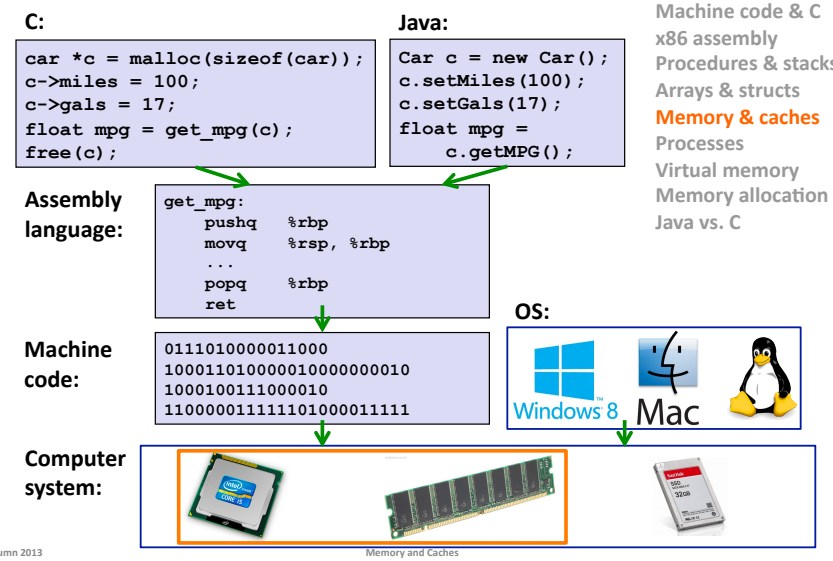
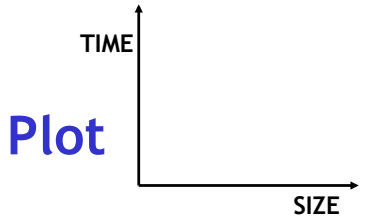
Roadmap



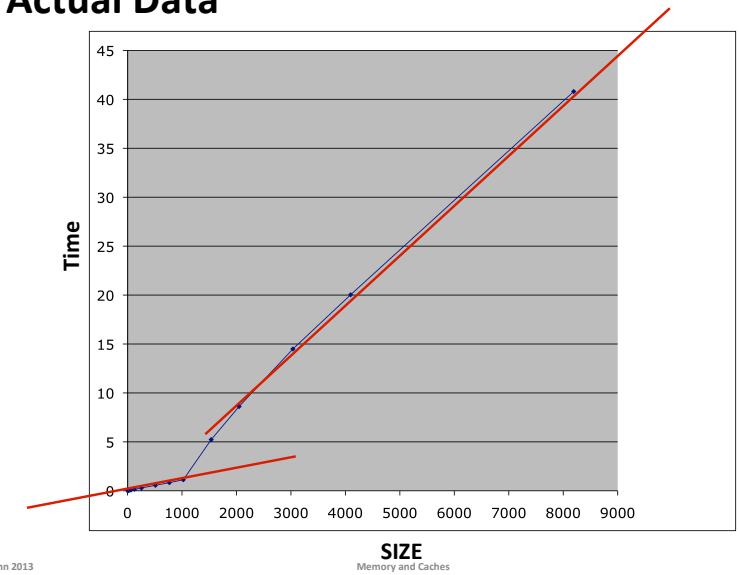
Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

How does execution time grow with SIZE?

```
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
  for (int j = 0 ; j < SIZE ; ++ j) {
     A += array[j];
  }
}
```



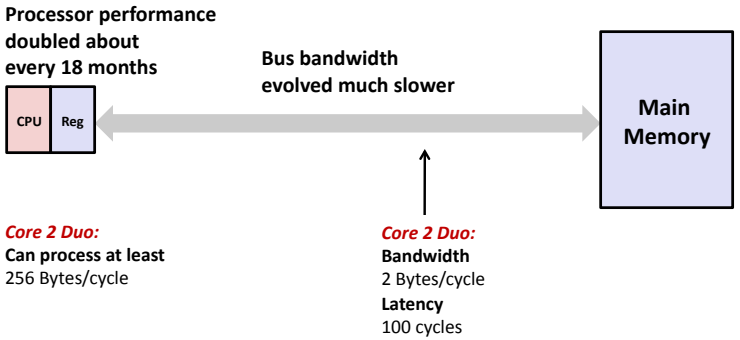
Actual Data



Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

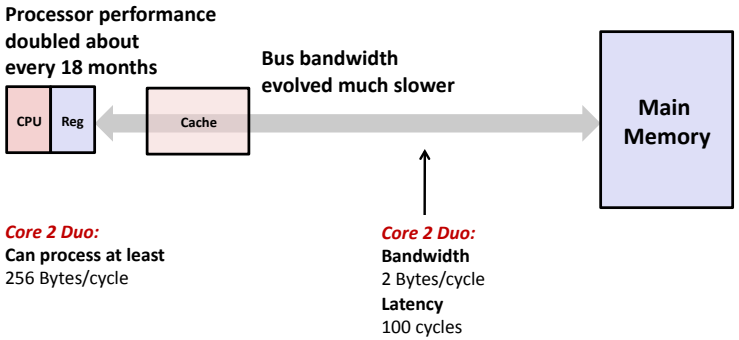
Problem: Processor-Memory Bottleneck



Problem: lots of waiting on memory

cycle = single fixed-time machine step

Problem: Processor-Memory Bottleneck



Solution: caches

cycle = single fixed-time machine step

Cache

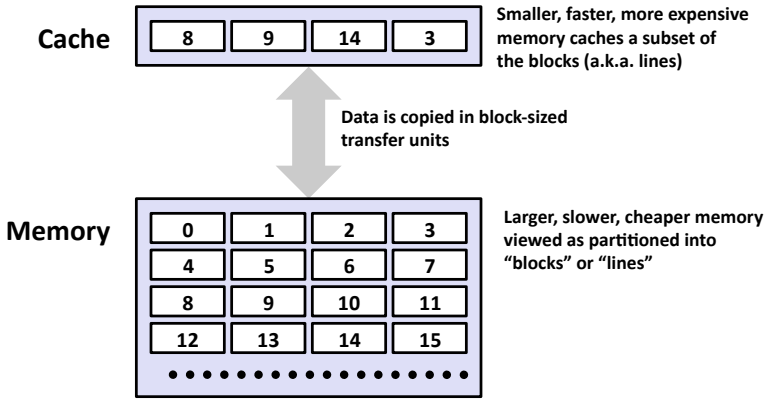
- **English definition:** a hidden storage space for provisions, weapons, and/or treasures

