CIS501: Caches

Shreyas S. Shivakumar

December 2, 2019

1 Lecture Notes

1. Memory

- Memory is tricky - it can either be *large* (number of bits that it can store) or it can be *fast* (time taken to access an entry), but it cannot be both.
- Volatile Memory:
  (a) **Static RAM (SRAM)** is very fast and is most often used in CPU *L1 and L2 caches*. The latency is in the *sub-nanoseconds*.
  (b) **Dynamic RAM (DRAM)** is optimized for density and is slower (in the *tens of nanoseconds*). DRAM is synonymous with regular PC ram and the speeds associated (3200MHz) with RAM is it's refresh rate. *Refreshing* RAM is not required in *Static RAM*.
- Non-Volatile Memory: Magnetic Disks, Flash RAM *etc.*
- Good caching strategies are important because if you have a *cache miss*, you will be forced to access lower levels of memory which are very expensive (*slow*) operations.

2. The Memory Wall

Processor speeds are increasing at a faster rate than memory speeds.

Clever designs are required to harness the effective speeds of the CPU because a 4GHz processor might be wasted on a system whose memory access latency is 12 milliseconds (*magnetic disk*).

3. Memory Hierarchy

As hinted above, a processor can only compute as fast as it can access memory. For *example:* a 3GHz processor can execute ADD instructions in 0.33ns. However, your average *main memory* latency is more than 33ns. And LDR and STR instructions require even more time.

But it's impossible to have memory that can *keep up* with the fast processor clock rates while having sufficient amount of space for running programs. How do you address this problem? *A **hierarchy** of memory.*

Organization:

The goal is to have multiple layers of memory between the *CPU* and the *Hard Disk* that try to deliver both *fast* memory and *large* memory in a hierarchical manner, with the *fastest* layers of memory, **L1 cache** (I$ and D$), **L2 cache** and **L3 cache** being close to the CPU, followed by **Main Memory** and eventually the **Disk**.

Level	Memory Type	Managed By
0	Registers	Compiler
1	Primary Cache - I$ and D$ (L1)	Hardware
2	Second and Third Cache (L2, L3)	Hardware
3	Main Memory	OS
4	Disk	OS

Note: Level 0, Level 1 and Level 2 are usually **on-chip**. Main Memory is made of *DRAM* and is usually of in the order of 8GB to 16GB on professional machines.

Note: Chips today (Intel Core i7) are usually 30-70% cache by area.

4. Locality

A key strategy in improving memory performance is *Locality.*

- **Spatial Locality:** if a program recently referenced a chunk of data, it is likely to refer to data *near* (in memory) that chunk of data soon. To improve performance, we can **proactively** fetch large chunks of data including data nearby and keep it available for fast access since the program will be likely to request it soon anyway. *Example:* arrays.

- **Temporal Locality:** if a program recently referenced data, it is likely to refer to that data *again* soon. To improve performance, we can **reactively** "cache" this data in small, fast memory so that the program can use it again quickly. *Example:* counter variables.

5. Cache Overview

- A cache is a hardware *hashtable*.
- A cache is typically organized as a number of individual cache **blocks** or cache **lines**. For *example:* a 4KB cache can be organized as 1024 $\times$ 4B ($4 \times 8 = 32$ bits) blocks / lines. Since there are 1024 blocks, you will need $log_2(1024) = 10$ bits to index into this cache. These are **Index Bits**.
- **Offset Bits:** to index to specific bytes within a cache block, the *Least Significant Bits (LSB)* are used. For the above *example*, since there are 4 bytes in a cache block, you would need $log_2(4) = 2$ offset bits along with the 10 *Index Bits*.

  Q: Offset Bits aside, Why should we use the remaining 10 bits of Least Significant Bits as Index bits instead of maybe the Most Significant Bits?

