Lecture 8 - Memory hierarchy and caching

Gidon Rosalki

2025-05-18

1 Overview

The main memory is a critical component of computing systems, and must scale to maintain performance growth and technology scaling benefits (in all matters of size, efficiency, cost, management algorithms, etc.). However, in reality there is the memory performance gap, also known as the CPU-DRAM latency gap. The processors got faster in accordance with Moore's law, however memory speed only increased by approximately 7% per year. So in a 10 year gap, the CPU got 100 times faster, but the difference in speed with memory caused it to only run instructions approximately 6 times faster. We can mitigate this through **caching**, where we keep frequently used data in a small and fast storage.

Consider the following example. I like to read books. My local library has all the books in the world, but for me to access a book from there takes days. I have a faster storage layer, my home library, from which book access is minutes. Finally, there is my bedside, from which access is seconds, but cannot store much. Since the faster storage layers have lower capacity, I swap between them in order to be able to read my current books most easily. This is caching, and we use it to minimise the average access time.

There is a hierarchy of the registers on the CPU, L1, L2, L3, and main memory, organised in order of increasing access time. The registers are made with flip flops, coming in at around 40 transistors per bit, the caches are made with SRAM, which comes in at 6 transistors per bit, the system memory is made with DRAM, which needs 1 transistor per bit, and finally the system storage is generally flash NAND, which is 1 - $\frac{1}{3}$ transistors per bit, and is the only one on this list that is not volatile, but oh so slow.

1.1 SRAM

SRAM, or Static Random Access Memory, is a series of cells arranged in rows and columns, where each cell is constructed of 2 not gates, and 2 transistors. The powering on and off of the relevant bit lines opens up the cells for reading / writing. We select the cells to be turned on using the address line, and then we write to them using the bit lines.

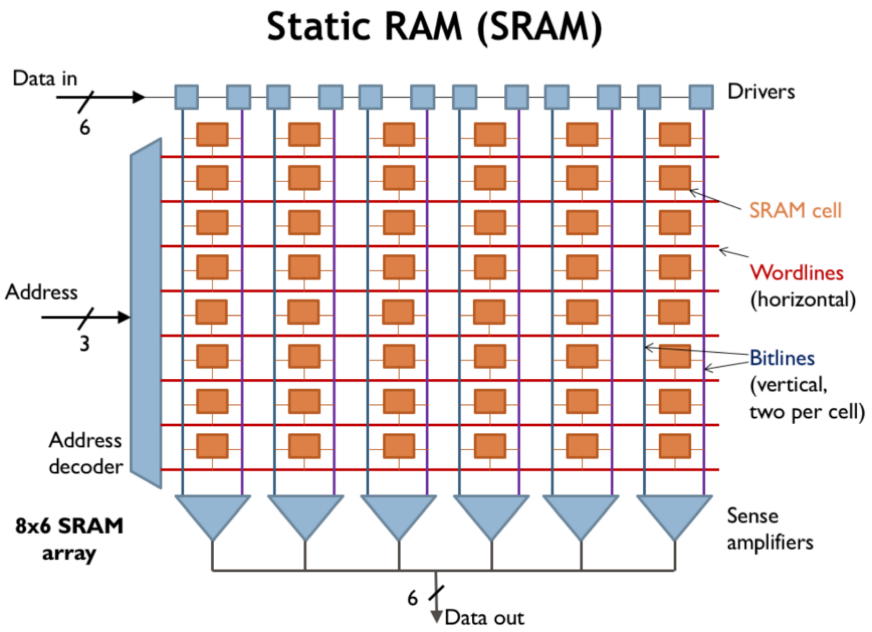

Figure 1: SRAM overview

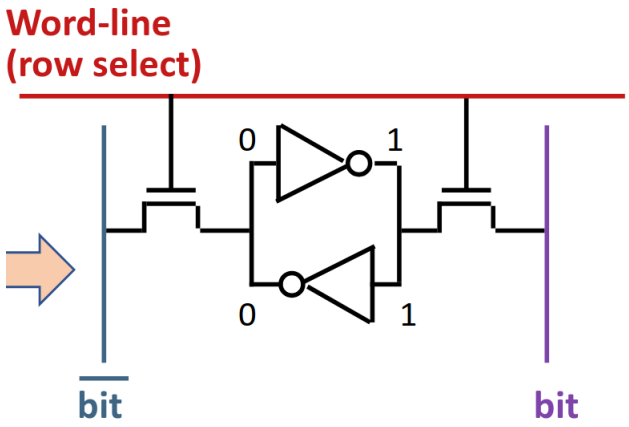

Figure 2: SRAM cell logic

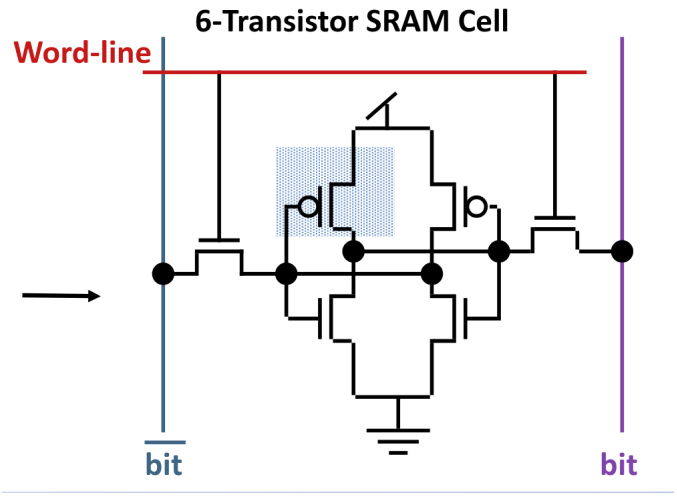

Figure 3: SRAM cell transistors

1.2 DRAM

DRAM only requires 1 transistor:

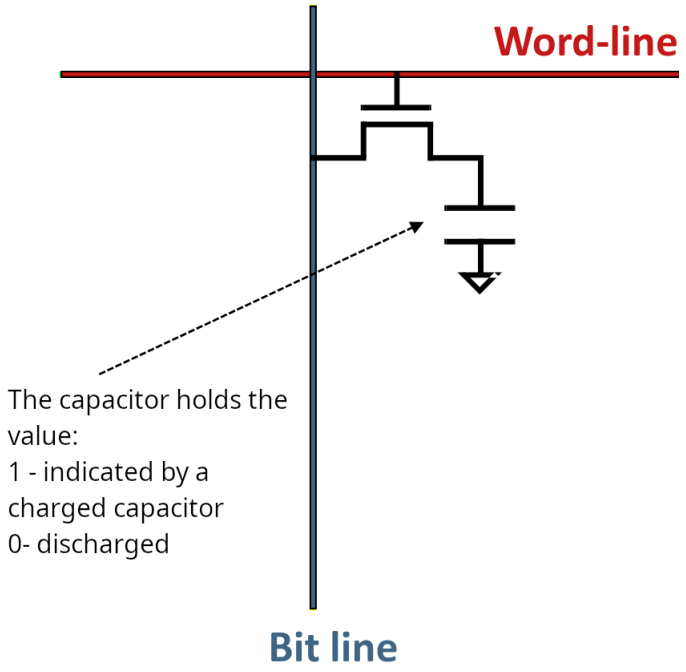

Figure 4: DRAM cell transistors

To write we drive the bit line to 1, selecct the row (word-line), and the capacitor charges, or discharges. To read we precharge the bit line to $\frac{V_{dd}}{2}$, select the row (word line), and the cell (capacitor), and bit line share charges. We sense the charge change (this operation is destructive), and then rewrite to restore the value. Since capacitors leak charge, this requires a periodic refresh of the capacitor charge, to prevent the memory from fading away. To achieve this, we perform a dummy read to every cell, every 64ms.

1.3 Flash memory

Here we again use 1 transistor per cell, and can store multiple bits per cell (up to 3 today). Using NOR gates leads to a faster to read block erase, and NAND is best for block operations for reading and writing. It reads 4-16kb at a time, writes 4-16kb at a time (1 to 0 change), and can erase a few Mb at a time (0 to 1 change). There is degradation with writes, and in some cases with multiple reads, which can cause needing a refresh.

2 Cache principles

2.1 Temporal and spatial locality

Temporal locality: After an item is accessed, it tends to be accessed again soon.
Spacial locality: After an item is accessed, nearby items tend to be accessed soon.

Temporal locality helps cacheing, since it tells us to keep recently accessed data closer to the processor, and spatial locality tell us to move contiguous blocks closer to the processor. By taking advantage of these principles or locality, we can use as much memory as is available in the slowest (and cheapest) technology, and at a speed close to what is offered by the fastest technology (on average).

2.2 Terminology

Hit: A successful search of data in the cache.
Miss: An unsuccessful search of data in the cache (the data was not in the cache).
Block: The basic unit that is loaded into the cache when a miss occurs. The minimum size of a block is a single byte.
Miss penalty: The time to replace a block with the corresponding block from the next level, plus the time to deliver this block between the two levels.

3 Cache design

3.1 Main idea

The cache holds a small part of the entire memory, meaning we need to map parts of the memory into the cache. The main memory is virtually partitioned into lines (blocks), with a typical block size of 64-128 bytes. The cache holds data by lines, and only a subset of the memory blocks can be mapped to the cache. The address of a line is used as a key.

3.2 Cache lookup

Case 1: Cache hit: The requested line is in the cache, and it returns the requested line very quickly.
Case 2: Cache miss: the requested line is not in the cache, and we need to fetch it from main memory (slow), and fill the cache. When doing this, we may need to evict another line from the cache to free up the space for the new line.

In a fully associative cache, any line can be mapped to any cache entry. The address is partitioned to the offset within a line (byte number), abnd the line number (so called *tag*). Each cache entry has a tag, and a valid bit. All tags are compared to the requested line number in parallel, and if one of the valid tags matches the line number, then we have a hit.

3.2.1 Direct mapped cache

A line is mapped to the fixed entry only. We lookup one line in one entry. The address is partitioned into:

- The offset within a line (byte number)
- The index (fixed cache entry number), this is also referred to as the set
- Tag (works as a key), the tag bits + valid bit

Conflicts are possible in this model, When multiple lines are mapped to the same entry, only one can reside in the cache.

3.2.2 2 way set associative

Conflicts are harmful, therefore multi-ways cache is a solution. Each set holds 2 entries, (way #0, and way #1). Each line can be mapped into one of two entries in the appropriate set.

3.2.3 4 way set associative

The previous concept of a 2 way set associative can be expanded to 4 ways, as with the following image:

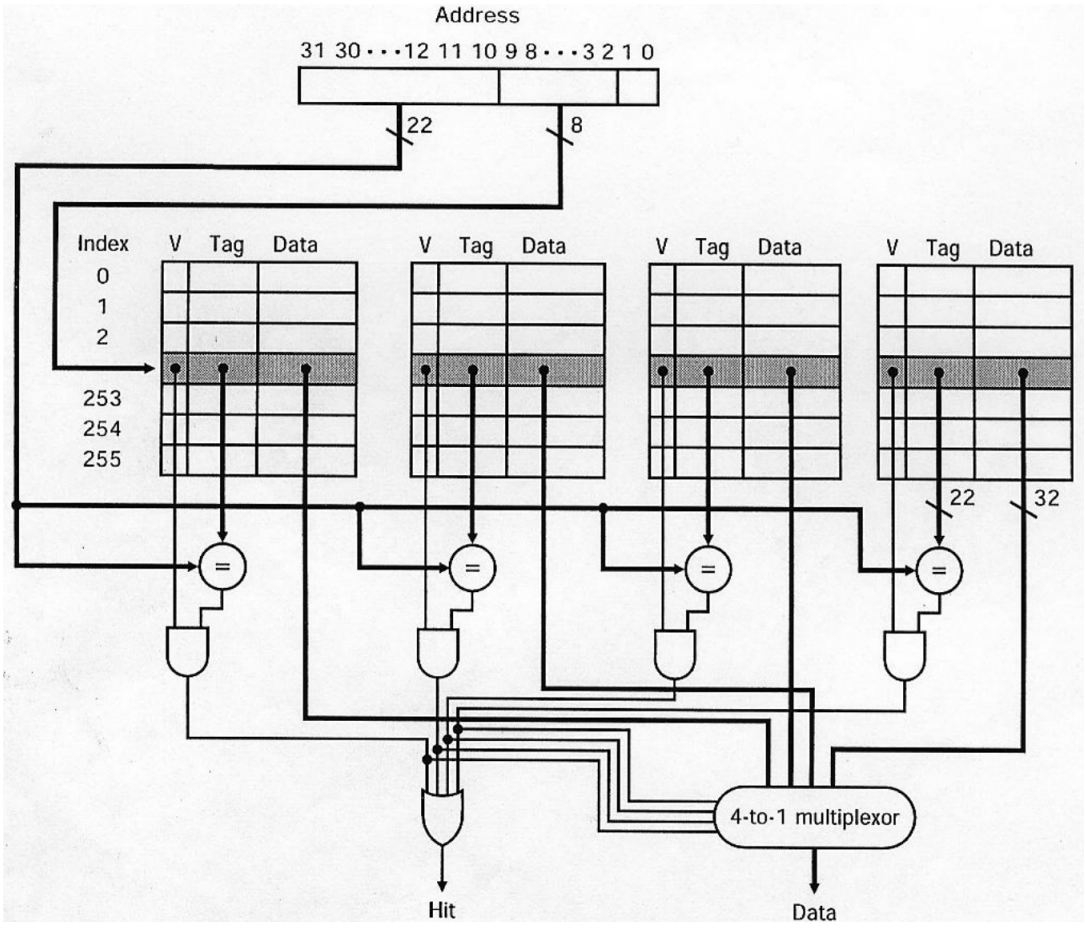

Figure 5: 4 way set associative cache

3.2.4 Cache organisation

Cache organisation is about mapping of main memory addresses to a smaller set of addresses - hash function. One must also map these locations through a way that enables lookup: Allowing us to find data quickly, should the data be cached (ie, located in the cache).

3.3 Types of cache misses

There are **compulsory** misses, which occur even in an infinitely large cache. This is because this is the first time we access the data. These are also called *cold* misses. We also have **capacity** misses. These occur in a fully associative cache, due to the working set being larger than the cache size. Next is **conflict** misses, which are due to associativity and mapping (the cache structure). This happens when different information is mapped to the same location in the cache, and so we dispose of previously cached information, even though the cache is not full. This is the cache being large enough, but the problem is in block mapping to sets. There is a fourth type, which will be discussed later in the semester, which is **coherency** (invalidation).

So fully associative is excellent, since its design helps it avoid conflict misses.

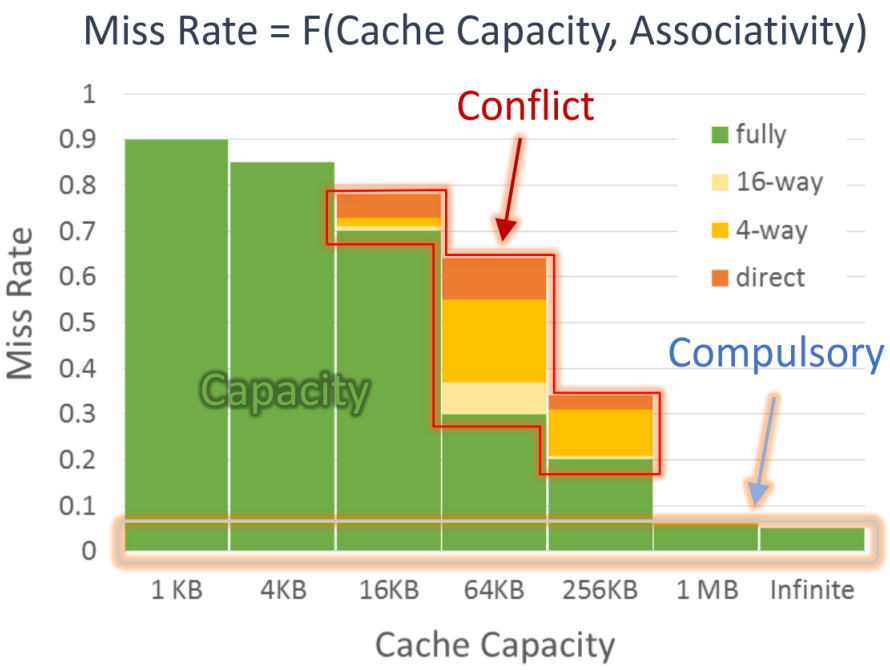

Figure 6: Cache miss rates

4 Cache performance

4.1 Terminology

The **miss rate** is the number of misses, divided by the total requests (accesses). The **hit rate** is the hits, divided by the total requests, also known as 1 minus the miss rate.

$$\text{Miss rate} = \frac{\text{misses}}{\text{total requests}}$$
$$\text{Hit rate} = \frac{\text{hits}}{\text{total requests}} = 1 - \text{miss rate}$$

The cache goal is to decrease the average memory access time (AMAT), which is defined

$$AMAT = \text{Hit time} + \text{Miss rate} \cdot \text{Miss penalty}$$

where the is the approximate lookup time, which is required for all accesses, to determine hit or misses, and the miss penalty is the added cycles on top of the lookup caused by misses. Consider if:

$$\begin{aligned}
\text{Hit rate} &= 0.9 \\
\text{Hit time} &= 4clk \\
\text{Miss rate} &= 0.1 \\
\text{Miss penalty} &= 200clk \\
AMAT &= 4 + 0.1 \cdot 200 \\
&= 24clk \\
AMAT \text{ (without cache)} &= 200clk
\end{aligned}$$

So we see here a roughly 10x improvement.

4.2 Impacts of miss penalties on performance

Suppose a processor executes at an ideal CPI of 1.1 (no cache misses), with 50% arithmetic/logic instructions, 30% ld/st, and 20% control, such that 10% of data memory operations miss with a 50 cycle miss penalty. Then

$$\begin{aligned}\text{CPI} &= \text{ideal CPI} + \text{average stalls per instruction} \\ &= 1.1 + 0.3 \cdot 0.1 \cdot 50 \\ &= 1.1 + 1.5 \\ &= 2.6\end{aligned}$$

so 58% of the time, the processor is stalled waiting for memory.

If we now add that 1% of the instructions cause a miss with a 50 cycle miss penalty, then

$$\begin{aligned}\text{CPI} &= \text{ideal CPI} + \text{average stalls per instruction} \\ &= 1.1 + 0.3 \cdot 0.1 \cdot 50 + 0.1 \cdot 50 \\ &= 1.1 + 1.5 + 0.5 \\ &= 3.1\end{aligned}$$

So now 65% of the time the processor is stalled waiting for memory.

4.3 Block size tradeoff

Having different sizes of block size has both benefits and drawbacks. The benefits of a larger block size are spacial locality, since if we access a given word, we are likely to soon access nearby words, which is very applicable to the code: executing a given instruction massively increases the likelihood that we will execute the following instructions. This also works well with sequential array accesses. However, this comes with the drawbacks that larger block sizes can mean larger miss penalties, it takes longer to load new blocks from the next level. Also, if it is too big relative to the cache size, then there are too few blocks, increasing the miss rate.
As we recall from earlier, we want to minimise the AMAT.

In conclusion, as the block size increases, then the miss penalty increase in a linear fashion. Additionally, as the block size increases, initially the miss rate decreases, but after a certain point it increase once more, as the fewer number of blocks compromises temporal locality. Finally, as the block size increases, the average access time decreases, until a certain point, where the increased miss penalty, and miss rate compromise this.

4.4 Multi level caches

We are motivated to have multi level caches, to try and limit this problem. Our L1 cache has

$$AMAT \approx \text{Hit time L1} + \text{Miss rate L1} \cdot \text{Miss penalty L1}$$

Next, the L2 reduces the L1 miss penalty, by reducing access to main memory, so on an L1 miss, we access L2 instead of the main memory, so

$$\text{Miss penalty L1} \approx \text{Hit time L2} + \text{Miss rate L2} \cdot \text{Miss penalty L2}$$

The L2 cache is larger, and has a better hit rate, but has a larger latency. For example, consider

- Hit time: L1 = 3clk, L2 = 6clk
- Miss rate: L1 = 8%, L2 = 3%
- Miss penalty: L2 = 200clk

Then, without L2:

$$AMATL1 \approx 3 + 0.08 \cdot 200 = 19clk$$

and with L2

$$\begin{aligned}\text{Miss penalty L1} &\approx 6 + 0.03 \cdot 200 \\ &= 12clk \\ AMATL1 &\approx 3 + 0.08 \cdot 12 \\ &= 3.96clk\end{aligned}$$

5 Replacement and write support

When a miss occurs, if there is a free space in the cache (lines that are not used), that can be used, then use it. Otherwise, we need to free space in the cache for the new line. In a direct mapped cache, a new line is mapped to a single entry, and thus we evict the old line from the entry. In an n-way associative cache, we choose a victim form all ways in the appropriate set, but how do we choose such a victim?

We have a few block replacement policies (for associative caches).

- Random replacement
- FIFO (First in First Out) replacement: Easy to implement in hardware, but may not use the temporal locality effectively.
- LRU (Least Recently Used): Evict the most unused line (as in, the one not accessed in the most time). This has good temporal locality, but becomes complex to implement in hardware as the number of ways grow. There are multiple approaches to give LRU approximation, for example a PLRU tree, or single bit PLRU.

5.1 Write handling

5.1.1 Write hit

Memory writes (store instructions) also need to update the cache. There are two approaches for this:

- Write through: Always update both the memory, and the cache. For this we generally use a write-buffer to avoid stalling the CPU. This has the benefits of coherence, both the cache, and the memory are always in the same state. However, this also has the negatives of a higher utilisation of memory bandwidth.
- Write back: Update only the cache, and update the memory at block eviction. This requires a "modified" bit to be stored as well in the cache (part of the tag array bits, along with the valid bit). This has a low store latency, and memory bandwidth, but requires snoops - coherency checks for multi processing.

5.1.2 Write miss

So when we're writing, what if the memory to which we want to write is not in the cache? Well, we once again have two ways to deal with this:

- Write allocate: We load the block into the cache (same as read miss), and then write to the cache. The implicit assumption is that more writes to this block are expected. This plays well with write back.
- Write no allocate: We write directly to main memory. This plays well with write through.

Unlike a read miss, the processor can continue execution in the presence of a write miss. However, this does have a catch of if there read from that memory location immediately following, then there is a problem, so thank goodness for write buffers.

5.2 Improving cache performance

5.2.1 Victim cache

The per set pressure may be non uniform, some sets may have more conflict misses than others. The solution is to dynamically allocate sets to ways. The victim cache gives a second chance for evicted lines. When a line is evicted from the L1 cache, it is placed in the victim cache. If the victim cache is full, then it evicts its LRU line. On the L1 cache lookup, we lookup in the victim cache in parallel. On a victim cache hit, the line is moved back to teh cache, and the evicted line is moved t the victim cache. Thus, this has the same access tie as a cache hit. This is particularly effective for a direct mapped cache. It enables combining the fast hit time of a direct mapped cache, and still reduces conflict misses.

5.2.2 Reducing miss penalty

The CPU does not need to wait for a full cache line to be loaded. Thus it can restart early, as soon as the requested word of the cache line arrives, we can send it to the CPU, and let the CPU continue execution. Additionally, we can request the missed word first from memory, and send it to the CPU as soon as it arrives. The CPU then continues execution while filling the rest of the words in the cache line. This is called critical word first, or wrapped fetch / requested word first. For example, the Pentium had a 64 bit = 8 byte nbus, with a 32 byte cache line, resulting in 4 bus cycles to fill the line.

5.2.3 Prefetching

The main idea here is to predict future memory accesses, and then a useful prefetch eliminates a potential cache miss. There are 2 types of prefetching:

- Software (i.e., a special prefetching instructions)
- Hardware

The code for prefetching is performed by a branch rpedictor. This has the limitations of relying on extra memory bandwidth, and if it is too aggressive / inaccurate, then it slows down demand fetches.

Prefetching has the quality metrics of

- Accuracy (the portion prefetches that become demand fetches)
- Coverage (the portion of cache misses removed by prefetching
- Timeliness (reduction of cache miss time)

For L1, cache prefetch schemes with high accuracy are used. Bigger caches usually leverage schemes with higher coverage.

6 Examples