- **CSE definition:** computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

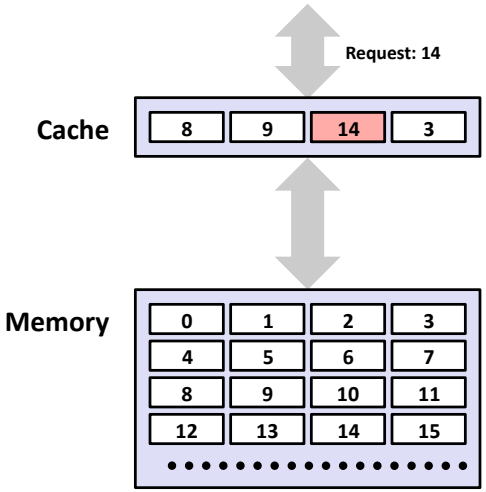
  more generally,

  used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)

General Cache Mechanics



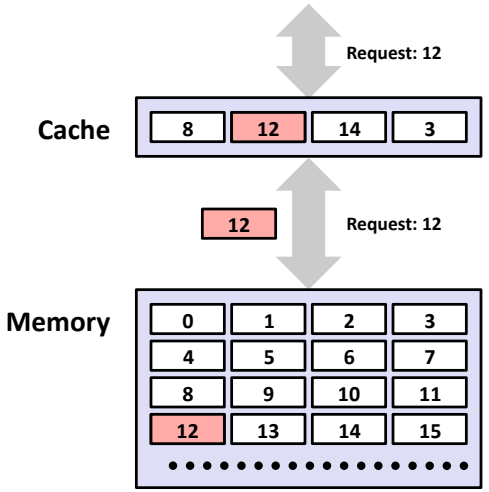
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

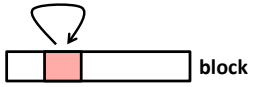
Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

Why Caches Work

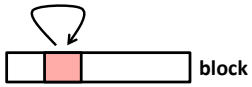
- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**
- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future
  - Why is this important?



Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future

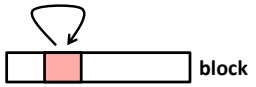


- **Spatial locality?**

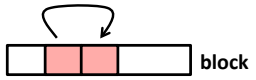
Why Caches Work

- **Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time



  - How do caches take advantage of this?

Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
        sum += a[i];
return sum;
```

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern
- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**

Locality Example #1

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Locality Example #1

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

1: a[0][0]
2: a[0][1]
3: a[0][2]
4: a[0][3]
5: a[1][0]
6: a[1][1]
7: a[1][2]
8: a[1][3]
9: a[2][0]
10: a[2][1]
11: a[2][2]
12: a[2][3]

stride-1

Locality Example #2

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Locality Example #2

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

1: a[0][0]
2: a[1][0]
3: a[2][0]
4: a[0][1]
5: a[1][1]
6: a[2][1]
7: a[0][2]
8: a[1][2]
9: a[2][2]
10: a[0][3]
11: a[1][3]
12: a[2][3]

stride-N

Locality Example #3

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?

Cost of Cache Misses

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory
- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    Cache hit time of 1 cycle
    Miss penalty of 100 cycles

cycle = single fixed-time machine step

Cost of Cache Misses

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory
- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    Cache hit time of 1 cycle
    Miss penalty of 100 cycles

cycle = single fixed-time machine step

  - Average access time:

check the cache every time

    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles
- **This is why "miss rate" is used instead of "hit rate"**

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
    = 1 - hit rate
  - Typical numbers (in percentages):
    - 3% - 10% for L1
    - Can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - Includes time to determine whether the line is in the cache
  - Typical hit times: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
  - Typically 50 - 200 cycles for L2 (trend: increasing!)

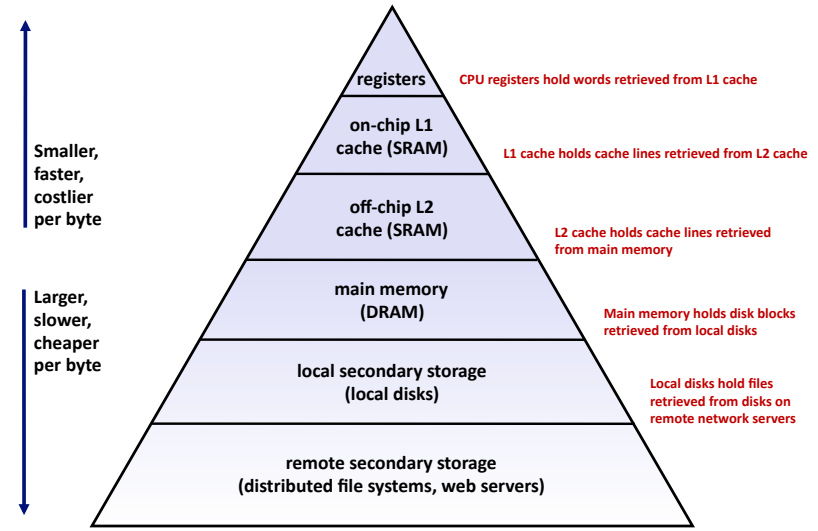
Can we have more than one cache?

- **Why would we want to do that?**

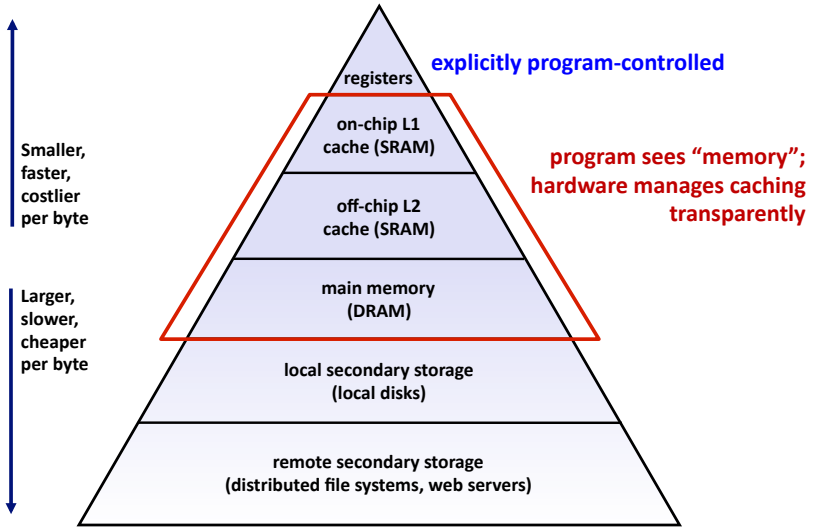
Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software systems:**
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality
- **These properties complement each other beautifully**
- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy**

An Example Memory Hierarchy



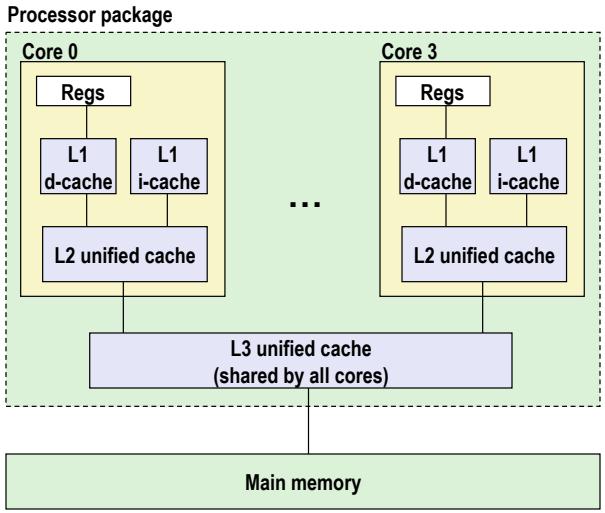
An Example Memory Hierarchy



Memory Hierarchies

- **Fundamental idea of a memory hierarchy:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- ***Big Idea:*** **The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.**

