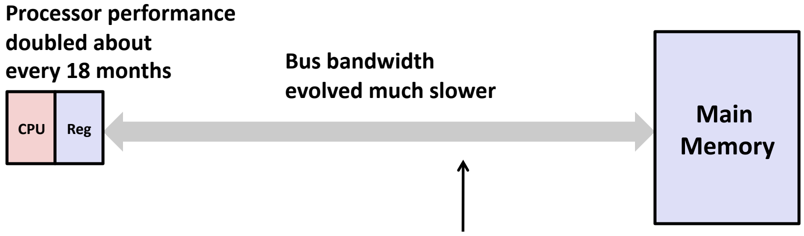
Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimizations that consider caches

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower



Core 2 Duo:
Can process at least
256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100 cycles

Solution: Caches

Cache

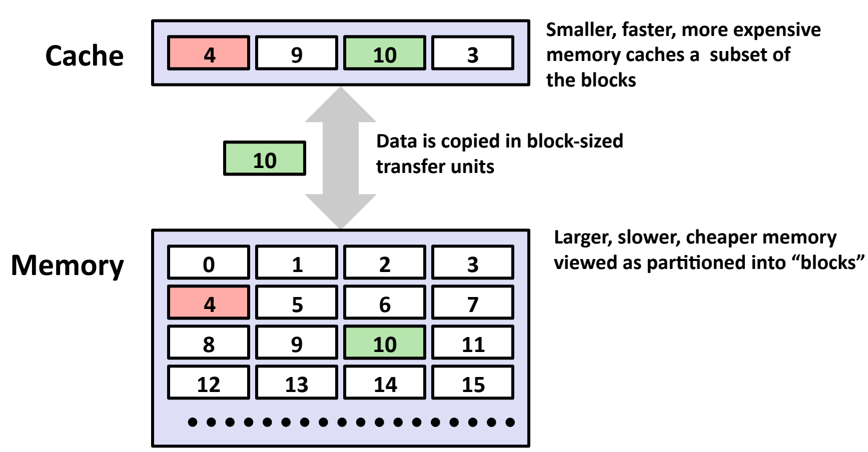
- **English definition:** a hidden storage space for provisions, weapons, and/or treasures

- **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)

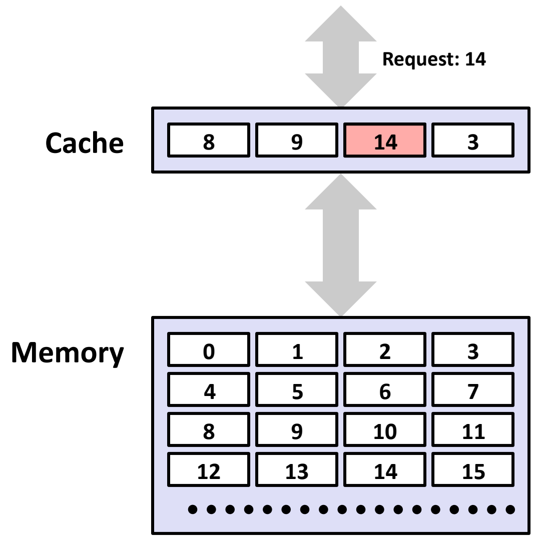
  more generally,

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

General Cache Mechanics



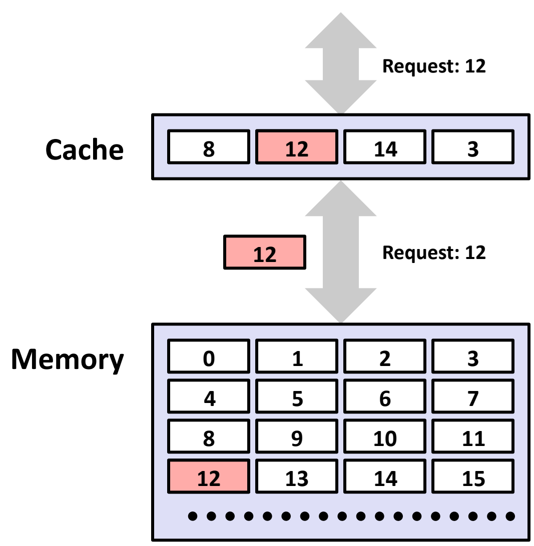
General Cache Concepts: Hit



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)

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 numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (**trend: increasing!**)

Lets think about those numbers

- **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
  - Average access 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"**

