System Architecture

Todd Davies

May 25, 2015

Introduction

The basic architecture of computer systems has been covered in first year course units which detailed both the instruction set architecture and the micro- architecture (hardware structure) of simple processors. Although these principle underlie the vast majority of modern computers, there are a wide range of both hardware and software techniques which are employed to increase the performance, reliability and flexibility of systems.

Aims

The aims of this course are to introduce the most important system architecture approaches. To give a wider understanding of how real systems operate and, from that understanding, the ability to optimise their use.

The syllabus includes:

- The motivation behind advanced architectural techniques.
- Caching
- The need to overcome latency. Caching as a principle, examples of caching in practice. Processor cache structure and operation.
- Pipelining
- Principles of pipelining. Implementation of a processor pipeline and its properties. Pipelining requirements and limitations. Additional support for pipelining.
- Multi-Threading
- Basic multi-threading principles. Processor support for multi-threading. Simultaneous multi-threading.
- Multi-Core
- Motivation for multi-core. Possible multi-core structures. Cache coherence.
- File System Support
- Implementation of file systems. RAID
- Virtual Machines
- Motivation for Virtual Machines. Language Virtual Machines. System Virtual Machines. Virtual Machine implementation. Binary Translation

Contents

1 Introduction

Performance is always an attribute in high demand in computer systems. Even though processors have become much more powerful over the last half century, there's still loads of stuff that we cannot do with current technology, such as synthesising HD video in realtime, or computing realistic game physics.

Since 2004/5, companies haven't been able to increase the speed of microprocessors at such a rapid rate due to physical limits, such as power dissipation and device variability. Our devices are still getting faster, but now architecture and the design of systems play a larger role in making stuff run faster. An example of this include making computation more parallel.

2 Caches

Not all technology has improved at the same relative speed. CPU's have become over three orders of magnitude faster over the past thirty years, but memory has increased by only one order of magnitude. This is problematic, since it means that we need to reconcile this gap in order to achieve efficient computation.

Processor caching is used to let the processor do useful computation while it's also waiting on the memory. Modern processors couldn't perform anywhere near how fast they do now without equally modern caching techniques, since the imbalance between the CPU and main memory is so high.

Caches (in general) provide a limited, but very fast local space for the CPU to use. They are used in lots of places all over computer science, including web browsers, mobile phone UI's etc. Likewise, a processor cache is a temporary store for frequently used memory locations.

The principle of locality is what makes caches work for processors, which is that the CPU will only use a small subset of memory over a short period of time. If this subset of memory can be loaded into the cache, then the computation can be sped up significantly.

Every 'cache miss' takes *at least* sixty times longer to execute than a 'cache hit' will (that's assuming there are no page faults etc). Circuit capacitance is the thing that makes electronic devices slow, and larger components have a larger capacitance, henceforth large memories are slow. Dynamic memories (DRAM) store data using capacitance, and are therefore slower than static memories (SRAM) that work using bistable circuits.

Even the wires between the processor and the memory have a significant capacitance. Driving signals between chips needs specialised high power interface circuits. An ideal situation would be to have everything on a single chip, however current manufacturing limitations prevent this; maybe one day we will be able to do this.

2.1 Why are caches expensive?

L1, L2 and (usually) L3 caches are SRAM instead of DRAM (which is what main memory is made from).

SRAM needs six transistors per bit, DRAM needs one.

SRAM is henceforth physically larger, taking up more space on the chip, which is expensive, since real estate costs money.

2.2 L1 Cache

The L1 cache is the first level of caching between the processor and the main memory. The L1 cache is around 32kb, which is very small in comparison to the size of the main memory, but this is driven out of necessity, since the cache needs to be small to be fast. The cache must be able to hold any arbitrary location in the main memory (since we don't know in advance what the CPU will want), and henceforth requires specialised structures to implement this.

2.3 Types of cache

The CPU will give the cache a full address of a word in memory that it wants to read. The cache will contain a small selection of values from memory that it has locally, but will ask the main memory for values that it does not have. This is called a *cache miss* and is expensive in comparison to a cache hit.

2.3.1 Fully associative

A **Fully Associative** cache is one where the cache is small (around 32,000 values), but stores both addresses and their corresponding data. The hardware compares the input address with all of the stored addresses (it does this in parallel). If the address is found, then the value is returned with no need to ask the RAM (cache hit), if the value isn't found, then a cache miss occurs, and the request must go to the main memory.

