outlineL16-w8TR-student

Monday, October 31, 2022 8:19 PM

outlineL16-w8TR-student

CS 354 - Machine Organization & Programming
Tuesday Oct 25th and Thursday Oct 27th,2022

Midterm Exam 2 - Thursday Nov 10th, 7:30 - 9:30 pm

- **UW ID required**
- **#2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches)**
  see "Midterm Exam 2" on course site Assignments for topics

Homework hw3: DUE on or before Monday, Oct 24th

Homework hw4: DUE on or before Monday, Nov 7th

Project p3: DUE on or before Friday, Oct 28th

Project p4A: DUE on or before Friday, Nov 4th

Project p4B: DUE on or before Friday, Nov 11th

Last Week

Locality & Caching Bad Locality Caching: Basic Idea & Terms Designing a Cache: Blocks	Rethinking Addressing Designing a Cache: Sets and Tags Basic Cache Lines Basic Cache Operation Basic Cache Practice

This Week

Direct Mapped Caches - Restrictive Fully Associative Caches - Unrestrictive Set Associative Caches - Sweet! Replacement Policies	Writing to Caches Cache Performance Impact of Stride Memory Mountain
Next Week: Assembly Language Instructions and Operands B&O Chapter 3 Intro 3.1 A Historical Perspective 3.2 Program Encodings 3.3 Data Formats 3.4 Accessing Information 3.5 Arithmetic and Logical Control 3.6 Control	

Direct Mapped Caches - Restrictive

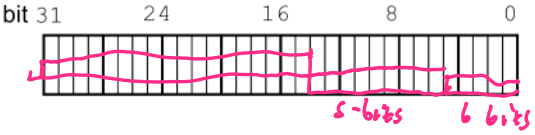
Direct Mapped Cache is a cache having S sets with 1 line per set where more blocks map to exactly one set

→ What is the address breakdown if blocks are 32 bytes and there are 1024 sets?

B = 32 = 2^5 S = 1024 = 2^{10}

5 b bits lo 5-bits

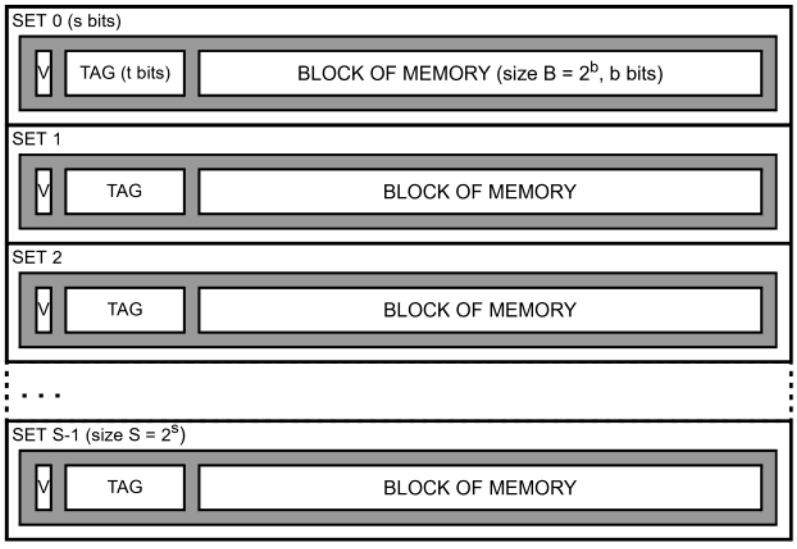
32-bit Address Breakdown



→ Is the cache operation fast O(1) or slow O(S) where S is the number of sets? FAST!

O(1) set selection + O(1) line matching

+ simple circuitry to determine if tag matches t-bits



→ What happens when two different memory blocks map to the same set?

Conflict miss - set stores one block only

thrashing - continually replace block in set

❋ *Appropriate for* larger caches (L3)

Fully Associative Caches - Unrestrictive

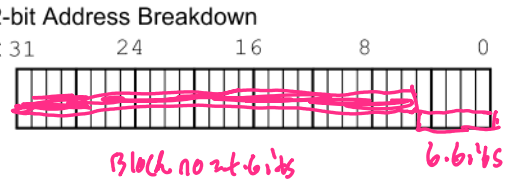
Fully Associative Cache is a cache having one set with E lines where any blocks can be stored in any line

→ What is the address breakdown if blocks are 32 bytes and there ~~are 1024 sets~~?

B = 32 = 2^5 → 5 6-bits

1 set.