Intel Core i7 Cache Hierarchy



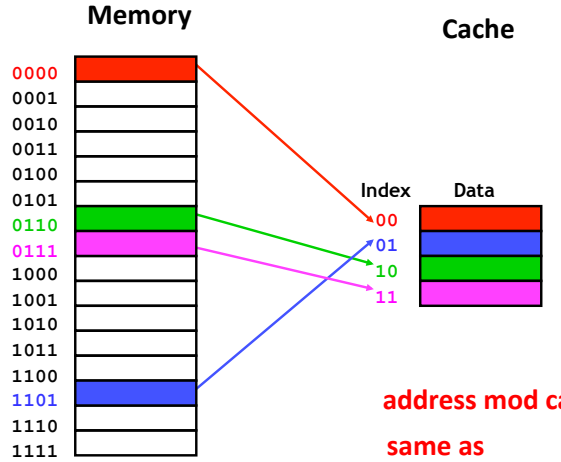
L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

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

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

Block size: 64 bytes for all caches.

Where should we put data in the cache?



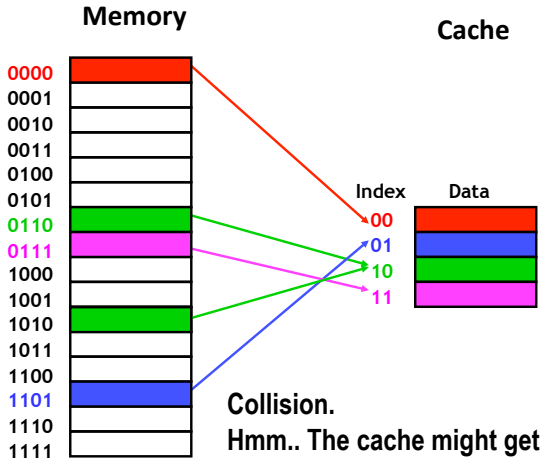
address mod cache size

same as

low-order $\log_2$(cache size) bits

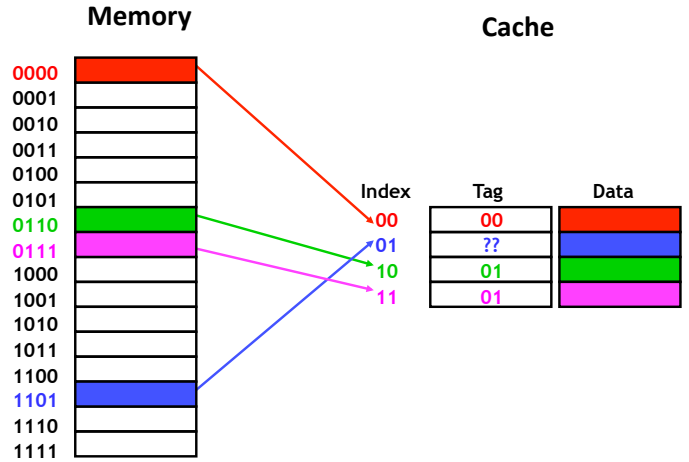
- How can we compute this mapping?

Where should we put data in the cache?



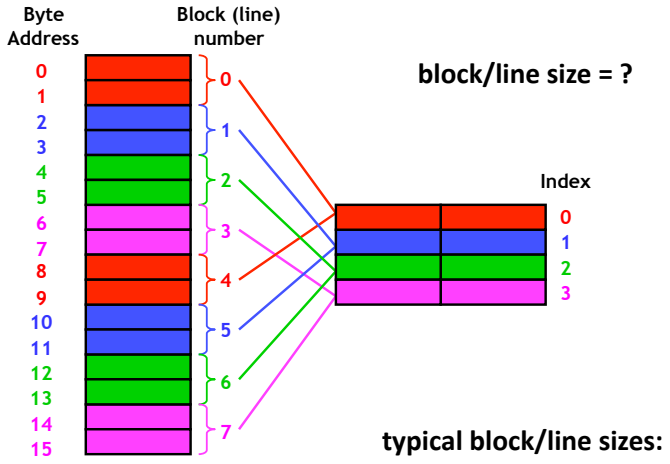
Collision.
Hmm.. The cache might get confused later!
Why? And how do we solve that?

Use tags to record which location is cached



tag = rest of address bits

What's a cache block? (or *cache line)*



A puzzle.

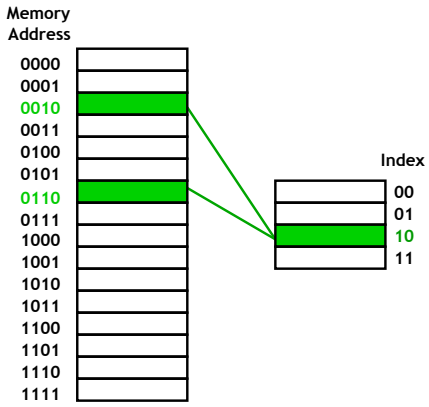
- What can you infer from this:
- Cache starts *empty*
- Access (addr, hit/miss) stream:
- (10, miss), (11, hit), (12, miss)

block size >= 2 bytes

block size < 8 bytes

Problems with direct mapped caches?

- **direct mapped:**
  - Each memory address can be mapped to exactly one index in the cache.
- **What happens if a program uses addresses 2, 6, 2, 6, 2, …?**
- ***conflict***

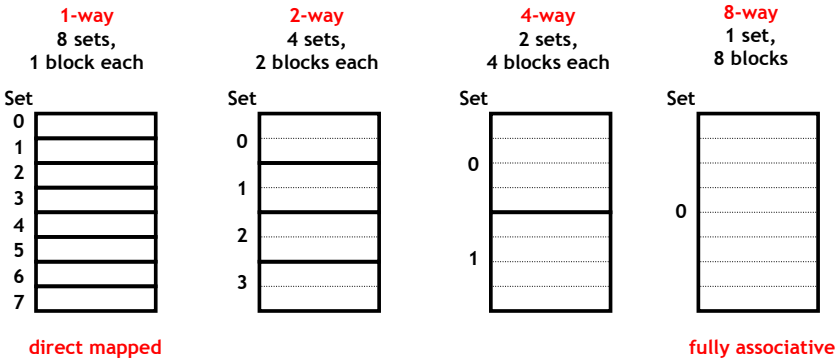


Associativity

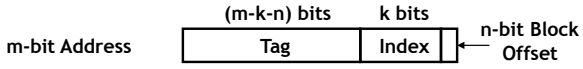
- What if we could store data in *any* place in the cache?

Associativity

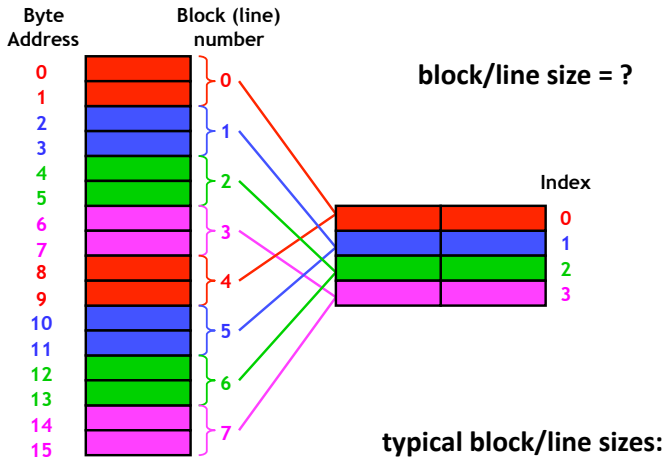
- What if we could store data in *any* place in the cache?
- That might slow down caches (more complicated hardware), so we do something in between.
- Each address maps to exactly one *set.*



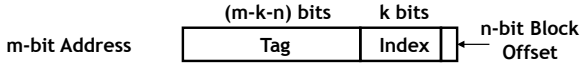
Now how do I know where data goes?



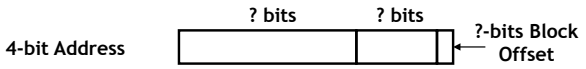
What's a cache block? (or *cache line)*



Now how do I know where data goes?

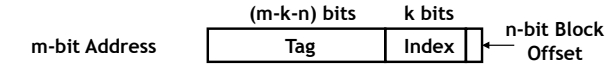


Our example used a 2^2-block cache with 2^1 bytes per block. Where would 13 (1101) be stored?



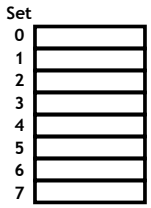
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 011 0011.



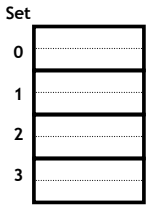
k = ?

1-way associativity
8 sets, 1 block each



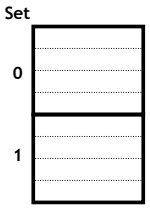
k = ?

2-way associativity
4 sets, 2 blocks each



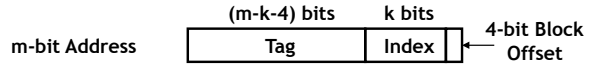
k = ?

4-way associativity
2 sets, 4 blocks each



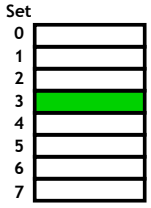
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 011 0011.