  Lower order bits have a higher entropy in the general execution of a program. Higher order bits also are at risk of hashing to the same entry.
- **Tag Bits:** How can you know what blocks, if any, are in the cache? In the previous *example*, the remaining bits (32 bits - 10 bits - 2 bits = 20 bits) are used to *tag* each cache word. These tag bits are entered into a separate (and parallel) memory structure called the **Tag Array**. Additional to the 20 bits is a single *Valid Bit* to indicate if there is information in the cache blocks or not.

  Q: Since this mechanism adds extra hardware (it is now doubled) to the cache, is it much slower?

  There is a small overhead, but since these two hardware structures can be **accessed in parallel**, performance is not affected too much.
- In modern implementations, more than one address is stored in each cache block since typical architectures have more than single byte granularity. Therefore, the entries in each block come from a contiguous set of addresses and inherently exhibit *spatial locality*.
- At this basic level, to remember are:

- Index Bits : which row in cache?
- Offset Bits : which column (byte) in cache?
- Tag Bits : different memory being mapped to the same cache block?
- Tag Array : where are tag bits stored?
- Valid Bit : is this a valid cache entry?

6. Cache Miss

What if the data that was requested isn't in the cache? Who handles this?

A **Cache Controller** is a finite state machine that is responsible for remembering the address of data that resulted in a *cache miss*. It accesses the next level of memory in order to acquire the necessary data. It must wait until it receives a response and then must write the required information and tags into the proper location in the cache.

7. Glossary

- Access : read or write to cache
- Hit : required data was found in cache
- Miss : required data was not found in cache
- Fill : bringing required data into the cache
- Miss Rate ($\%_{misses}$) : ratio of misses to accesses
- Access Latency (t_{access}) : time to check cache
- Miss Latency (t_{miss}) : time to read data into cache

Note: A **Fill** usually also entails the costs involved in *evicting* data that was already in the cache to make room for the new data that is incoming.

8. Performance Equation

$$t_{average} = t_{access} + (\%_{misses} \times t_{miss}) \tag{1}$$

Note: t_{access} is paid regardless of whether it was a hit or a miss. It is the cost associated with reading or writing to the cache.

Note: Since a cache miss is technically a function of all the latency associated with retrieving data from the different layers of memory, this above equation is usually **recursive** up-to the number of layers in the memory hierarchy.

What's the simplest way to reduce $\%_{misses}$?

Increase capacity! Miss rates will decreases monotonically but at a certain size of **working set** you will notice diminishing returns. And with the increased capacity, t_{access} inevitably increases, resulting in lower performance. Intuitively, remembering more $\implies$ miss less, **but** remembering more $\implies$ access it slower.

Example #1:

If you have a simple pipeline with $CPI_{base} = 1$ and 30% of instructions are LDR and STR. If the L1 cache has $\%_{miss} = 2\%$ for the Instruction Cache (I$) and $\%_{miss} = 10\%$ for the Data Cache (D$) with a miss penalty of $t_{miss} = 10$ cycles for both, what is the new CPI_{new}?

$$CPI_{new} = CPI_{base} + \%_{I-miss} \times t_{I-miss} + \%_{D-miss} \times t_{D-miss}$$
$$CPI_{new} = 1.0 + 0.02 \times 10 + 0.1 \times 10 = 1 + 0.2 + 0.3 = 1.5$$

9. Cache Organization Example

A system has 4-bit addresses ($\rightarrow$ 16B memory) and an 8B cache with a block size of 2B. What are the *Index Bits*, *Offset Bits* and *Tag Bits*?

The number of rows (sets) is: $\frac{\text{cache capacity}}{\text{block size}} = \frac{8B}{2B} = 4$

Offset Bits: The number of bytes in a block is 2, therefore you will need $log_2(2) = 1$ offset bits.

Index Bits: There are 4 rows (sets) of cache blocks / lines, therefore you will need $log_2(4) = 2$ index bits.

Tag Bits: The remaining bits will be used as tag bits, i.e 4 (total) - 1 (offset) - 2 (index) = 1 tag bit.

10. Performance Optimizations

As we have noticed, increasing the size of the cache doesn't necessarily result in improvement in performance. We need to think of other mechanisms

that can help manipulate the $\%_{misses}$ by changing how the cache is organized, given that it has a fixed capacity.

