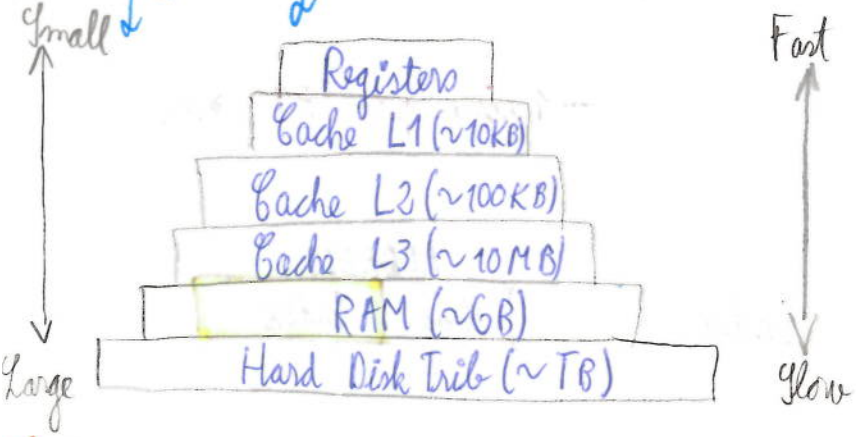
CACHING (fast, but small)

Processor Cache = small amount of very fast memory used as temporary store for frequently used memory locations (both instructions and data)

Role of CPU Cache

- CPU clock frequencies are much faster than memory access time
- CPU would be unable to run at full speed without the addition of the cache

Memory Hierarchy



L2 is { 16x bigger than L1
4x slower than L1
still 10x faster than RAM

L3 is { 2x faster than RAM

Cache → SRAM (needs 6 transistors per bit)

Main Memory → DRAM (needs one)

SSHD (hybrid) = contains a large hard disk drive and and SSD cache to improve performance of frequently accessed data

Caches exploit temporal and spatial locality.

Temporal locality

- memory locations accessed recently are likely to be accessed again within a short time span
- instructions in a loop

Spatial locality

- memory locations located close to recently accessed addresses are likely to be accessed as well
- data in arrays, sequence of instructions

Types of cache

1. Fully Associative Cache (small cache - expensive) - n lines per SET → just (1)
- random access, makes full use of the cache
- cache stores **full address** and **data**
- compare input address with all stored addresses (in parallel)
- if (address found) cache hit
  else cache miss [must go to main memory (slow), cache should be updated]

2. Direct Mapped Cache (based on a hash table) - 1 line per SET → (n)
- fully associative has a lot of expensive comparison operations, so we want to avoid this
- use standard RAM to implement cache functionality
- **address** divided into two parts: **tag** and **index**
- **index** used to address RAM directly
- **tag** is stored and **compared** with incoming tag, if same), data is read
- many addresses map to the same index

(using least significant bit as index exploits principle of spatial locality to minimize displacing recently used data

3. Set Associative Cache (uses a MUX) - way more expensive)
- is simply a small number of direct mapped caches operating in parallel
- replacement strategy more flexible
- hit rate gets better than single DMC
- if 4 DMCs, 4 addresses with the same index but a different tag can be stored

Cache Write Strategy → Cache Hit

- memory writes more complex than reads
- do an address comparison to check if it already exists in cache
- hit, update value in cache, but do not always need to write to memory, long write times

Write Through = every cache write also done to memory (super slow)

Write Through with buffer = buffered, so the write is non-blocking i.e. the processor does not have to wait, but writes may back up

Copy Back (Write) = write is only done to cache (marked as dirty). Only write back to memory when cache entry is replaced

Cache Write Strategy → Cache Miss (fastest strategy => Write Allocate & Copy Back)

Write Allocate = find location, reject if necessary, assign cache location and write value. Write through to RAM or rely on copyback

Write Around (write no-allocate) = just write to RAM, do not write to cache

Valid Bit - indicate meaningful data (it isn't just initialized to 0 or smth.)

→ How to improve "hit rate"

Exploit spatial locality - use a wider cache line (each entry will give more data than just a word). Use the lowest bits to select the word in the cache line. Data is transferred from RAM in bursts equal to the width of the line. Larger line size means less misses, but a line size which is too big means data may not be used or RAM access will take longer

Harvard architecture - separate caches for instructions and for data

Multiple level caches

- big caches are slow and L1 needs to run at processor speed. Put L2 between RAM and L1 → 16 times bigger but 4 x slower than L1. (10x faster than the RAM). L1D (data) and L1I (instructions) both share L2. L3 is large (8MB) its performance is only 2X RAM

Types of cache misses

1. Compulsory misses

- When we first start the computer, the cache is empty, so until the cache is populated, we're going to have a lot of misses

2. Capacity misses (we want to put ALL our elements in the cache)

- since the cache is limited in size, we can't contain **all** of the pages for a program, so some misses will occur

3. Conflict misses - multiple blocks compete for the same set (Not happening in fully associativ

- ~~in a direct mapped or set associative cache, there is competition between memory locations for places in the cache. If the cache was fully associative, then this wouldn't occur~~

Cache performance

- in order to fill a cache from empty, it takes $\frac{\text{cache size}}{\text{line size}}$ (= number of lines) memory accesses
- if we multiply this by the time it takes for a single memory access, then we can work out how long it will take to fill the cache

Cache consistency

- make sure the values stored in the CPU cache are consistent with those in the main memory
there are situations they can **disagree**. (if IO writes / reads directly to RAM)

Solutions

1) Non-cacheable

- make areas of memory that IO can access non-cachable or clear the cache before and after IO takes place

