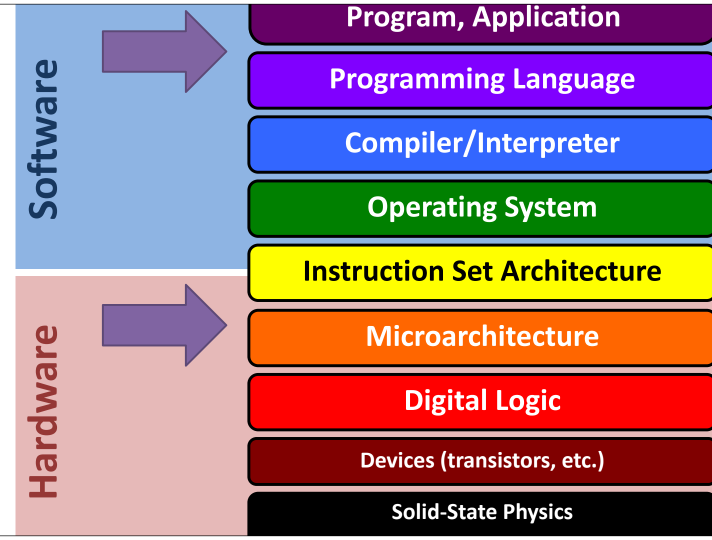
CS 240
Foundations of Computer Systems

Memory Hierarchy and Cache

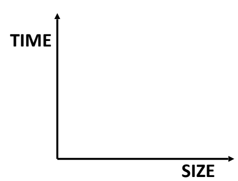
Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

https://cs.wellesley.edu/~cs240/

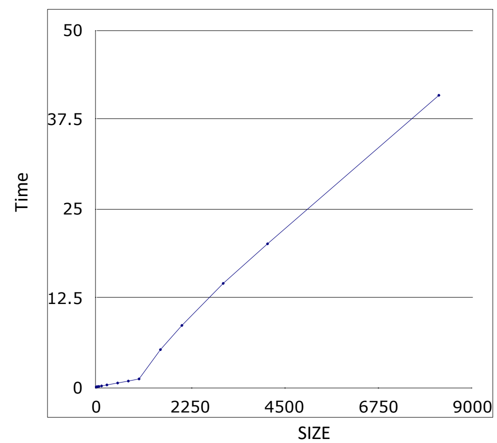


How does execution time grow with SIZE?

