15-213

"The course that gives CMU its Zip!"

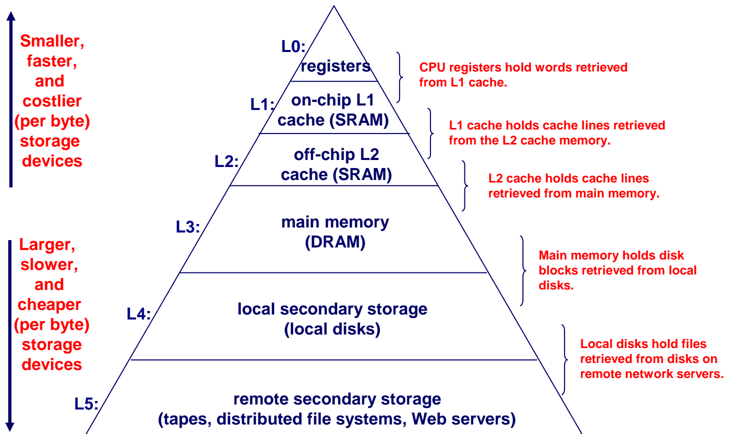
Cache Memories
September 28, 2007

Topics

- Locality of reference
- Caching in the memory hierarchy
- Generic cache memory organization
- Direct mapped caches
- Set associative caches

class10.ppt

An Example Memory Hierarchy



Locality

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
- **Temporal locality:** Recently referenced items are likely to be referenced in the near future.
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

Locality Example:

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

- Data
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**
- Instructions
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Locality Example

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?

```
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;
}
```

Locality Example

Question: Does this function have good locality?

```
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;
}
```


Locality Example

Question: Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

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

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


Caches

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

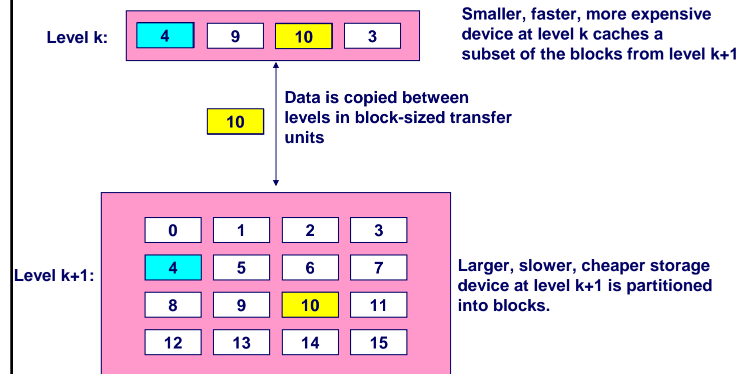
Fundamental idea of a memory hierarchy:

- For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

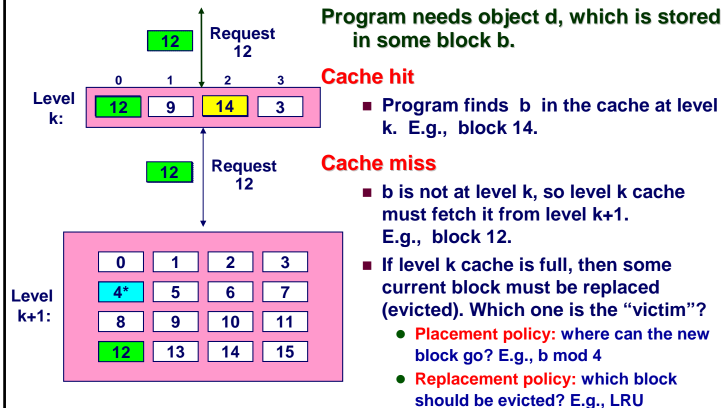
Why do memory hierarchies work?

- 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.
- Net effect: A large pool of memory 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.

Caching in a Memory Hierarchy



General Caching Concepts



Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

General Caching Concepts

Types of cache misses:

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
- 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+1.
  - 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.

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 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

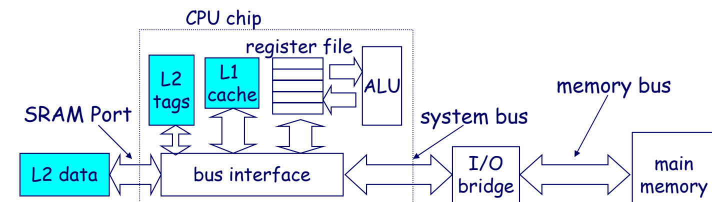
Cache Memories

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 L1, then in L2, then in main memory.

Typical system structure:



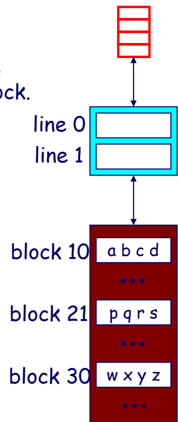
Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.

The tiny, very fast CPU register file has room for four 4-byte words.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The small fast L1 cache has room for two 4-word blocks.



The big slow main memory has room for many 4-word blocks.

General Organization of a Cache

Cache is an array of sets.

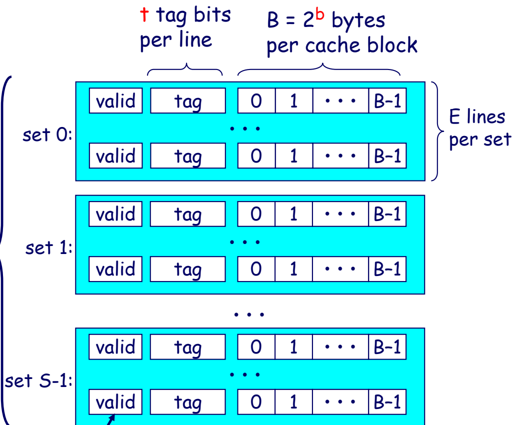
Each set contains one or more lines.

Each line holds a block of data.

$S = 2^s$ sets

t tag bits per line

$B = 2^b$ bytes per cache block



1 valid bit per line

Cache size: $C = B \times E \times S$ data bytes

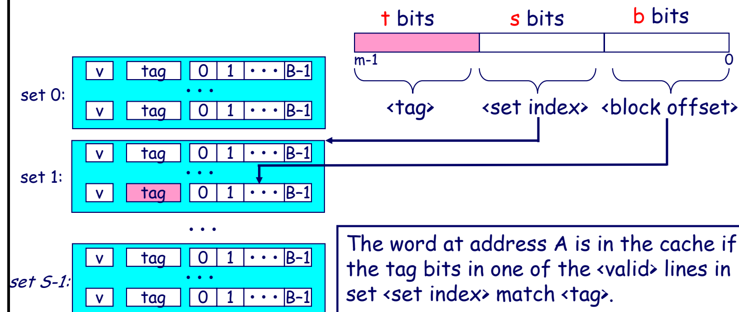
Addressing Caches

Address A:

t bits | s bits | b bits

m-1 … 0

<tag> <set index> <block offset>



The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

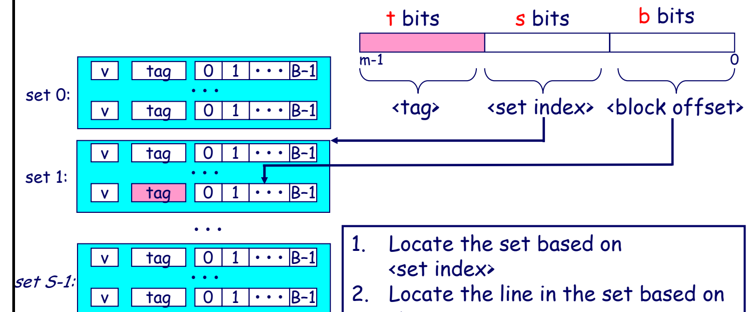
Addressing Caches

Address A:

t bits | s bits | b bits

m-1 … 0

<tag> <set index> <block offset>

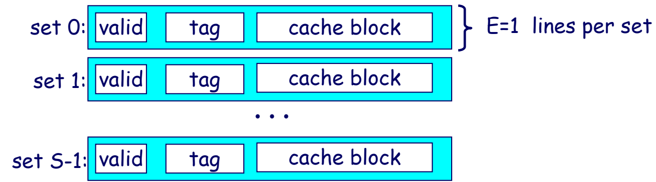


1. Locate the set based on <set index>
2. Locate the line in the set based on <tag>
3. Check that the line is valid
4. Locate the data in the line based on <block offset>

Direct-Mapped Cache

Simplest kind of cache, easy to build
(only 1 tag compare required per access)

Characterized by exactly one line per set.

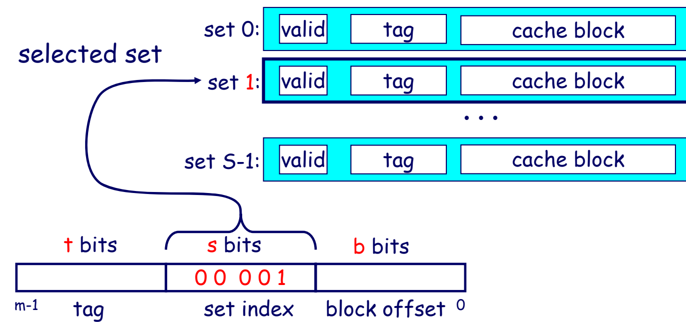


Cache size: C = B x S data bytes

Accessing Direct-Mapped Caches

Set selection

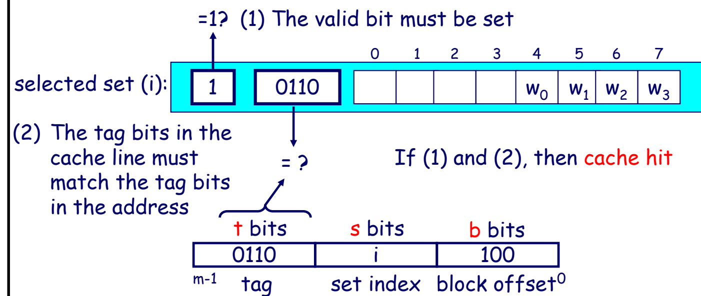
- **Use the set index bits to determine the set of interest.**



Accessing Direct-Mapped Caches

Line matching and word selection

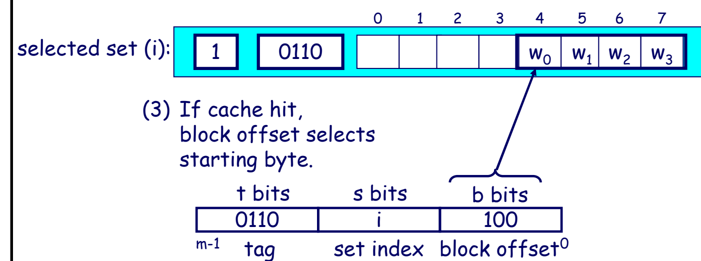
- **Line matching: Find a valid line in the selected set with a matching tag**
- **Word selection: Then extract the word**



Accessing Direct-Mapped Caches

Line matching and word selection

- **Line matching: Find a valid line in the selected set with a matching tag**
- **Word selection: Then extract the word**



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 entry/set

Address trace (reads):

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

v	tag	data
1	0	M[0-1]
1	0	M[6-7]

Set Associative Caches

Characterized by more than one line per set

set 0: valid | tag | cache block; valid | tag | cache block — E=2 lines per set

set 1: valid | tag | cache block; valid | tag | cache block

• • •

set S-1: valid | tag | cache block; valid | tag | cache block

E-way associative cache

Accessing Set Associative Caches

Set selection

- **identical to direct-mapped cache**

set 0: valid | tag | cache block; valid | tag | cache block

selected set → set 1: valid | tag | cache block; valid | tag | cache block

• • •

set S-1: valid | tag | cache block; valid | tag | cache block

t bits	s bits	b bits
	0 0 0 0 1	
tag	set index	block offset

m-1 ... 0

Accessing Set Associative Caches

Line matching and word selection

- **must compare the tag in each valid line in the selected set.**

=1? (1) The valid bit must be set

selected set (i): 1 | 1001 | 0 1 2 3 4 5 6 7; 1 | 0110 | w_0 w_1 w_2 w_3

(2) The tag bits in one of the cache lines must match the tag bits in the address

= ?

If (1) and (2), then cache hit

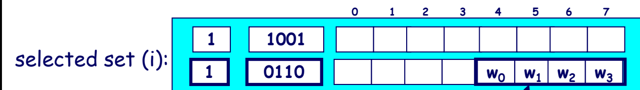
t bits	s bits	b bits
0110	i	100
tag	set index	block offset

m-1 ... 0

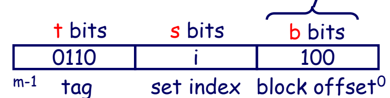
Accessing Set Associative Caches

Line matching and word selection

- Word selection is the same as in a direct mapped cache



(3) If cache hit, block offset selects starting byte.



2-Way Associative Cache Simulation

t=2 s=1 b=1

xx	x	x

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

Address trace (reads):

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

v	tag	data
1	00	M[0-1]
1	10	M[8-9]
1	01	M[6-7]
0		

Replacement Algorithms

- *When a block is fetched, which block in the target set should be replaced?*

Optimal algorithm:

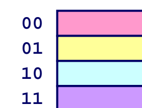
- replace the block that will not be used for the longest period of time
- must know the future

Common Algorithms:

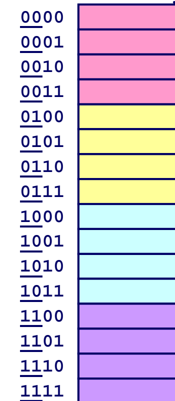
- Least recently used (LRU)
  - replace the block that has been referenced least recently
  - tracking this information requires some effort
- Random (RAND)
  - replace a random block in the set
  - trivial to implement

Why Use Middle Bits as Index?

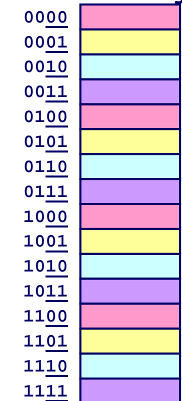
4-line Cache



High-Order Bit Indexing



Middle-Order Bit Indexing



High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing

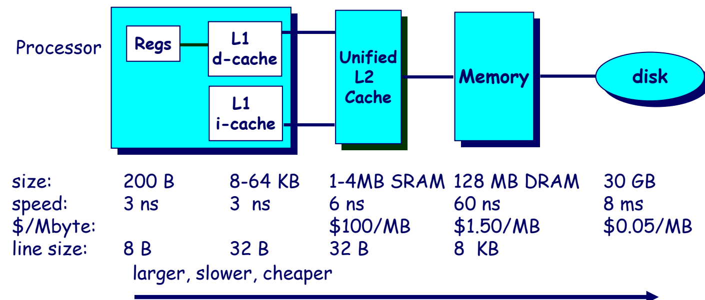
- Consecutive memory lines map to different cache lines
- Can hold S*B*E-byte region of address space in cache at one time

Maintaining a Set-Associate Cache

- How to decide which cache line to use in a set?
  - Least Recently Used (LRU), Requires $\lceil \lg_2(E!) \rceil$ extra bits
  - Not recently Used (NRU)
  - Random
- Virtual vs. Physical addresses:
  - The memory system works with physical addresses, but it takes time to translate a virtual to a physical address. So most L1 caches are virtually indexed, but physically tagged.

Multi-Level Caches

Options: separate data and instruction caches, or a unified cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper →

What about writes?

Multiple copies of data exist:
- L1
- L2
- Main Memory
- Disk

What to do when we write?
- Write-through
- Write-back
  - need a dirty bit
  - What to do on a write-miss?

What to do on a replacement?
- Depends on whether it is write through or write back

Cache Performance Metrics

Miss Rate
- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  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:
  1-3 clock cycles for L1
  10-14 clock cycles for L2

Miss Penalty
- additional time required because of a miss
  - Typically 100-300 cycles for main memory

Impact of Cache and Block Size

Cache Size

- Effect on miss rate?
- Effect on hit time?

Block Size

- Effect on miss rate?
- Effect on miss penalty?

Impact of Associativity

- Direct-mapped, set associative, or fully associative?

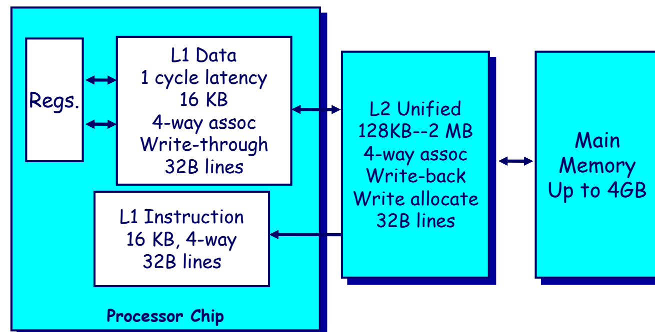
Total Cache Size (tags+data)?

Miss rate?

Hit time?

Miss Penalty?

Intel Pentium III Cache Hierarchy



Summary

- Caching works!
- Programming for good *temporal* and *spatial* locality is critical for high performance.