2) IO use data cache

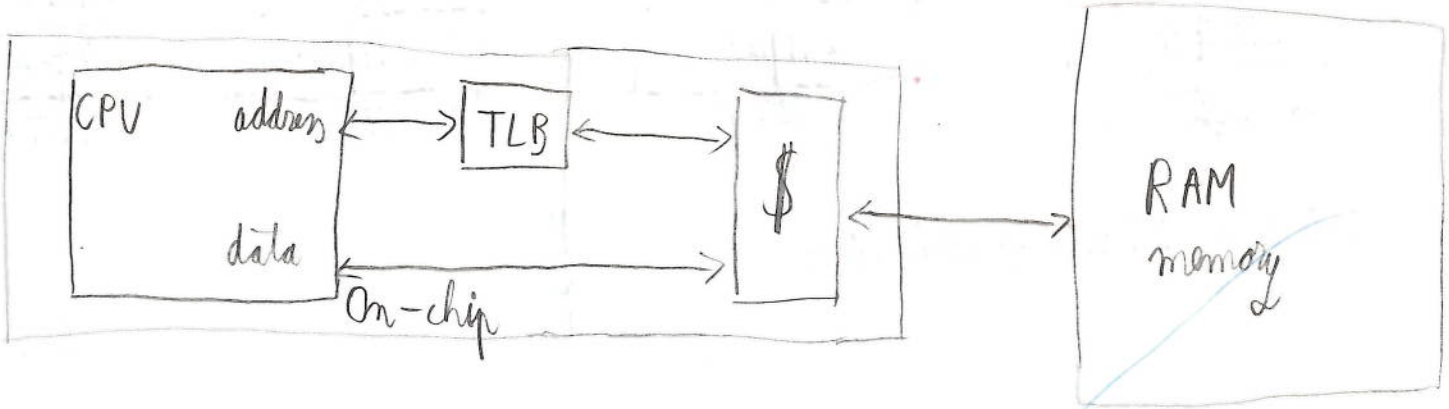
- IO writes first in CPU's L1D (data) cache before accessing the memory
- slows down the cache

3) Snoop on IO activity

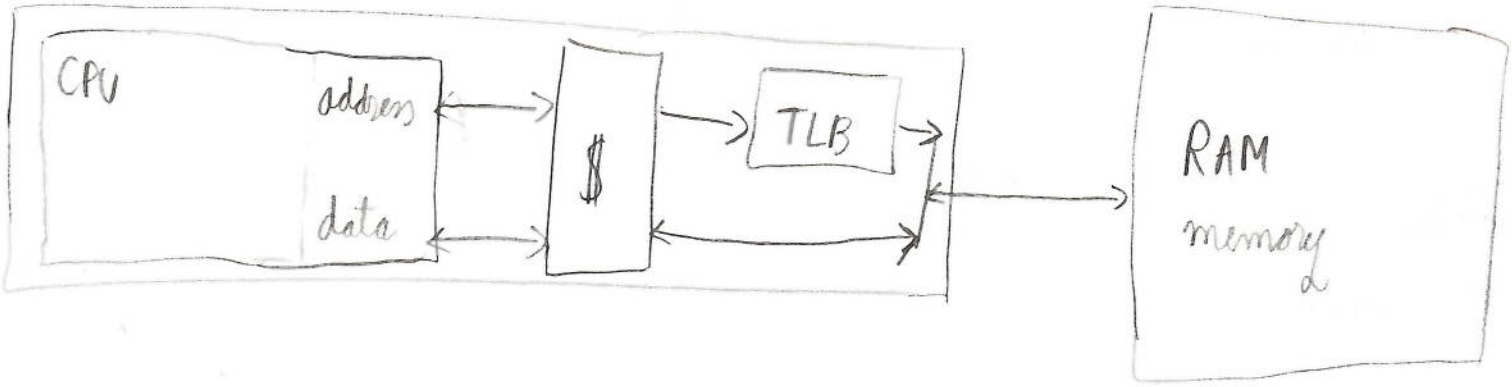
- we could have hardware logic in the cache that will look at the reads and writes to memory from IO and make sure the cache is consistent with memory for those addresses

Virtual Addresses (uses a TLB → Translation - Lookside Buffer)

CPU has **virtual addresses**
Memory has **physical addresses**
} Which address does the cache store? → we have to decide that



- slow, since they must pass through extra logic before hitting the cache



- makes snooping hard to implement along with other functional difficulties

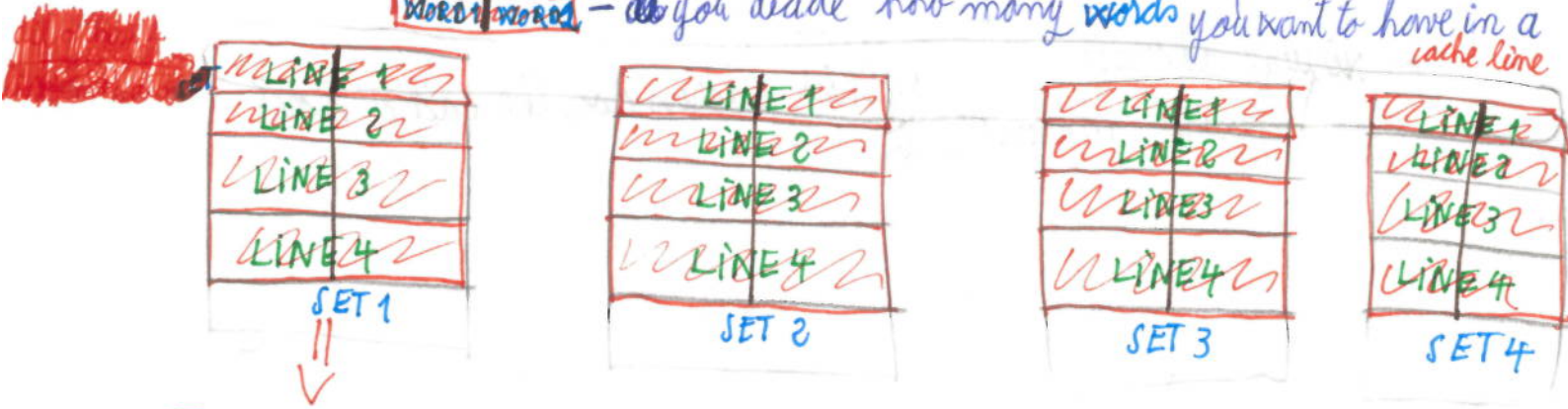
The answer is to have the TLB operate in parallel to the cache, since translation only affects the high order bits of the cache (the low order bits remains the same), and only the tag is changed by address translation.

