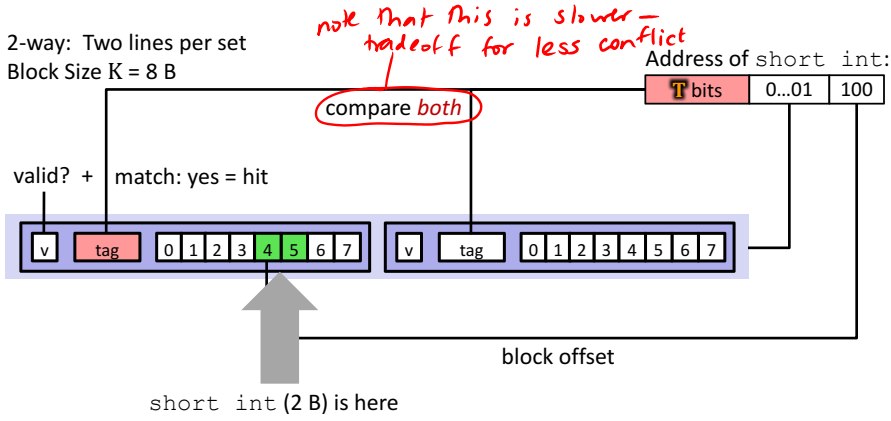


Example: Set-Associative Cache (N = 2)

2-way: Two lines per set
Block Size K = 8 B



Example: Set-Associative Cache (N = 2)



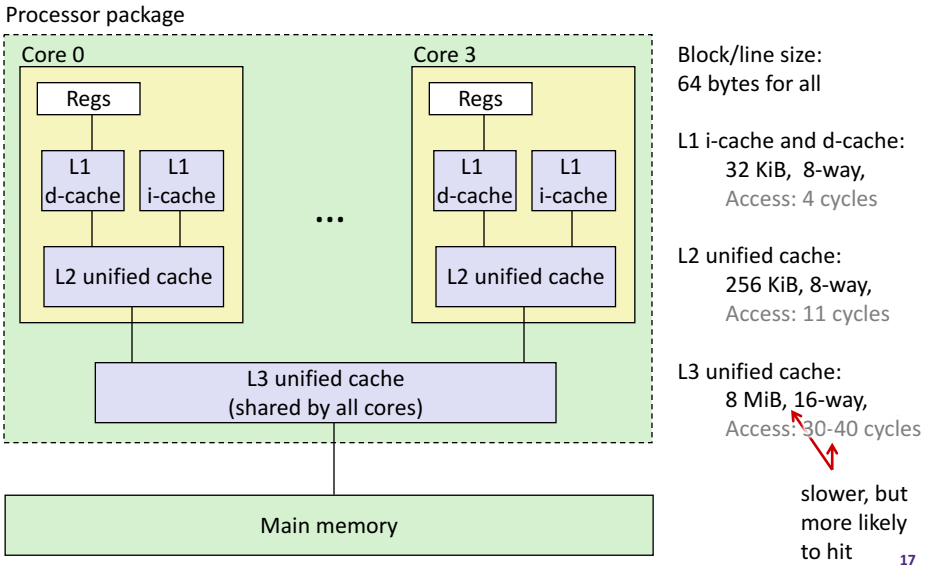
No match?

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

Types of Cache Misses: 3 C's!

- Compulsory ("required") (cold) miss
  - Occurs on first access to a block
- Conflict miss
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, … could miss every time
  - Direct-mapped caches have more conflict misses than N-way set-associative (where N > 1)
- Capacity miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won't fit, even if cache was *fully-associative*)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses

Core i7: Associativity



Block/line size:
64 bytes for all

L1 i-cache and d-cache:
32 KiB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KiB, 8-way,
Access: 11 cycles

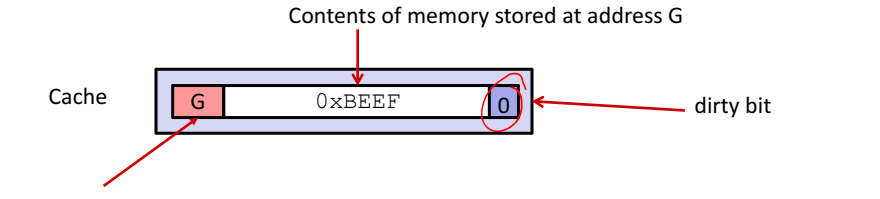
L3 unified cache:
8 MiB, 16-way,
Access: 30-40 cycles

