

Cache Memories

CS-281: Introduction to Computer Systems

Instructor:

Thomas C. Bressoud

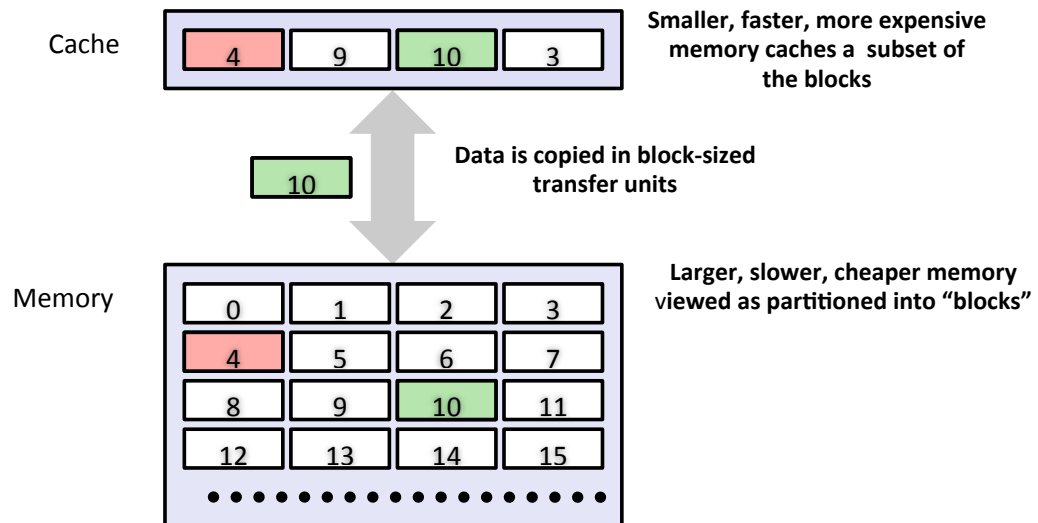
1

Caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- 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.