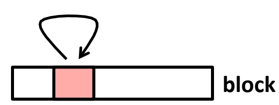
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block
- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block i must be placed in slot (i mod size)), direct-mapped
    - if more than one, n-way set-associative (where n is a power of 2)
  - 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
- **Capacity miss**
  - Occurs when the set of active cache blocks (the working set) is larger than the cache (just won't fit)

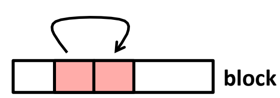
Why Caches Work

- **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



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]

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]

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

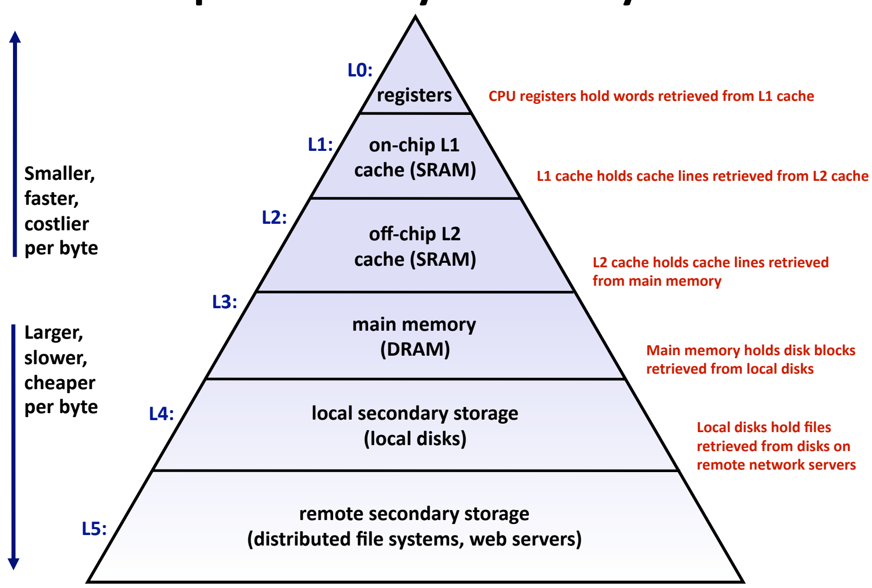
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    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?

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



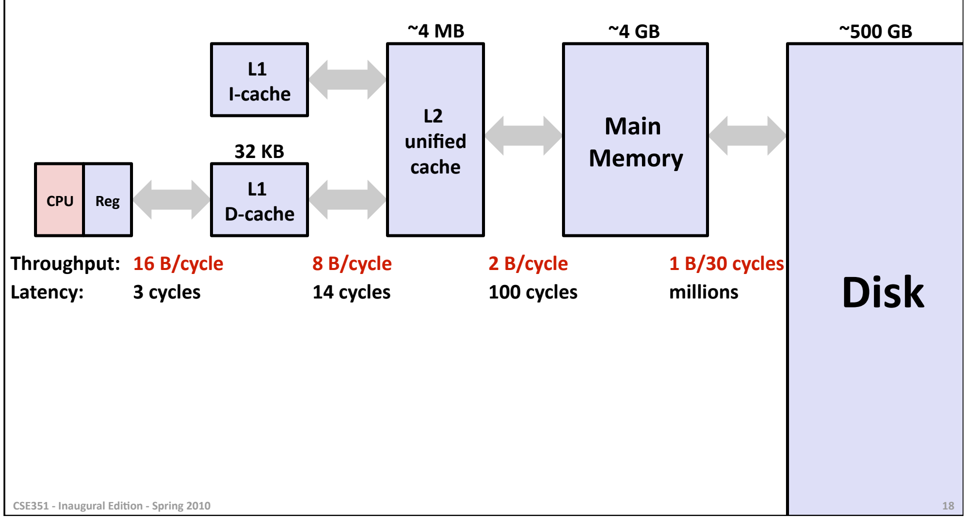
Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+OS
Buffer cache	Parts of files	Main memory	100	OS
Network cache	Parts of files	Local disk	10,000,000	File system client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web server

