Foundations of Computer Systems
Fall 2022

Daniel Grant

Contents

Chapter 1

Storage and Caching

1.1 Day 25

November 2, 2022

1.1.1 Data Storage

Example 1.1 (Hardware used for Data Storage)
→ **Data storage technologies can be broken into the following categories:**

Disks

1. Hard Disk (HDD)
2. Solid State Drive (SSD)

Differences Between the two disk types:

- SDD is faster and smaller than HDD
- SDD is also more expensive than HDD

Random Access Memory (RAM):

1. Dynamic RAM (DRAM)
2. Static RAM (SRAM)

Registers, Such as:

- rax
- rbx
- rcx
- rdx
- rsi (*etc*)

→ **In terms of storage access time, the following is true:**

- Disks are the slowest in terms of access time.
- SSD is marginally faster than disks.
- DRAM is faster than SSD by a large margin, but still slower than registers (CPU)

Definition 1.1: CPU-Memory gap

Although the CPU is the fastest component in a computer in terms of access time, it is still progressing in terms of access speed much faster than other types of data storage.

The problem we should be concerned about it the gap between the CPU and the memory creates idle time for the CPU as is it significantly faster than memory.

Need something to bridge the gap between the CPU and the memory (*hint: Caching*).

→ **Caching vs. Memory**

- Caching is smaller, faster, and more expensive; It caches a subset of the blocks of memory
- Memory is larger, slower, and cheaper; It is partitioned into blocks.

1.1.2 Caching

Definition 1.2: Caching

A smaller, faster storage to speed up larger, slower storage.

Note:-
SRAM is used for hardware caches, and DRAM is used for main memory.

→ **Main Topics in Caching:**

1. Cache Operation
2. Cache Design
3. Cache Effectiveness

1.1.3 Cache Operation

Example 1.2 (Cache Operation)
Uses fixed size data blocks. The size of the block is a parameter set for each system. The "block" is the size of unit transfer between the cache and memory (An entire block will be copied from memory to cache even if you only referenced a small amount of that memory)

Note:-
→ *Cache Block Size Comparison*
Smaller blocks would lead to more blocks in general in the cache.
However, larger blocks would lead to fewer cache misses.

For example, you might access memory from some address in the middle of a block, and then load it into the cache in the hopes of getting "cache hits" later on.

In this example, we are accessing both the cache and the memory, but the time it takes to access the cache is much faster than the time it takes to access the memory, so moving to cache is nearly negligible.

In the case of a cache ”miss”, will perform a cache eviction.

Definition 1.3: Cache Eviction

Changing something stored in the cache to something else stored in memory.

Cache Design

→ **Design Goals for Caches:**

- Maximize cache hit rate
- Minimize complexity and cost

Definition 1.4: Write-Miss

Types of write-misses:

1. write-allocate cache (bring updated block into cache)
2. no-write-allocate cache (leave cache alone)

Example 1.3 (Write-through Cache)
→ **Write-Miss**
On a write-hit, will update cache and memory immediately.

→ **Write-Miss**
Uses the no-write-allocate policy.

Example 1.4 (Write-back Cache)
→ **Write-Hit**
On a write-hit, will IGNORE memory, and only update the cache.

When evicting a block, if it a block that has been changed, then it will be different than the block saved in memory, so will update memory on eviction.

This will be preferable in the scenario that a cache is updated numerous times in a row, as it will not update memory on each access.

Deferring memory updates to a later time.

→ **Write-Miss**
Uses the write-allocate policy.

→ **Cache Placement Policy:**

- Direct-Mapped Cache: Each memory block is mapped to 1 code block, so each cache is tied to a certain part of the memory (given a piece of memory, it has an associated cache).
  1. easy to know if memory is in the cache

2. easy to know which cache to use for some memory
3. easy to implement
4. If multiple pieces of memory are often being accessed, but are associated to the same cache, a *Conflict Miss* occurs.

- Fully-Associative Cache
- Set-Associative Cache

1.2 Day 26

November 4, 2022

1.2.1 More on Cache Placement Policy

Definition 1.5: Fully Associative Cache

A cache is fully associative if it can map any block to any cache line.

$\longrightarrow$ *Benefits of a fully associative cache:*

- Do not need to keep track of where a block is stored in the cache.
- Can store any block in any cache line.

$\longrightarrow$ *Downside of a fully associative cache:*

- Need to search the entire cache to find a block.

Definition 1.6: Direct Mapped Cache

A cache is direct mapped if it can map any block to a single cache line.

$\longrightarrow$ *Benefits of a direct mapped cache:*

- Can store any block in a single cache line.
- Can find a block in the cache in one step.

$\longrightarrow$ *Downside of a direct mapped cache:*

- Need to keep track of where a block is stored in the cache.

Definition 1.7: Set Associative Cache

Blocks are mapped to a set of cache locations.

$\longrightarrow$ *Benefits of a set associative cache:*

- Can store any block in a set of cache lines.
- Can find a block in the cache in one step.

$\longrightarrow$ *Downside of a set associative cache:*

- Need to keep track of where a block is stored in the cache.

1.2.2 Metadata in a Cache line

Different parts of a Cache Line

- *Tag*: The tag is used to identify the block that is stored in the cache line.
  - This is important as there is more than one block that can be stored in a cache line.
  - Also, for any one cache line, there is more than one block that can be stored in it.

- *Valid bit*: The valid bit is used to indicate whether the cache line contains a valid block.
  - If the valid bit is 0, then the cache line does not contain a valid block.
  - If the valid bit is 1, then the cache line contains a valid block.

- *Dirty bit*: The dirty bit is used to indicate whether the block in the cache line has been modified.
  - If the dirty bit is 0, then the block in the cache line has not been modified.
  - If the dirty bit is 1, then the block in the cache line has been modified.

- *Data*: The data is used to store the actual data of the block.

1.2.3 Cache Bit Breakdown

Example 1.5 (Cache Bits)

1. Offset Bit
   - For example, might need 3 bits to represent the offset.
   - This is because the block size is 8 bytes, and $2^3 = 8$.
2. Index Bit
   - For example, might need 10 bits to represent the index.
   - This is because the cache size is 1024 bytes, and $2^{10} = 1024$.
3. Tag Bit
   - Then, would have 19 bits to represent the tag.
   - This is because the address size is 32 bits, and $32 - 3 - 10 = 19$.

1.2.4 Data Lookup in a Cache

1. Line Selection
   - Might want to check what Cache line the data is in. (We can do this by using the tag)
2. Line Matching
   - Check if the tag matches the tag in the cache line.
3. Word Extraction
   - If the tag matches, then we can extract the data from the cache line.

Question 1: When is the dirty bit set?

The dirty bit is flipped from `0` to `1` when the block is modified.

It is flipped from `1` to `0` when the block is written back to memory.

1.3 Day 27

November 7, 2022

1.3.1 Direct-Mapped Caches (redux)

Note:-

Review from last time

- Caches are broken into cache lines
- Cache lines are broken into blocks
- Many more blocks of memory then cache lines; so multiple blocks of memory mapped to each cache

---

- OFFSET determined by block size
- INDEX determined by number of cache lines
- TAG determined by address size and cache size

- Direct-mapped caches are prone to conflict misses

Example 1.6 (Cache Memories)

Same TAG and INDEX, but different OFFSET:
$\longrightarrow$ different indexes of same memory mapped to same cache line (same block)

Same INDEX and OFFSET, but different TAG:
$\longrightarrow$ different memory blocks mapped to same cache line

Same TAG and OFFSET, but different INDEX:
$\longrightarrow$ completely different memory blocks mapped to different cache lines

$\longrightarrow$ **Reading Cache Memory**

For example, read `01000100` with value 5. Then:

- TAG = `010`
- INDEX = `001`
- OFFSET = `00`

Procedure:

1. Check at index `001` in cache to see if take `010` is in the cache. If not, then read from memory and put in cache.

2. Since `010` is in the cache, then read the value at offset `00` in the cache line.

Note:-

If the dirty bit is set, then write the cache line back to memory before reading a new block from memory.