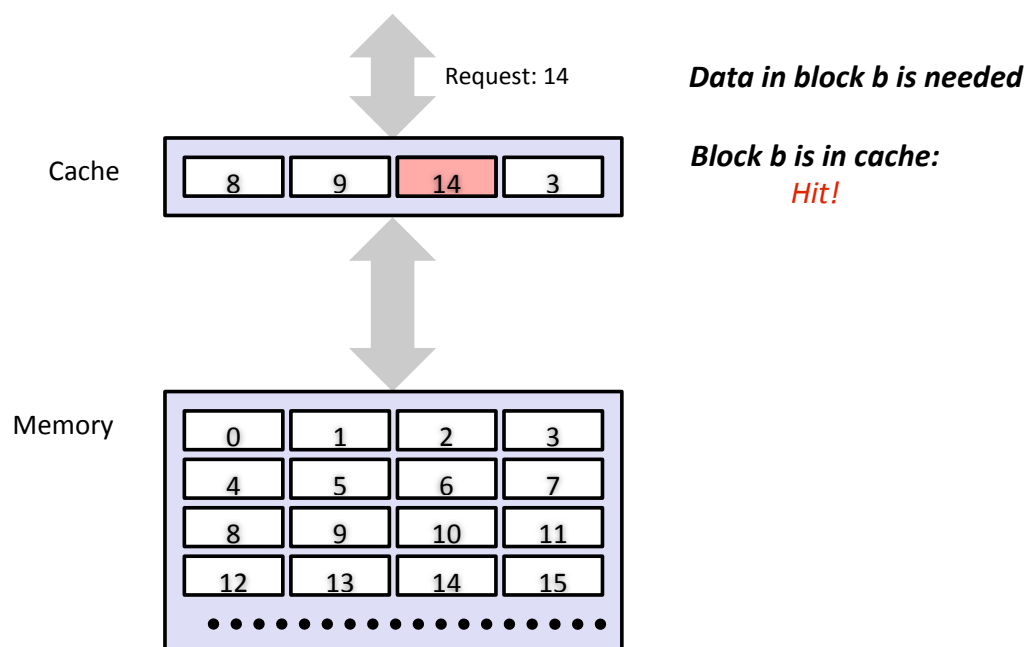
2

General Cache Concepts



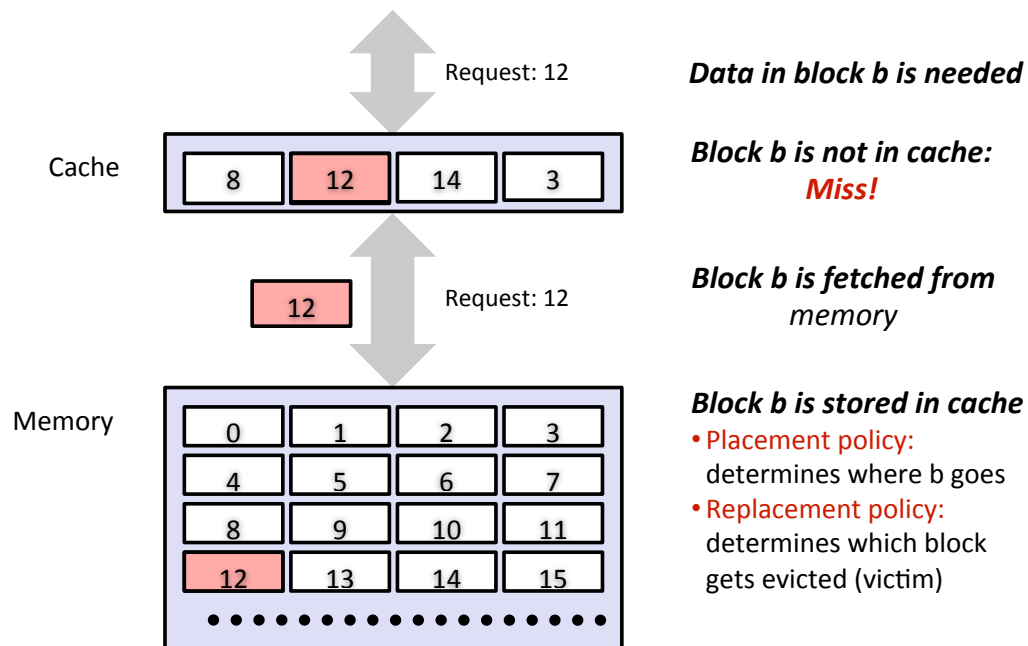
3

General Cache Concepts: Hit



4

General Cache Concepts: Miss



5

General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
 - Cold misses occur because the cache is empty.
- **Conflict miss**
 - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
 - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
 - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
 - Occurs when the set of active cache blocks (**working set**) is larger than the cache.

6

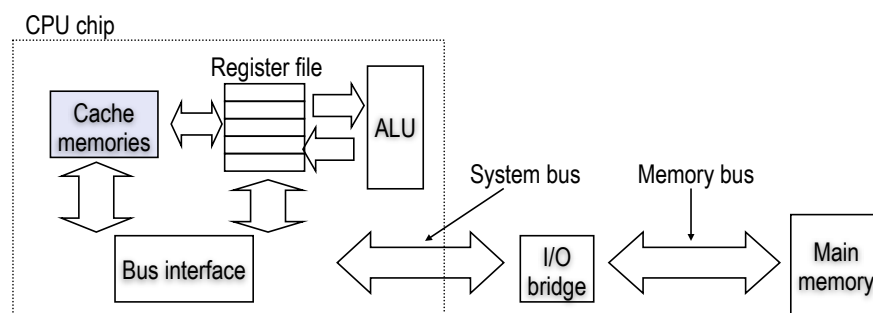
Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes 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	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