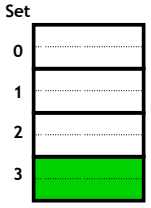
k = 3

1-way associativity
8 sets, 1 block each



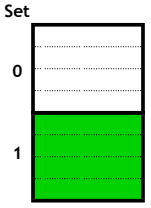
k = 2

2-way associativity
4 sets, 2 blocks each



k = 1

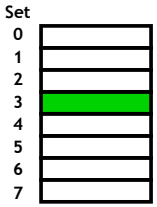
4-way associativity
2 sets, 4 blocks each



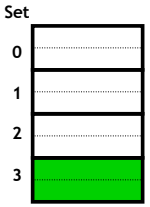
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, which one should we replace?

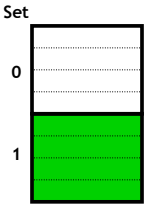
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



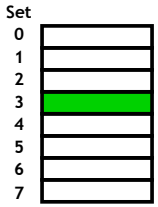
4-way associativity
2 sets, 4 blocks each



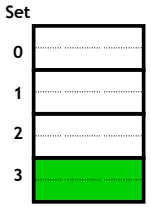
Block replacement

- Replace something, of course, but what?

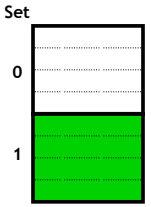
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each

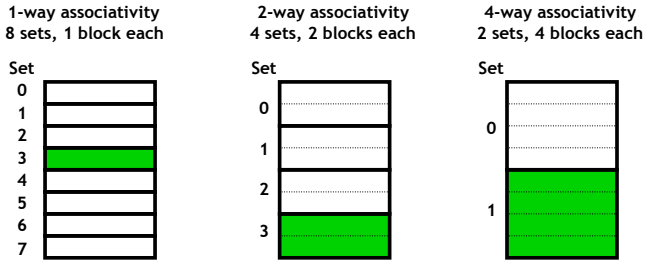


4-way associativity
2 sets, 4 blocks each



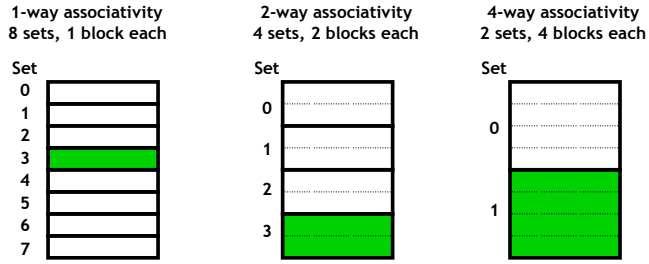
Block replacement

- **Replace something, of course, but what?**
  - Obvious for direct-mapped caches, what about set-associative?



Block replacement

- **Replace something, of course, but what?**
  - Caches typically use something close to **least recently used (LRU)**
  - (hardware usually implements "not most recently used")



Another puzzle.

- What can you infer from this:

- Cache starts *empty*
- Access (addr, hit/miss) stream

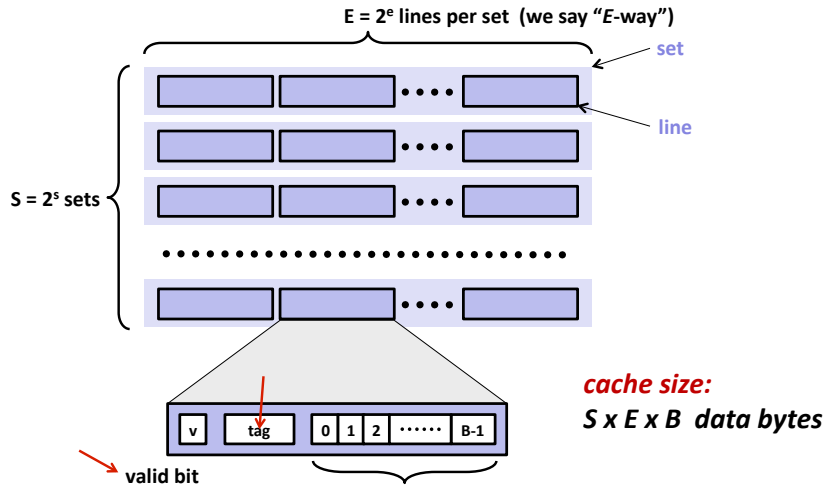
- (10, miss); (12, miss); (10, miss)

12 is not in the same block as 10

12's block replaced 10's block

direct-mapped cache

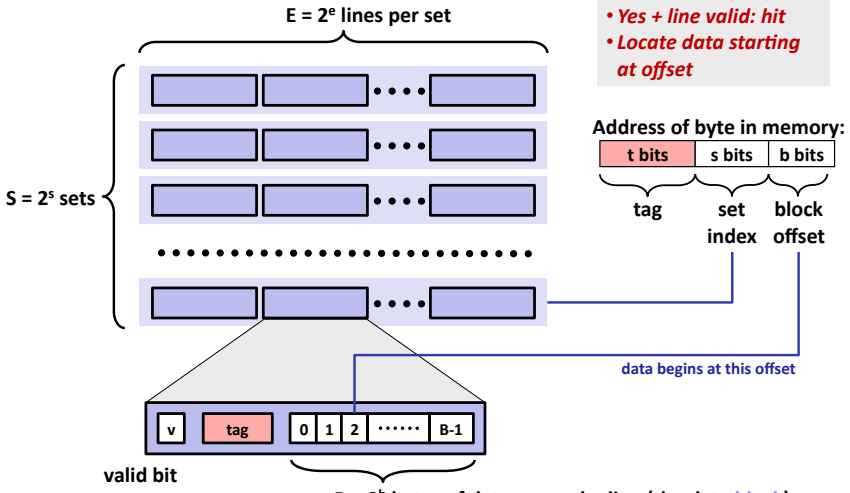
General Cache Organization (S, E, B)



cache size:
S x E x B data bytes

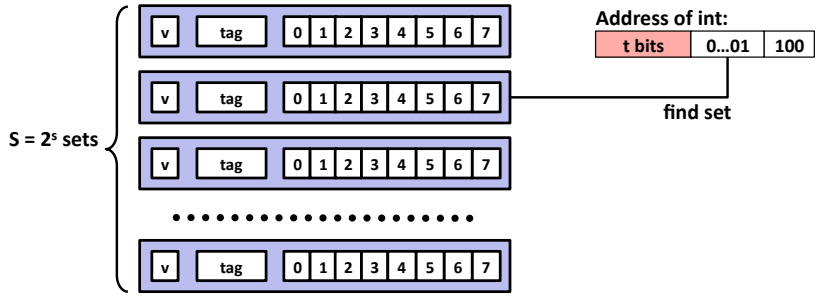
Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



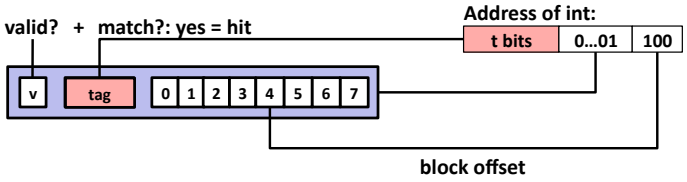
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes



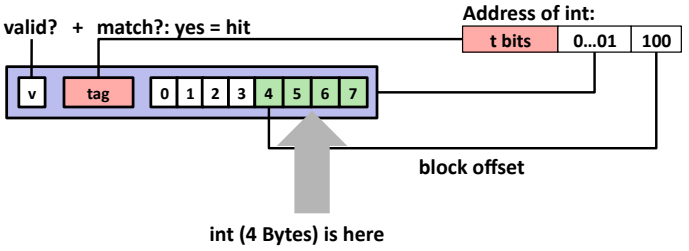
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Example (for E = 1)

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

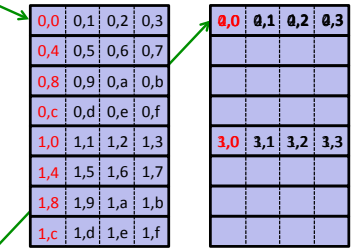
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Assume sum, i, j in registers
Address of an aligned element of a: `aa...ayyyyyxxx000`

Assume: cold (empty) cache
3 bits for set, 5 bits for offset

`aa...ayyy yxx xx000`

0,0: `aa...a000 000 00000`



32 B = 4 doubles

4 misses per row of array
4*16 = 64 misses

32 B = 4 doubles

every access a miss
16*16 = 256 misses

Example (for E = 1)

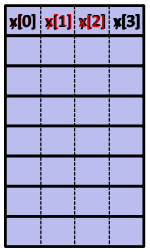
```
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