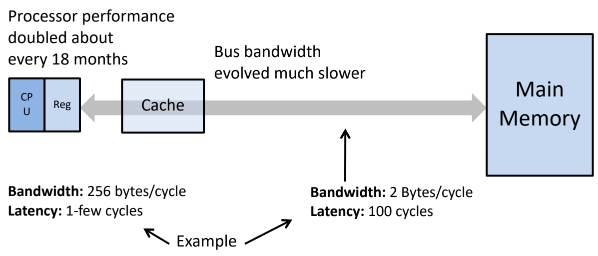
```
int array[SIZE];
fillArrayRandomly(array);
int s = 0;

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



Reality



Processor-memory bottleneck



Solution: caches

Cache

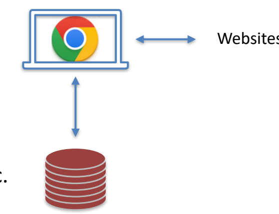
English:

n. a hidden storage space for provisions, weapons, or treasures
v. to store away in hiding for future use

Computer Science:

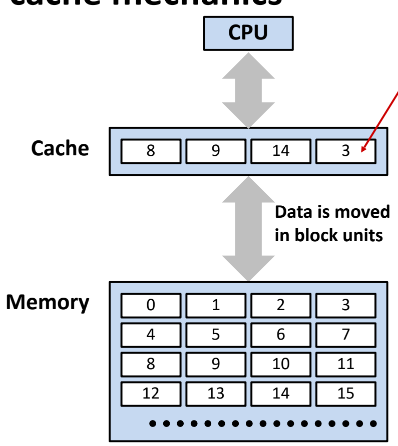
n. a computer memory with short access time used to store frequently or recently used instructions or data

v. to store [data/instructions] temporarily for later quick retrieval



Also used more broadly in CS: software caches, file caches, etc.

General cache mechanics

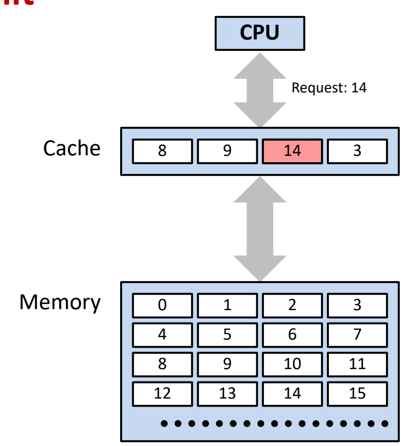


Block: unit of data in cache and memory.
(a.k.a. line)

Smaller, faster, more expensive.
Stores **subset of memory blocks**.
(lines)

Larger, slower, cheaper.
Partitioned into blocks (lines).

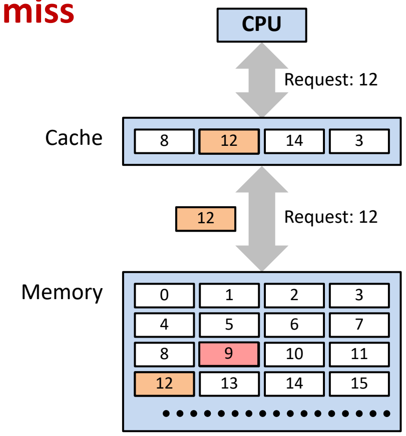
Cache hit



1. Request** data in block **b.

2. Cache hit:
Block b is in cache.

Cache **miss**



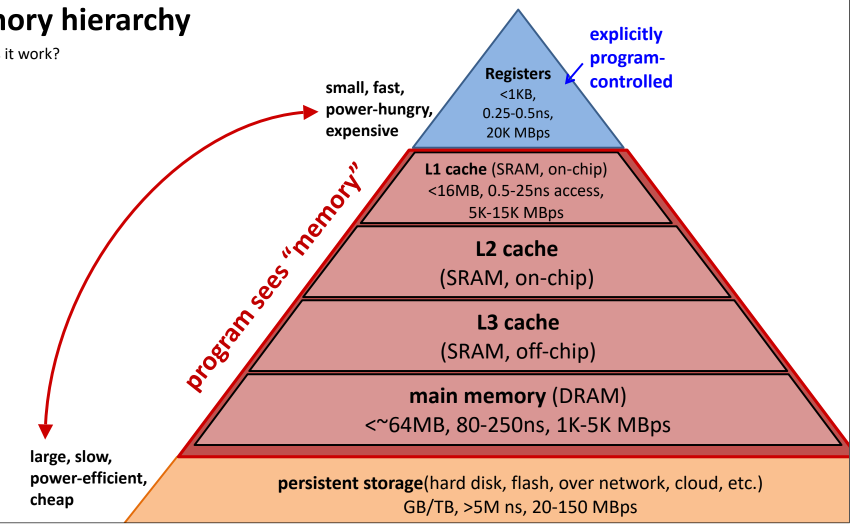
1. ***Request*** *data in block* ***b.***
2. ***Cache miss:***
   block is not in cache
3. ***Cache eviction:***
   Evict a block to make room, maybe store to memory.
4. ***Cache fill:***
   Fetch block from memory, store in cache.

Placement Policy:
where to put block in cache

Replacement Policy:
which block to evict

Memory hierarchy

Why does it work?

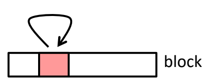


Locality: why caches work

Programs tend to use data and instructions at addresses near or equal to those they have used recently.

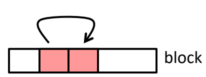
Temporal locality:

Recently referenced items are ***likely*** to be referenced again in the near future.



Spatial locality:

Items with nearby addresses are ***likely*** to be referenced close together in time.



How do caches exploit temporal and spatial locality?

Locality #1: Basic iteration over array

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

What is stored in memory?

Locality #2: iteration over 2D array

ex

row-major M x N 2D array in C

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

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


Locality #3: iteration over 2D array

ex

row-major M x N 2D array in C

Swapped loop order

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

    for (int j = 0; j < N; j++) {
        for (int 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 #4

What is "wrong" with this code?

How can it be fixed?

```
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;

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


Cost of cache misses

Miss cost could be 100 × hit cost.

99% hits could be **twice** as good as 97%. How?

Assume cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:

97% hits: (0.97 * 1 cycle) + (0.03 * 100 cycles) = 3.97 cycles

99% hits: (0.93 * 1 cycle) + (0.01 * 100 cycles) = 1.93 cycles

hit/miss rates

Cache performance metrics

Miss Rate

Fraction of memory accesses to data not in cache (misses / accesses)

Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

Hit Time

Time to find and deliver a block in the cache to the processor.

Typically: **1 - 2 clock cycles** for L1; **5 - 20 clock cycles** for L2

Miss Penalty

Additional time required on cache miss = main memory access time

Typically **50 - 200 cycles** for L2 *(trend: increasing!)*

Cache organization

Block

Fixed-size unit of data in memory/cache

Placement Policy

Where in the cache should a given block be stored?