Example of 4-way set associative (useful for your understanding - 2015 exam)

= 4 • Direct Mapped Caches per 1 SET

[redacted] is a cache line; multiple cache lines form a SET

WORD1 WORD2 - you decide how many words you want to have in a cache line



This is a SET with 4 cache lines

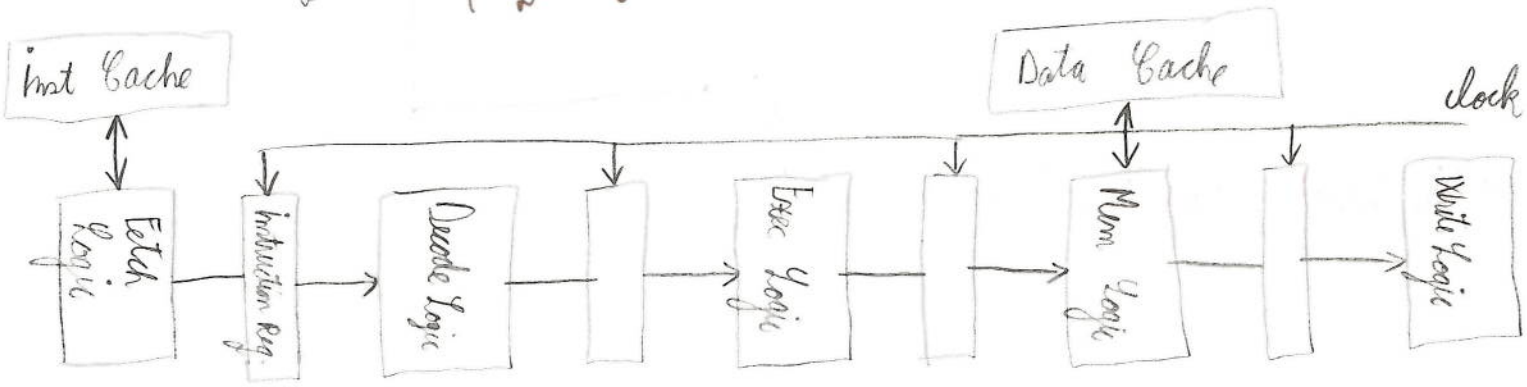
You don't know how many SETS you have. This is just an example. If you know how many sets you have, if you know the cache size and [redacted] set size.

number of SETS = cache size / set size

SET SIZE = number of lines • line size

PIPELINES

Pipelining is a technique in which the instructions are split into different stages in a way that multiple instructions can overlap their execution. It provides a more efficient utilisation of the processor resources while at the same time allows increasing **clock frequency**.



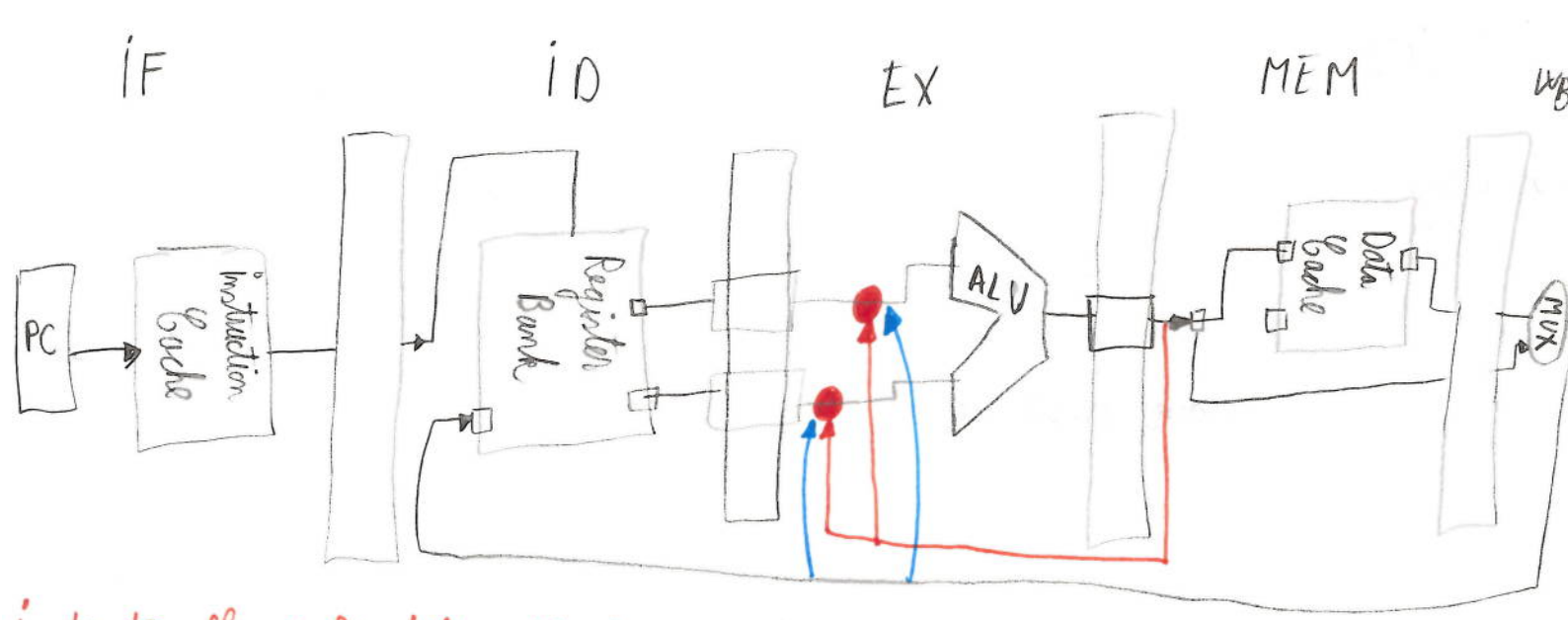
Problems that can arise with the use of pipelining

Data Hazard = an instruction needs to read from a register that is written by a previous instruction before it is stored in the register bank which will require to stall the pipeline until the data is available

Solution: – **forwarding / bypassing** (adding connections from the output of the ALU[stage 3] and Memory Access[stage 4] to the input of the ALU)
– reordering of instructions during compilation

Control Hazard = when executing a branch, the following instruction is not decided until stage 2 if it is unconditional or stage 3 if it is conditional, so the instruction fetched after a branch may not be the one that needs to be executed after the branch instruction

