

1 Review

1.1 Give the formulas for splitting an address into tag, index, and offset (TIO).

- $\text{Offset} = \log_2(\text{size of block in bytes})$
- $\text{Index} = \log_2(\text{number of blocks} / \text{associativity})$
- $\text{Tag} = \log_2(\text{memory size in bytes}) - \text{Index} - \text{Offset}$
- $\text{Size of cache} = \text{number of blocks} * \text{block size}$

1.2 Define the 3 C's of cache misses and how to improve each.

(a) Compulsory

First reference to a line. Will always happen, even in an infinitely large cache.
To improve: increase line size or prefetching

(a) Capacity

Cache is too small, so the requested line was evicted.
To improve: increase cache size (but increased hit time)

(a) Conflict

Multiple blocks map to the same set and evict each other even when other sets are empty. Cannot exist in a fully associative cache (only one set)!
To improve: increase cache size and increase associativity (but increased hit time)

2 Cache Misses (Sp'24 MT1, Q4)

- 2.1 Assume you have a 4 KiB 2-way set associative L1 data cache with 64-byte cache lines, and that you are using 32-bit addresses. How many bits are required for the tag, index, and offset?

Offset: 6 bits, $\log_2(64) = 6$

Index: 5 bits Number of lines = 4 KiB / 64 B = $\frac{2^{12}}{2^6} = 2^6$ lines

Number of sets = lines / associativity = $\frac{2^6}{2} = 2^5$ sets $\rightarrow \log_2(2^5) = 5$ bits

Tag: 21 bits, $32 - 5 - 6 = 21$

- 2.2 Now consider the following program. Assume that each element is a long (8 bytes) and that the matrix elements are stored in row-major order. Assume that the first element in the matrix is aligned to the start of a cache line, that sum, i, and j are stored in registers, and the same cache as above.

```
int sum = 0;
int matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        sum += matrix[i][j];
    }
}
```

- (a) What is the number of cache misses?

Notice:

- We traverse in row-major order and no element is ever reused
- One cache line holds 64 B / 8 B = 8 elements

Each full j loop (one iteration of i) touches an entire 64-element row $\rightarrow 64$ elements / 8 elements per line = 8 lines $\rightarrow 8$ compulsory misses per j loop.

(8 misses per j loop) * (64 j loops) = **512 compulsory misses**

- (a) How many hits?

Total accesses - total misses = $64 \cdot 64 - 512 = 3584$

- (a) How many compulsory misses?

512 - again, elements are never reused

- (a) How many conflict misses?

0

(a) How many capacity misses?

0

2.3 Bonus: How can we improve our hit rate?

Every miss is compulsory, so we either make each miss bring in more useful data (larger cache lines) or fetch lines before they are needed (prefetching, e.g. a next-line / stream prefetcher works well for this sequential access pattern).

3 Sectored Cache (Sub-blocks)

100	1		1		1		1	
300	1		1		0		0	
204	0		1		0		1	

3.1 What problem was this optimization trying to solve? Explain.

Large cache lines keep tag overhead low and exploit spatial locality, but every miss must fetch the whole line, which costs memory bandwidth and increases the miss penalty. A sectored cache keeps one tag per large line but adds a valid bit per sub-block, so on a miss only the sub-block that was accessed needs to be fetched: the tag cost of a big line with the fetch cost of a small one.

Exam Practice (Sp'22 MT1, Q3)

We have an 8 KiB, two-way set associative cache with 16-byte cache blocks. The replacement policy is LRU. Each cache block is broken up into four smaller sub-blocks. Each element in `buf` is 4 bytes. Assume `sum`, `i`, and `j` are stored in registers.

```
// Loop A
sum = 0;
for (int i = 0; i < 128; i++)
    for (int j = 0; j < 32; j++)
        sum += buf[i*32 + j];

// Loop B
sum = 0;
for (int j = 0; j < 32; j++)
    for (int i = 0; i < 128; i++)
        sum += buf[i*32 + j];
```

3.2 What is the number of misses for Loop A and for Loop B with the sectored cache?

Each sub-block is $16 \text{ B} / 4 = 4$ bytes, so it can only hold one element. Since a sectored cache doesn't fetch the neighboring sub-blocks on a miss, every access is a compulsory miss!

$32 \cdot 128 = 4096$ accesses $\rightarrow$ **4096 misses** for each loop.

3.3 What is the number of misses for Loop A and for Loop B if the cache is not sectored (no sub-blocks)?

Loop A: Each block holds 4 elements. The `j` loop does 32 sequential accesses per iteration $\rightarrow$ 8 compulsory misses per `i` iteration. $128 \cdot 8 =$ **1024 compulsory misses**.

Loop B. With j fixed, each step of i adds 128 bytes = $0x80$ to the address. The lower 4 bits are the block offset, so the $0x8$ lands in the set-index field: every access moves +8 sets. Total sets = $(8 \text{ KiB} / 16 \text{ B}) / 2 = 256$, so one pass of i uses only $256 / 8 = 32$ sets (every 8th set, 0, 8, 16 256)

There's 128 accesses per i loop, so with 32 sets $\rightarrow$ 4 distinct blocks per set, but each set is only 2-way. Under LRU, by the time a block is needed again (128 accesses later, at the next j) the set has seen 3 other blocks and evicted it. Example, set 0: $\text{buf}[0]$ and $\text{buf}[32*32]$ fill the two ways; $\text{buf}[32*64]$ evicts $\text{buf}[0]$; $\text{buf}[32*96]$ evicts $\text{buf}[32*64]$. At $j = 1$, $\text{buf}[1]$ needs $\text{buf}[0]$'s block, but it's already gone so miss.