(a) Increasing Block Size

(b) Adding Associativity

11. Increasing Block Size

This seems valid and is directly tied to the notion of *spatial locality*.

Q: How do the different bits change with increasing block size?

If the size of the blocks increases, each block will have more bytes, and therefore will require **more offset bits** to individually index into them. Since the size of the cache is fixed, this will mean that the number of sets / rows will actually reduce, resulting in **fewer index bits**. The tag bits remain unchanged.

Q: Can any of these go to 0 bits?

This is possible, if the block size is made equal to the size of the entire cache, this would mean that there is a single cache block / line, which will require no index bits.

*Q: Since the number of tag bits stays the same during the increase in block size, why does the overhead associated (**tag overhead**) with the tag reduce?*

While the tags remain at the same size, the number of total tags is reduced. Since each tag is associated with a set / row, and the number of sets / rows reduces, you will have fewer tag entries resulting in a lower overhead.

$$\text{Tag Overhead} = \frac{\text{Number of Tag Bits}}{\text{Block Size in Bits}}$$

*Q: What are the implications of larger block sizes on **performance?***

Since the block size is larger, more adjacent data is brought in per operation resulting in more **spatial pre-fetching** which can reduce the $\%_{misses}$ upto a point. However, after a certain point $\%_{misses}$ starts to increase because (a) potentially **useless data is also transferred** and (b) we might be **prematurely replacing** useful data.

Q: How does increasing block size affect the miss penalty / latency t_{miss}?

Technically, since the blocks are larger, they should take longer to **read, transfer** and **fill**. However, if you use **Critical World First (or) Early Restart** you can avoid this effect on isolated misses. This technique allows for the requested word to be fetched first and the remaining words are filled in the background. This will not, however, improve the situation where a cluster of misses occurs as this will turn into a *bandwidth problem* under those circumstances.

12. Associativity

Cache Conflicts: If you had had 16B of Main Memory and an 8B cache, can multiple addresses in memory map to the same location in the cache? **Yes**. You will essentially have 8 situations of conflict.

For *example:* addresses 0**01**0 and 1**01**0 will get mapped to the same location in the cache because they have the same **index bits**. Even if 0**01**0 was the only element in the cache, with 7 remaining bytes free, 1**01**0 would still conflict with it. Can we fix this? **Yes**, but this will take further re-organization of the cache structure.

Introduction:

In a **Direct Mapped Cache**, there is only one cache block where a memory block (*example:* 12) can be found, and if there are 8 blocks (and rows), then memory block 12 will get mapped to (12 modulo 8 = 4), i.e block id 4 in the cache.

In a **2-Way Set Associative Cache**, there would be four sets (rows) and memory block 12 must be in either one of the two (12 modulo 4 = 0) cache blocks.

In a **Fully Associative Cache**, the memory block 12 can appear in any of the eight cache blocks.

*Q: What is the effect of increasing **associativity** on the different bits?*

If associativity increases, the number of index bits will decrease and the number of tag bits will increase. For *example:*, a direct mapped cache will require more index bits than a 2-way set associative cache.

Q: What is the lookup procedure?

As before, the **index bits** are used to find the set / row. The data and tags in all frames are read in parallel. If any of them match and the **valid bit** is set, then it is a *Hit*.

Replacement Policies

With N-Way Associative Caches, what happens during a *Cache Miss*? Which block in a set should be evicted?

- Random
- FIFO
- **Least Recently Used (LRU)** : works well in practice. Requires additional **LRU bits** - for a 2-Way Set Associative cache, this is just a single bit.
- Not Most Recently Used (NMRU) : approximation of LRU, easier to implement
- Belady's : replace what will be used furthest in the future - impractical but good for simulation and pre-recorded traces

*Q: What is the impact of associativity on **performance**?*

With higher associative caches, you will have a lower $\%_{misses}$, but at a certain point you will have diminishing returns because t_{access} will increase.

Q: Does associativity need to be powers of 2?

No it does not!