Solution: – stall the pipeline once a branch instruction is decoded
– add NOP instructions after each branch
– use branch prediction



Instruction Level Parallelism (ILP)

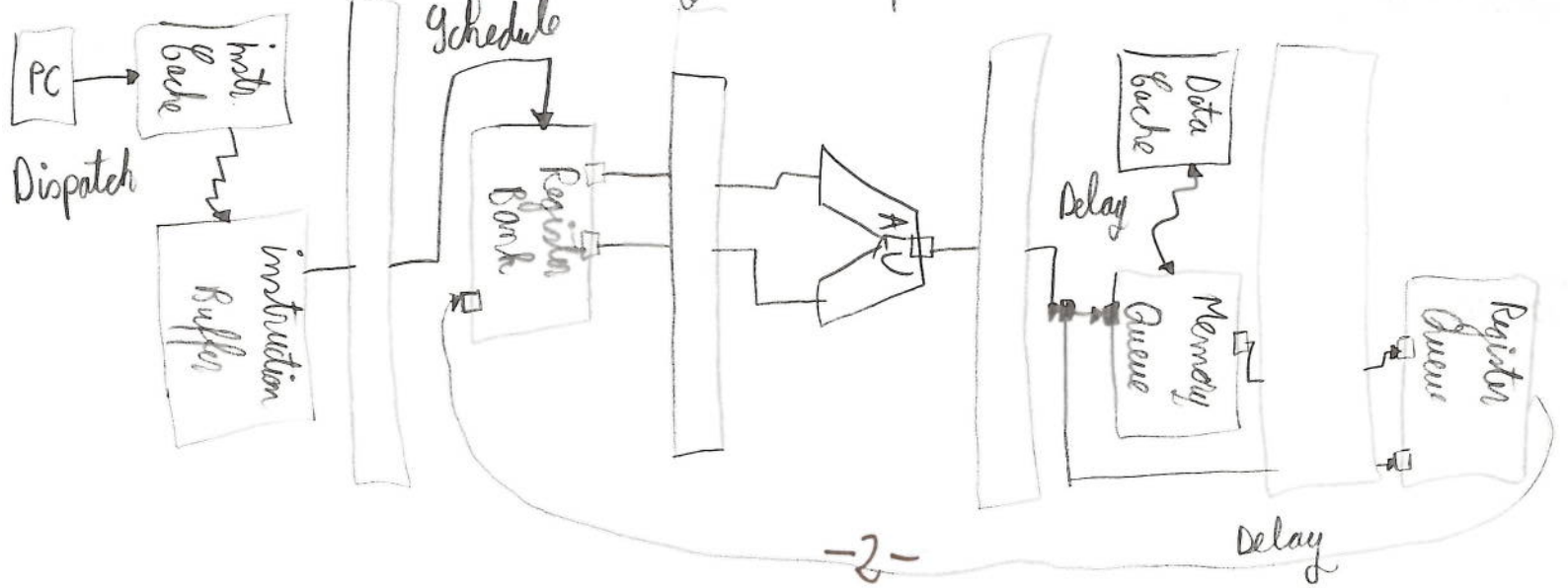
- need to fetch multiple instructions per cycle and to decode multiple instructions per cycle.
- need multiple ALUs

Superscalar

- two instr can execute in parallel, but the access rate to registers and cache will be doubled (need a dual ported register bank and dual ported cache)
- use a 'dispatch unit' in the fetch stage which uses hardware to examine the instruction dependencies; only issue if they are independent

Out of order execution processor

- an instruction buffer needs to be added to store all issued instructions
- a scheduler is in charge of sending non-conflicted instruction to execute



~~VIRTUALIZATION~~ and STORAGE

Permanent Storage:

Three categories:

1) Write Once, Read Many (WORM):
- once you've written, you can't write over it e.g. CD-ROM, DVD-ROM

2) Write Many, Read Many:
- the writes are fully reversible for the purposes of the computer e.g. HDD

3) Write (not too) Many, Read Many:
- devices will slightly wear on each write making them less effective
e.g. Rewritable CD (hundreds to thousands write cycles), flash (thousands to low millions)

Hard drives

- consist of multiple magnetic disks (around 4) laid on top of each other that spin around and can be written/read by a 'head'
- each disk can store around 2TB and rotate at four speeds: 5400RPM, 7200RPM, 10000RPM or 13000RPM

Seek Time = time it takes for the **head** to reach the target **track** on the platter

Search Time = time for the target sector to arrive under the head

Transfer Rate = amount of data that can be read per unit time

Disk Access Time = Seek Time + Search Time + Transfer Time

Example: compute access time of a disk: sector size is 512 B, seek time 8.5 ms, the disk rotates at 7200 RPM and the transfer speed is 177 MB/s

$$\text{Search time} = \frac{0.5 \text{ rotations} \cdot 60}{7200} = 4{,}16 \text{ ms}$$

$$\text{Transfer time} = \frac{512 \text{ B}}{177 \cdot 10^6 \text{ B/s}} = 2{,}89 \mu\text{s}$$

Disk access time = seek time + search time + transfer time

$$= 8{,}5 \text{ ms} + 4{,}16 \text{ ms} + 2{,}89 \cdot 10^{-3} \text{ ms}$$

An internal processor in the hard drive will re-order the operating system's sector requests so that they are in the most efficient order for retrieval

Why are disks slow?

1) High seek time
- multiple platters (=> more tracks / sectors per cylinder) ⟹ the head moves less

2) High search time (rotation speed)
- increase the rotation speed (server disks up to 15000 RPM)

3) Low sustained transfer rate
- "stripe" file system across multiple disks
- apply cache

RAID (Redundant Array of Independent Disks) → a type of storage virtualization

What happens when a disk fails?

RAID 0 Lose all data (hope there's more than one RAID layer)

RAID 1 Business as usual, hot-swap the failed disk

RAID 2-6 Operate in degraded mode

- If a data drive failed, then every read must be reconstructed
- If a parity drive failed, then there is a low performance impact (while the system recomputes the parity bits with a new drive)

SSD vs. HDD

- SSD's are much faster than hard drives since they don't have moving parts, so the data access is much faster
- SSD are made of flash memory. They have a Floating Gate Field Effect Transistor that can store 0's and 1's
- SSD have wear levelling is when logical block addresses are mapped to physical addresses differently over time so that specific blocks aren't worn out

age Virtualization

- RAID allows us to span file systems onto multiple drives by striping and mirroring.