7

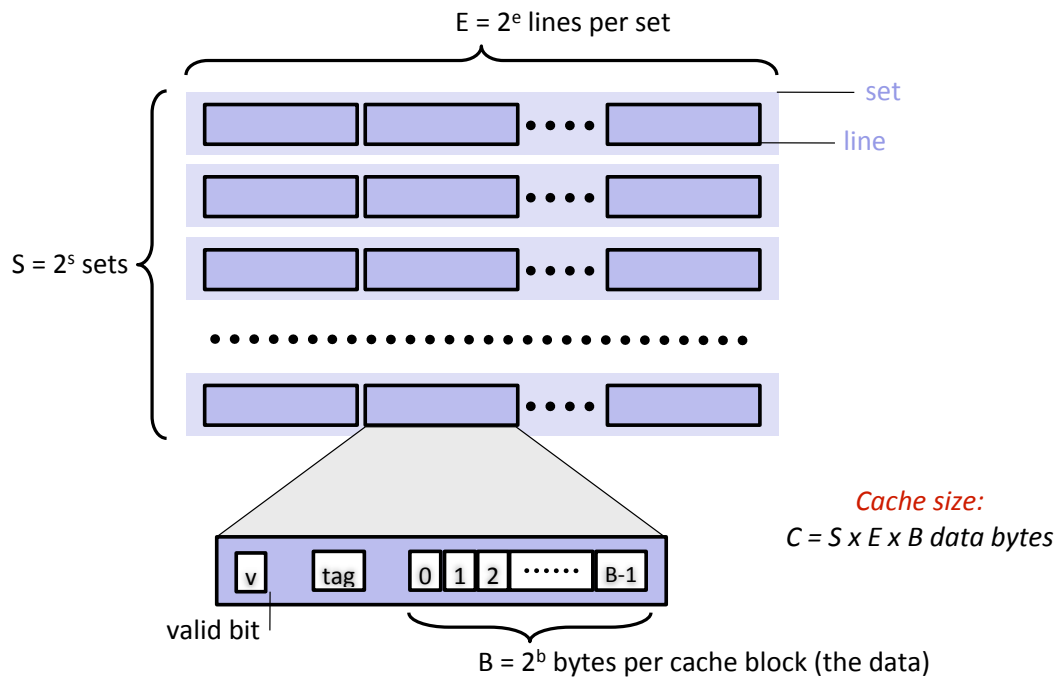
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
 - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



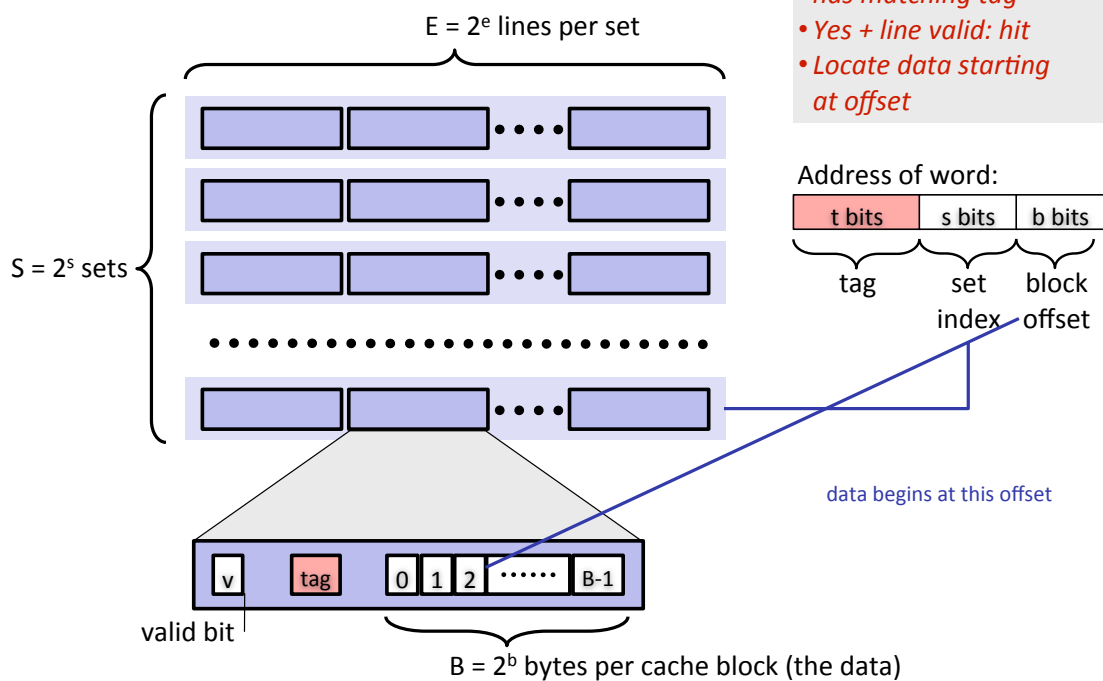
8

General Cache Organization (S, E, B)