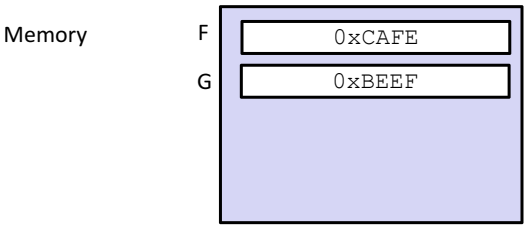
slower, but more likely to hit

What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What to do on a write-hit?
  - Write-through: write immediately to memory and all caches in-between
  - Write-back: defer write to memory 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, update line in cache
    - Good if more writes or reads to the location follow
  - No-write-allocate: ("write around") just write immediately to memory
- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally

Write-back, write-allocate example



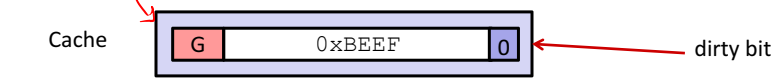


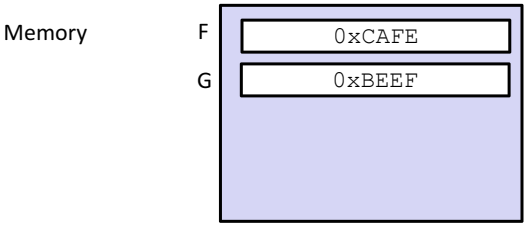
In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

Write-back, write-allocate example

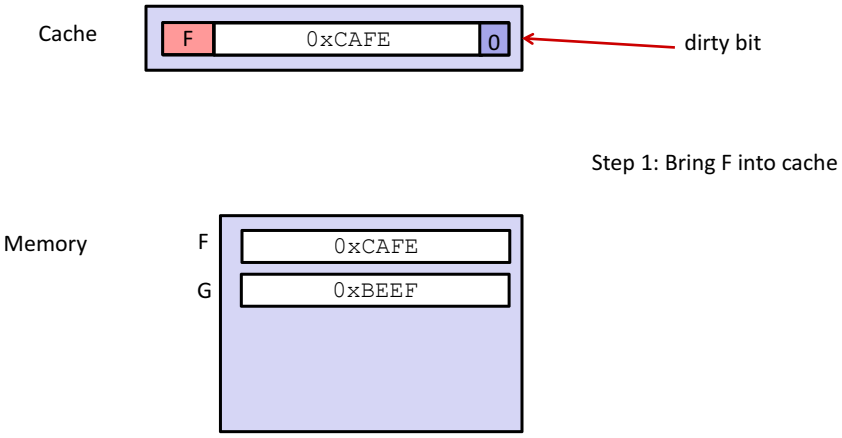
```
mov 0xFACE, F
```





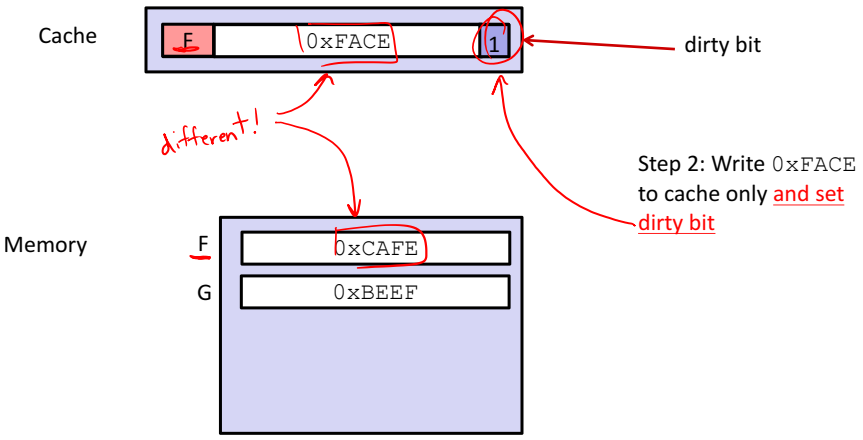
Write-back, write-allocate example

```
mov 0xFACE, F
```