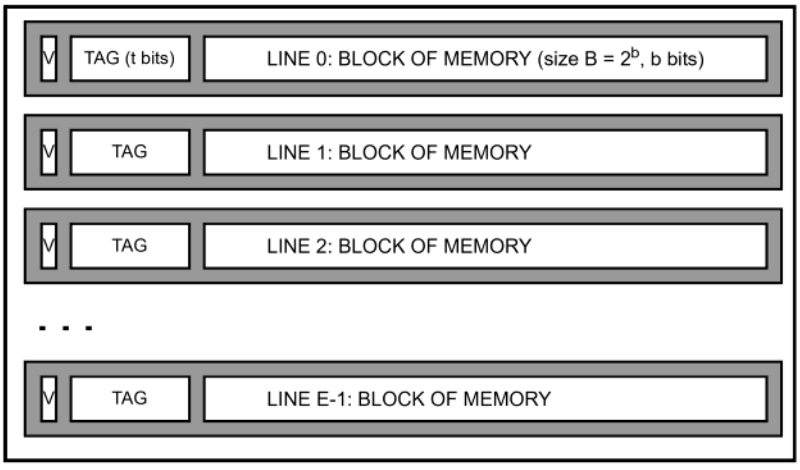
32-bit Address Breakdown



→ Is the cache operation fast O(1) or slow O(E) where E is the number of lines? slow!

O(1) set selection + O(E) line searching

- complex circuitry to match t-bits in any line
- more t-bits to compare



→ What happens when two different memory blocks map to the same set?

choose a free line + reduces conflicts

→ How many lines should a fully associative cache have?

First, determine cache and block size (B)

C = S · E · B

C = 6 · B ⇒ E = 6/B

C = 1 · E · B

❋ *Appropriate for* small caches (L1)

Set Associative Caches - Sweet!

Set Associative Cache is a cache commonly used today

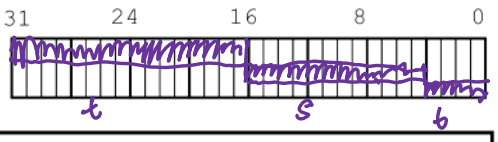
S sets w/ E lines per set

A memory block maps to one set, can be any line in that set

→ What is the address breakdown if blocks are 32 bytes and there are 1024 sets?

B=32 → 5 b bits S=1024 → 10 s bits

32-bit Address Breakdown

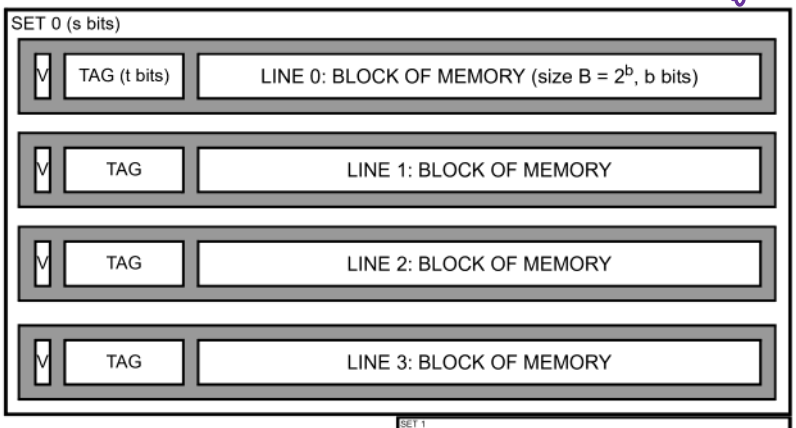


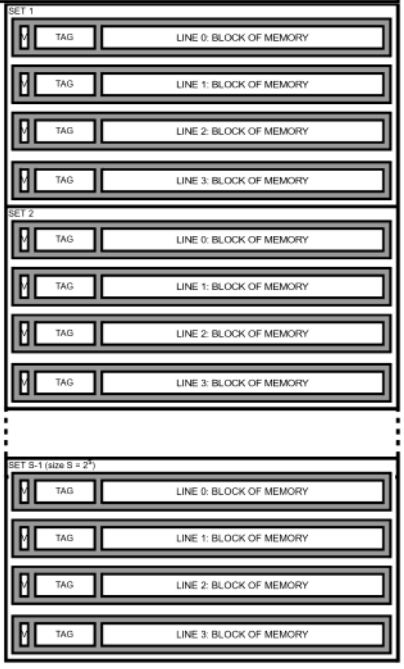
+ Fast

O(1) set selection

O(E) line matching

+ reduces conflict misses





Let E be number of lines in each set

→ "associativity" of the set

4-way associativity

E = 4 is

E = 1 is Direct map cache

❋ ***C = (S, E, B, m)*** ← m bits in address

Let C be cache size in bytes

C = S·E·B

→ How big is a cache given (1024, 4, 32, 32)?

C = 1024·4·32 = $2^{10}\cdot 2^{2}\cdot 2^{5} = 2^{17}$ = 128K

→ What happens when E+1 different memory blocks map to the same set?

Conflict miss → use replacement policy to pick a victim

Replacement Policies

Assume the following sequence of memory blocks

are fetched into the same set of a 4-way associative cache that is initially empty:

b1, b2, b3, b1, b3, b4, b4, b7, b1, b8, b4, b9, b1, b9, b9, b2, b8, b1

1. *Random Replacement*

→ Which of the following four outcomes is possible after the sequence finishes? Assume the initial placement is random.

	L0	L1	L2	L3
1.	b9	b1	b8	b2
2.	b1	b2	--	b8
3.	b1	b4	b7	b3
4.	b1	b2	b8	b1

2. *Least Recently Used* (LRU)

Track the line last used
maintain LRU queues - when line is used move to front, LRU is at back of queues
use status bits to track LRU

→ What is the outcome after the sequence finishes? Assume the initial placement is in ascending line order (left to right below).

L0 L1 L2 L3
b1 b2 b3

never longer than 4

3. *Least Frequently Used* (LFU)

Must keep track of how often a line is used
each line has a counter
- counter is zeroed when line gets a new block
- counter is incremented when line is accessed
- if tie choose randomly

→ Which blocks will remain in the cache after the sequence finishes?

L0 L1 L2 L3
b1 b2 b3 b4
1? 1 1

L0 L4 L2 L3
b1 b9 b8 b4

❋ *Exploiting replacement policies* to improve performance is HARD!
So programmers don't

Writing to a Cache

- *Reading data copies* a block of memory into cache levels
- *Writing data requires that* these copies are consistent

Write Hits

occur when writing to a block that is in the cache

→ When should a block be updated in lower memory levels?

1. ***Write Through***: write this cache and update lower level immediately
   - must wait for lower level to do write
   - more bus traffic passing all write-through
   + Or bypass cache with a write buffer
2. ***Write Back***: update next lower level when changed line is evicted
   + faster, no wait
   - must track if line is changed.
     add a "dirty bit" = line has changed.

update now / update next level now / write now, do next level later / later

Write Misses

occur when writing to a block that is not in the cache.

→ Should space be allocated in this cache for the block being changed?

1. ***No Write Allocate***: write directly to next level (bypassing this cache)
   - must wait for lower-level cache to do write
2. ***Write Allocate***: read block into this cache, write to it
   - must wait to get block from next lower level
   - more bus traffic, must block to cache from lower level

Typical Designs

1. **Write Through** paired with no write allocate - get data to next lower level
2. **Write Back paired** with write allocate - keep data in current cache level "set dirty bit"

→ Which best exploits locality?

2. also symmetric with read

Cache Performance

Metrics

hit rate # of hits compared / # memory accesses, higher is better

hit time time to determine a hit (shorter = better)

= set selection + line matching

miss penalty, Additional time to process a miss (shorter = better)

Larger Blocks (S and E unchanged)

hit rate better (larger cache size), more spatial locality per block

hit time about the same (essentially)

miss penalty worse, longer to transfer larger blocks

THEREFORE Block sizes are relatively small

More Sets (B and E unchanged)

hit rate Better, helps with temporal locality

hit time worse, set selection will be slower

miss penalty same

THEREFORE faster caches have fewer sets i.e. (L1)

slower caches (L3) can be larger

More Lines E per Set (B and S unchanged)

hit rate better, more temporal locality, decreases conflict misses

hit time line matching will take longer → worse

miss penalty worse, more lines slows choosing a victim block

THEREFORE faster caches have fewer lines per set. (L1)

slower caches have more lines (L3)

Intel Quad Core i7 Cache (gen 7) in pic

all: 64 byte blocks, use pseudo LRU, write back

→ L1: 32KB, 4-way Instruction & 32KB 8-way Data, no write allocate ← 2 L1 Cache

→ L2: 256KB, 8-way, write allocate

→ L3: 8MB, 16-way (2MB/Core shared), write allocate

C

Impact of Stride

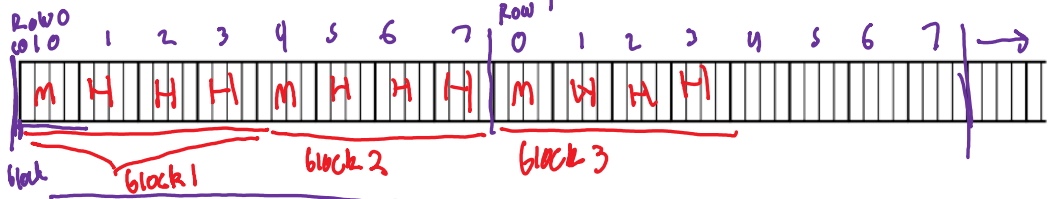
Stride Misses % of misses = $\min(1, (\text{wordsize} \cdot k)/B) \cdot 100$