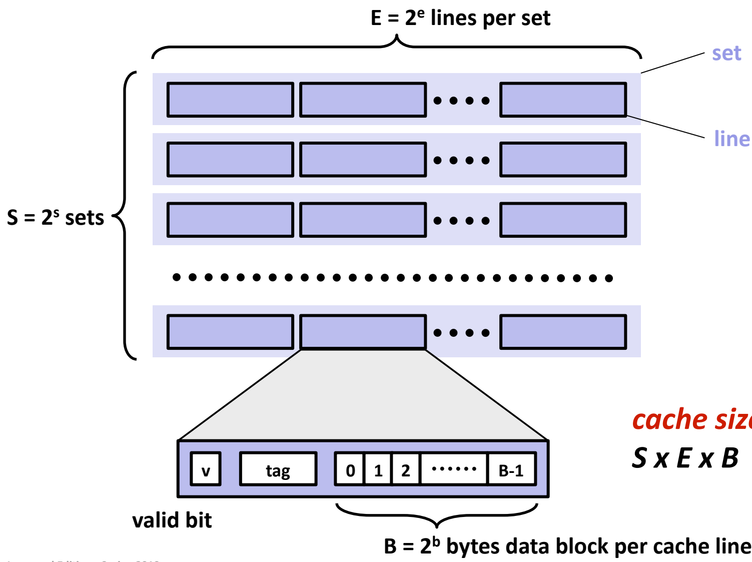
Memory Hierarchy: Core 2 Duo

Not drawn to scale

L1/L2 cache: 64 B blocks



General Cache Organization (S, E, B)

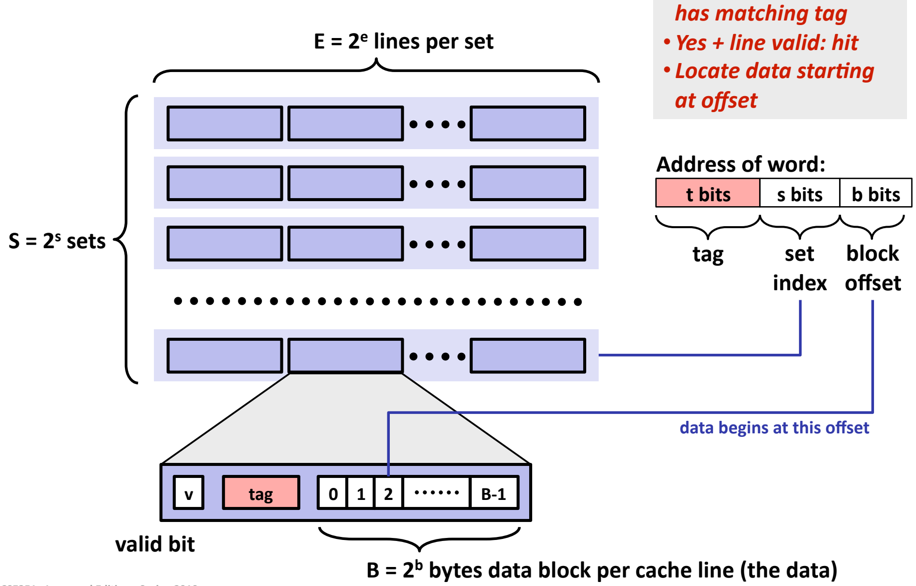


cache size:

S x E x B data bytes

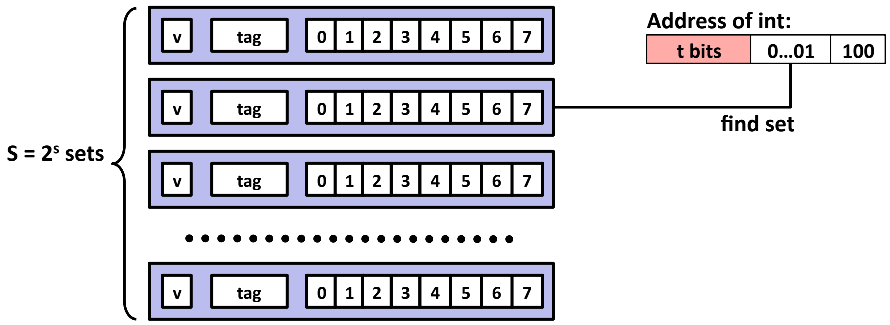
Cache Read

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



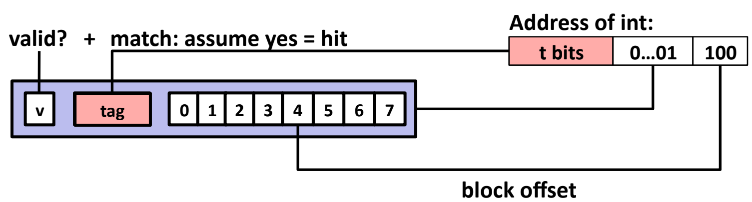
Example: Direct-Mapped Cache (E = 1)