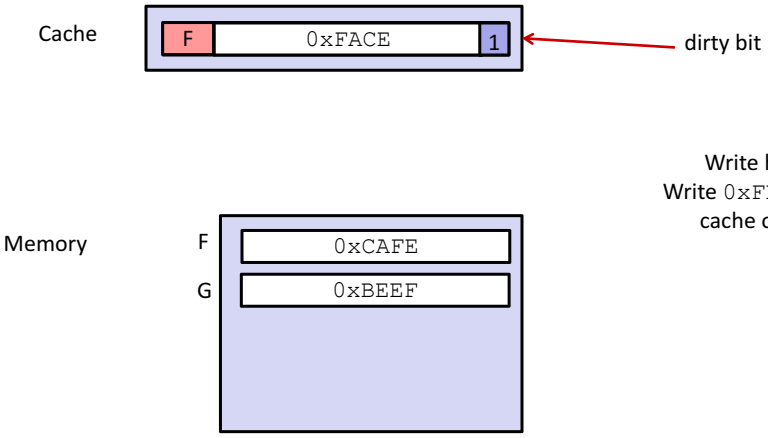
Write-back, write-allocate example

```
mov 0xFACE, F
```



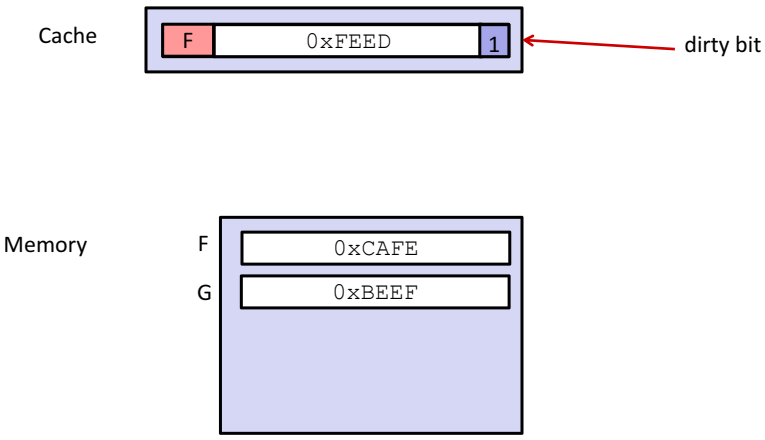
Write-back, write-allocate example

```
mov 0xFACE, F     mov 0xFEED, F
```



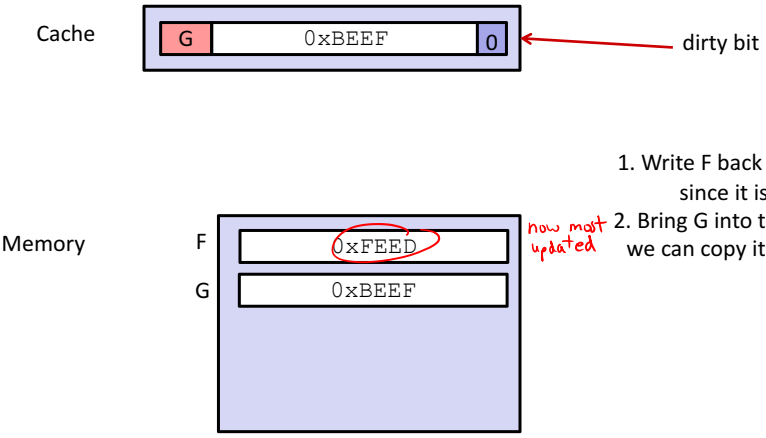
Write-back, write-allocate example

```
mov 0xFACE, F     mov 0xFEED, F          mov G, %rax
```



Write-back, write-allocate example

```
mov 0xFACE, F     mov 0xFEED, F          mov G, %rax
```



1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into `%rax`

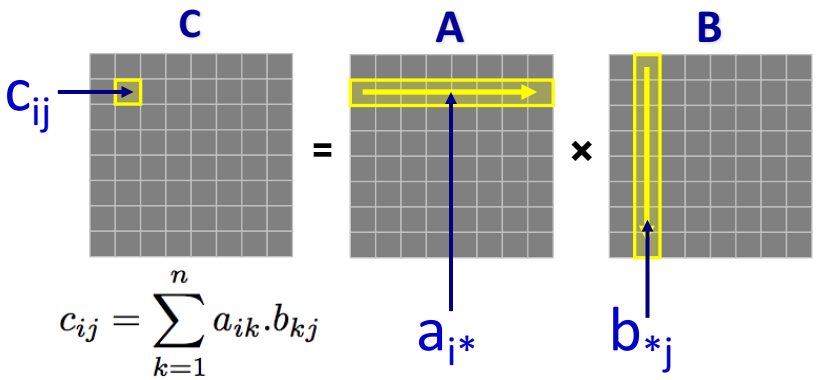
Where else is caching used?