Caches rely on locality in order to function effectively. There are two types of locality; temporal locality, which is the principle that if you use an address once, you may use it again soon (e.g. loops), and spatial locality, where if you use an address once, you are also likely to use addresses nearby (e.g. arrays).

Spatial locality is exploited better by having a bigger data area in the cache (returning say 512 bits for every address instead of just one word)

The cache hit rate is the ratio of cache hits to misses. We usually need a hit rate of around 98% to hide the speed of memory from the CPU. Instruction hit rates are usually better than data hit rates (although they are in the same cache remember). The reason for this is that instructions are accessed in a more regular pattern, incrementing by one word every time, or looping around etc (have higher locality).

When we do a cache miss (on a read), we should add the missed value to the cache. In order to do this, we need a cache replacement policy to find room in the cache to put the new value. Common cache replacement algorithms include:

See my `COMP25112` notes for in depth stuff about LRU, Round Robin and more cache replacement algorithms (though they'll be under *page* replacement algorithms or scheduling algorithms there).

Last Recently Used (LRU) - slow, good for hit rates

Round Robin - not as good, easier to implement

Random - Easy to implement, works better than expected.

Memory writes are more complicated than reads. If we've already got the value in the cache, then we change the value in the cache. We can use three write strategies for cache writes (on hits) to ensure that the changes are propagated to memory:

- Write through (slow)
- Write through + buffer (faster, slow when heavily used)
- Copy back on cache replacement.

If however, we have a miss, then:

- We can find a location in the cache and write to that, then rely on copy back later, or write back straight away.
- We can skip the cache, and write directly to RAM. Subsequent reads to this memory location will put it in the cache again (this is good if you're initialising datastructures with zeroes).

The fastest strategy of these two is write allocate or copy back (the first one). However, the main memory and the cache aren't coherent, which can be a problem for stuff like multiprocessors, autonomous IO devices etc. This can cause problems, which we will deal with later.

A cache is coherent with memory if the values stored in the cache are the same as those in memory. This is always desirable (since otherwise you could have 'old' data somewhere in the system), but sometimes hard to achieve.

Each cache line is at least half address and half data, but often, we store more data per address, so will have 64 bytes of data per 32 bit address.

A fully associative cache is ideal, but this is expensive (in terms of silicon and power).

2.3.2 Directly mapped

We can use standard RAM to create a directly mapped cache, which mimics the functionality of an ideal cache. Usually, this uses static RAM, which is more expensive than dynamic RAM, but is faster.

The address is divided into two parts, the tag and the index. The tag is the higher order bits of the address and the index is the lower order bits. The index is used to address the slot in the cache, and is made of the number of bits required to do this (so 15 bits for a 32kbit cache for example). The tag is used to check that the data stored for this index value is the correct address, this is probably described better by looking at Figure 1.

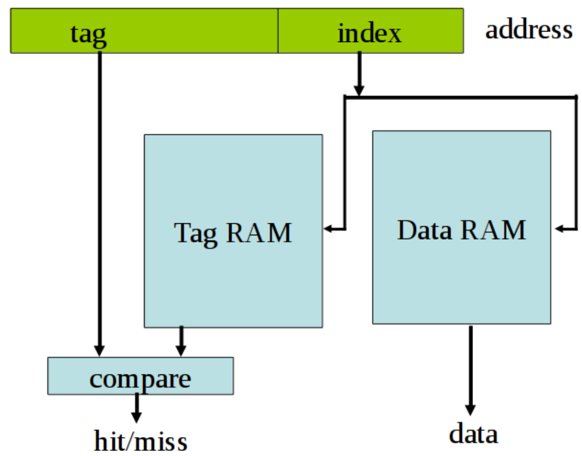


Figure 1: A direct mapped cache has one slot for multiple different memory addresses, the index determines what slot the value goes in, and the tag checks that the correct value is being read.

A direct mapped cache is really just a hash table implemented in hardware, but the collision policy is eviction (so if two addresses have the same index, then the one currently in the cache will be evicted to make room for the new one).

This is a very cheap replacement policy (since no algorithm is implemented to decide what cache line to evict next), and it exploits spatial locality to minimise displacing recently used data. However, it does have a lower hit rate than more complex algorithms since the replacement strategy is so inflexible.

2.3.3 Set associative

Set associative caches are a compromise. They comprise of a number of directly mapped caches operating in parallel. If one matches, we have a hit and select the appropriate data. This is good because we can have more flexible cache replacement strategies. In a 4 way set associative cache, we could choose any one of the four caches for example. The hit rate of set associative caches improves with the number of caches, but increasing the number increase the cost.

3 Practical caches

3.1 Cache control bits

When the system is started, the cache is empty. We need a bit for each cache entry to indicate that the data is meaningful (i.e. it isn't just an uninitialised zero or something), which we do by having a valid bit. We also need a dirty bit if we're using the 'write back' caching strategy (see above), rather than the 'write through' strategy.

3.2 Exploiting spatial locality

In order to exploit spatial locality, we need to have a wider 'cache line', where each entry will give you more data than just one word. Each entry tag could correspond to two, four, eight etc words. The principle of spatial locality says that if we access memory at one location, we'll probably want to access nearby locations in the near future too.

The 16/32 byte figure might be outdated now!

The lowest bits are used to select the word in the cache line. Most cache lines are 16 or 32 bytes, which is 4 or 8 32bit words. The data is transferred from RAM in bursts equal to the width of the line size, using specialised memory access modes.

The line size is important, since we want to have a line size of multiple words to exploit spatial locality, but if the line is too big, then parts of it will never be used. The number of cache misses decreases as you increase the cache line size, until one point, where the line size will be too long to use all the words, and then the number of misses will increase.

3.3 Separate instruction and data caches

Studies have shown that programs typically do one access to memory for data per three access for an instruction. The data accesses are usually (except in the rare case of self modifying programs) in separate address ranges, and therefore these different access patterns can be exploited by having separate instruction and data caches. This is called a Harvard architecture.

To cater for this, the L1 CPU cache is usually split up into the L1I cache for instructions and the L1D cache for data.

3.4 Multi level caches

As chips get bigger, in theory, we should build bigger caches to perform better. However, big caches are slow, and the L1 cache needs to run at processor speed. We can instead put another cache between the RAM and the L1 cache, and keep the L1 cache the same size.

The L2 cache is typically around sixteen times bigger than the L1 cache, but also four times slower. It's still ten times faster than RAM though. The L1D and L1I caches both share the L2 cache.

If a chip has an L3 cache, then it is usually quite large (maybe around 8Mb), but its performance is only about twice as good as that of RAM.

3.5 Cache misses

There are three types of cache misses, called the three C's

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.

Capacity misses
: Since the cache is limited in size, we can't contain all of the pages for a program, so some misses will occur.

Conflict misses
: 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 misses due to this wouldn't occur.

We can calculate the impact of cache misses, predicting on average, how long the CPU will wait for a reply from memory. If the L1 cache has a hit rate of 98% and takes one cycle to return a value, the L2 cache has a hit rate of 90% and takes four cycles to return, the L3 cache has a hit rate of 70% and takes ten cycles, while the main memory takes 100 cycles, what is the average access time?

First, we can calculate how many accesses (as a percentage) each level will get:

L1	98%
L2	$((100 - 98) * 0.9) = 1.8\%$
L3	$((2 - 1.8) * 0.7) = 0.14\%$
Main Memory	$((0.2 - 0.14) * 1) = 0.06\%$

We can then use those as decimals to calculate how many clock cycles the CPU will have to wait on average:

$$
\begin{aligned}
\text{Average} &= 0.98(1) + 0.018(1+4) + 0.0014(1+4+10) + 0.0006(1+4+10+100) \\
&= 0.98 + 0.09 + 0.021 + 0.069 \\
&= 1.16
\end{aligned}
$$

3.6 More cache performance

In order to fill a cache from empty, it takes $\frac{\text{Cache size}}{\text{Line size}}$ memory accesses. If we multiply this by the time it takes for a single memory access (say 10µs), then we can work out how long it will take to fill the cache (assuming each access is to a unique memory address). We can derive how many CPU cycles this takes from this.

3.7 Cache consistency

We need to make sure that the values stored in the CPU cache are consistent with those in main memory. There are situations when they can disagree, for example if IO reads or writes directly to memory (perhaps using DMA), then that value could be different from whatever is in the cache.

There are a number of solutions to this:

Non-cacheable
: One solution is to make areas of memory that IO can access non-cachable, or clear the cache before and after the IO takes place.

IO use data cache
: Another is to have the IO go directly through the CPU's L1D (data) cache before accessing memory, but this tends to slow down the cache (which is obviously bad for the speed of the system, but we'll look why in Section 4).

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. This will be covered more in Section 6.2.1.

3.8 Virtual Addresses

Since the CPU deals with virtual addresses when accessing memory, and uses a Translation Lookaside Buffer to derive the correct physical address. However, which address does the cache store? Does it sit before the TLB, or after it between the CPU and memory?

If we make addresses go through the TLB before they reach the cache, then this is slow, since they must pass through extra logic etc before hitting the cache. However, if we make the cache store virtual addresses, and have the TLB sit inbetween the cache and memory, this makes snooping hard to implement along with other functional difficulties. This is illustrated in Figure 2

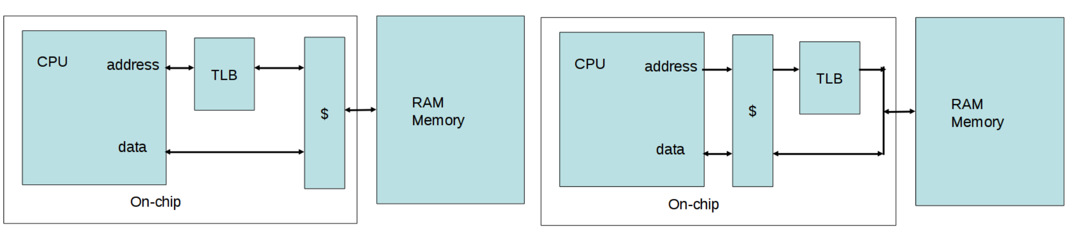


Figure 2: Two possible placements of the Translation Lookaside Buffer

The answer is to have the TLB operate in parallel to the cache. Since address translation only affects the high order bits of the cache (the low order bits are the offset which remains the same). The cache index is selected from the low order offset bits, and only the tag is changed by address translation. This is shown in Figure 3.

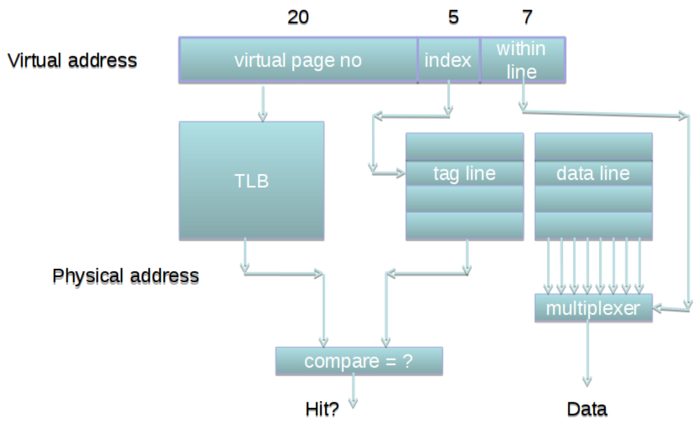


Figure 3: The TLB operating in parallel with the CPU cache

4 Pipelines

The fetch execution cycle is very repetitive, if we can optimise it, then we can potentially improve the performance of the system a lot! In very simple systems, each fetch and execute would take one clock cycle; in detail, it'd look like this:

Fetch

- **IF** - Fetch instruction from memory
- **ID** - Decode instruction; select registers

Execute

- **EX** - Perform an operation or calculate an address
- **MEM** - Access an operand in memory
- **WB** - Write to registers

If all that takes one clock cycle, each stage will only be active for about $\frac{1}{5}$ of a clock cycle, or in other words, each CPU component spends 80% of its time doing nothing!

If we can get all of those components of the CPU working at the same time, then we can speed up the clock speed by five times! In order to do this, we can use a pipeline, with buffers that are flushed every clock cycle inbetween each stage of the pipeline, as shown in Figure 4.

Although we have divided our processor into five stages, you can split it up into more or less than that (the most simple of which is just to divide into fetch and execute as we did in `COMP12111` in the first year. Modern processors use a lot of stages, maybe around thirty.)