In this example, cache blocks are 16 bytes; 8 sets in cache
- How many block offset bits?
- How many set index bits?

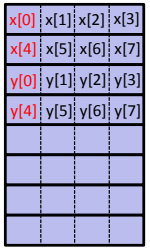
Address bits: ttt....t sss bbbb
- B = 16 = 2^b: b=4 offset bits
- S = 8 = 2^s: s=3 index bits

0: 000....0 000 0000
128: 000....1 000 0000
160: 000....1 010 0000

if x and y have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128

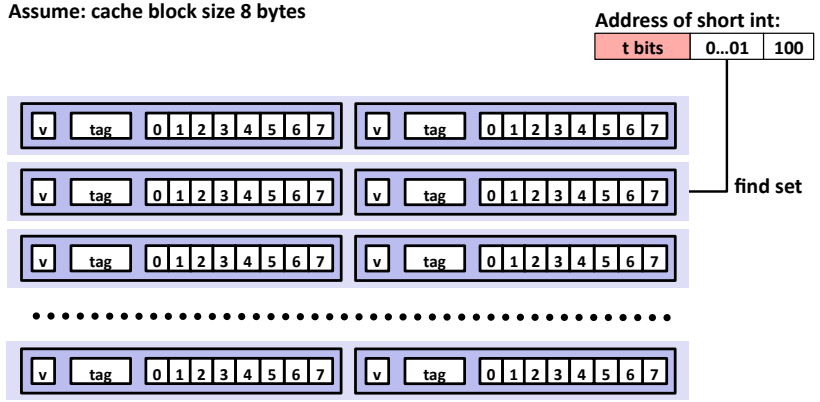


if x and y have unaligned starting addresses, e.g., &x[0] = 0, &y[0] = 160



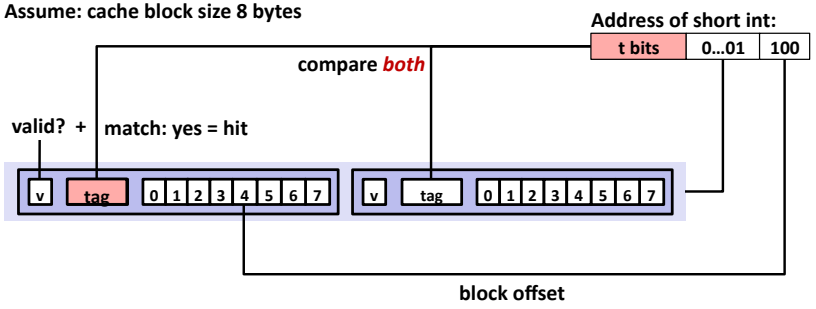
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