- Software caches are more flexible
  - File system buffer caches, browser caches, etc.
  - Content-delivery networks (CDN): cache for the Internet (e.g. *Netflix*)
- Some design differences
  - Almost always fully-associative
    - so, no placement restrictions
    - index structures like hash tables are common (for placement)
  - More complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single "block" transfers
    - may fetch or write-back in larger units, opportunistically

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

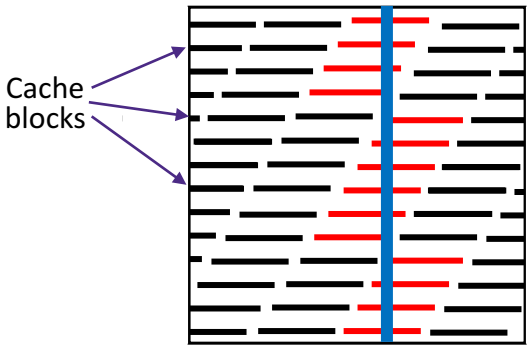
Example: Matrix Multiplication



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

Matrices in Memory

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

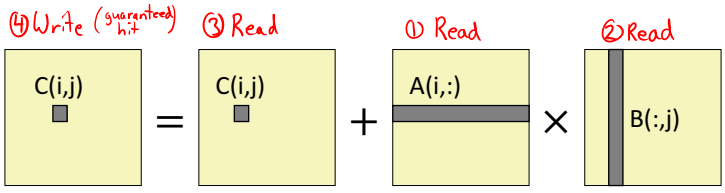


COLUMN of matrix (blue) is spread among cache blocks shown in red

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*n+j] += a[i*n+k] * b[k*n+j];
```

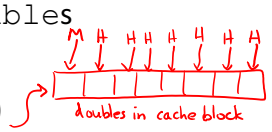
check mem access pattern



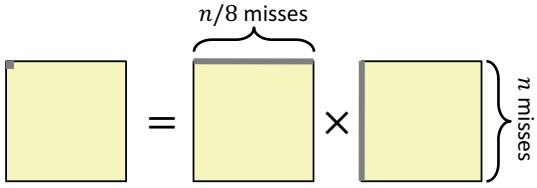
Cache Miss Analysis (Naïve)

Ignoring matrix C

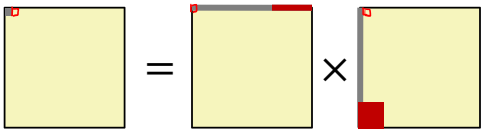
- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles
  - Cache size C $\ll n$ (much smaller than n)



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



  - Afterwards in cache: (schematic)

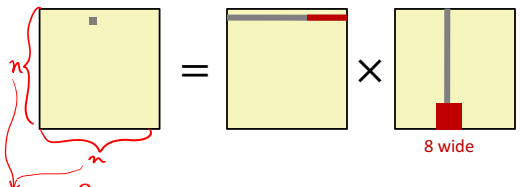


Cache Miss Analysis (Naïve)

Ignoring matrix C

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

- Other iterations:
  - Again: $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$

once per element

Linear Algebra to the Rescue (1)

- 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)

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 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*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

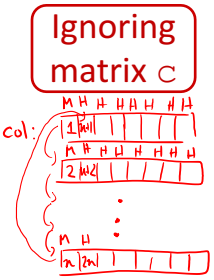
6 nested loops may seem less efficient, but leads to much faster code!

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

Cache Miss Analysis (Blocked)

Ignoring matrix c

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



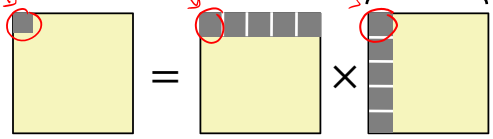
simultaneously

r^2 elements per block, 8 per cache block

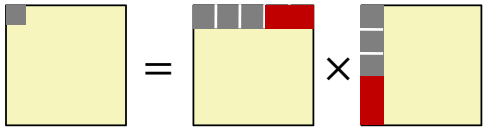
n/r blocks

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

n/r blocks in row and column



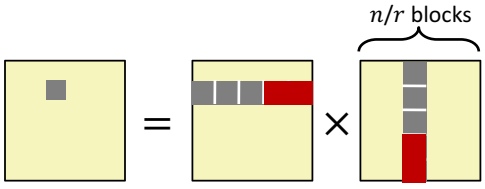
  - Afterwards in cache (schematic)



Cache Miss Analysis (Blocked)

Ignoring matrix C

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size C $\ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$
- Other (block) iterations:
  - Same as first iteration
  - $2n/r \times r^2/8 = nr/4$



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

Matrix Multiply Summary

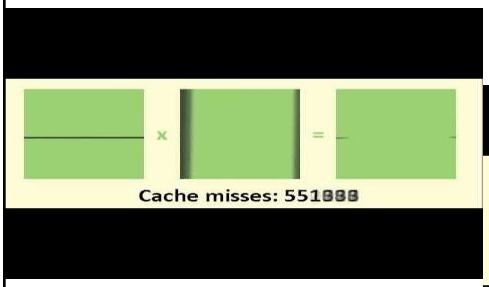
- Naïve: $(9/8) \times n^3$
- Blocked: $1/(4r) \times n^3$
  - If r = 8, difference is 4*8 * 9/8 = 36x
  - If r = 16, difference is 4*16 * 9/8 = 72x
- Blocking optimization only works if the blocks fit in the cache
  - Suggests largest possible block size up to limit $3r^2 \leq C$
- Matrix multiplication has inherent temporal locality:
  - Input data: $3n^2$, computation $2n^3$
  - Every array element used $O(n)$ times!
  - But program has to be written properly

38

Matrix Multiply Visualization

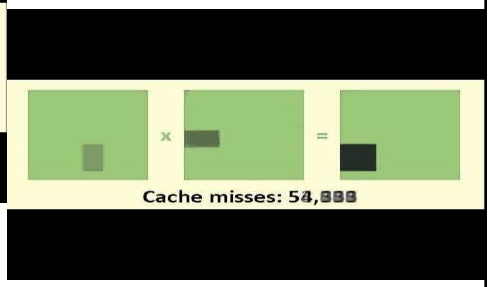
- Here n = 100, C = 32 KiB, r = 30

Naïve:



≈ 1,020,000
cache misses

Blocked:



≈ 90,000
cache misses

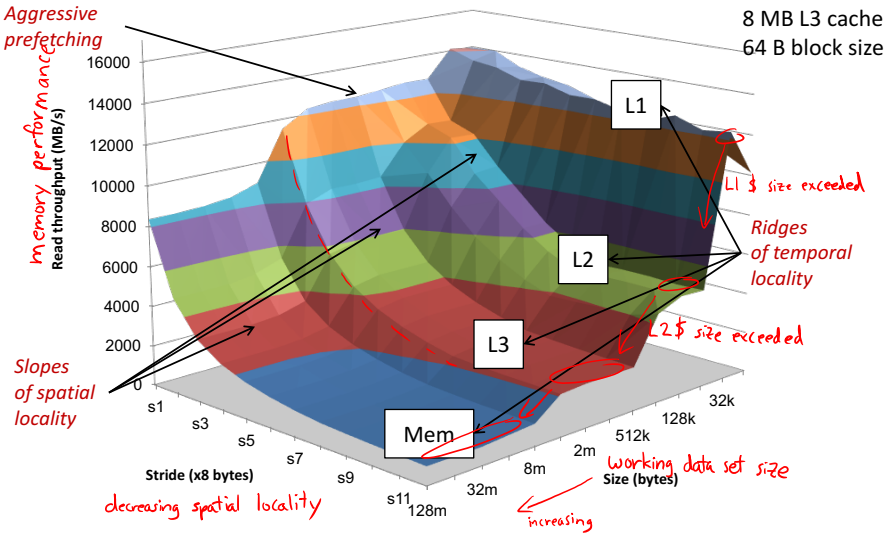
Cache-Friendly Code

- 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

general rules of thumb

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



Learning About Your Machine

- **Linux:**
  - `lscpu`
  - ls /sys/devices/system/cpu/cpu0/cache/index0/
    - Ex: cat /sys/devices/system/cpu/cpu0/cache/index*/size
  - `cat /proc/cpuinfo | grep cache | sort | uniq`
- **Windows:**
  - `wmic memcache get <query>` (all values in KB)
  - Ex: `wmic memcache get MaxCacheSize`
- Modern processor specs: http://www.7-cpu.com/