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



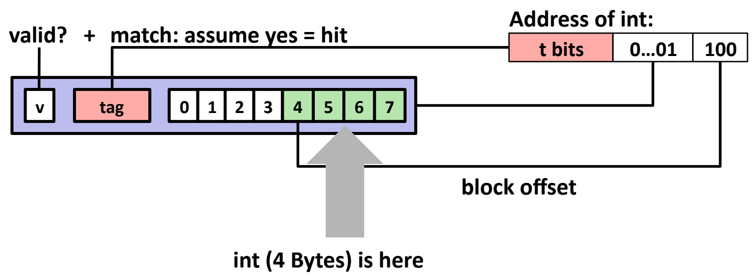
Example: Direct-Mapped Cache (E = 1)

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

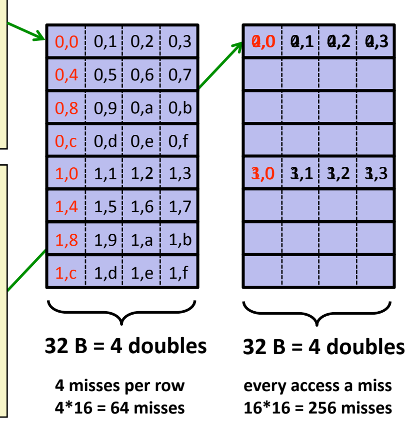
Example (for E =1)

Assume sum, i, j in registers
Address of an aligned element of a: aa....aaxxxxyyyy000

Assume: cold (empty) cache
3 bits for set, 5 bits for byte
aa....aaxxxx xyy yy000

```
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;
}
```

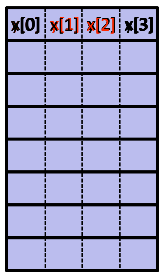
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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



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;
}
```

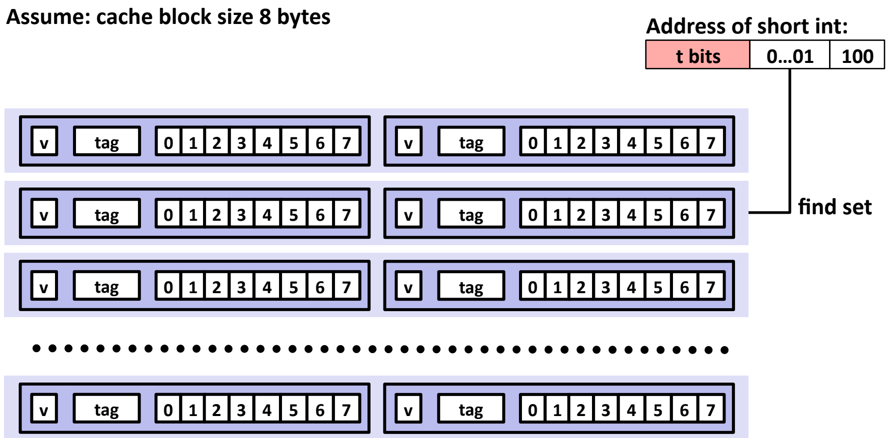
if x and y have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128



if x and y have unaligned starting addresses, e.g., &x[0] = 0, &y[0] = 144

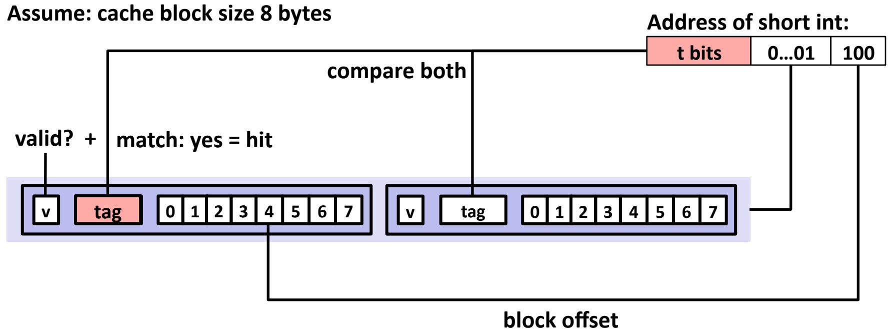
E-way Set-Associative Cache (Here: E = 2)

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



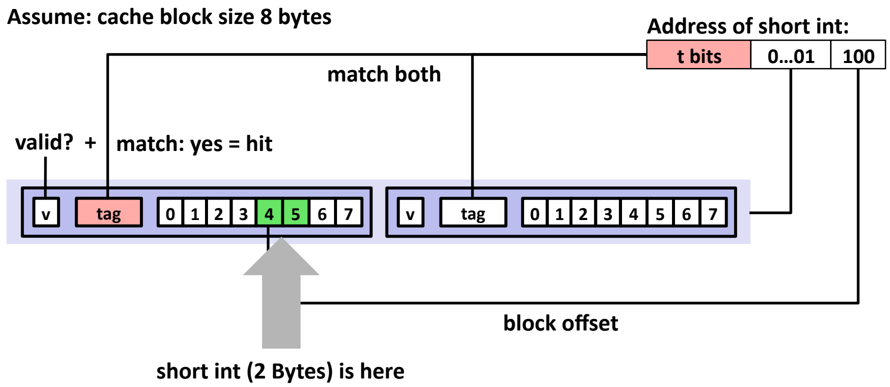
E-way Set-Associative Cache (Here: E = 2)

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;
}
```