13. Caches and Stores

Everything we have talked about so far has been in the context of reading from the cache, such as during a LOAD instruction. What about *STORE* instructions that write to the cache?

This will mainly involve the Data Cache (D$) since the Instruction Cache (I$) is basically read-only.

Issues arise with *write* operations because as we've hinted at before, there are **multiple copies** of a particular item across different levels of memory. When a *write/store* operation is performed, how do we **keep track** of all these different copies and update them accordingly? And how to we guarantee **consistency** across the different memory levels?

Another problem is with **tag / data access**: With read operations both tag and data arrays could be accessed in parallel. If there was a *tag mismatch*, that meant that the data was wrong and you could just *stall* until the right data is fetched. What about during a **write**?

We cannot **read tag** and **write data** in parallel because if you happen to write the wrong data - you've made a permanent change and it is hard to recover from this. Therefore, with *write* operations, this is a **two-step** process where:

- You first match the tag
- Then write to the matching *way*

This could introduce structural hazards and will require bypassing to avoid unnecessary stalls.

14. Propagating Writes

Q: Since lower levels of memory (basically the entire memory hierarchy) need to be updated by write instructions, when should you propagate these updated values?

There are two approaches to this: **Write-Through** and **Write-Back**. Modern systems will often use both of these techniques at different levels in memory.

Q: Suppose on a STORE instruction, we wrote data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be ***inconsistent****. How can we fix this?*

Write-Through : we maintain consistency by always writing the data into both the memory and the cache. *Write-Back* or *Copy-Back* : When a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced.

Write-Through:

- This is the *simplest approach*.
- This method does require extra bandwidth since you are basically updating the different memory levels each time the cache is updated.
- No special hardware is needed.

Write-Back:

- In this approach, a *write* is propagated only when a block is replaced. This is useful when a processor generates writes faster than the writes can be handled by main memory.
- There is a **dirty bit** associated with each block.
- A **clean block** indicates that all occurrences of this value are consistent, and that there is another copy of this block somewhere. Therefore, evicting a clean block is more straightforward and easy to do.
- A **dirty block** is one that has been updated, meaning that there is a *mismatch* across it's *copies*. These are more challenging to handle.
- We use an additional *hardware structure* called the **Write-Back Buffer (WBB)**. The *intuition* here is to take advantage of caching and only update one level of the hierarchy. Delay the process of updating the rest of the hierarchy for a later time, i.e propagate these changes at a different pace. This does however require an additional amount of bookkeeping to make sure that the changes do actually propagate.
- The contents of the *Write-Back Buffer* will be pushed down the hierarchy and at each stage it will be labeled *dirty* until it reaches the *main memory* and *disk*.
- This method uses less bandwidth
- *Q: Why use a Write-Back Buffer?* The WBB helps by being an intermediate storage element so that blocks can be **quickly evicted** and placed here thus freeing up room in the faster memory level.

15. Write Misses

Next we will see how *write misses* are actually handled and some optimizations that can be made to improve efficiency:

Q: How are write misses handled?

Write-Allocate: fills block from next level, then write it. This is commonly used, reduces read misses but requires additional bandwidth. **Write-Non-Allocate:** just writes to the next level, does not allocate. Results in more read misses but uses less bandwidth. Usually used with *Write-Through* propagation.

*Q: A LOAD instruction during a **read miss** can't go on without the data so it must stall, but does a STORE instruction during a **write miss** need to stall?*

No, we can actually keep going. And to do this, we require another hardware structure called the **Store Buffer**. STORE instructions put addresses / values into the store buffer and keep going. The caveat is that **LOAD** instructions must now also check the *Store Buffer* in addition to Data Memory.

16. What are the three types of cache misses?

- **Compulsory** Miss : It has never seen this address before, and no cache can help with this type of miss, even an infinitely sized cache.
- **Capacity** Miss : The miss occurred because the cache was too small. This miss would occur even if it was a fully associative cache.
- **Conflict** Miss : Miss caused because the associativity was too low.
- *Coherence Miss* : Miss caused by external invalidation.