9

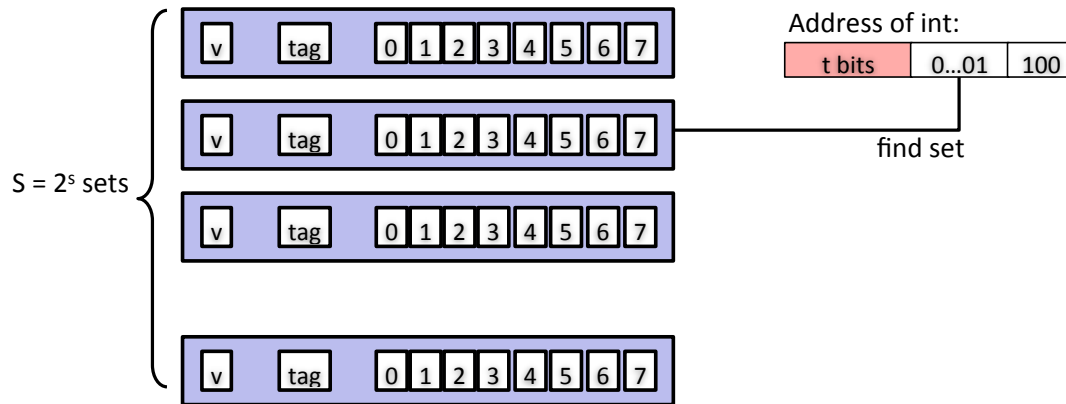
Cache Read



10

Example: Direct Mapped Cache (E = 1)

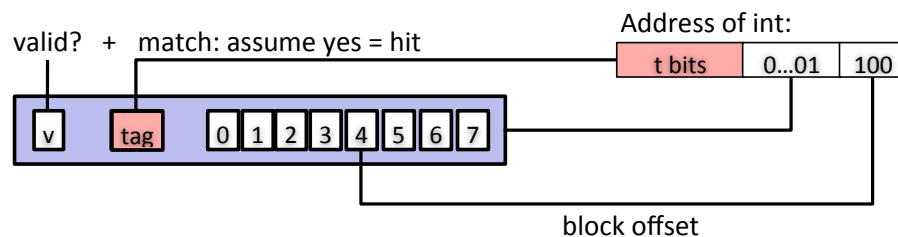
Direct mapped: One line per set
Assume: cache block size 8 bytes



11

Example: Direct Mapped Cache (E = 1)

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

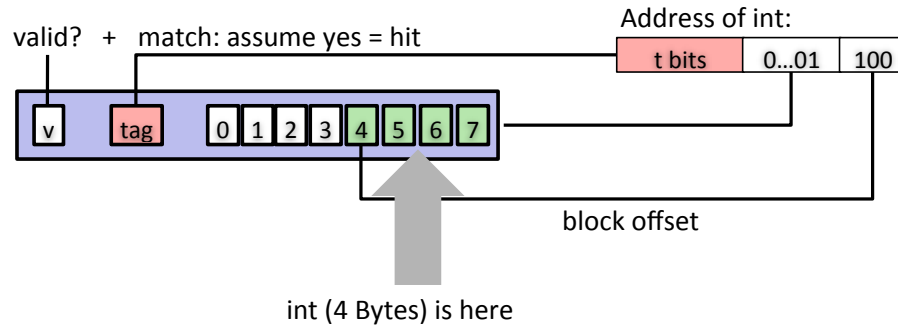


12

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

13

Direct-Mapped Cache Simulation

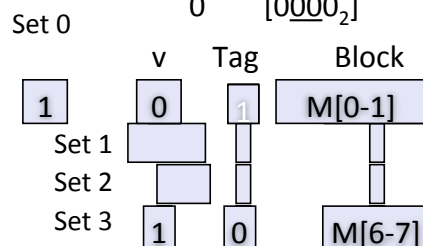
t=1 s=2 b=1

x xx x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

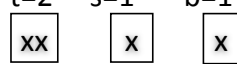
Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	miss



14

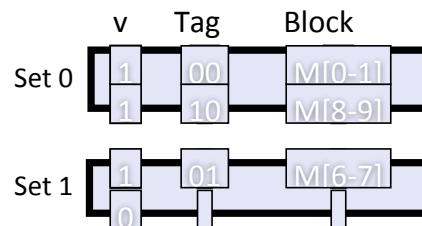
2-Way Set Associative Cache Simulation

t=2 s=1 b=1


M=16 byte addresses, B=2 bytes/block,
 S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	hit