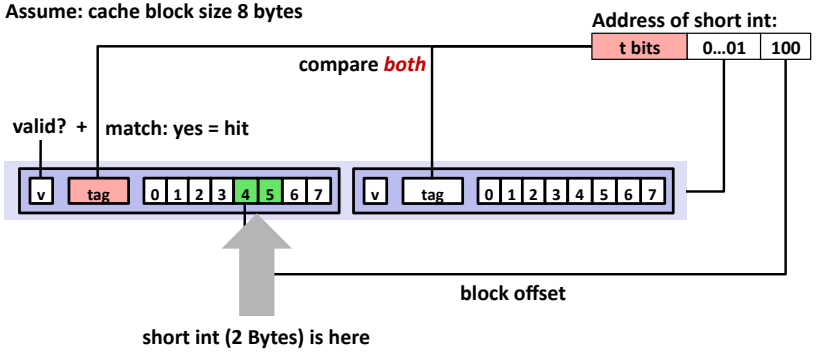
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



No match:

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

Example (for E = 2)

```
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

If x and y have aligned starting addresses, e.g. &x[0] = 0, &y[0] = 128, can still fit both because two lines in each set

x[0]	x[1]	x[2]	x[3]	y[0]	y[1]	y[2]	y[3]
x[4]	x[5]	x[6]	x[7]	y[4]	y[5]	y[6]	y[7]

Types of Cache Misses

- **Cold (compulsory) 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, ... would miss every time
  - direct-mapped caches have more conflict misses than n-way set-associative (where n is a power of 2 and n > 1)
- **Capacity miss**
  - Occurs when the set of active cache blocks (the working set) is larger than the cache (just won't fit)

What about writes?

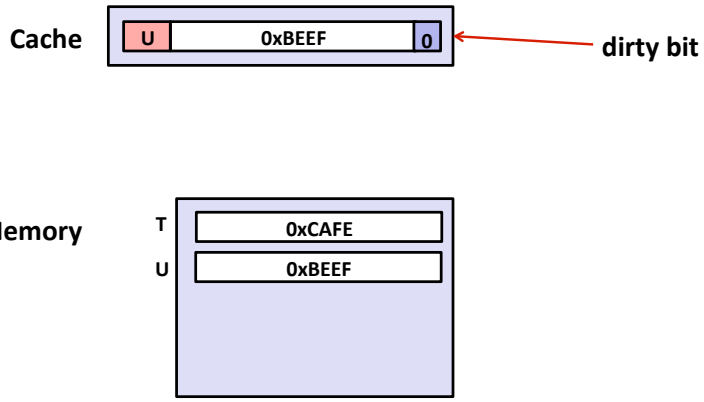
- **Multiple copies of data exist:**
  - L1, L2, possibly L3, main memory
- **What is the main problem with that?**

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, all caches in between.
  - Write-back: defer write to memory until line is evicted (replaced)
    - Need a *dirty bit* to indicate if line is different from memory or not
- **What to do on a write-miss?**
  - Write-allocate: load into cache, update line in cache.
    - Good if more writes or reads to the location follow
  - No-write-allocate: just write immediately to memory.
- **Typical caches:**
  - Write-back + Write-allocate, usually ← **why?**
  - Write-through + No-write-allocate, occasionally

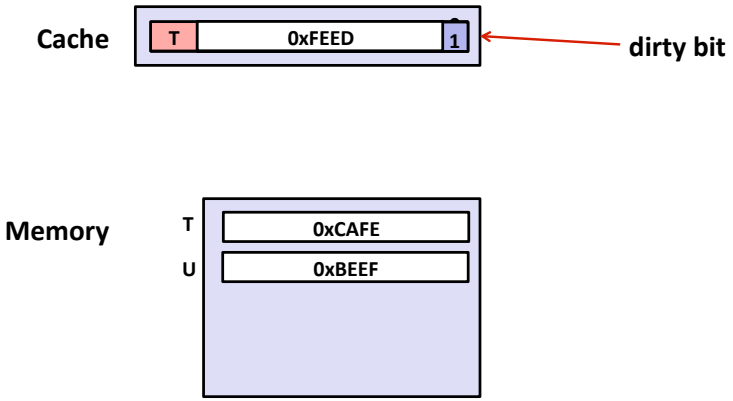
Write-back, write-allocate example

mov 0xFACE, T



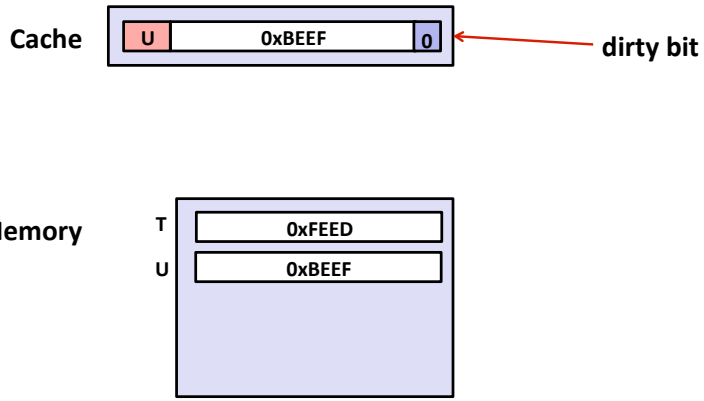
Write-back, write-allocate example

mov 0xFACE, T mov 0xFEED, T mov U, %rax

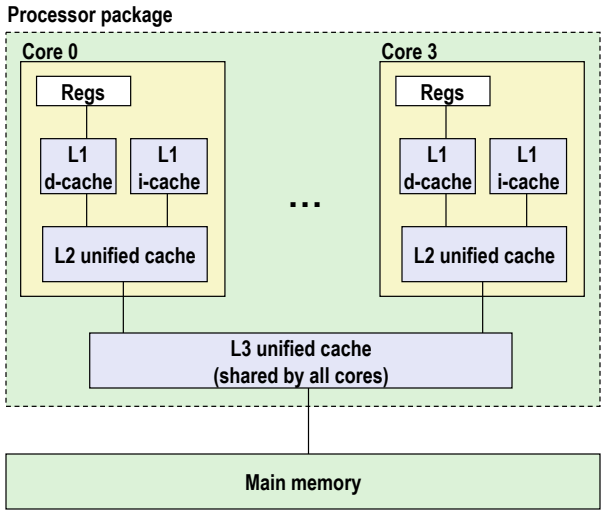


Write-back, write-allocate example

mov 0xFACE, T mov 0xFEED, T mov U, %rax



Back to the Core i7 to look at ways



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

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

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

Block size: 64 bytes for all caches.

slower, but more likely to hit

Where else is caching used?

Software Caches are More Flexible

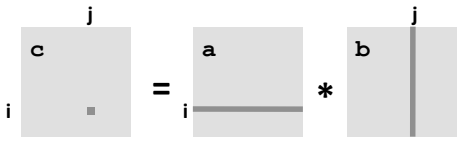
- **Examples**
  - File system buffer caches, web browser caches, etc.
- **Some design differences**
  - Almost always fully-associative
    - so, no placement restrictions
    - index structures like hash tables are common (for placement)
  - Often use complex replacement policies
    - misses are very 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?**
  - Proper choice of algorithm
  - Loop transformations

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
```