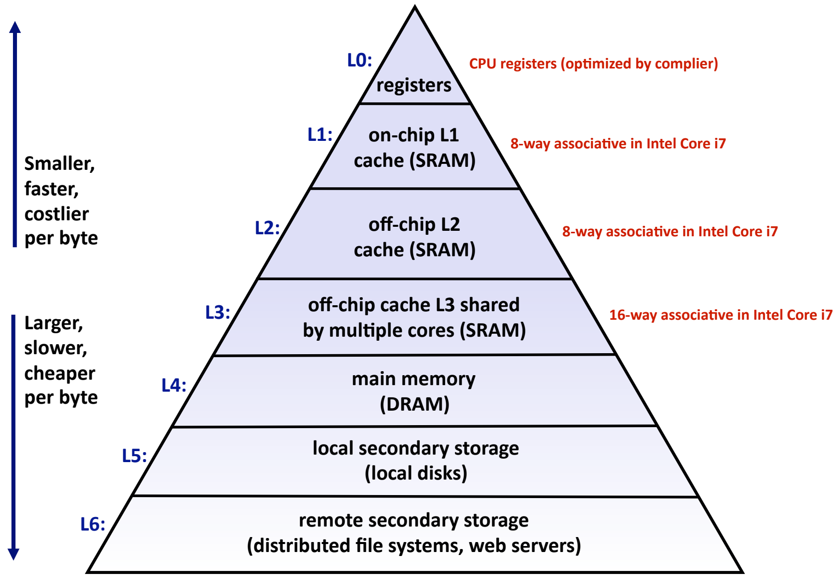
if x and y have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128 still can fit both because 2 lines in each set

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]

Fully Set-Associative Caches (S = 1)

- **All lines in one single set, S = 1**
  - E = C / B, where C is total cache size
  - S = 1 = (C / B) / E
- **Direct-mapped caches have E = 1**
  - S = (C / B) / E = C / B
- **Tags are more expensive in associative caches**
  - Fully-associative cache, C / B tag comparators
  - Direct-mapped cache, 1 tag comparator
  - In general, E-way set-associative caches, E tag comparators
- **Tag size, assuming m address bits (m = 32 for IA32)**
  - $m - \log_2 S - \log_2 B$

Typical Memory Hierarchy (Intel Core i7)



What about writes?

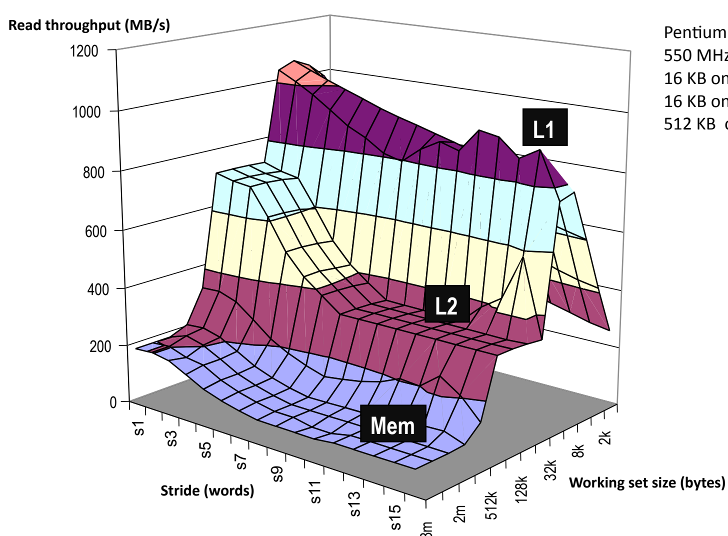
- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- **Typical**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