Only the $j = 0$ pass (1024) is compulsory; the rest are conflict misses caused by the power-of-two stride, not by array size (one pass is only 2 KiB). **Every access misses: $32 \times 128 = 4096$ misses.**

3.4 Qualitatively explain whether our sectoring scheme has improved the average memory access time of Loop A and Loop B. If so, why? If not, why not?

Loop A: Sectoring does **not** improve AMAT.

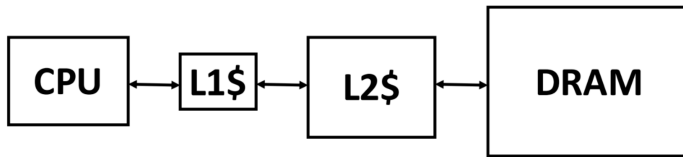
- Hit time: same
- Miss rate: 4x increase from non-sectored to sectoring
- Miss penalty: slightly less than a 4x decrease (less memory bandwidth per miss, plus some fixed overhead per miss)

Loop B: Sectoring **does** improve AMAT.

- Hit time: same
- Miss rate: same (everything misses either way)
- Miss penalty: decreased, because each miss moves less data from memory

Takeaway: sector caches hurt when your code exploits spatial locality!

4 Hierarchical Caches



4.1 What problem was this optimization trying to solve? Explain.

A single cache can't be both fast (which needs it to be small) and have a low miss rate (which needs it to be large). A hierarchy puts a small, fast L1 in front of a larger, slower L2 so that most L1 misses hit in L2 instead of paying the full DRAM latency, reducing the effective miss penalty without slowing down L1.

Exam Practice

Assume we have a two-level cache hierarchy: $L1 \rightarrow L2 \rightarrow \text{Memory}$.

	Hit Time	Local Miss Rate
L1	2 cycles	8%
L2	15 cycles	25%
Main Memory	200 cycles	-

4.2 What is the AMAT of this system?

$$AMAT_{L2} = 15 + 0.25 \cdot 200 = 65 \text{ cycles}$$

$$AMAT = 2 + 0.08 \cdot 65 = 7.2 \text{ cycles}$$

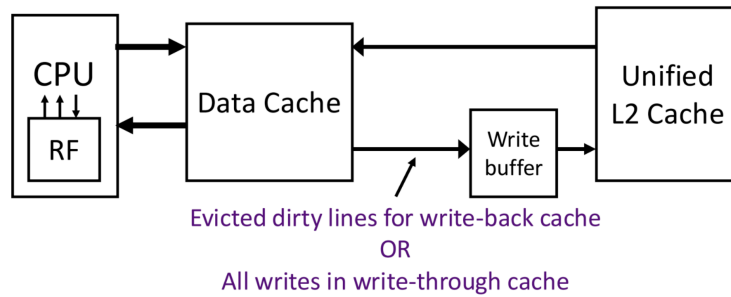
4.3 Say we double the size of L1. The L1 local miss rate falls from 8% to 5%, and the L2 local miss rate rises from 25% to 35%. Compute the new AMAT. Then explain in at most two sentences why the L2 local miss rate went up even though the hierarchical AMAT got better.

$$AMAT_{L2} = 15 + 0.35 \cdot 200 = 85 \text{ cycles}$$

$$AMAT = 2 + 0.05 \cdot 85 = 6.25 \text{ cycles}$$

The 3% drop in L1 miss rate outweighs L2's extra cycles. L2's local miss rate rose only because the bigger L1 now filters out the easy (high-locality) accesses, leaving L2 a harder stream of requests.

5 Write Buffers



5.1 What problem was this optimization trying to solve? Explain.

Writes to the next level of the hierarchy are slow, but the CPU usually doesn't need to wait for a write to finish. A write buffer is a small queue between the cache and the next level that holds pending writes (every store in a write-through cache, or evicted dirty lines in a write-back cache), so the cache can retire the write immediately and drain it in the background instead of stalling. Loads must check the buffer so they don't read stale data.

Exam Practice (Sp'22 MT1)

5.2 How do write buffers affect write miss penalties in a write-through cache? (A write miss occurs when the program attempts to STORE to an address that is not in the cache.) Explain your reasoning.

Improved. Without a buffer, the cache blocks until the store reaches the next level; with a buffer, the store retires immediately and drains in the background. This matters most in a write-through cache, where every store (both hits and misses) goes to the next level.

5.3 How do write buffers affect read miss penalties? (A read miss occurs when the program attempts to LOAD from an address that is not in the cache.) Explain your reasoning for both a write-back and a write-through cache.

Write-back: Improved. A read miss that evicts a dirty line normally has to wait for that line to be written out first; the buffer absorbs the writeback so the fetch can start immediately.

Write-through: Almost no change. Write-through already pays on every store to keep memory in sync, and in exchange evictions are free (nothing to write back), so there is no writeback for the buffer to hide on a read miss.