$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}.$$

memory access pattern?

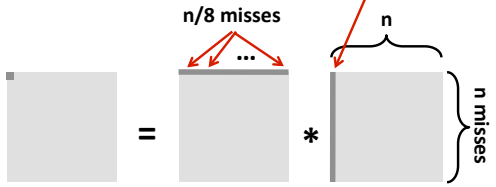
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n, **not** left-shifted by n)

spatial locality:
chunks of 8 items in a row
in same cache line

each item in column in
different cache line

- **First iteration:**
  - n/8 + n = 9n/8 misses
    (omitting matrix c)

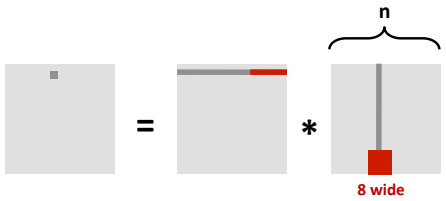


  - Afterwards in cache:
    (schematic)

Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n)

- **Other iterations:**
  - Again:
    n/8 + n = 9n/8 misses
    (omitting matrix c)

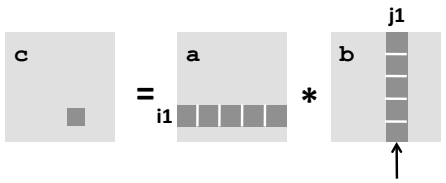


- **Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

once per element

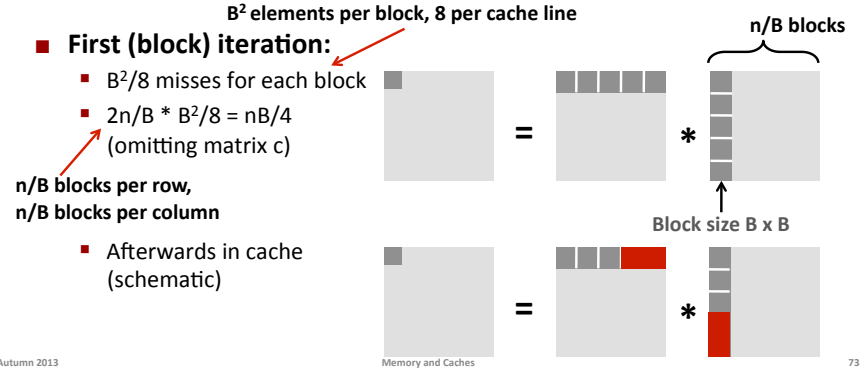
Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



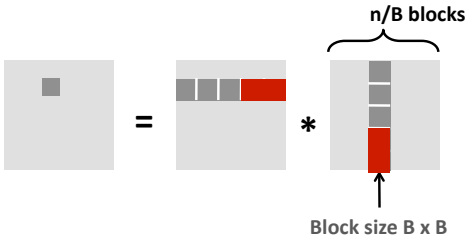
Cache Miss Analysis

- **Assume:**
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- **Assume:**
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- **Other (block) iterations:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$
- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$



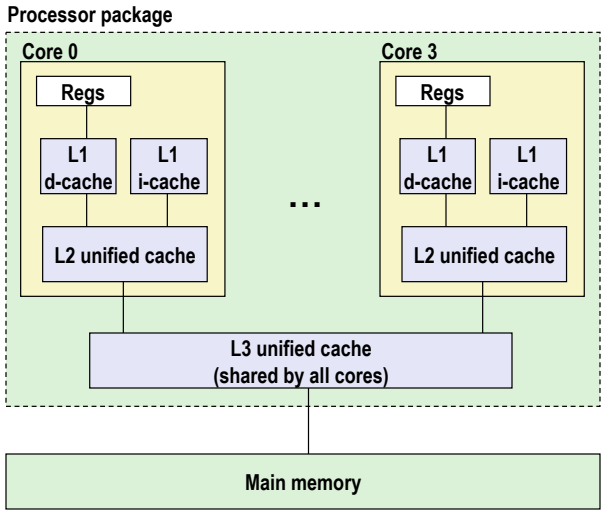
Summary

- **No blocking:** **(9/8) * n^3**
- **Blocking:** **1/(4B) * n^3**
- **If B = 8 difference is 4 * 8 * 9 / 8 = 36x**
- **If B = 16 difference is 4 * 16 * 9 / 8 = 72x**
- **Suggests largest possible block size B, but limit $3B^2 < C$!**
- **Reason for dramatic difference:**
  - 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

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 sizes, line sizes, associativities, 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

Intel Core i7 Cache Hierarchy



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

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

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

Block size: 64 bytes for all caches.

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip