EECS 370: Introduction to Computer Organization Notes

Noah Peters

April 17, 2023

Abstract

Lecture notes for EECS 370 at the University of Michigan. LaTeXtemplate by Pingbang Hu.

Contents

Chapter 3

Caches

Lecture 17: Introduction to Caches

3.1 Memory Hierarchy

14 Mar. 12:00

We often need a lot of memory, LC2K alone can handle 2^{18} bytes of memory. We have several choices for memory:

- **SRAM**: Static Random Access Memory
  - fast: 2ns access time or faster
  - decoders are big
  - expensive, high area requirement
- **DRAM**: Dynamic Random Access Memory
  - slower: 50ns access time
  - must stall for dozens of cycles each memory load
  - less expensive
- **Flash**
  - slow: access time varies wildly
  - less expensive than DRAM
  - non-volatile
- **Disks**
  - obnoxiously slow: 3,000,000ns access time
  - dirt cheap
  - non-volatile

As seen above, there are trade-offs among each type of memory. Ideally, we would have cheap *and* fast memory. So, we can use a combination of memory types to optimize the common case via strategic **locality of reference**.

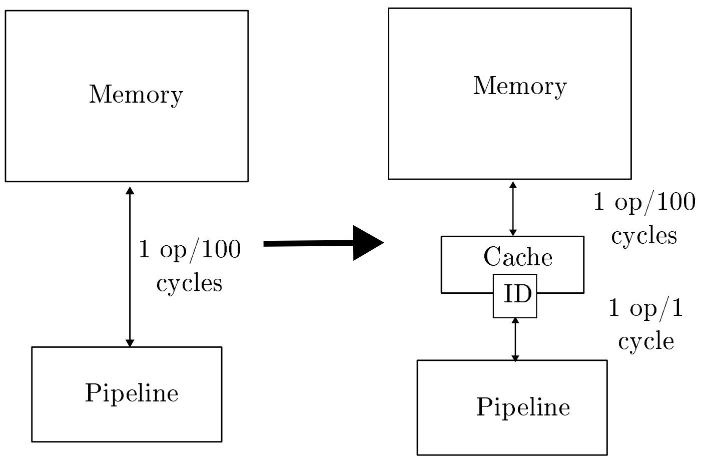


Figure 3.1: Caches Overview

Definition 3.1.1. The **architectural** view of memory is what the machine language (or programmer) sees, i.e. just a big array.

Note. Breaking up the memory system into different pieces (cache, main memory, disk) is **not architectural**. The machine language doesn't know about it.

Can use our variety of memories as follows:

- use small array of SRAM for the **cache**
- use a larger amount of DRAM for **main memory**
- use a lot of flash and/or disk for **virtual memory**

3.2 Cache Basics

Whenever memory returns data, we can store it in a cache. We'll need to store:

- the data
- a tag denoting its memory location
- a "valid" status bit

Then for our next memory access, we can first check if the tag matches the address we are attempting to access.

Definition 3.2.1 (Cache Hit). A **hit** occurs when data for a memory access is found in the cache.

Definition 3.2.2 (Cache Miss). A **miss** occurs when data for a memory access is *not* found in the cache.

Definition 3.2.3 (Hit/Miss Rate). The **hit/miss rate** is the percentage of memory accesses that hit/miss in the cache.

3.2.1 CAMs

Definition 3.2.4. CAMs: content addressable memories are akin to a set of data matching a query. Instead of an address we send a key to the memory, asking whether the key exists and, if so, what value it is associated with. Memory answers: yes/no and gives associated value (if there is one).

We apply *operations* on CAMS:

- **Search**: the primary way to access a CAM
  - send data to CAM memory
  - return "found" or "not found"
  - if found, return location of where it was found or its associated value
- **Write**: send data for CAM to remember

3.2.2 Cache Organization

Cache memory can copy data from any part of main memory. Cache memory has two parts:

- the **tag (CAM)**: holds the memory address
- the **block (SRAM)**: holds the memory data

A **hit** in the cache occurs when a tag match is found. The microprocessor sends the address to the CAM containing the tags and searches for the tag. If there's a search result hit, the corresponding block is forwarded to the microprocessor. If not, the address is forwarded to main memory:

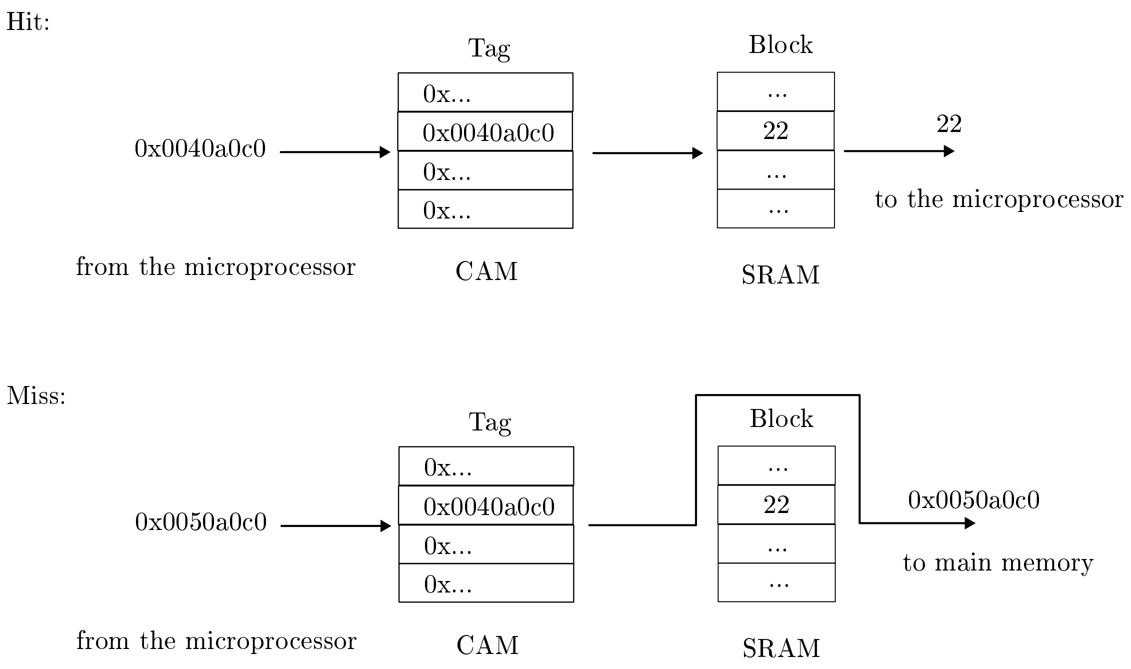


Figure 3.2: Hardware view of caches.

Problem 3.2.1. Given:

- *cache* has 1 cycle access time
- *main memory* has 100 cycle access time
- *disk* has 10,000 cycles access time

What is the average access time for 100 memory references if 90% of the cache accesses are hits and 80% of the accesses to main memory are hits? Assume main memory access time includes tag array access to determine hit/miss.

Answer. $0.9 \cdot 1 + 0.1 \cdot (100 + 0.2 \cdot 10000) = 210.9$ ⊛

3.2.3 Cache Operation

Every cache *miss* will get the data from memory and *allocate* a cache line to put the data in (just like any CAM write). However... what do we replace in the cache? Does an optimal replacement policy exist?

Definition 3.2.5 (Temporal locality). The principle of **temporal locality** in program references says that if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location.

Remark. Locality is a property of *programs* (not hardware).

Specifically, temporal locality says that the **least recently referenced (LRU)** cache line should be *evicted* to make room for the new line. Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced.

Definition 3.2.6 (Average Access Latency). Average Access Latency = cache latency · hit rate + memory latency ·

Lecture 18: Cache Organization: Block Size and Writes

16 Mar. 12:00

Definition 3.2.7 (k-Way Set Associative LRU). A set associative LRU with k **ways**, has essentially k cells within each cache line's block. The LRU is set to be the *way* with the highest *count*. 20:43

Each cache hit now essentially brings in two blocks of data, since each block in memory is paired up with an adjacent block. Now, we can have the LSB of our memory address correspond to either block A or block B of the retrieved block, and the remaining bits correspond to a (now-shortened) memory address.

Definition 3.2.8 (Spatial Locality). **Spatial locality** in a program says that we reference a memory location (e.g., 1000), we are more likely to reference a location near it (e.g., 1001) than some random location.

3.3 Cache Organization

We now consider the design of a cache.

3.3.1 Block Size

- choice of block size found by simiulating lots of different block sizes and seeing which ones give the best performance
- most systems use a block size between 32 and 128 bytes
- *longer sizes* reduce overhead by reducing the number of CAM entries and reducing the size of each CAM entry

Problem 3.3.1. Given a cache with the following configuration:

- total size is 8 bytes

- block size is 2 bytes
- fully associative
- LRU replacement
- memory address size is 16bits and is byte addressable

1. How many bits are for each tag? How many blocks in the cache?
2. For the following reference stream, indicate whether each reference is a hit or miss: 0, 1, 3, 5, 12, 1, 2, 9, 4.
3. What is the hit rate?
4. How many bits are needed for storage overhead for each block?

Answer.

1. 16 bits total - 1 bit for block offset = **15 bits** for each tag. 8 bytes total / 2 bytes per block = **4 blocks**.
2.
   - 0: never in cache before $\rightarrow$ **miss**
   - 1: 0 was just cached and block size is 2 bytes $\rightarrow$ **hit**
   - 3: 2 nor 3 have been cached yet $\rightarrow$ **miss**
   - 5: 4 nor 5 have been cached yet $\rightarrow$ **miss**
   - 12: 12 nor 13 have been cached yet $\rightarrow$ **miss**
   - 1: 1 was cached and hasn't been booted from cache $\rightarrow$ **hit**
   - 2: 3 was cached recently $\rightarrow$ **hit**
   - 9: LRU was 4/5 block, so 4/5 booted from cache, 8/9 in now $\rightarrow$ **miss**
   - 4: 4/5 block just booted, so not in cache; boots 12/13 $\rightarrow$ **miss**
3. $3/9 = 0.33$
4.
   - Tag: 15 bits
   - Valid: 1 bit
   - LRU: $\log_2(4) = 2$ bits

   $\Rightarrow$ total: **18 bits**

⊛

3.3.2 Stores and the Cache

Where should we write the result of a store?

- If that memory location is in the cache?
  - Send it to the cache
  - should we also send it to memory (**write-through policy**)
- If that memory location is *not* in the cache?
  - Allocate the line, i.e. put it in the cache (**allocate-on-write policy**)
  - Write it directly to memory without allocation (**no allocate-on-write policy**)

Add memory diagram from lecture

Definition 3.3.1 (Write-Through Policy)**.** When storing a value from the processor to memory and the memory location is in the cache, we store the value in the cache and simultaneously change the value in memory.

Definition 3.3.2 (Allocate-On-Write Policy)**.** When storing a value from the processor to memory and the memory location is *not* in the cache, we must first *read* the desired block from memory, before storing into the block in the cache and memory.

Definition 3.3.3 (No Allocate-On-Write Policy)**.** When storing a value from the processor to memory and the memory location is *not* in the cache, we can ignore the cache and simply update memory with the given value.

Remark. This method may sometimes be less advantageous if we are going to read the value from memory soon after writing to it. However, write data streams and read data streams are often independent, so this isn't usually the case.

Lecture 19: Direct Mapped and Set Associative Caches

3.4 Write Back Caches

21 Mar. 12:00

We can design the cache to *not* write all stores to memory immediately. We can do this by keeping the most recent copy in the cache and update the memory *only when* that data is evicted from the cache.

Definition 3.4.1 (Write-Back Policy)**.** Under this policy, when storing a value from the processor, we update the *cache* but *do not* update the value in *memory.*

We don't need to write-back all evicted lines, only those blocks that have been stored into. We can keep a **dirty bit** that *resets when the line is allocated* and *set when the block is stored into.* If a block is "dirty" when evicted, write its data back into memory.

V	D	LRU	Tag	Data block (bytes)

Figure 3.3: Contents of a cache line.

Writes Summary

Store With **No Allocate**	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache and *Mem*
Miss?	Write to Mem	Write to Mem
Replace Block?	If evicted block is dirty, write to Mem	Do nothing

Store With **Allocate**	Write-Back	Write-Through
Hit?	Write Cache	Write to Cache and *Mem*
Miss?	Read from Mem to cache, alllocate to LRU block, write to cache	Read from Mem to cache, alllocate to LRU block, write to cache *and Mem*
Replace Block?	If evicted block is dirty, write to Mem	Do nothing

Problem 3.4.1. Consider the following cache:

- 32-bit memory addresses
- byte addressable
- 64 KB cache
- 64 B cache block size
- **write-allocate**
- **write-back**
- *fully* associative

This cache will need 512 kilobits for the data area (64 kilobytes times 8 bits per byte). Note that here, 1 kilobyte = 1024 bytes. Besides the actualy cached data, this cache will need other storage. Consider **tags, valid bits, dirty bits, bits to track LRU, etc.**
How many additional bits (not counting data) will be needed to implement this cache?

Answer. We have 2^{10} blocks:

$$64 \text{ KB} = 2^{16} \text{ b} \rightarrow \frac{2^{16} \text{ b}}{2^{6}\text{b / block}} = 2^{10} \text{ blocks.}$$

The **tag's** size:

$$32 \text{ b for mem addresses} - 6 \text{ b for block offset} = 26 \text{ b for tag.}$$

The **valid** and **dirty bits** only need 1 bit each. To keep track of LRU we can calculate the number of bits needed as follows:

$$2^{16}/2^{6} = 2^{10} \text{ blocks in the cache} \rightarrow 10 \text{ bits needed to keep track of LRU}$$

So, the total number of additional bits needed for the cache is

$$26 + 1 + 1 + 10 = \textbf{38 bits} \text{ (per block).}$$

⊛

Problem 3.4.2. Suppose that accessing a cache takes **10 ns** while accessing main memory, and in the case of cache-miss it takes **100 ns**.

a What is the average memory access time if the cache hit rate is 97%?

b If the cache size is increased, causing the hit rate to rise by 1% and the time for accessing the cache by 2 ns. Will this improve performance?

Answer.

a
$$10 \text{ ns} + 0.03 \cdot 100 \text{ ns} = 13 \text{ ns.}$$

b
$$12 \text{ ns} + 0.02 \cdot 100 \text{ ns} = 14 \text{ ns.}$$

⊛

3.5 Direct-Mapped Caches

A block can go to *any* location. This means that when we are checking tags, we *check every cache tag* to determine whether the data is in the cache. This parallel approach is what is used for **fully associative caches**, which we have studied so far. However, this method is *slow*.

We can redesign the cache to eliminate the requirement for parallel tag lookups.

Definition 3.5.1 (Direct-Mapped Caches)**.** Direct-mapped caches partition memory into as many regions as there are cache lines. Each memory region maps to a **single cache line** in which data can be placed. You then only need to **check a single tag**: the one associated with the *region the reference is located in.*

Remark. The memory regions that map to the same cache line are placed as far apart in memory as possible, so that blocks that are spatially close are not competing for the same cache line.

Since two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time, a 0% hit rate is possible if more than one block accessed in an interleaved manner map to the same index.

Problem 3.5.1. How many tag bits are required for the following cache specs? What are the overheads of this cache?

- 32-bit address, byte addressed
- direct-mapped 32k cache
- 128 byte block size
- write-back

Answer. The number of bits required for the line index is

$$15 - 7 = 8. \ (2^{15} = 32 \text{ kb cache})$$

The tag size is

$$32 \text{ b total} - 7 \text{ b for block offset} - 8 \text{ b for line index} = \textbf{17 b for tag}.$$

So the overhead in this cache is

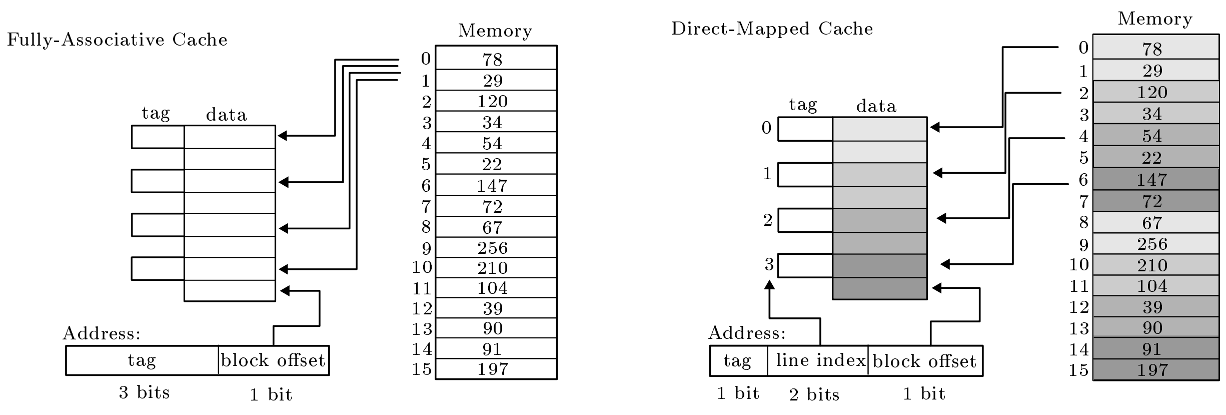
$$\begin{aligned} 17 + 1 \text{ b for valid bit} + 1 \text{ b for dirty bit} &= 19 \text{ b } / \text{ cache line} \\ 19 \text{ b } / \text{ line} \cdot 256 \text{ lines} &= 4864 \text{ bits} \\ 4864 \text{ b}/32 \text{ KB} &= \textbf{1.9\% overhead}. \end{aligned}$$

⊛

Lecture 20: Set-Associative Caches and 3 C's

23 Mar. 12:00

Unlike fully-associative caches, in direct-mapped caches the tag array (CAM) doesn't have to be searched *before* the data array (SRAM). Instead, we can do a **direct lookup** and search the tag array and data array in **parallel**, both of which are faster for cache lookups. The *downside* to direct-mapped caches, however, is that there is an increased chance for collisions in the cache.



(a) A basic *fully-associative* cache.

(b) A basic *direct-mapped* cache.

Figure 3.4: Fully-associative and direct-mapped caches.

3.6 Set Associative Caches

We can achieve the advantages each the fully-associative and direct-mapped cache structures using **set associative caches**.

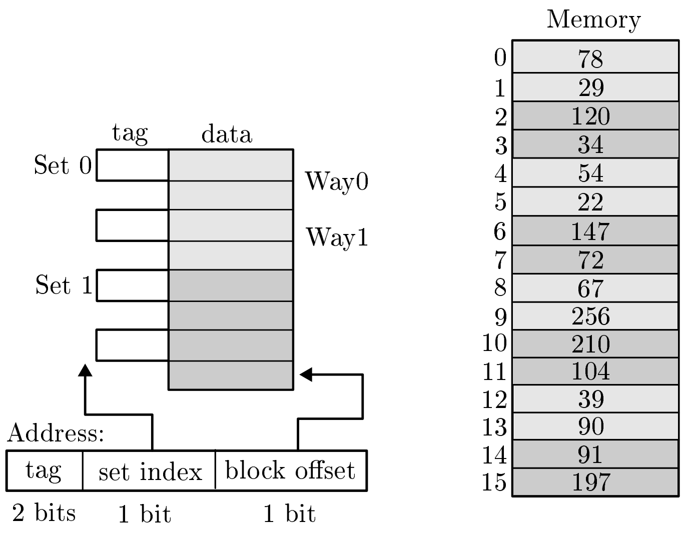


Figure 3.5: A basic set-associative cache.

Definition 3.6.1 (Set Associative Cache)**.** Set associative caches:

- Partition memory into regions (like direct-mapped but fewer partitions)
- Associate a region to a set of cache lines; check tags for all lines in a set to determine a hit
- Treat each line in a set like a small fully associative cache (with LRU policy)

Problem 3.6.1. For a 32-bit address and 16 KB cache with 64-byte blocks, show the breakdown of the address for the following cache configurations:

a Fully Associative Cache

b 4-Way Set Associative Cache

c Direct-Mapped Cache

Answer. Since the cache size is 16 KB $= 2^4 \cdot 2^{10} = 2^{14}$ bytes and there are 64 bytes per block we have

$$2^{14}/2^6 = 2^8 \text{ blocks.}$$

a For the **fully associative** cache:

$$\begin{aligned} 64 \text{ byte blocks} &\rightarrow 6 \text{ bits for block offset} \\ 32 \text{ b - } 6 \text{ b} &= 26 \text{ bits for tag} \end{aligned}$$

b For the **4-way associative** cache:

$$\begin{aligned} 64 \text{ byte blocks} &\rightarrow 6 \text{ bits for block offset} \\ 2^8 \text{ blocks}/4 \text{ blocks per set} = 2^6 \text{ sets} &\rightarrow 6 \text{ bits for set index} \\ 32 \text{ b - } 12 \text{ b} &= 20 \text{ bits for tag} \end{aligned}$$

c For the **direct-mapped** cache:

$$\begin{aligned} 64 \text{ byte blocks} &\rightarrow 6 \text{ bits for block offset} \\ 2^8 \text{ different blocks} &\rightarrow 8 \text{ bits for line index} \\ 32 \text{ b} - 6 - 8 &= 18 \text{ bits for tag} \end{aligned}$$

⊛

Remark. Note that we've used *block sizes, number of sets, number of ways* that are all be power of 2. This allows us to properly space blocks, sets, and data, etc.

3.7 The 3 C's of Cache Misses

Definition 3.7.1 (Compulsory Miss). A **compulsory** miss, also called a "cold start" miss, occurs the first time there is a reference to any block, which always results in a miss.

Definition 3.7.2 (Capacity Miss). A **capacity** miss occurs when the cache is too small to hold all the data, and a hit would have occurred if the cache were large enough.

Definition 3.7.3 (Conflict Miss). A **conflict** miss occurs when the cache is not associated enough. One set is getting a disproportionate amount of accesses, where a hit would have occurred with a fully associative cache.

3.7.1 Classifying Cache Misses

To classify cache misses we can use different forms of simulation:

- Simulate with a cache of unlimited size (cache size = memory size) → any misses must be *compulsory* misses
- Simulate again with a fully associative cache of intended size → any new misses must be *capacity* misses
- Simulate a third time, with the actualy intended cache → any *new* misses must be *conflict* misses

3.7.2 Fixing Cache Misses

We have various methods of reducing each type of cache miss:

- *Compulsory*: increase block size (and thus reduce total number of blocks)
- *Capacity*: build bigger cache
- *Conflict*: increase associativity

Problem 3.7.1. Consider a cache with the following configuration: write-allocate, total size of 64 bytes, block size is 16 bytes and 2-way associative, memory address size is 16 bits and byte-addressable, replacement policy is LRU, and cache is empty at the start.

For the following memory accesses, indicate whether the reference is a hit or miss, and the type of miss:

0x00, 0x14, 0x27, 0x08, 0x38, 0x4A, 0x18, 0x27, 0x0F, 0x40

Answer. Note that the block size is 16 bytes, so FA and SA have 4 blocks. SA is 2-way.

Address	Infinite	FA	SA	3 Cs
0x00	M	M	M	Compulsory
0x14	M	M	M	Compulsory
0x27	M	M	M	Compulsory
0x08	H	H	H	N/A
0x38	M	M	M	Compulsory
0x4A	M	M	M	Compulsory
0x18	H	M	H	N/A
0x27	H	M	M	Capacity
0x0F	H	M	M	Capacity
0x40	H	H	M	Conflict

⊛

3.8 Cache Parameters vs. Miss Rate

3.8.1 Cache Size

As cache total data (not including tag) capacity increases, temporal locality can be used better. However, it's not *always* better. Too large of a cache adversely affects hit and miss latency, too small of a cache doesn't exploit temporal locality well. The **working set** (i.e. the whole set of data executing application references within a time interval) size is constant, so expanding cache size beyond that isn't necessarily advantageous.

3.8.2 Block Size

Block size is the data that is associated with an address tag. Too small blocks don't exploit spatial locality well and have larger tag overhead. Too large blocks have too few total number of blocks, so we're less likely to transfer useless data.