- direct-mapped, set associative

Replacement Policy

What if there is no room in the cache for requested data?

- least recently used, most recently used

Write Policy

When should writes update lower levels of memory hierarchy?

- write back, write through, write allocate, no write allocate

Blocks

Divide address space into fixed-size aligned blocks.
power of 2

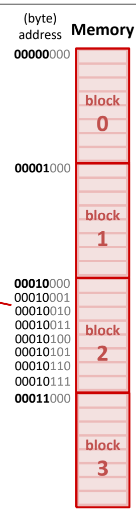
Example: block size = 8

full byte address

00010010

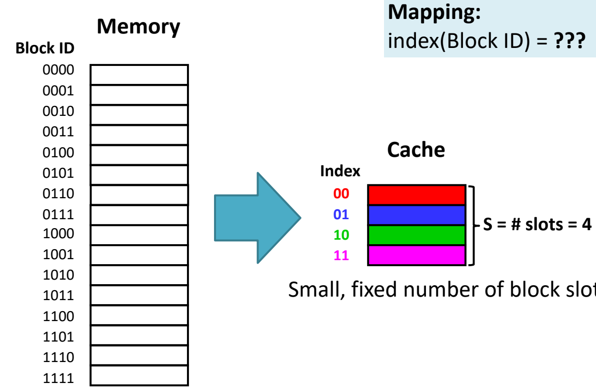
Block ID
address bits - offset bits

offset within block
$\log_2$(block size)



Placement policy

Mapping:
index(Block ID) = **???**

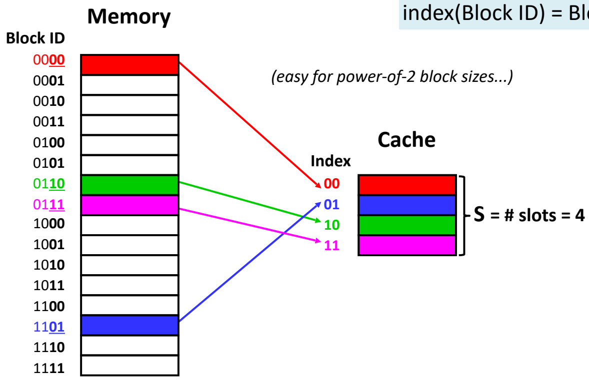


Small, fixed number of block slots.

Large, fixed number of block slots.

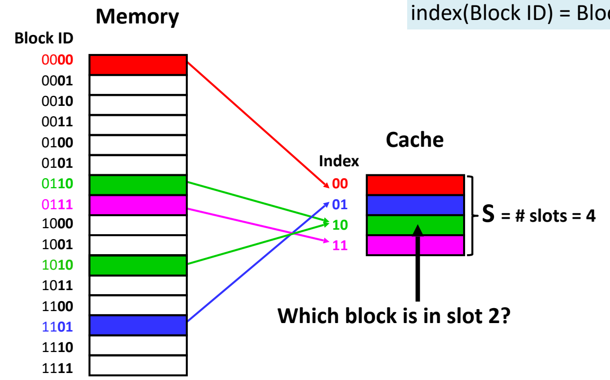
Placement: *direct-mapped*

Mapping:
index(Block ID) = Block ID *mod* S



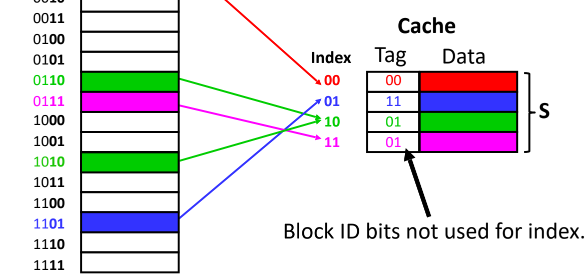
Placement: mapping ambiguity?

Mapping:
index(Block ID) = Block ID *mod* S

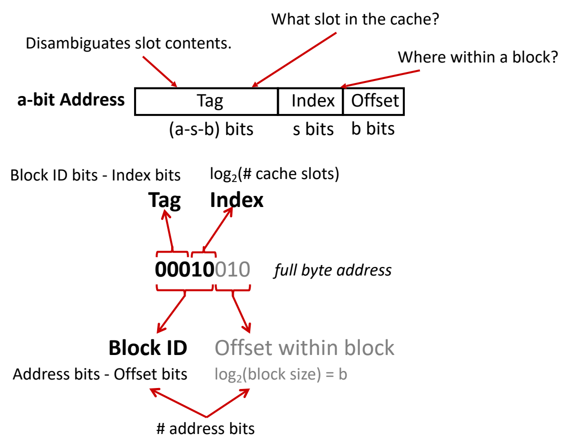


Placement: tags resolve ambiguity

Mapping:
index(Block ID) = Block ID *mod* S



Address = tag, index, offset



Cache size puzzle

ex

Cache starts *empty.*

Access (address, hit/miss) stream:

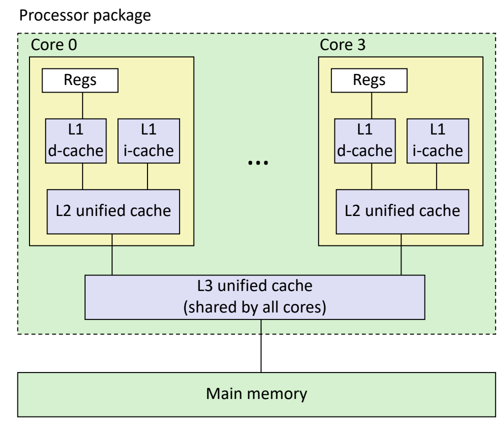
(0xA, miss), (0xB, hit), (OxC, miss)

What could the block size be?

1. First, convert the hex to integers
2. Remember that blocks must be aligned to the block size
3. Hint: there are two possible block sizes!

Example memory hierarchy

Typical laptop/desktop processor
(c.a. 201_)



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

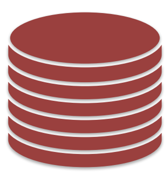
Software caches

Examples

- File system buffer caches, web browser caches, database caches, network CDN caches, etc.

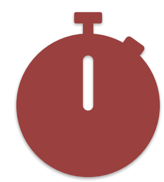
Some design differences

- Often use complex replacement policies
- Not necessarily constrained to single "block" transfers



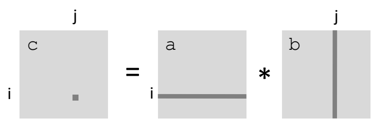
Cache-friendly code

- Locality, locality, locality.
- Programmer can optimize for cache performance
  - Data structure layout
  - Data access patterns
    - Nested loops
    - Blocking
- All systems favor "cache-friendly code"
  - Performance is hardware-specific
  - Generic rules capture most advantages
    - Keep working set small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on *inner loop* code



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

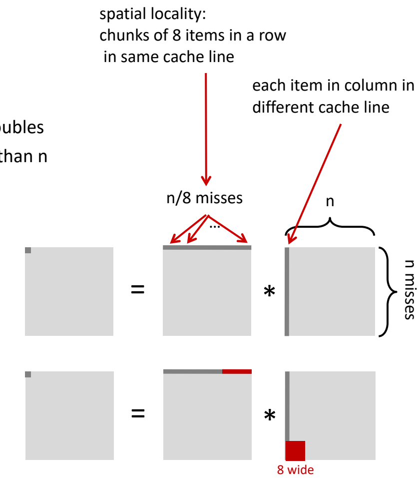
Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C is much smaller than n

First iteration:

- n/8 + n = 9n/8 misses (omitting matrix c)

Afterwards in cache: (schematic)



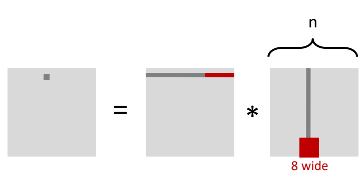
Cache Miss Analysis

Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C is much smaller than n

Other iterations:

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



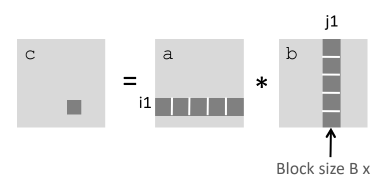
Total misses:

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

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];
}
```



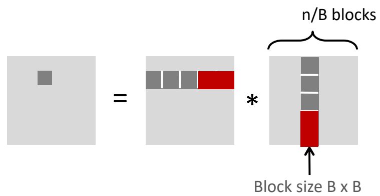
Cache Miss Analysis

Assume:

- Cache block = 64 bytes = 8 doubles
- Cache size C << n (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

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

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

```
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++)
            // CODE HERE
}
```


Exercise: order these 3 functions by locality

ex

```
typedef struct {
    int vel[3];
    int acc[3];
} point;
```

```
#define N 100
point p[N];
```

```
void clear1(point *p, int n) {
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}
```

```
void clear2(point *p, int n) {
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++)
            p[i].vel[j] = 0;
        for (j=0; j<3; j++)
            p[i].acc[j] = 0;
    }
}
```

```
void clear3(point *p, int n) {
    int i, j;
    for (j=0; j<3; j++){
        for (i=0; i<n; i++)
            p[i].vel[j] = 0;
        for (i=0; i<n; i++)
            p[i].acc[j] = 0;
    }
}
```