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

The Memory Mountain



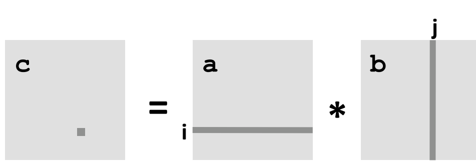
Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

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 to achieve?**
  - Proper choice of algorithm
  - Loop transformations
- **Cache versus register-level optimization:**
  - In both cases locality desirable
  - Register space much smaller
    + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

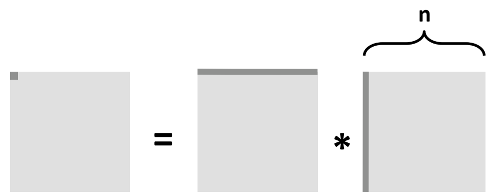
Example: Matrix Multiplication

```
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];
}
```

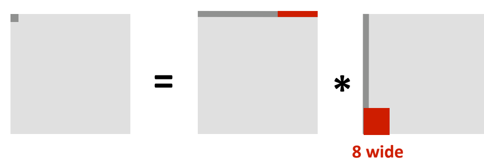


Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- **First iteration:**
  - n/8 + n = 9n/8 misses
    (omitting matrix c)

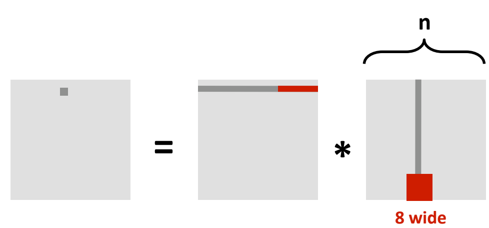


  - Afterwards in cache:
    (schematic)



Cache Miss Analysis

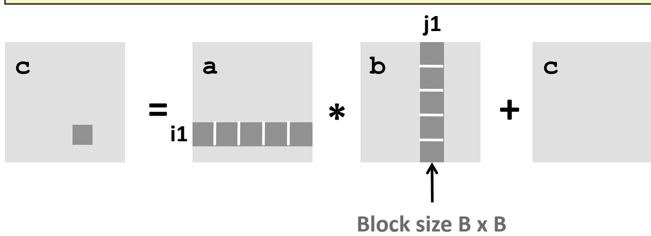
- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- **Other iterations:**
  - Again:
    n/8 + n = 9n/8 misses
    (omitting matrix c)



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

Blocked Matrix Multiplication

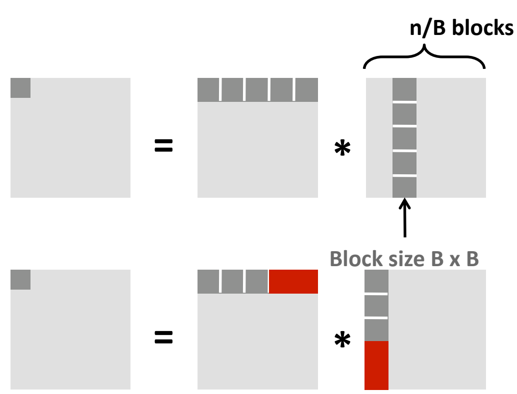
```
c = (double *) calloc(sizeof(double), n*n);

/* 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; i++)
                      for (j1 = j; j1 < j+B; j++)
                          for (k1 = k; k1 < k+B; k++)
                              c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache Miss Analysis

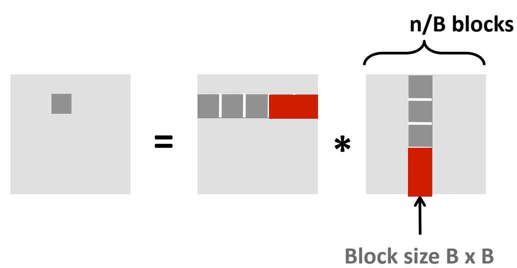
- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Four blocks fit into cache: $4B^2 < C$

- **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 = 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 $4B^2 < C$!**
  (can possibly be relaxed a bit, but there is a limit for B)
- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly