Memory Hierarchy

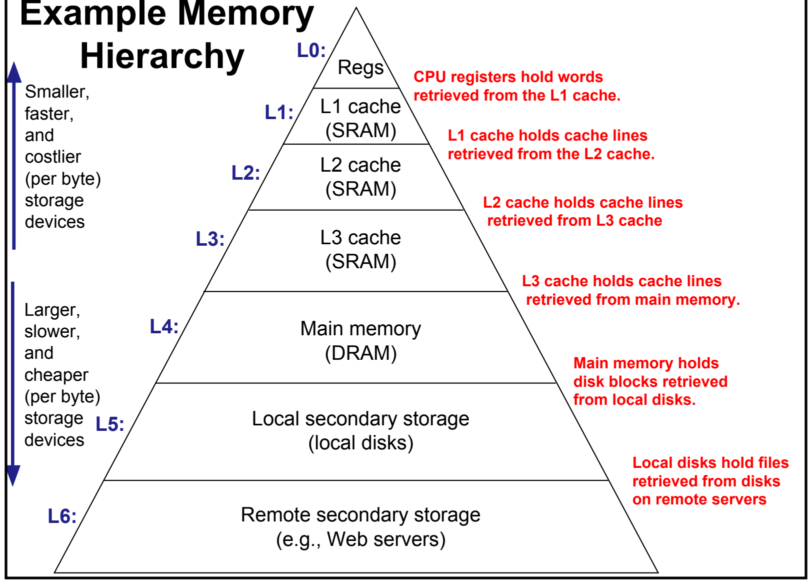
Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

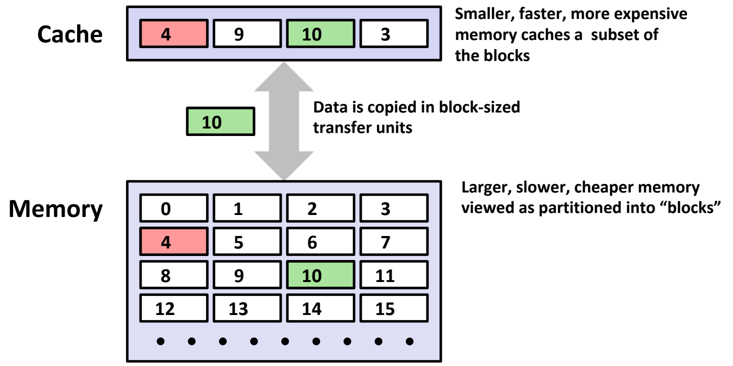
Cache Memory Organization and Access

Example Memory Hierarchy



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
CPU registers hold words retrieved from the L1 cache.
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache.
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache
L3:
L3 cache (SRAM)
L3 cache holds cache lines retrieved from main memory.
L4:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L5:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote servers
L6:
Remote secondary storage (e.g., Web servers)

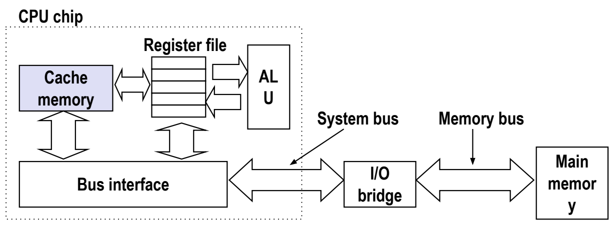
General Cache Concept



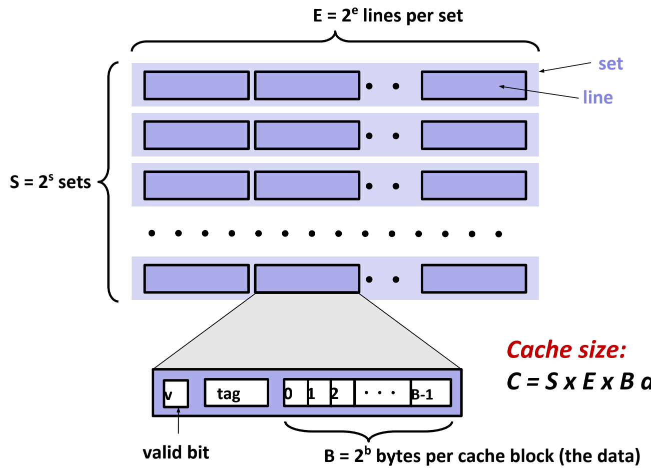
Cache
4
9
10
3
Smaller, faster, more expensive memory caches a subset of the blocks
10
Data is copied in block-sized transfer units
Memory
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
Larger, slower, cheaper memory viewed as partitioned into "blocks"

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 cache
- Typical system structure:



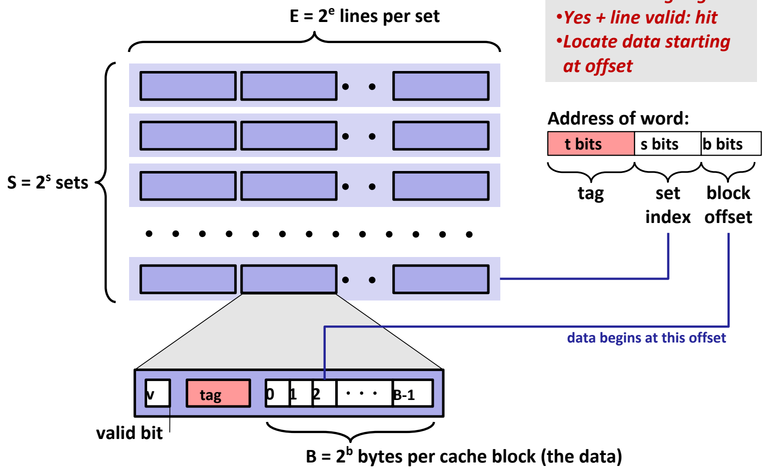
General Cache Organization (S, E, B)



Cache size:
C = S x E x B data bytes

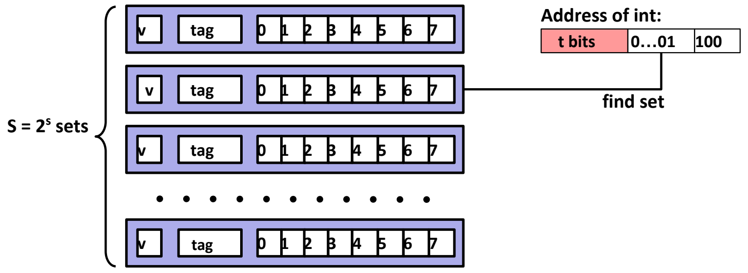
Cache Read

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



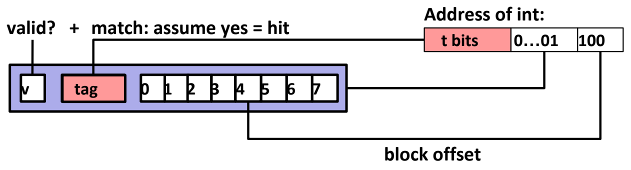
Example: Direct Mapped Cache (E = 1)

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



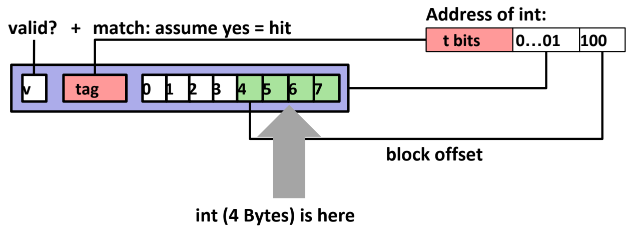
Example: Direct Mapped Cache (E = 1)

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



If tag doesn't match: old line is evicted and replaced

Example: Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

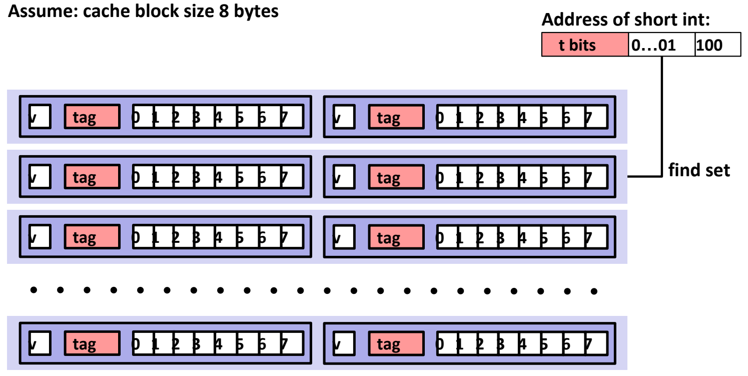
Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	miss
1	$[0\underline{00}1_2]$,	hit
7	$[0\underline{11}1_2]$,	miss
8	$[1\underline{00}0_2]$,	miss
0	$[0\underline{00}0_2]$	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

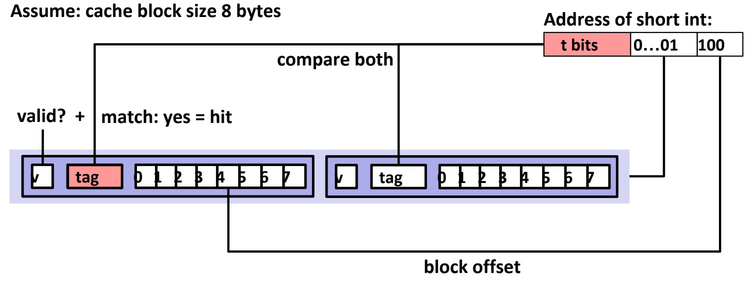
E-way Set Associative Cache (Here: E = 2)

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



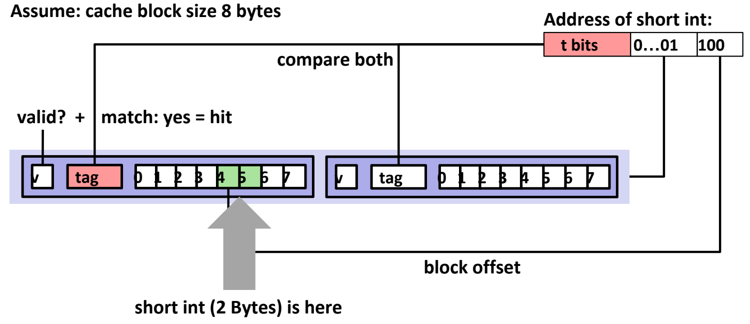
E-way Set Associative Cache (Here: E = 2)

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: 2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

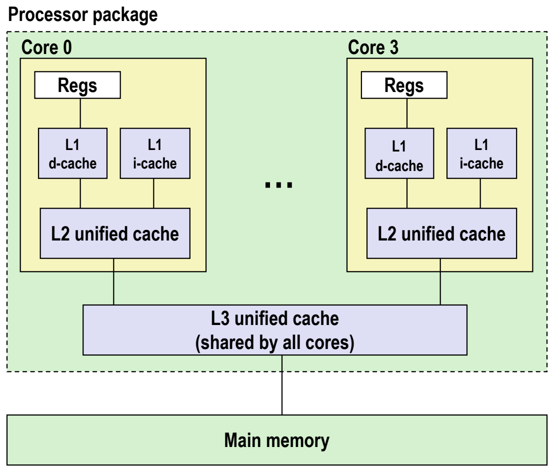
0	[0000$_2$],	miss
1	[0001$_2$],	hit
7	[0111$_2$],	miss
8	[1000$_2$],	miss
0	[0000$_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, 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 straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Intel Core i7 Cache Hierarchy Example



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

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

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 very 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:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
    - typically **50-200 cycles** for main memory

Ouch!

Let's 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: 0.97*1 cycle + 0.03 * 100 cycles ≈ 1 cycle + 3 cycles = **4 cycles**
    99% hits: 0.99*1 cycle + 0.01 * 100 cycles ≈ 1 cycle + 1 cycle = **2 cycles**
- **This is why "miss rate" is used instead of "hit rate"**

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**) because there is a good chance that they are stored in registers.
  - Stride-1 reference patterns are good (**spatial locality**) because subsequent references to elements in the same block will be able to hit the cache (one cache miss followed by many cache hits).

Rearranging Loops to Improve Spatial Locality

Matrix Multiplication Example

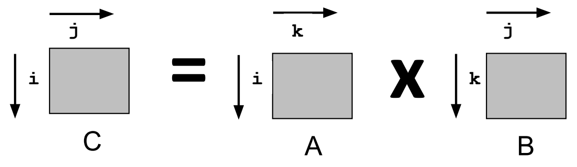
Variable `sum` held in register

```
/* 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;
  }
}
```

- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - 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



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];
```

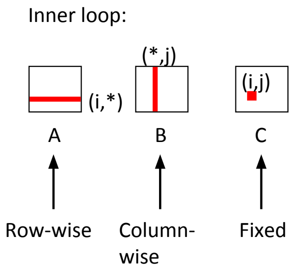
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:

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

  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

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;
  }
}
```



We can reorganize the loops in several different ways.
Which organization gives the best cache performance?

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)

```
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)

?

```
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)

Measuring Cache Misses

`valgrind` with cachegrind - cache simulator

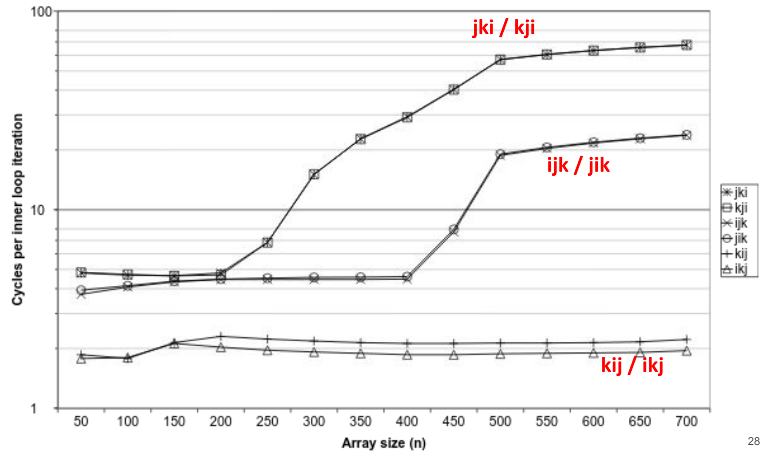
```
       D1mr        D1mw          function
-----------------------------------------
2,977,062,519         0             ijk
4,004,000,016         0             jki
1,253,500,008         0             jik
  750,750,003         0             kji
  252,004,004        12             ikj
  252,004,004        12             kij
```

L1 cache = 1024B
16 way associative
32 B cache line

```
       D1mr        D1mw          function
-----------------------------------------3
,133,750,020          0             ijk
5,005,000,020          0             jki
5,005,000,020          0             kji
2,820,375,018          0             jik
  189,003,003          9             ikj
  189,003,003          9             kij
```

L1 cache = 1024B
4 way associative
32 B cache line

Core i7 Matrix Multiply Performance



Learn about your machine's cache

- `lshw` **command - list hardware information**
  - `sudo lshw -C memory`
- `lscpu` **command - display information about the CPU architecture**
  - `lscpu`
- `dmidecode` **command -**
  - `sudo dmidecode -t cache`

Note: some of these may not work well in a virtual machine environment.

Cache Summary

- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.