17. What is a way of reducing Conflict Misses?

Conflict Misses occur as a result of insufficient associativity. Adding more associativity to the cache is *expensive* and rarely needed. But what if you could get *associativity on demand* if/when you actually need it? This is what a **Victim Buffer (VB)** will provide.

The **Victim Buffer** is a fully associative cache that sits on the L1 cache path. It is *small* and *very fast*. It holds blocks that were evicted from some other level and can provide these at a very low latency when required. If you have a *miss* and find it in the victim buffer, you can quickly place the block back in the L1 cache. It will be faster than accessing the L2 cache.

18. What is a way of reducing Compulsory Misses?

Bring data into the cache **proactively / speculatively**. Anticipate upcoming miss addresses and bring them into the cache. This is called *Prefetching* and can be done using simple hardware table-driven prefetching. It can identify common strides and access patterns and use predictors to identify and fetch data.

Software Prefetching exists too which is usually inserted by the programmer or the compiler.

19. What is a way of reducing Capacity Misses?

Capacity misses are usually a result of poor spatial or temporal locality. **Code Restructuring** is a simple yet effective way to handle this problem in many cases.

Loop interchange: Depending on whether your programming language is *row-major* or *column-major*, a simple interchange of iterator variables can result in a significant increase in performance.

Loop blocking: If accesses are spread out over time, you can exploit temporal locality to fetch and store a cache sized amount of data and more quickly access it.

20. Designing a Cache Hierarchy

- Higher level caches *(L1 cache)* optimize for t_{access}. They are more frequently accessed. They have lower capacity and smaller block sizes.
- Lower level caches *(L2 and L3 cache)* optimize for $\%_{misses}$. They are less frequently accessed. They have higher capacity, associativity and block size.

Design Choice #1: **Unified** Cache (I$ and D$) or **Split** Cache. A *Split Cache* minimizes structural hazards, but results in smaller, fixed capacities for both data and memory. A *Unified Cache* results in fewer capacity misses since unused instruction capacity can be used for data.

Design Choice #2: **Inclusive** Memory or **Exclusive** Memory. In an *Inclusive* design, a block in L1 is always in L2. In an *Exclusive* design, a block is either in L1 or L2, never in both.

21. Hierarchy Performance

$$t_{avg} = t_{avg-L1}$$
$$t_{avg-L1} = t_{access-L1} + (\%_{misses-L1} \times t_{misses-L1})$$
$$t_{avg-L1} = t_{access-L1} + (\%_{misses-L1} \times t_{avg-L2})$$
$$t_{avg-L1} = t_{access-L1} + (\%_{misses-L1} \times (t_{access-L2} + (\%_{misses-L2} \times t_{misses-L2})))$$

and so on...

Note: When calculating **miss rate**, there are two ways of performing this calculation (a) misses per instruction or (b) misses per cache access. Depending on the which approach you use, the percentage may vary.

Misses per instruction is a more straightforward approach and is synonymous to a *global miss rate*. Misses per access is trickier and is a more *local miss rate*.

Multi-level Performance Calculation:

Let's say that a system has 30% of it's instructions are memory operations. In the L1 cache, the $t_{access} = 1$ cycle and $\%_{miss} = 5\%$. In the L2 cache, the $t_{access} = 10$ cycles and $\%_{miss} = 20\%$. In the main memory $t_{access} = 50$. What is the CPI of this system?

Percentage of instruction misses in the L1 cache is : 30% of 5% $\implies$ 1.5%.

Percentage of instruction misses in the L2 cache is : 20% of 1.5% $\implies$ 0.3%.

$$CPI = 1 + (1.5\% \times 10) + (0.3\% \times 50)$$
$$CPI = 1 + 0.15 + 0.15 = 1.3$$

2 Textbook Notes

1. Why do we care about good memory architectures?

All programs spend much of their time accessing memory and the memory system is necessarily a major factor in determining performance.

2. What is the general idea behind having a memory hierarchy?

The goal is to present the user with as much memory (*i.e size*) as is available in the cheapest technology, while providing access at the *speed* offered by the fastest memory.