15

A Higher Level Example

```
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;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
 a[0][0] goes here



32 B = 4 doubles

blackboard

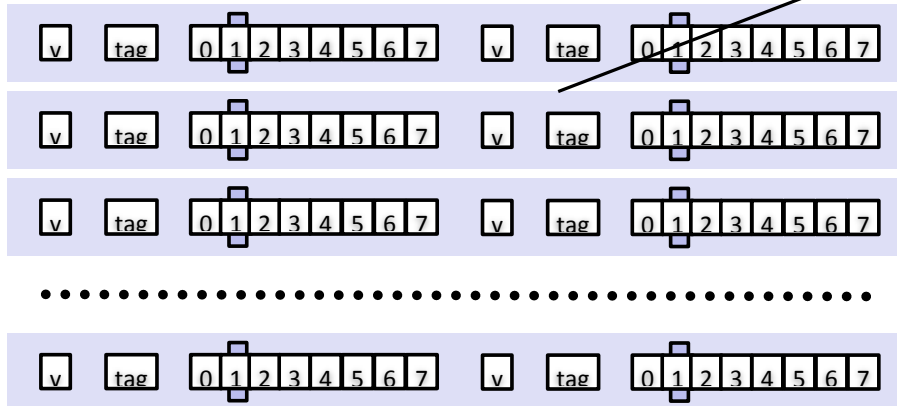
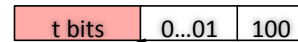
16

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

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



find set

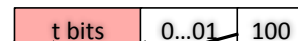
17

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

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



compare both

valid? + match: yes = hit



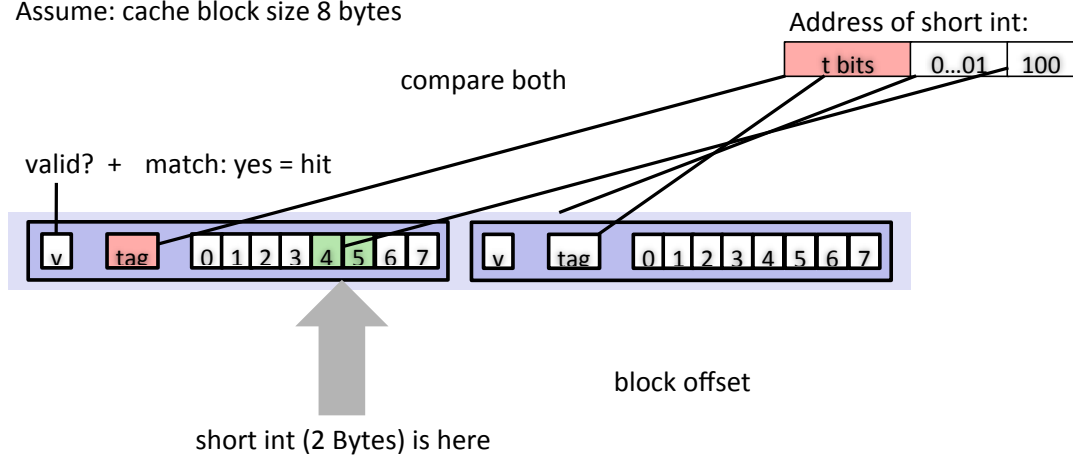
block offset

18

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), ...

19

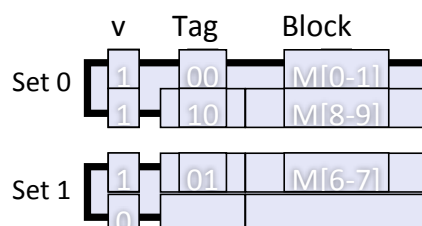
2-Way Set Associative Cache Simulation

t=2 s=1 b=1

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	hit



20

A Higher Level Example

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

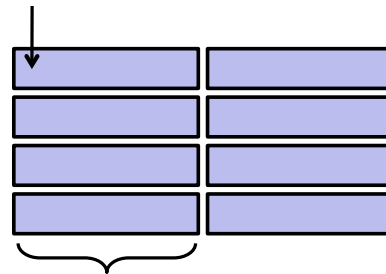
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(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;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

blackboard

21

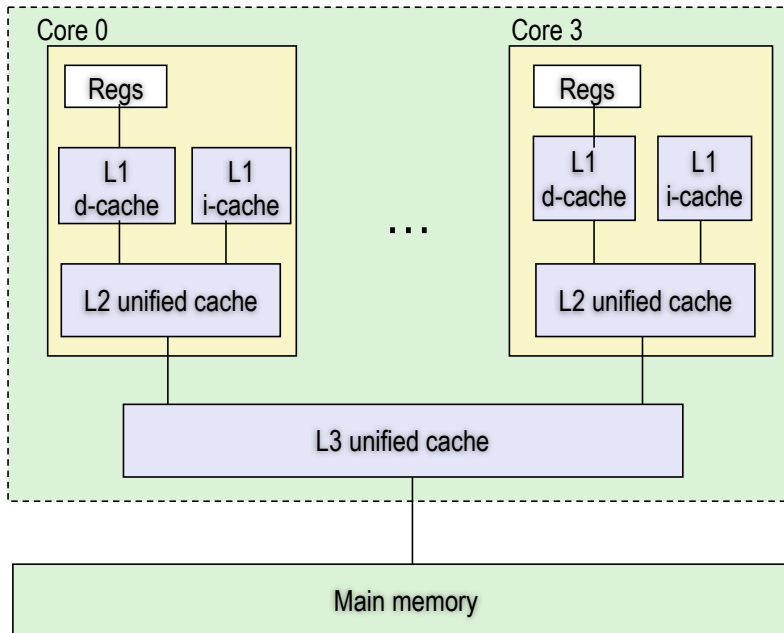
What about writes?

- 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**

22

Intel Core i7 Cache Hierarchy

Processor package



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.

23

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!)

24

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: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- This is why “miss rate” is used instead of “hit rate”

25

Writing Cache Friendly Code

- Make the common case go fast
 - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
 - Repeated references to variables are good (**temporal locality**)
 - Stride-1 reference patterns are good (**spatial locality**)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.

26

The Memory Mountain

- **Read throughput** (read bandwidth)
 - Number of bytes read from memory per second (MB/s)
- **Memory mountain**: Measured read throughput as a function of spatial and temporal locality.
 - Compact way to characterize memory system performance.

27

Memory Mountain Test Function

