CS429: Computer Organization and Architecture
Cache II

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: April 22, 2019 at 10:32

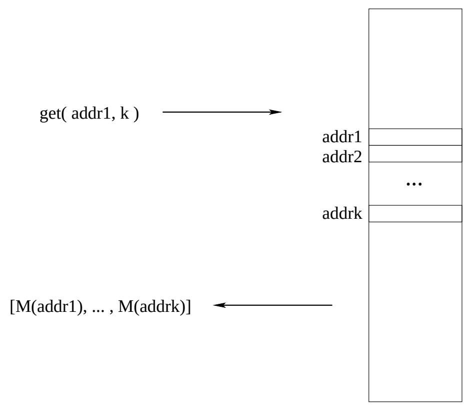
Cache Vocabulary

Much of the cache organization described in these slidesets applies to the L1 and L2 caches; not to other caches such as the TLB, browser caches, etc.

- Capacity
- Cache block/line
- Associativity
- Cache set
- Set index
- Tag
- Block offset
- Hit rate
- Miss rate
- Placement policy
- Replacement policy

The Memory Abstraction

Conceptually, memory is a large array of bytes that can be accessed from your program by specifying a starting address and a byte count.



Like any other memory optimization, caching must maintain this abstraction.

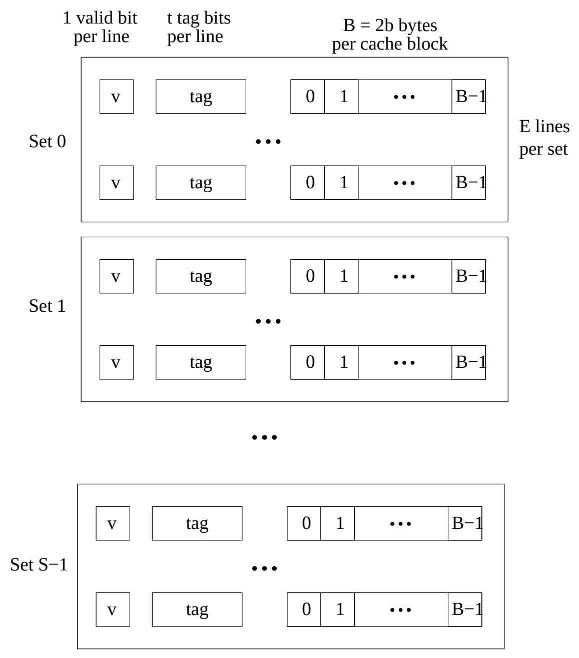
Organization of Cache Memory

Cache is an array of $S = 2^s$ sets.

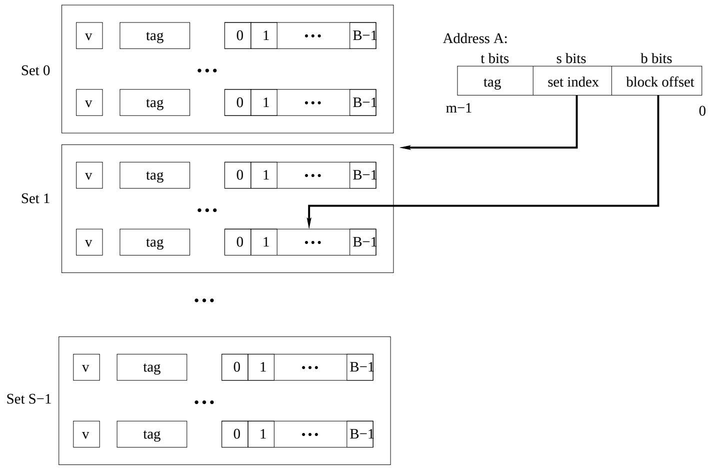
Each set contains $E \geq 1$ lines.

Each line holds a block of data containing $B = 2^b$ bytes

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



Addressing Caches



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* for that line.

The word contents begin at offset *block offset* from the beginning of the block.

Placement and Replacement

What is the placement policy for such a cache?

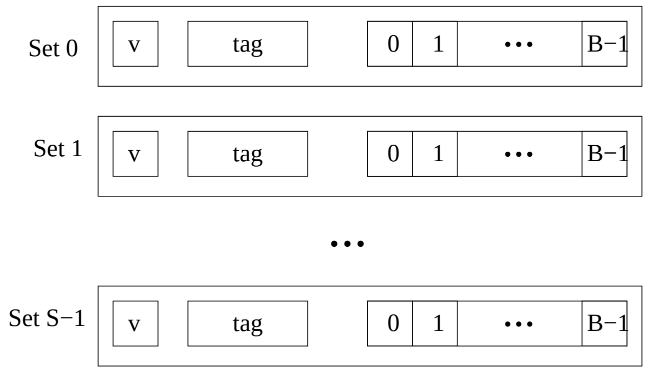
An address must be placed into some line in the set matching the *set index* of the address.

What is the replacement policy for such a cache?

This can vary. A common replacement policy is: replace the *least recently used* (LRU) line in the set with a new line containing the accessed address.

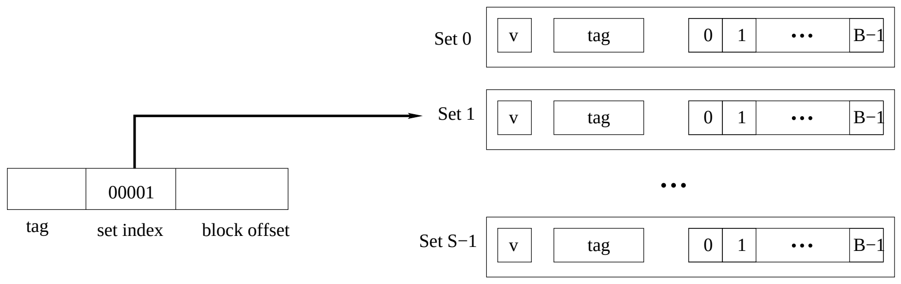
Direct-Mapped Cache

This is the simplest kind of cache, characterized by exactly one line per set (i.e, $E = 1$).



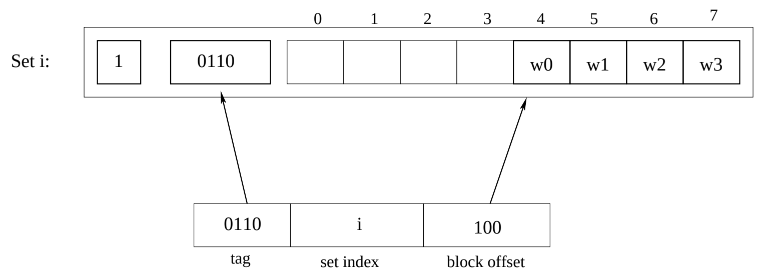
Accessing Direct-Mapped Caches

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



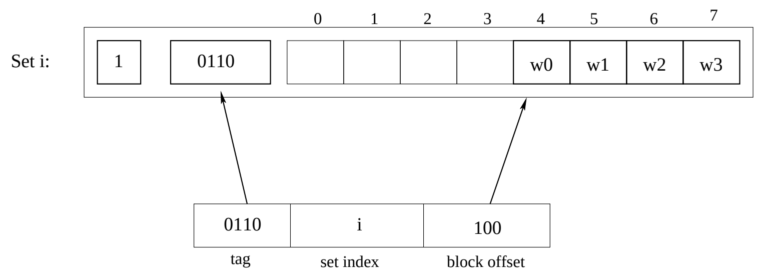
Direct-Mapped Caches: Matching and Selection

- *Line matching:* Find a valid line in the selected set with a matching tag.
- *Word selection:* Extract the word using the block offset.



1. The valid bit must be set.
2. The tag bits in the cache line must match the tag bits in the address.
3. If (1) and (2), then cache hit, and block offset selects starting bits.

Bytes in the Line



Suppose that the set index is 4 bits. What are the characteristics of this cache?

1. How big are addresses on this machine?
2. How many sets are there in the cache?
3. How many lines per set?
4. How many bytes per line?
5. How big is the block offset?
6. Suppose $i = 0101$. What range of bytes are in this line?

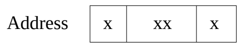
Direct-Mapped Cache Simulation

Suppose:

- $M = 16$ byte addresses;
- $B = 2$ bytes/block;
- $S = 4$ sets;
- $E = 1$ line/set.

Address trace (reads):

1. $0 = [0000_2]$
2. $1 = [0001_2]$
3. $13 = [1101_2]$
4. $8 = [1000_2]$
5. $0 = [0000_2]$



(1) 0 [0000] (miss)

v	tag	data
1	0	M[0−1]

(3) 13 [1101] (miss)

v	tag	data
1	0	M[0−1]
1	1	M[12−13]

(4) 8 [1000] (miss)

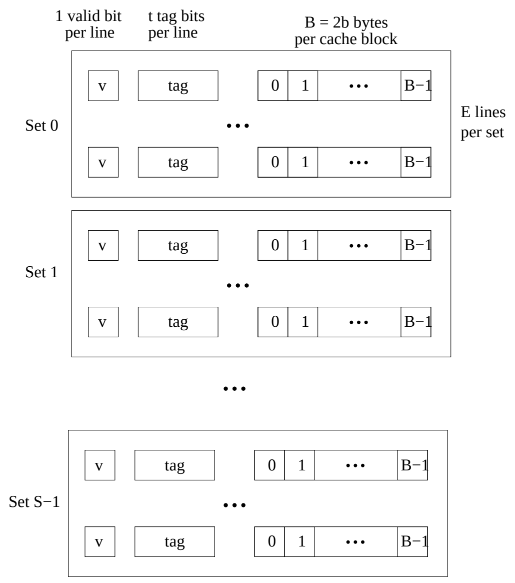
v	tag	data
1	1	M[8−9]
1	1	M[12−13]

(5) 0 [0000] (miss)

v	tag	data
1	0	M[0−1]
1	1	M[12−13]

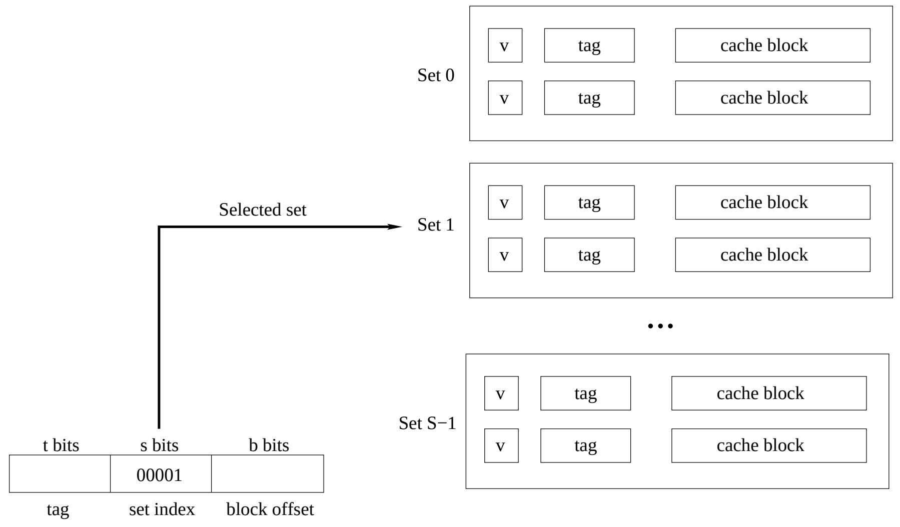
Set Associative Caches

These are characterized by more than one line per set.



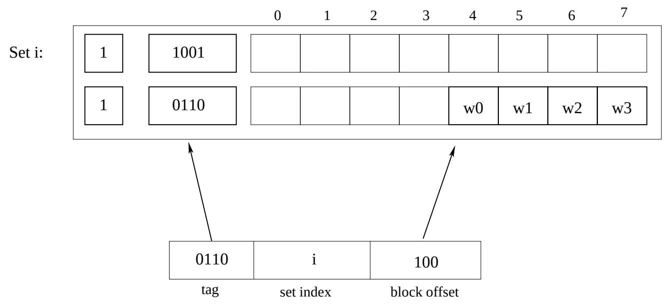
Accessing Set Associative Caches

Set selection is identical to that for direct-mapped cache.



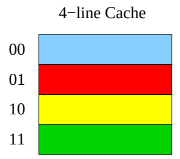
Accessing Set Associative Caches

For *Line Matching* and *Word Selection*, we must compare the tag in each valid line in the selected set.



1. The valid bit must be set.
2. The tag bits in one of the cache lines must match the tag bits in the address.
3. If (1) and (2), than cache hit, and block offset selects starting byte.

Why Use Middle Bits as Index?

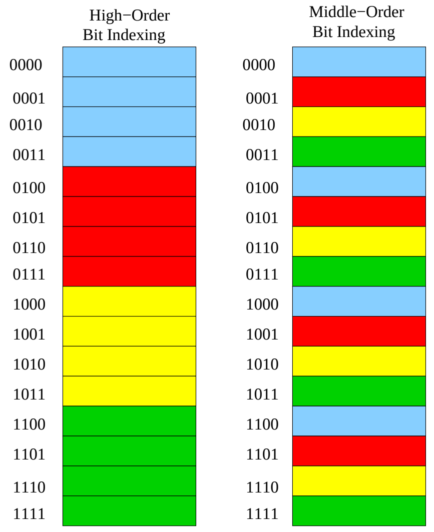


High-Order Bit Indexing

- Adjacent memory blocks map to same cache entry.
- Poor use of spatial locality.

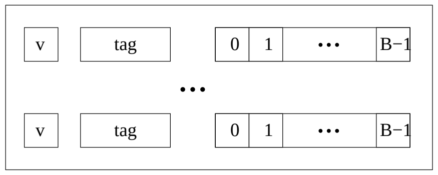
Low-Order Bit Indexing

- Consecutive memory blocks map to different cache lines.
- Can hold a larger region of address space in cache at one time.



Fully Associative Caches

A *fully associative* cache is one that has only one set.



- There are no set index bits in the address.
- Otherwise, accessing is the same as for a set associative cache.
- Tends to require more hardware to perform the associative search on a larger number of lines.

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in the 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 (including time to determine whether the line is in the cache).
- Typical numbers: 1-3 clock cycles for L1; 5-12 clock cycles for L2.

Miss Penalty

- Additional time required because of a miss.
- Typically 100-300 cycles for main memory.

Memory System Performance

Average Memory Access Time (AMAT)

$$T_{access} = (1 - p_{miss}) \cdot t_{hit} + p_{miss} \cdot t_{miss}$$
$$t_{miss} = t_{hit} + t_{penalty}$$

Assume 1-level cache, 90% hit rate, 1 cycle hit time, 200 cycle miss time (hit time plus miss penalty).

$$t_{access} = (1 - 0.1) \cdot 1 + 0.1 \cdot 200 = 0.9 + 20 \approx 21$$

AMAT $= 21$ cycles, even though 90% only take one cycle. This shows the importance of a high hit rate.

Memory System Performance II

How does AMAT affect overall performance? Recall the CPI equation (pipeline efficiency).

$$CPI = 1.0 + lp + mp + rp$$

- load/use penalty (lp) assumed memory access of 1 cycle.
- Further, we assumed all load instructions were 1 cycle.

Cause	Name	Instr. Freq.	Cond. Freq.	Stalls	Product
Load	lp	0.30	0.7	21	4.41
Load/Use	lp	0.30	0.3	21+1	1.98
Mispredict	mp	0.20	0.4	2	0.16
Return	rp	0.02	1.0	3	0.06
Total					6.61

More realistic AMAT (20+ cycles) really hurts CPI and overall performance.

Memory System Performance and the Pipeline

Suppose your pipelined Y86 machine had a 1-level cache, 90% hit rate, 1 cycle hit time, 200 cycle miss time.

$$t_{access} = (1 - 0.1) \cdot 1 + 0.1 \cdot 200 = 0.9 + 20 \approx 21$$

Recall that the clock speed of the pipeline is constrained by the *slowest* stage (M).

What does this mean for the pipelined Y86 if the slowest stage has such huge variability? What could you do?

Memory System Performance and the Pipeline II

What does this mean for the pipelined Y86 if the slowest stage has such huge variability? What could you do?

1. Run the clock 200 times more slowly to accommodate the longest memory access? Obviously not.
2. Stall the pipeline when you have a cache miss? There's really no other alternative. Why not?

Reads from memory really can't come from anywhere else. Forwarding won't help; the value is not in any pipeline register.

Improving Memory System Performance

$$T_{access} = (1 - p_{miss}) \cdot t_{hit} + p_{miss} \cdot t_{miss}$$

$$t_{miss} = t_{hit} + t_{penalty}$$

How can we reduce AMAT?

- Reduce the miss rate.
- Reduce the miss penalty.
- Reduce the hit time.

There have been numerous inventions targeting each of these. Which would matter most?

Issues with Writes

If you *write* to an item in cache, the cached value becomes *inconsistent* with the values stored at lower levels of the memory hierarchy.

There are two main approaches to dealing with this:

Write-through: immediately write the cache block to the next lowest level.

Write-back: only write to lower levels when the block is evicted from the cache.

Write-through requires updating multiple levels of the memory hierarchy (causes bus traffic) on every write.

Write-back reduces bus traffic, but requires that each cache line have a *dirty bit*, indicating that the line has been modified.

Write Strategies

How to deal with write misses?

Write-allocate loads the line from the next level and updates the cache block.

No-write-allocate bypasses the cache and updates directly in the lower level of the memory hierarchy.

Write-through caches are typically no-write-allocate. Write-back caches are typically write-allocate. Why does this make sense?

Writing Cache Friendly Code

- Can write code to improve miss rate.
- Repeated references to variables are good (temporal locality).
- Stride-1 reference patterns are good (spatial locality).

Examples: Assume cold cache, 4-byte words, 4-word cache blocks.

```
int sumarrayrows (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;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols (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;
}
```

Miss rate = 100%

Some Questions to Consider

- What happens where there is a miss and the cache has no free lines? What should we evict?
- What happens on a write miss?
- What if we have a multicore chip where cores share the L2 cache but have private L1 caches? What bad things could happen?

Concluding Observations

A programmer can optimize for cache performance.

- How data structures are organized.
- How data are accessed.
- Nested loop structure.

All systems favor "cache friendly code."

- Getting absolute optimum performance is very platform specific (cache sizes, line sizes, associativities, etc.)
- But you can get most of the advantage with generic code.
- Keep the working set reasonably small (temporal locality).
- Use small strides (spatial locality).