A volume group is a set of drives in a pool and storage space in such a group is divided into physical extents.

A logical volume is made of physical extents.

These abstractions allow us to add more drives, extend partitions, take snapshots of a file system

Storage Area Networks

- implement Logical Volume Management features across multiple servers

ZFS

- combines file system and Logical Volume management

VIRTUALIZATION and STORAGE

Virtualization isolates the details of the hardware from the software that uses it. You can break it down into two broad categories:

Process Virtualization

- run a process under the control layer of software, e.g. the JVM running Java bytecode

System Virtualization

- run an operating system under the control of a layer of software, e.g. VMware

Goals of System Virtualization

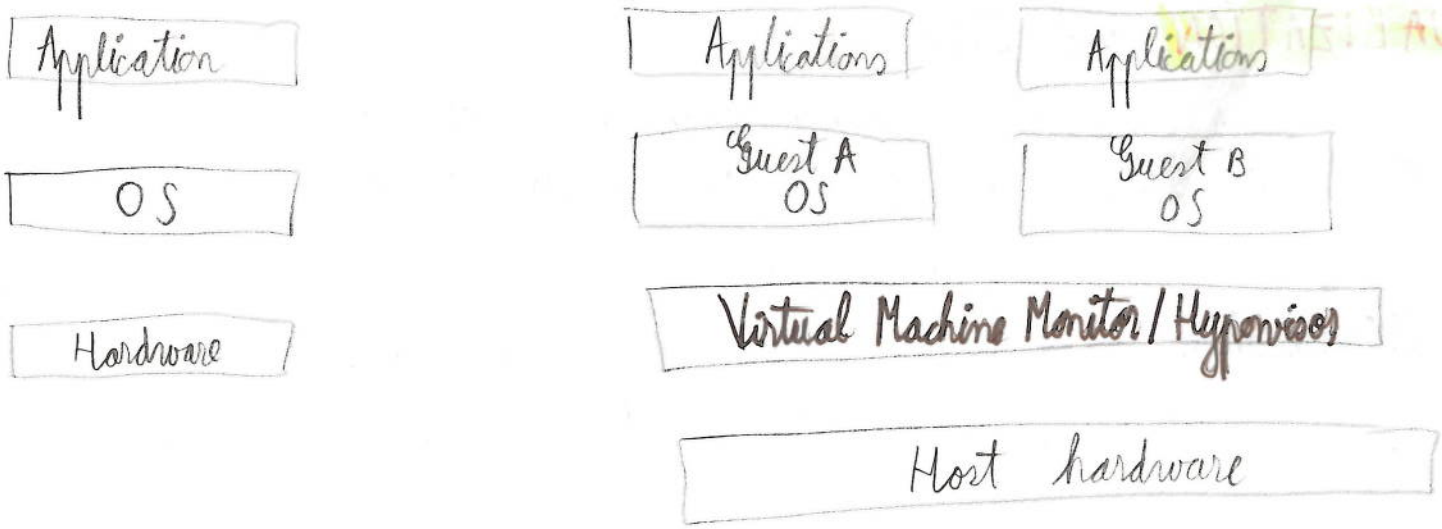
- isolation of the guest OS from the exact details of the hardware (e.g. CPU type, memory configuration, peripheral devices)

What can virtualization do?

- translate between technologies (different instruction sets, system calls etc.)
- change the level of abstraction (providing garbage collection, debugging etc.)
- make the system resources look different (emulate CD drives, reduce RAM amount for a virtual machine)

Reverse Debugging

- when you hit a breakpoint, the debugger lets you step back through the code. This is often implemented by having the VM keep track of what each instruction did and reversing the operation each time you step back.



Unvirtualized

Virtualized

VMM (Virtual Machine Monitor)

- a virtualised OS runs on top of a VMM
- handles physical resources access for the guest OS (unprivileged) since it runs in a privileged mode such as: timers, CPU registers, CPU flags, device control registers (DMA), memory mapping (page table)
- when a guest OS tries to access resources its not allowed to, it will trigger a trap instruction on the VMM, which allows the VMM to check the bounds of access for that instruction/OS and proceed accordingly
- some instructions behave differently according to what mode they are in, so the VMM must be able to handle that

...tions on VMs

At **start**, the VMM saves current registers, loads VM's initial registers and jump...s the VM's PC. At **stop**, the VMM will save its registers into its own memory space (stopped so that CPU can be shared)

Quiescent – in a state of inactivity or dormancy – best time to stop a VM

Freeze – save VM's state into a file. Because of this we can move a VM onto a different machine, snapshot its state, quickly start it

Live migration

= move a VM from one machine to another without **pausing** execution

Phases:

Warm-up phase

- VMM copies all memory pages from source to destination while the VM is still running

Stop-and-copy phase

- VM stopped on source, dirty pages copied, then VM resumed on destinatio...
  - **Downtime** = time between stopping the VM on source and resuming it on destination

Load Balancing

- management software monitors "load" on all physical machines
- if loads are mismatched, migrate a VM from a loaded to a less-loaded machine

High Availability

- for critical applications, keep a standby VM available on a different hardware syst...
- regularly copy active VM image to standby VM (but don't activate it)
- activate standby VM if active VM stops responding (VM crashes? VMM crashes? Hardware system fails?)

Rapid provisioning = deployment of an OS image, including libraries and applications that has been previously configured and saved, perhaps in an archive of virtual machines, onto a hardware system

How can it be implemented using System Virtualization?

By copying the file containing a "frozen" image of the required VM to the host system and resuming it under control of the Virtual Machine Monitor / Hypervisor

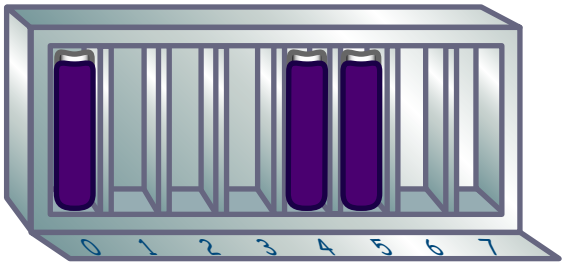
Checkpointing and restoring

Every time the VMM/ Hypervisor pauses a Virtual Machine, enough state is saved to be able to resume the VM later. Checkpointing saves this state in a file, alongside an image of the "physical" memory of the VM and its configuration, for later usage. Restoring this checkpoint is a matter of moving the memory image in the correct place, reloading relevant hypervisor state from that stored earlier, and resuming execution

Cache Associativity

Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

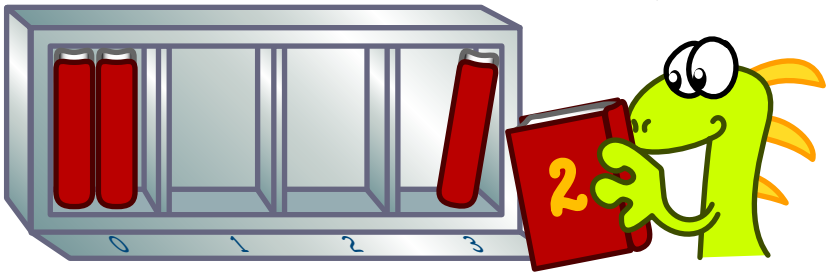
Direct Mapped



Tag	Index	Offset

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

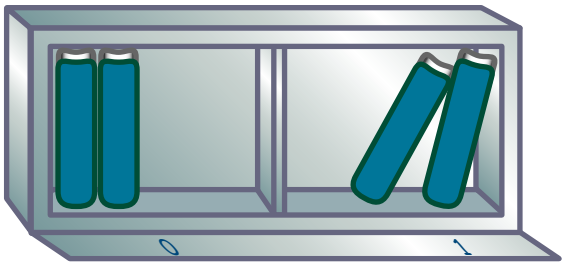
2-Way Set Associative



Tag	Index	Offset

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

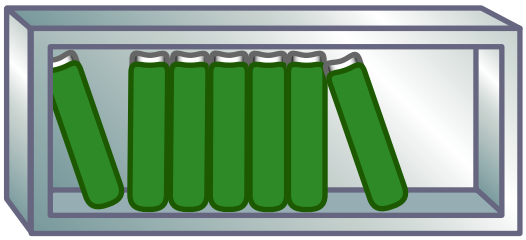
4-Way Set Associative



Tag	Index	Offset

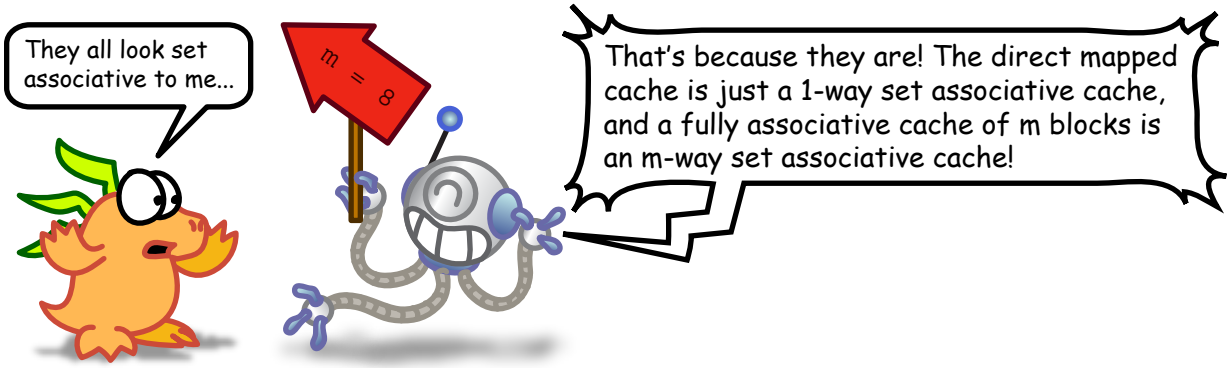
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Fully Associative



Tag	Offset

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!



Katrina Yim

I will assume that you are familiar with what caches are and why they are used. This paper will contain an explanation of the types of organizational schemas of caches.

We begin with a very simple view of the cache: caches are divided into **blocks** and each block may be of a various size. The number of blocks in a cache is usually a power of 2. For now, we will assume that each block of the cache contains only one byte.

Thus, if we were to depict what I have just said, our current cache would look like this:

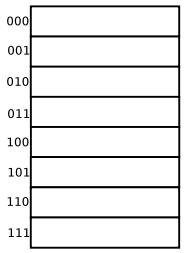


Figure 1: A cache example

As you can see, this cache contains eight blocks and each block can accomodate, as stated, 1 byte (or 8 bits) of information.

The next question is very important: *Where should we put data in the cache?* This question and its answers relates directly to the organizational schemas of the caches.

I will begin with the simplest approach: direct mapping — each main memory address maps to exactly one cache block. The following figure depicts the idea:

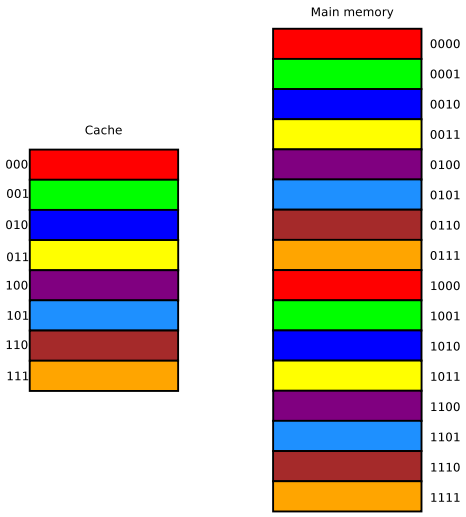


Figure 2: Direct mapping

Here we had a mapping of a 16 byte memory to an 8 byte cache. The direct

mapped cache can store specific lines of memory into the same cache block. For example, the address 'b0000 will always go into the cache block with index 'b000. (the colouring may help you see the designated cache blocks for every main memory line) This organizational schema directly maps a memory line into a specific cache block, hence the name **direct mapping**.

How can we compute this mapping? Well, if you think about it, what we have done is that we have logically split the main memory into chunks, each chunk having the size of the whole cache. (in our example we have divided the 16 byte main memory into 2 chunks of 8 bye — the size of the cache). We usually call this chunks **cache pages** and the size of such a page varies from one organizational schema to another. In the case of direct mapping, the cache page size is equal to the cache size. You can see that the mapping is cyclic: the first 8 bytes are mapped in a unique way, then the 9th one is mapped to the place of the first one, the 10th one is mapped to the place of the second one and so forth. One way to figure out which cache block a particular memory address should go to is to use the mod operator. Thus, if the cache contains 2^k blocks, then the data at memory address i would go to the cache block with index $(i \mod 2^k)$.

More easily, if we want to do this operation directly in the hardware it is enough to take the least significant k bits of a binary value to compute the value $\mod 2^k$. Just look in the example: memory address 'b0000 maps to cache block 'b000 and so does memory address 'b1000. It is all about the least significant bits.

One problem that arrises is that other addresses might also map to the same cache block. As we have seen, that is the case of main memory addresses 'b0000 and 'b1000. How can we distinguish between them? The answer is fairly simple: notice that the lest significant *3* bits are the same in the case of these 2 addresses so we can distinguish between them using the remaining bits: the first one. This is the concept of **tag**. For a cache which has 2^k blocks, each containg 1 byte of data, each main memory address of length m bits will be split in 2 parts: **the index** part which is the least significant k bits of the address which will help us locate the cache block this memory address corresponds to and **the tag** part which is the most significant m-k bits which will help us distinguish between the memory addresses that map to the exact same cache block.

Thus, a cache block entry would look like this:

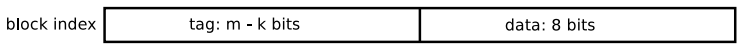


Figure 3: Cache block entry

This is what happens when the processor request an address: it will first look at the cache. The least significant k bits of the address requested are used as an index to find the appropiate cache block and then the most significant m-k bits (**the tag**) are compared to the tag value that is already present in the tag part of the cache block. If these 2 values are equal, we have a **cache hit**, otherwise a **cache miss**.

It is obviously clear that this organizational schema requires some additional information to be kept inside a cache block: the tag. Also, this schema is very inflexible when it comes to cache block replacement policy: by having a single location for a cache line, we do not exploit *time locality* — no matter how much a

cache block has been used, if another memory address is requested that maps to the same cache block, the previous data that was held there will be overwritten.

It is now time to also talk about *space locality*. Think about our initial assumption: that each cache block will store just 1 byte of data. This defeats the purpose of *spatial locality* since we do not copy in the cache values that are next to the value requested — we only copy the value requested. Solution? Again, simple: what we can do is make the cache block retain more than 1 byte. Let's say that the each cache block can retain n bytes. The following figure depicts this:

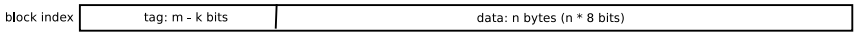


Figure 4: Cache block entry 2

What exactly have we done here? We can now logically split the main memory into chunks, each chunk being of size n *bytes*. Usually, n is a power of two and this size of n *bytes* is called **cache line**. It is time for **block addresses**. Recall that we have split the main memory into **blocks** of size n bytes. How many such blocks are there? The answer is simple: the total size of the memory divided by the size of the cache line. How do we find out the block address of a main memory address i? The answer is again simple: we do the integer division i / n. The following figure depicts an example:

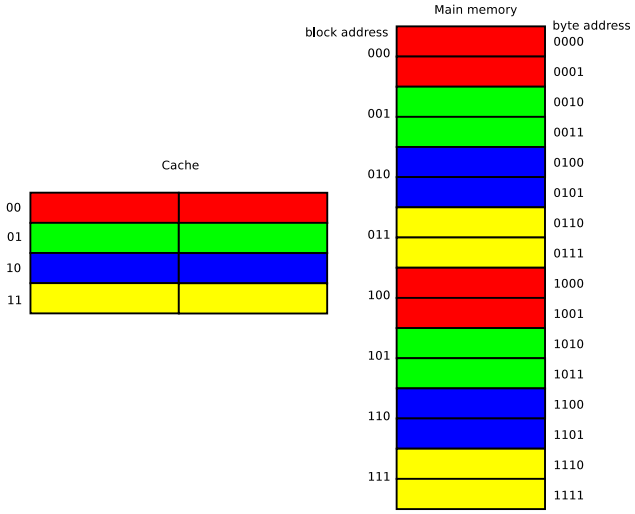


Figure 5: Direct mapping with bigger cache line

Here we have a cache that has 4 blocks and can accomodate 2 bytes in each block. (the tag is left out for simplicity purposes). This means that the main memory can be divided into *blocks* of size 2 bytes each. What is different here is that we do not do direct mapping on byte addresses, but on block addresses. Given a main memory address, for example 'b1101 (that is 13 in decimal) we can find its block address by dividing 13 with 2, which gives us 6 ('b110 in binary). If n (the size of the cache line) is a power of two, let's say 2^b then the integer division would just mean to ignore the first b bits in the address. In our

case n was equal to 2 so b was 1 and you can see clearly that the set of the block addresses are obtained from erasing the trailing bit of the set of the byte addresses. Once we know the block address we can map it as before: by doing the modulo division with the number of cache blocks. (or cache entries if you prefer). Thus, to continue our example, if we have block address 'b110 (6 in decimal) and 4 cache entries, the remainder will be 2. ('b10 — this can be also obtained by taking the least significant 2 bits out of the block address).

So: a short recap — how to arrive from byte address to cache entry index? Given a cache with 2^k entries, each entry consisting of n bytes where n is a power of 2 of the form 2^b, and a main memory address width of m bits, we can obtain the main memory block address by taking the most significant m - b bits from the original address, and then we can find the index of the cache entry by taking the least significant k bits of of these m - b.