```

/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);
    cycles = fcyc2(test, elems, stride, 0); /* warm up the cache */
    return (size / stride) / (cycles / Mhz); /* call test(elems, stride) */
}

```

28

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

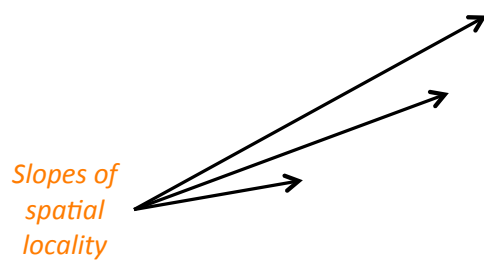
All caches on-chip

29

The Memory Mountain

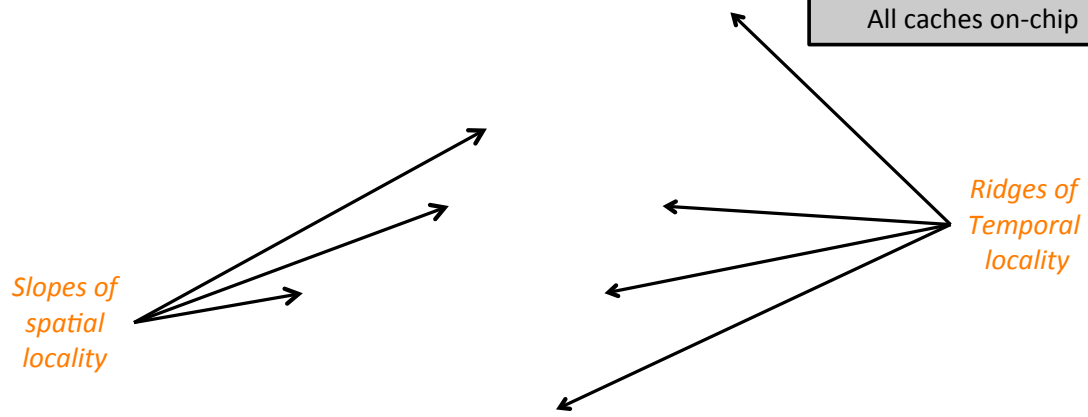
Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip



30

The Memory Mountain



31

Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

32

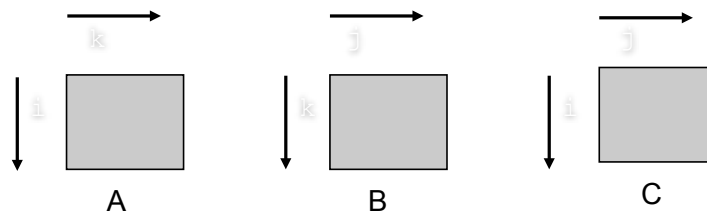
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 64-bit words)
- Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop



33

Matrix Multiplication Example

Description:

- Multiply N x N matrices
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
 - but may be able to hold in register

```

/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```

Variable *sum* held in register

34

Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
 - each row in contiguous memory locations
- Stepping through columns in one row:
 - `for (i = 0; i < N; i++)`
 `sum += a[0][i];`
 - accesses successive elements
 - if block size (B) > 4 bytes, exploit spatial locality
 - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
 - `for (i = 0; i < n; i++)`
 `sum += a[i][0];`
 - accesses distant elements
 - no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

35

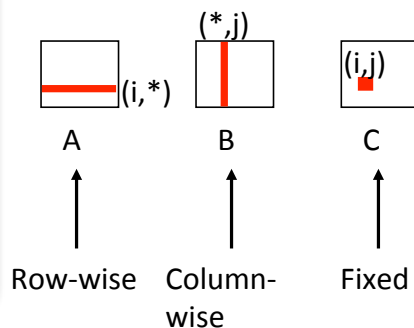
Matrix Multiplication (ijk)

```

/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

36

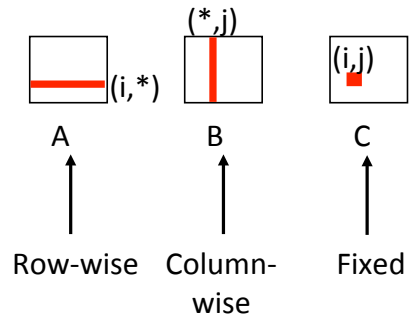
Matrix Multiplication (jik)

```

/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

37

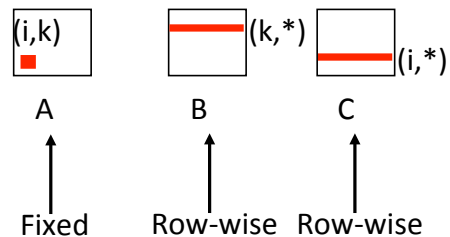
Matrix Multiplication (kij)

```

/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

38

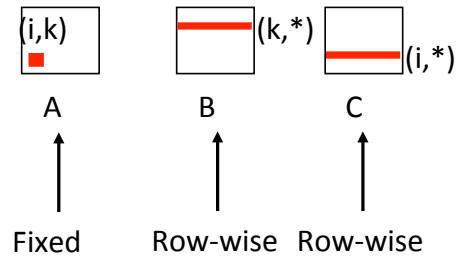
Matrix Multiplication (ikj)

```

/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

39

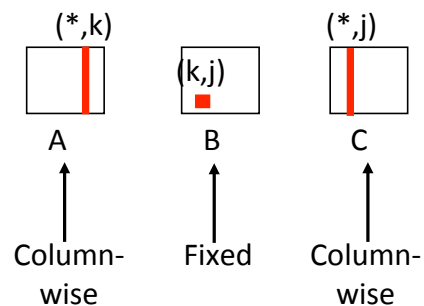
Matrix Multiplication (jki)

```

/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

40

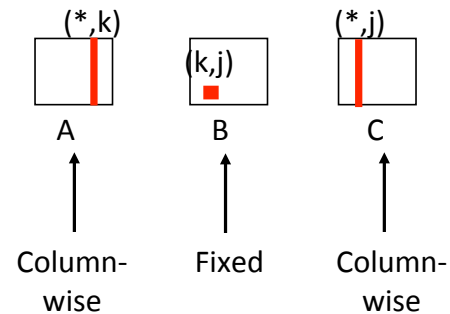
Matrix Multiplication (kji)

```

/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

41

Summary of Matrix Multiplication

```

for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```

for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}

```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```

for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}

```

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

42

Core i7 Matrix Multiply Performance

jki / kji

ijk / jik

kij / ikj

43

Today

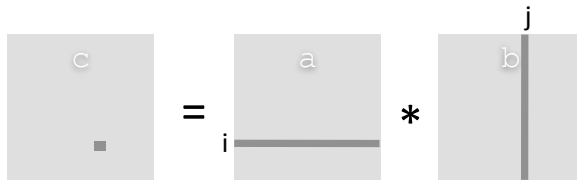
- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

44

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];
}
```