(Could in theory need to access memory twice to read a block from memory, in the case of needing to update memory, and then write in a new piece of data for the cache block)

$\rightarrow$ **Write cache-miss**

In the case of a cache miss on a write operation, then the cache line is read from memory, and then the value is written to the cache line. You can either change the memory based on the write operation, in which case the dirty bit will be set to 0, or can just change data in cache and set dirty bit to 1.

1.3.2 Cache Associativity

Associativity is the ability to check multiple cache lines for a given tag.

Definition 1.8: Cache sets

A cache set is a group of cache lines that are all indexed by the same index bits.

- Direct-mapped caches have only one cache set
- Fully associative caches have one cache line per cache set
- n-way associative caches have n cache lines per cache set

Example 1.7 (2-way Set Associative Cache)

- Half as many cache lines as direct-mapped cache, but twice as many cache sets.
- Have to look at 2 cache lines to see if a cache miss occurs.

$\rightarrow$ **Steps for Cache Extraction for Associative caches:**

1. Set selection: determine which cache set to look at
2. Tag comparison: determine if the tag is in the cache set
   (Hardware checks all lines in parallel)
3. Hit or miss: if tag is in cache set, then hit, otherwise miss

Note:-

As Associativity increases, the hardware gets more and more costly to produce, but the performance of the cache (and therefore CPU) increases.

1.3.3 Cache Replacement Policy

What cache lines to evict in a given set?

Definition 1.9: Cache Replacement Policy

A cache replacement policy is a policy for choosing which cache line to evict when a cache set is full.

- **LRU**: evict the cache line that was least recently used
  this is mostly used
- **FIFO**: evict the first cache line that was put in the cache set
- **Random**: evict a random cache line
- **Optimal**: evict the cache line that will not be used for the longest time

1.4 Day 28

November 9, 2022

Review from Last Time

- Replacement Policy: Choose <u>LRU</u> (least recently used)

Table 1.1: Cache Set

Set	LRU	line 1	line 2
	↑ bit		

1.4.1 Caching Model

- **S** = number of sets (s set bits $\rightarrow 5 = 2^5$)
- **E** = number of lines per set (associativity level)
- **B** = bytes per block (b offset bits $\rightarrow B = 2^b$)

Example 1.8 (Direct and Fully Associative Caches)

1. Direct Mapped Cache:
   - **E** = 1
2. Fully Associative Cache:
   - **S** = 1

Note:-
In the case of there being an invalid cache line, if you are using a LRU Cache, you would want to replace that line rather than changing the least recently used line.

Question 2: Can Adjacent blocks be mapped to the same cache?

For example:

- block 0: addresses 0x0000 - 0x0003
- block 1: addresses 0x0004 - 0x0007

These two blocks are adjacent, but they are not mapped to the same cache line.

This holds true for all cache types. So, the answer to the question "Can Adjacent blocks be mapped to the same cache?" is **no**.

1.4.2 Why do Caches work well?

Definition 1.10: The Principle of Locality

Some things in memory are accessed more frequently than other memory locations.

Locality is divided into **spatial locality** and **temporal locality**.

Temporal Locality

- **Temporal Locality** is the tendency for a program to access the same memory locations repeatedly.
- **Example:**
  - A program that accesses an array of integers will tend to access the same integers repeatedly.

Spatial Locality

- **Spatial Locality** is the tendency for a program to access memory locations that are close to each other.
- **Example:**
  - A program that accesses an array of integers will tend to access the integers that are close to each other.

```
$sum \leftarrow 0$ for $i \leftarrow 0 ton$ do
    $sum \leftarrow sum + A[i]$
return $sum$
```