Faster memories are more expensive per bit than the slower memories and thus are smaller. The three primary technologies that are present in a hierarchy are *SRAM*, *DRAM* and *Magnetic Disk / Flash*.

3. Glossary

- **Block / Line:** The *minimum unit of information* that can be either present or not present in a cache.
- **Hit Rate:** The fraction of memory accesses *found* in a level of the memory hierarchy.
- **Miss Rate:** The fraction of memory accesses *not found* in a level of the memory hierarchy.
- **Hit Time:** The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.
- **Miss Penalty:** The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requester.

4. Ideal Scenario

If the *hit rate* is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

5. What is a *True Hierarchy*?

In a true hierarchy, data cannot be present in level i unless it is also present in level *i+1*.

6. What is a *Direct Mapped Cache*?

If each word in memory can go to exactly one place in the cache, then it is straightforward to find the word if it is in the cache.

$$\text{location} = (\text{block address}) \ \% \ (\text{number of blocks in the cache})$$

But...

Because each cache location can contain the contents of a number of different memory locations, (*i.e there is a many-to-one mapping*), how do we know whether the data in the cache corresponds to a requested word? **Tag bits!**

Tag Field: A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

Valid Bit: A field in the tables of a memory hierarchy that indicates that the associate block in the hierarchy contains valid data.

Q: When would a valid bit be required?

Think of the situation when a processor first starts up. The cache does not have good data and the tag fields present are actually meaningless. Without the tag bits, we wouldn't be able to tell between random data and actually meaningful information.

7. Cache Miss

The processing of a cache miss creates a *pipeline stall* as opposed to an interrupt, which would require saving the state of all the registers.

8. What are the two directions in which performance can be improved?

- Reducing the *miss rate* by reducing the probability that two different memory blocks will contend for the same cache location.
- Reducing the *miss penalty* by adding additional levels to the hierarchy - multi-level caching.

9. A *Write Buffer* is used in *Write-Back* to improve performance. When does a *Write Buffer* fail?

A *Write Buffer* can stall when the buffer is full and if a write occurs. It is difficult to quantitatively analyze the exact penalty introduced by a write buffer stall. In a simple model, the stalls associated with the write buffer are ignored.

10. What is a *Fully Associative Cache*?

A cache structure in which a block can be placed in any location in the cache.

To find a given block in a fully associative cache, *all the entries* in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly *increase the hardware cost*, effectively making fully associative placement practical only for caches with small number of blocks.

Note: In a fully associative cache, there is effectively only one set, and all the blocks must be checked in parallel. Thus, there is no need for an *index field* and the entire address (*excluding offset*) is compared with the *tag entry*.

11. What is a *Set Associative Cache*?

A cache that has a fixed number of locations (*at least two*) where each block can be placed.

In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called a n-way set associative cache. An n-way set-associative cache consists of a number of **sets**, each of which consists of n blocks. Each block in the memory maps to a unique **set** in the cache.

$$\text{location} = (\text{block address}) \ \% \ (\text{number of } \textit{sets} \text{ in the cache})$$

12. What happens when *associativity* is increased?

This usually results in a decrease in miss rate. However, there is a potential increase in access time / hit time, so there is a *tradeoff* to be made.

13. Where can blocks be placed?

Scheme	Number of *sets*	Blocks per *set*
Direct Mapped	Number of blocks in cache	1
Set Associative	$\frac{\textit{Number of blocks in cache}}{\textit{Associativity}}$	Associativity
Fully Associative	1	Number of blocks in cache

14. How is a block found?

Associativity	Location method	Comparisons required
Direct Mapped	Index bits	1
Set Associative	Index set, *search elements*	Associativity
Fully Associative	Search cache entries	Size of the cache

15. What block is replaced during a *miss*?

Typically, either the *Least Recently Used (LRU)* or *Not Most Recently Used (NMRU)*, i.e an approximation of LRU or just Random selection.

16. How are writes handled?

Each level in the hierarchy can use either write-through or write-back.