3.8.3 Associativity

How many blocks can map to the same index? Larger associativity causes lower miss rates, less variation among programs, but diminishing returns. Smaller associativity causes lower costs, faster hit time.

3.9 Instruction vs. Data Caches

We've been focusing on caching loads and stores (i.e. data), but what about cache for instructions? Instructions should be cached as well! We have two options:

1. Treat instruction fetches as normal data and allocate cache lines when fetched

2. Create a second cache (called the **instruction cache** or ICache) which caches instrucations only; more common in practice.

We can integrate caches into a pipeline by replacing instruction memory with *Icache* and replacing data memory with *Dcache*. But there are some issues: what happens when both caches miss at same time? What about added latency?

Lecture 21: Cache Wrap-Up

3.10 Cache Wrap-Up

28 Mar. 12:00

Problem 3.10.1. The *grinder* app running on LC2K with full data forwarding and all brances predicted not-taken has the following frequencies:

45% R-type, 20% branches 15% loads, 20% stores

In *grinder*, 40% of branches are taken and 50% of LWs are followed by an immediate use. The I-cache has a miss rate of 3% and the D-cache has a miss rate of 6% (no overlapping of misses). On a miss, the main memory is accessed and has a latency of 100 ns. The clock frequency is 500 MHz.

What is the CPI of *grinder* on the LC2K?

Answer. Since the clock frequency is 500 MHz, we have a 2 ns cycle time. So the stalls per cache miss have a length of

$$100 \text{ ns}/2 \text{ ns} = 50 \text{ cycles}.$$

So the CPI will equal:

$$CPI = 1 + \text{data hazard stalls} + \text{ctrl hazard stalls} + \text{I-Cache stalls} + \text{D-Cache stalls}$$

$$= 1 + 0.15 \cdot 0.5 \cdot 1 + 0.2 \cdot 0.4 \cdot 3 + 1 \cdot 0.03 \cdot 50 + 0.35 \cdot 0.06 \cdot 50 = \mathbf{3.865}.$$

⊛

Problem 3.10.2. Say you have the following:

- *program*: generates 2 billion loads, 1 billion stores, each 4 bytes in size
- *cache*: a 32-byte block which gets a 95% hit rate on the program

How many bytes of memory would be read and written if:

a We had no cache?

b We had a write-through cache with a no-write allocate policy?

c We had a write-back cache with a write-allocate policy? (Assume 25% of misses result in dirty eviction)

Answer.

a *No cache*: All stores go to memory and are 4 bytes each, so 1 billion · 4 bytes = 4 billion bytes for writes. Similarly, there are 4 bytes per read, so there are 2 billion · 4 bytes = 8 billion bytes for reads.

b *Write-through, no allocate*: No allocate means we bypass cache on a miss, and write-through means we write to cache and memory on hit. So, again all stores will go to memory and are still 4 bytes each, so there are **4 billion bytes of writes**. Only loads that miss in the cache go to memory, but they read the full cache block. So there are 2 billion · 0.05 · 32 bytes = **3.2 billion bytes of reads**.

c *Write-back, write-allocate*: In this case, store misses result in a cache block being read, causing 1 billion stores $\cdot 0.05 \cdot 32$ bytes $= 1.6$ billion bytes to be read due to store misses. Similarly, *load* misses result in cache block being read, causing an additional 2 billion loads $\cdot 0.05 \cdot 32$ bytes $=$ 3.2 billion bytes to be read. Thus, there are **4.8 billion bytes read**. We only write to memory when there are dirty evictions, which can be done by both loads and stores. Since there are $0.05 \cdot 1$ billion stores $\cdot 32$ bytes $\cdot 25\% = 0.4$ billion bytes of writes due to store misses. There are 2 billion loads $\cdot 0.05 \cdot 32$ bytes $\cdot 25\% = 0.8$ billion bytes of writing due to load misses, resulting in **1.2 billion bytes of writes** in total.

⊛

Problem 3.10.3. Given a 200 MHz processor with 8 KB instruction and data caches and a with memory access latency of 20 cycles. Both caches are 2-way associative. A program running on this processor has a 95% icache hit rate and a 90% dcache hit rate. On average, 30% of the instructions are loads or stores. The CPI of this system, if caches were ideal would be 1.

Suppose you have the following two options for the next generation processor, which do you pick?

1. Double the clock frequency $\rightarrow$ assume this will increase your memory latency to 40 cycles, and the base CPI of 1 can still be achieved after this change
2. Double the size of your caches $\rightarrow$ this will increase the instruction cache hit rate to 98% and the data cache hit rate to 95%; assume the hit latency is still 1 cycle

Answer. See end of lecture. ⊛

Problem 3.10.4. If you have a 16 KB cache with 32-byte cache lines that is four-way set-associative on a computer with 32-bit addresses:

a. How many sets do you have?

b. How many bits do you need for the Offset? Index? Tag?

Answer.

a.

$$\begin{aligned} 2^14 \text{ bytes}/2^5 \text{ bytes per block} &= 2^9 \text{ blocks} \\ 2^9 \text{ blocks}/2^2 \text{ blocks per set} &= 2^7 \text{ sets} \end{aligned}$$

b.

$$\begin{aligned} \log(2^5 \text{ bytes per block}) &= 5 \text{ bits for Offset} \\ \log(2^7 \text{ sets}) &= 7 \text{ bits for set Index} \\ 32 - 7 - 5 &= 20 \text{ bits for Tag} \end{aligned}$$

⊛

Problem 3.10.5. Describe why we have an Icache and a Dcache on nearly all computers rather than just one unified cache.

Answer. It's advantageous to separate them since we typically just read instructions, while we need to also write to data. Separating them allows for better bandwidth since you are going to different places for different things. ⊛

Problem 3.10.6. Describe the primary **advantage** of *write-back* caches over *write-through* caches.

Answer. Each are better in the following scenarios:

1. **Write-Back** is better when we are writing to the same place *many, many times*.
2. **Write-Through** is better when we are writing to the same place *only once*.