R is stride length in words
B is block size in bytes

Example:

```
int initArray(int a[][8], int rows) {
   for (int i = 0; i < rows; i++)
      for(int j = 0; j < 8; j++)
         a[i][j] = i * j;
}
```

→ Draw a diagram of the memory layout of the first two rows of a:



Assume: a is aligned with cache blocks
→ is too big to fit entirely into the cache
→ words are 4 bytes, block size is 16 bytes
direct-mapped cache is initially empty, write allocate used

→ Indicate the order elements are accessed in the table below and mark H for hit or M for miss:

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	1 M	2 H	3 H	4 H	5 M	6 H	7 H	8 H
1	9 M	10 H	11 H	12 H	13 M	H	H	H
...								

plug into equation:

% misses = $\min(1, \text{word size} \cdot k)/B) \cdot 100 = \min(1, (4 \cdot 1)/16) \cdot 100 = \min(1, 0.25) \cdot 100 = 25\%$ misses

→ Now exchange the i and j loops mark the table again:

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	1 M							
1	2 M							
...	3 M							

$i \to \infty$

$\therefore$ 100% miss rate

% misses = $\min(1, (4 \cdot 8)/16) \cdot 100$
$= \min(1, 2) \cdot 100$
$\therefore 100\%$

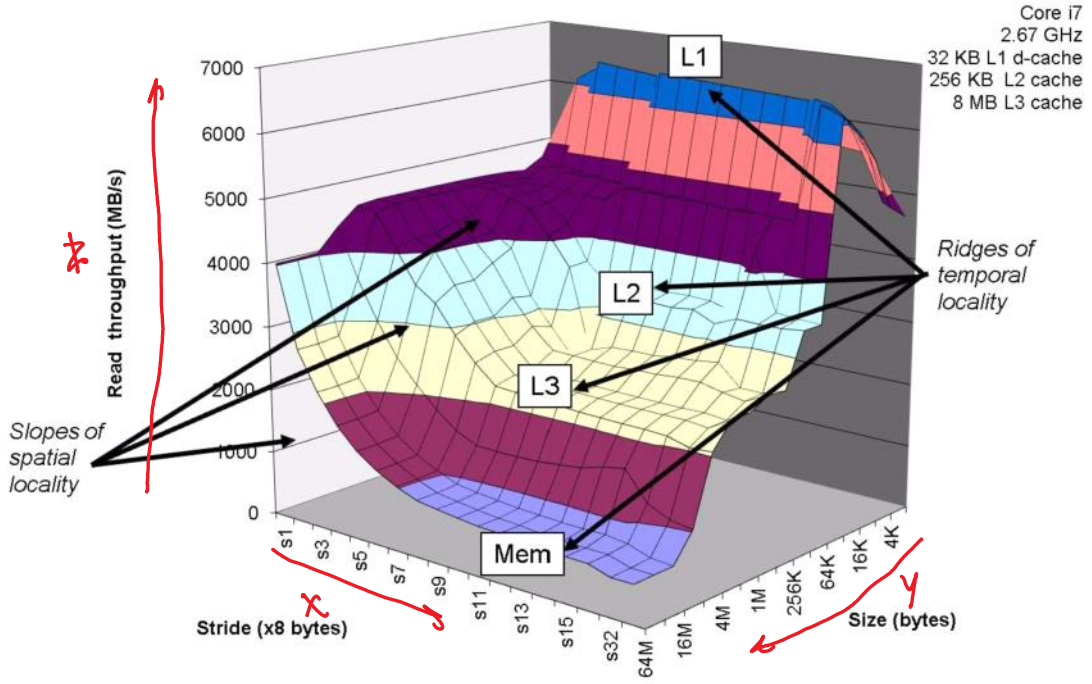
Memory Mountain

Independent Variables

x stride - 1 to 16 double words step size used to scan through array
y size - 2K to 64 MB arraysize

Dependent Variable

z read throughput - 0 to 7000 MB/s



Computer Systems, A Programmer's Perspective
Second Edition, Bryant and O'Hallaron

Temporal Locality Impacts

factor of ~10 (size) between L1 (6000 MB/s) and (600 MB/s)

Spatial Locality Impacts

(4000 MB/s)
factor of ~7 between stride=1 and 16 (600 MB/s)

* *Memory access speed is not characterized by a single value*
It is a landscape that can be exploited through utilizing spatial + temporal locality.