Thus, when we access one byte of data in memory, we will copy its entire block(cache line) into the cache, to hopefully take advantage of spatial locality. But now we store more than one byte in each cache line! How can we refer to a single byte (the one requested)? Remember that the cache line size is n or 2^b. We can use b bits to address each individual byte in the cache line. Luckily for us, those b are the lease significant bits of the original address. We call this the **offset** and we use it to locate the requested byte inside the cache line. To make things simpler, byte i of a memory block is always stored in byte i of the corresponding cache line. We should not forget about the **tag** as well. Thus, when an address arrives at the cache controller it is split as follows:

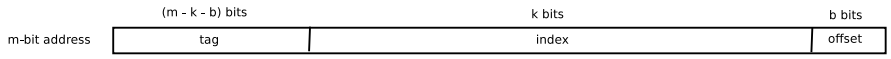


Figure 6: Logical split of a memory address

Remember that 2^b is the size (in bytes) of the cache line, 2^k is the number of cache entries and m is the size in bits of the memory address.

Now: what I have just explained is not exactly accurate. We can not address individual bytes in a machine. We can only address words: and the word size may vary. If, for example a words contains 4 bytes and a cache line contains 64 bytes (which, by the way, is usual nowadays), then you do not need 6 bits to address the words. You only need 4. Thus, the trailing 2 bits of the address are unused. Very sad for them. This is the final figure which depicts accurately how an address is logically split in direct mapping.

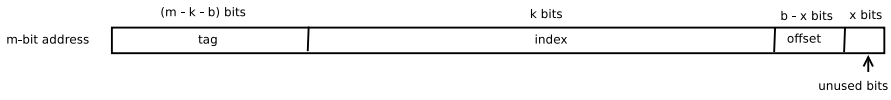


Figure 7: Logical split of a memory address 2

Here, the meanings are just the same, but the new introduced variable x has the property that 2^x = *sizeof(word)*.

We have finally solved the problem of **spatial locality**! What about the **temporal locality**? What happens if a program uses addresses (in decimal) 2, 6, 2, 6, ...? We would constantly need to rewrite the cache because there is only

one location in the cache that corresponds to each address. We need a solution to address the **temporal locality** as well.

Another proposed organizational schema is a type of cache called**fully associative**. Here, we permit the data block to be placed in any available cache block instead of forcing the data to be placed at a specific cache block, thus we remove the index.

But what are the prices of full associativity?

- Because there is no index in the address any more, the entire address must be used as the tag, thus we require more space for the cache.
- A piece of data could be anywhere in the cache, so we must check every cache block. To do this in parallel (because if we do it sequentally would defeat the purpose of caching since it would take a lot of time) we would need a lot of comparators, thus the complexity of the hardware would be enourmous for large caches with many cache blocks.

There is also, an intermediate possibility: **set associative** cache. This types of cache combines the fully associative schema with the direct mapped schema. The idea is quite simple: the cache is divided into groups of blocks, called **sets** and each memory address maps to exactly one set in the cache, but data may be placed in any block within that set. If each set has 2^x blocks, then we call this type of cache **2^x-way associative cache**.

How do we determine where a memory address belongs in an associative cache? The answer is simple: in a similar way. Take, for example, this figure:

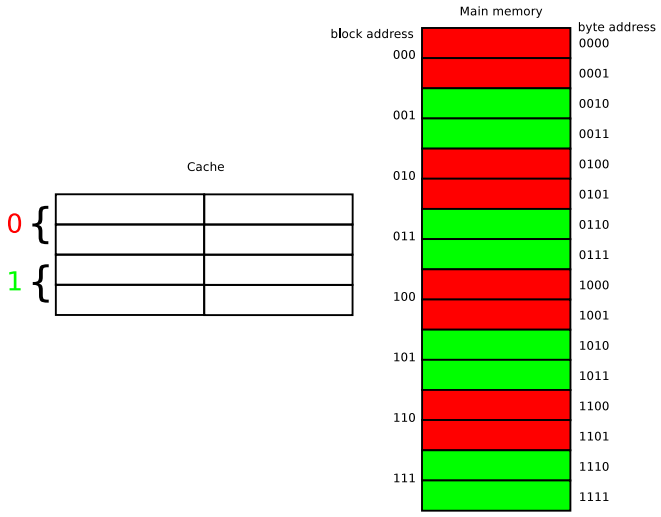


Figure 8: 2-way set associative cache

Here we have a cache that has 4 blocks, each block containing 2 bytes. These 4 blocks are divided into 2 sets, each set containing 2 blocks. Again, because the cache line size is 2 bytes, the main memory is divided into chunks of 2 bytes. Coressponding to these chunks, we have 8 conceptual blocks in the main memory. Each of these 8 blocks are directly mapped to one of the two sets, but the data(the conceptual blocks of the main memory — recall that we copy in

the cahce the whole block, not individual bytes) can be placed anywhere in the 2 blocks of each set. For example, main memory address 'b0001 has block address 'b000. This block directly maps to set 0, but block 'b000 can be placed either in the first block of the cache or the second block of the cache. To deal with this, we need to have 2 comparators. Similarly, to deal with 2^x*-way associativity* we need 2^x comparators.

Our arithmetic computations now compute a *set index* — we are mapping to sets, not directly to cache blocks. Thus, we have:

- *block address = memory address / size of cache line*
- *set index = block address mod / number of sets*
- *block offset(or cache line offset) = memory address / size of cache line*

Finally, a main memory address can be partinioned as follows:

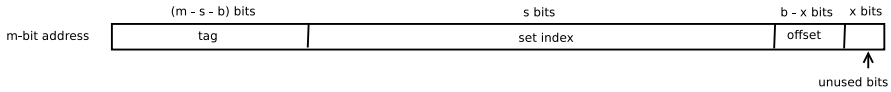


Figure 9: Logical split of a memory address 3

Remember: m is the number of bits of the main memory address, 2^s is the number of sets (2^s is equal to *size of cache in bytes / size of cache line in bytes / number of ways of associativity*), 2^x is the size of a word in bytes and 2^b is the size of the cache line.

To illustrate this, I will give you an example: my machine has a L1 cache of 32KB, 8-way associative and a cache line size of 64 bytes. The number of sets must be, therefore:

$$\frac{32*2^{10}bytes}{64bytes}\frac{1}{8ways} = 64sets$$

This is accurate, as it can be found from */sys/devices/system/cpu* (works only on Linux).