⊛

Chapter 4

Virtual Memory

Lecture 22: Virtual Memory Basics

4.0.1 Memory Problems

4 Apr. 12:00

We run many programs on the same machine, each requiring (possibly) GBs of storage. Further, unrelated programs shouldn't have access to each other's storage. We also want to address the issues of expense, being able to run the same program on machines with different amounts of DRAM, and importantly: it would be nice to be able to *enforce different polices* on *different portions* of the memory (e.g. read-only).

4.1 Virtual Memory Basics

The solution to this is to build new hardware and software that automatically translates each memory reference from a **virtual address** (which the programmer sees as an array of bytes) to a **physical address** (which the hardware uses to either index DRAM or identify where the storage resides on disk).

Definition 4.1.1 (Virtual Memory)**. Virtual memory** hardware changes the virtual address the programmer sees into the physical one the memory chips see.

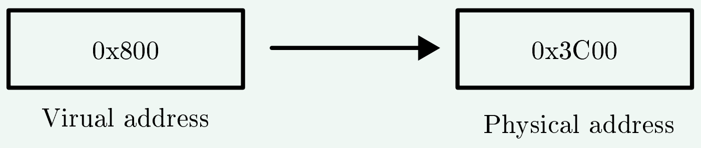


Figure 4.1: Basics of virtual memory.

Virtual memory enables multiple programs to share the physical memory. This provides the following 3 capabilities to the programs:

1. **Transparency**: don't need to know how other programs are using memory
2. **Protection**: no program can modify the data of any other program
3. **No DRAM limit**: each program can have more data than DRAM size

4.1.1 Transparency and Protection

The operating system uses **page tables** to keep track of processes. Each process has its own page table, each containing address translational information (i.e. virtual address → physical address).

4.1.2 No DRAM Limitation

We can use disk as temporary space in case memory capacity is exhausted. This temporary space in disk is called **swap partition**.

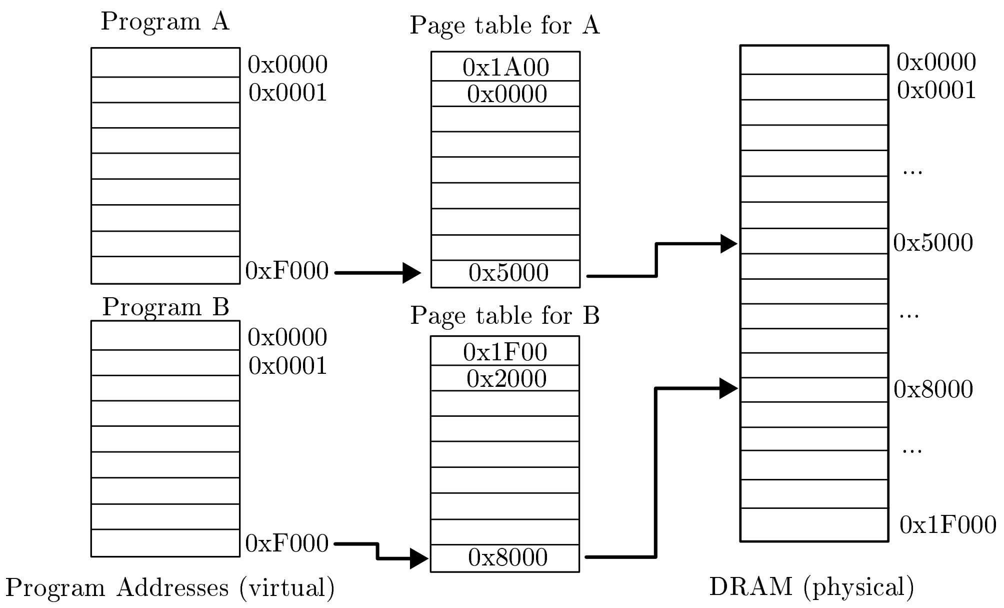


Figure 4.2: Page table basics.

4.2 Pages

Now we aim to develop a method of simulating virtual sets of memory for applications. We can divide memory in chunks of **pages** (e.g. 4 KB for x86).

Definition 4.2.1 (Page). Memory is divided into **pages**. The size of physical page = size of virtual page. A virtual address consists of a virtual page number and a page offset field (low order bits of the address).

Definition 4.2.2 (Page Table). A **page table** contains a map of all the *virtual* addresses and their corresponding *physical* addresses in memory.

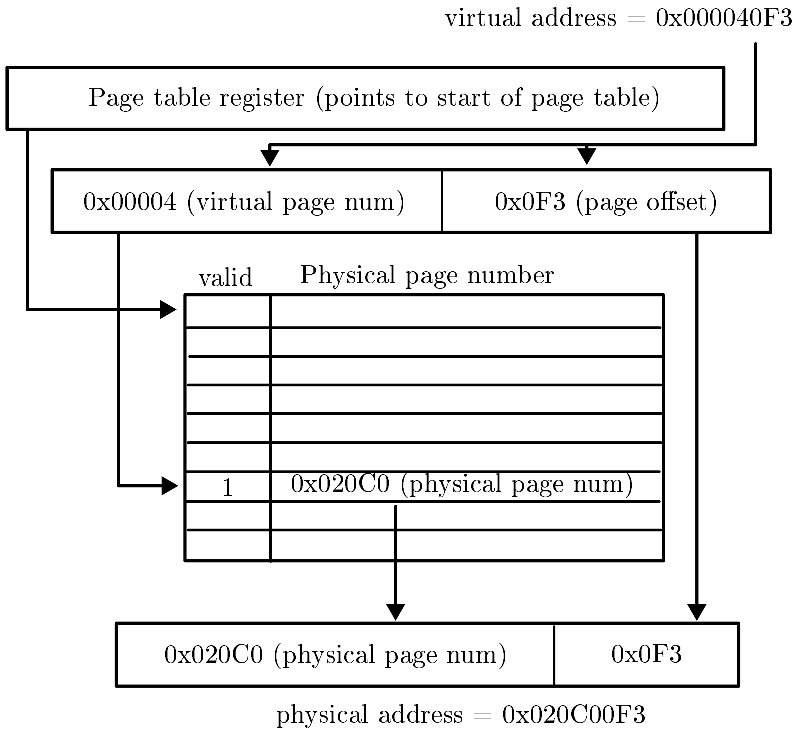

virtual address = 0x000040F3
Page table register (points to start of page table)
0x00004 (virtual page num)
0x0F3 (page offset)
valid
Physical page number
1
0x020C0 (physical page num)
0x020C0 (physical page num)
0x0F3
physical address = 0x020C00F3

Figure 4.3: Page table components.

Definition 4.2.3 (Page Fault). A **page fault** occurs when there is an access to the page table and the data is on the *disk* (i.e. not in DRAM). When this happens, the following occurs:

1. Stop the current process
2. Pick page to replace
3. Write back data if dirty
4. Get referenced page
5. Update page table(s)
6. Reschedule process

Remark. The operating system handles page faults and finds locations on the disk.

Problem 4.2.1. Given the following:

- 4 KB page size
- 16 KB DRAM
- Page Table stored in physical page 0; can't be evicted
- 20 bit, byte-addressable virtual address space
- Page Table has the following initial configuration:
  - Virtual page 0 in physical page 1
  - Virtual page 1 in physical page 2
  - No valid data in other physical pages

Fill in the following table.

Remark. Like caches, we will use LRU when we need to replace a page.

Answer. Given only the virtual addresses and the conditions above, we can derive the rest. We first calculate the number of virtual addresses possible:

$$\begin{aligned} 4\text{ KB } &\to 2^{12}\text{ bytes } \to \text{ 12 bits for offset} \\ \text{20 bit system} &\to 20 - 12 = \text{8 bits for virt. pg num} \\ &\Rightarrow 2^8 \ \textbf{ virtual pages} \end{aligned}$$

Further, we see that there are

$$\text{16 KB DRAM/4 KB pg size} = \ \textbf{4 pages}$$

in physical memory.

Virtual Addr	Virt Page	Page Fault?	Physical Addr
0x00F0C	0x00	n	0x1F0C
0x01F0C	0x01	n	0x2F0C
0x20F0C	0x20	y (into 3)	0x3F0C
0x00100	0x00	n	0x110
0x00200	0x00	n	0x1200
0x30000	0x30	y (2)	0x2000
0x01FFF	0x01	y (3)	0x1FFF
0x00200	0x00	n	0x1200

⊛

Lecture 22: Page Tables Cont.

6 Apr. 12:00

Problem 4.2.2. How big is a page table given the following specs:

- 32-bit virtual address
- 1 GB $= 2^{30}$ bytes of *physical* memory
- 4 KB $= 2^{12}$ bytes per *page*

Answer. First we can calculate the number of bits required for a **physical page number**:

$$\text{30 bits for physical mem } - \text{ 12 bits for page offset } = \text{18 bits}$$

Each entry also has some extra bits (valid, read only, dirty, etc.), so let's round it off and say each entry is 3 bytes (18 bits for PPN, 6 extra bits).
Next, we calculate the number of entries in the page table by noting we have 1 entry per virtual page. We can calculate the number of virtual page numbers (and thus entries in the page table) as

$$\text{32 bits for virtual addr } - \text{ 12 bits for page offset } = \text{20 bits } \to 2^{20}\text{ virtual pages }.$$

Finally, we see that the total size of the page table is

$$\text{num virtual pgs } \cdot \text{ size of each pg table entry } = 2^{20} \cdot 3\text{ bytes } \approx 3\text{ MB}.$$

⊛

The above demonstrates how a single-level page table occupies a rather large amount of continuous space in physical memory. How can we better organize the page table? We have two options:

1. **Flat** (single-level): use a single, massive array (what we've been doing)

- One page table lookup
- Always takes up a lot of memory

2. **Hierarchical** (multi-level): use pointer page that points to child pages spread throughout memory
    - Dynamically adjusts memory usage
    - Typically uses much less memory than single-level
    - Two (or more) page table lookups
    - Used in most modern systems

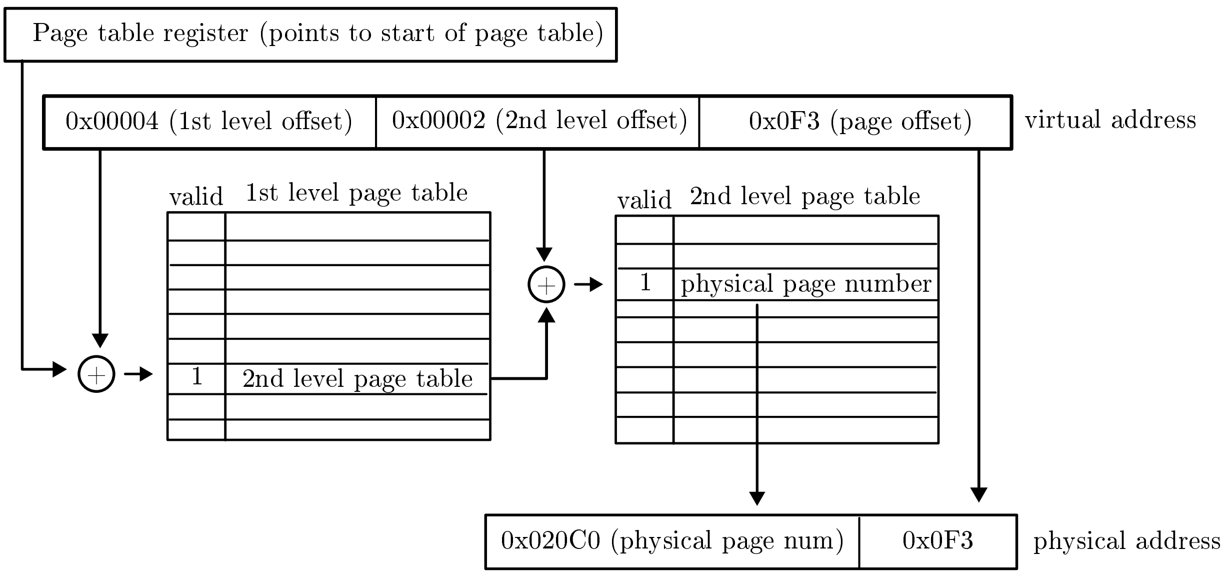


Figure 4.4: Hierarchical page table configuration.

Problem 4.2.3. How much memory is used for the following memory access pattern with a 12 bit page offset?

$$0x00000ABC$$
$$0x00000ABD$$
$$0x10000ABC$$
$$0x20000ABC$$

Answer. Since the leftmost 5 hex digits indicate our 1st and 2nd level page table addresses, the accesses will be as follows:

$0x00000ABC$ // Page fault
$0x00000ABD$
$0x10000ABC$ // Page fault
$0x20000ABC$ // Page fault

So, the memory used can be calculated as:

$$4 \text{ KB for 1st level PT } + 3 \cdot 4 \text{ KB for each 2nd level PT } = 16 \text{ KB}.$$

Problem 4.2.4. Design a two-level virtual memory system of a byte addressable processor with **24-bit long addresses**, no cache in the system, **256 KB memory** installed, and no additional memory can be added. Further, it must have the following specifications:

- **Virtual memory page: 512 bytes**
- Each second-level page table must fit exactly in one memory page, and 1st level page table entries are 3 bytes each

Visually, we have the following:

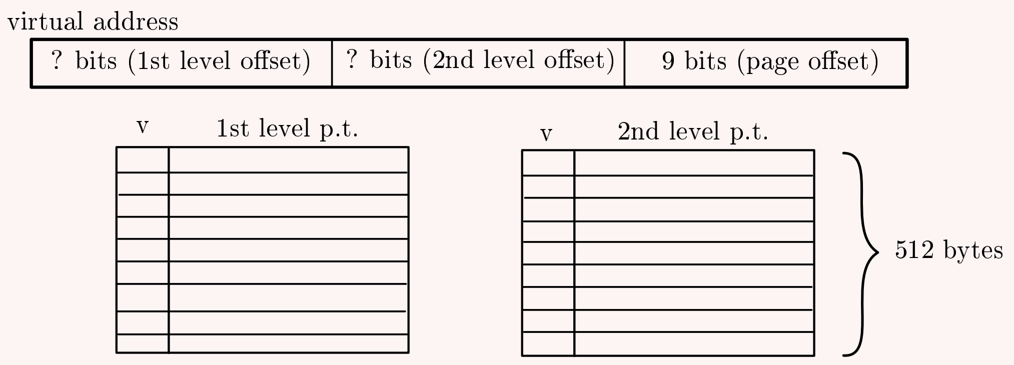


Figure 4.5: The multi-level VM configuration.

Answer. First we can note the **total physical memory size** is 256 KB $= 2^{18}$ bytes. From this, we can calculate the number of **physical pages**:

$$2^{18} \text{ bytes of mem } / 2^{9} \text{ bytes per page } = 2^{9} \text{ physical pages.}$$

2^9 physical pages $\rightarrow$ 9 PPN bits $\rightarrow$ 2 bytes per entry in 2nd level p.t. (rounding to whole byte).

Thus, the number of **entries per 2nd level p.t.** is

$$512 \text{ bytes } / \text{ } 2 \text{ bytes per entry } = 256 \text{ entries }.$$

$$2^8 \text{ entries } \Rightarrow 8 \text{ bits for 2nd level offset}$$

Finally, we can calculate the number of bits required for the **first level offset** is

$$24 \text{ total bits } - 8 \text{ 2nd level offset bits } - 9 \text{ pg offset bits } = 7 \text{ bits.}$$

So the **1st level page table size** is

$$2^7 \cdot 24 \text{ bits per entry } = 2^7 \cdot 3 \text{ bytes } = 384 \text{ bytes }.$$

4.2.1 Other Virtual Memory Translation Functions

The virtual memory must also keep track of other things pertinent to its operation such as LRU, page data location (i.e. is it on physical memory, the disk, or if unitialized). It must also track access permissions (e.g. read only for instructions); this is **how your system detects segmentation faults**.

Lecture 23: Translation Look-aside Buffers

To translate a virtual address into a physical address, we must first access the page table in physical memory. If it's an n-level page table, we must do n total loads before getting the physical page number. Then, we access physical memory again to get the data. This is **very slow**. 11 Apr. 12:00

4.3 Translation Look-Aside Buffer (TLB)

We fix this performance problem by avoiding main memory in the translation from virtual to physical pages. We do this by buffering common translations in a **translation look-aside buffer**, a fast cache memory dedicated to storing a small subset of valid virtual-to-physical page translations.

Definition 4.3.1 (Translation Look-Aside Buffer (TLB)). A fast cache dedicated to storing a small subset of valid page table entries.

Remark. These generally have a very low (<1%) miss rate.

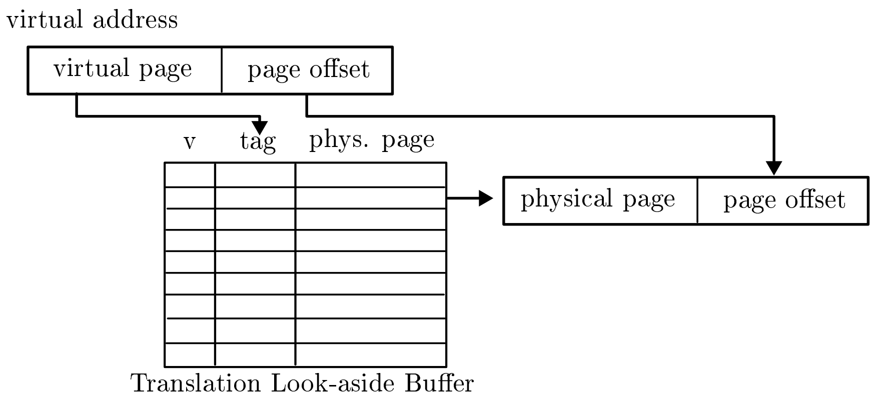


Figure 4.6: Translation look-aside buffer configuration.

We can put the TLB lookup in our processor pipeline **after the virtual address is calculated and before the memory reference is performed**. This can be before or during the data cache access. In the case of TLB miss, we need to perform the virtual to physical address translation during the memory stage of the pipeline.

The overall process for **loading a program** in memory is as follows:

1. Ask OS to create new process
2. Construct a page table for the process
3. Mark all page table entries as invalid with a pointer to the executable file containing the binary
4. Run the program and get an immediate page fault on first instruction

Appendix