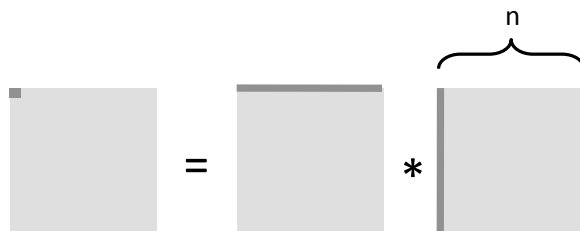
45

Cache Miss Analysis

- Assume:
 - Matrix elements are doubles
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (much smaller than n)

- First iteration:

- $n/8 + n = 9n/8$ misses



- Afterwards in cache:
(schematic)



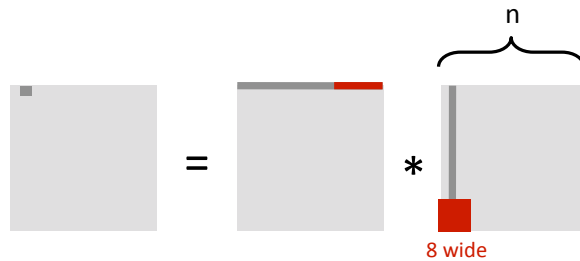
46

Cache Miss Analysis

- Assume:
 - Matrix elements are doubles
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (much smaller than n)

- Second iteration:

- Again:
- $n/8 + n = 9n/8$ misses



- Total misses:

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

47

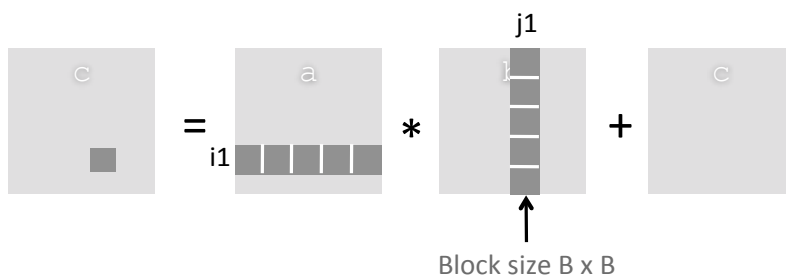
Blocked Matrix Multiplication

```

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];
}


```



48

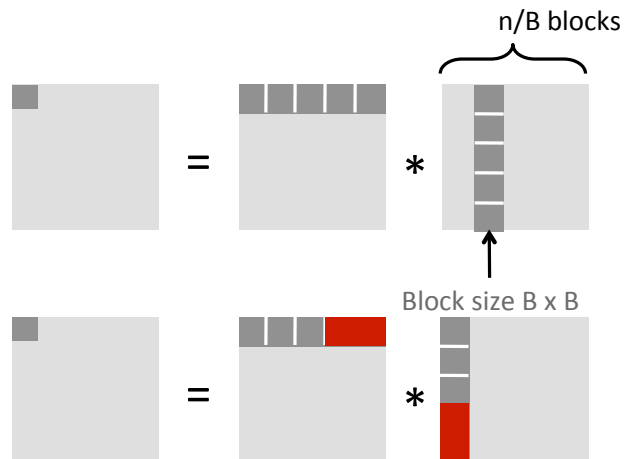
Cache Miss Analysis

Assume:

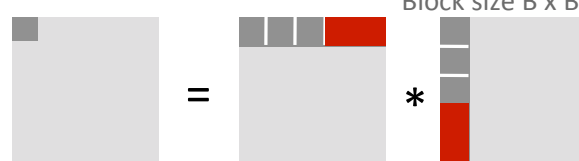
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks  fit into cache: $3B^2 < C$

First (block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$
(omitting matrix c)




- Afterwards in cache



49

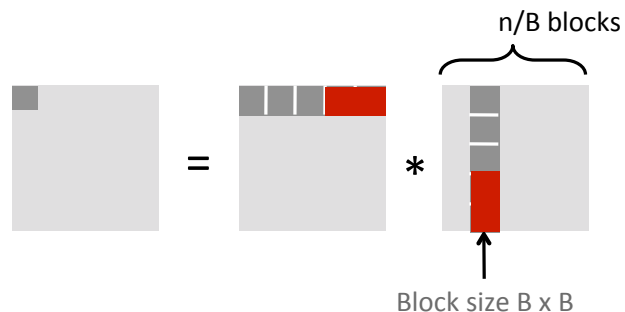
Cache Miss Analysis

Assume:

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

Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

50

Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Suggest 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 elements used $O(n)$ times!
 - But program has to be written properly

51

Concluding Observations

- 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)

52