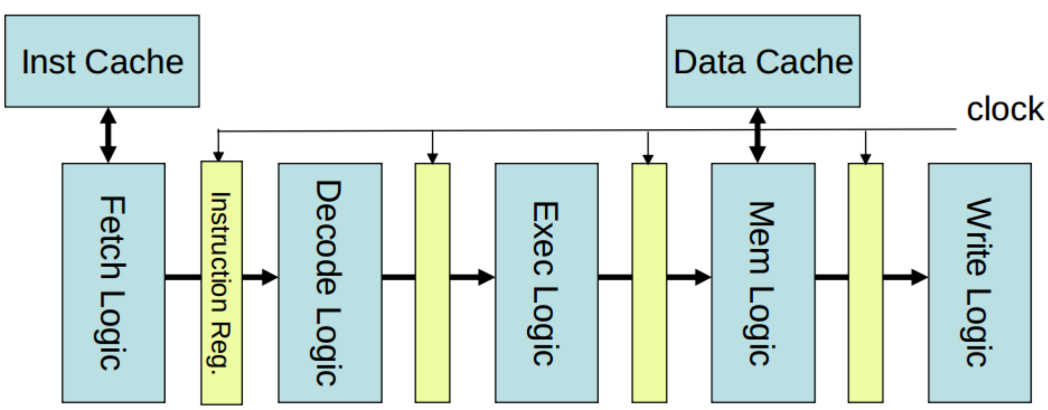


Figure 4: Using buffers between the different stages of the pipeline

Now, we can still only execute one instruction per cycle, but we can also increase our clock speed by five times, since each stage of the pipeline is isolated by a buffer from the next, and only does $\frac{1}{5}$ of the work of a ‘normal’ CPU cycle.

4.1 Control Transfer Problem

Using a pipeline is fab if your instructions occur solely in a serial manner, however, what if your program branches? The processor will only know that a branch is happening at the `ID` stage of the pipeline, by which time we’ve already fetched the next instruction!

If we come across a branch at the `ID` stage, then the fetched instruction at the `IF` stage will have to be ignored all the way down the pipeline, so we would waste one clock cycle (or more specifically, we would waste $\frac{1}{5}$ of the work of five clock cycles!). The ignored instruction is said to be a *bubble* in the pipeline.

So far, we’ve assumed that our branch instruction has relied on no conditional flags. If it did however, we would need to wait until the `EX` stage of the pipeline before we knew what the outcome of the conditional evaluation would be. This has the potential to create two bubbles, since if the branch was to occur, the instructions at both the `IF` and `ID` stages of the pipeline would have to be ignored.

Other control hazards are possible, it’s not just branching that’s the issue. Conditional instructions (e.g. `MOVGE`) can also be cause control hazards.

These bubbles are called **control hazards**, and they occur when it takes one or more pipeline stages to detect a branch. Longer pipelines are more likely to suffer from control hazards more, since more of their pipeline will have been processed by the time an instruction is detected to be a branch.

4.1.1 Branch Prediction

The main technique used to mitigate control hazards is **branch prediction**. If we can remember what address a branch directed us to fetch next from what it did when we executed that branch previously, then we could pre-emptively load that instruction in the `IF` stage instead of fetching the instruction at the PC.

In order to do this, we use a **branch target buffer**. This maps the virtual address of one branch instruction onto the virtual address of the instruction that is branched to, for example:

Branch instruction address	Next instruction address
`0xd4f30d2C`	`0xd4f30d60`
`0xd4f30d0f`	`0xd4f30ddd`
⋮	⋮
`0xd4f30c4f`	`0xd4f30d6c`

Table 1: An abstraction of what the datastructure inside a Branch Target Buffer could be like

Now, we will always get unconditional branches right (after we’ve done it for the first time), and we will get branches that are part of loops (such as a `for` or a `while` loop) right most of the time. If we predict a branch incorrectly, we just get a bubble like we would if we had no branch prediction, since the steps after the incorrectly predicted branch will just be aborted. For efficiency, branch prediction is a win-win situation.

Branch prediction is easy to understand, but implementing it is expensive. In practice branch predictors use the history of each branch (maybe taking the mode of the last five branches), and the context of the branch (i.e. how did we get to this point) in order to make a more accurate prediction.

4.2 Data Hazards

If we’re using a pipeline to process instructions, then we have multiple instructions at various stages of execution at the same time. This can cause problems, if multiple instructions that are being executed in parallel operate on the same resources. For example, if we execute the ARM Assembly code:

```
ADD   R1, R2, R3
MUL   R0, R1, R1
```

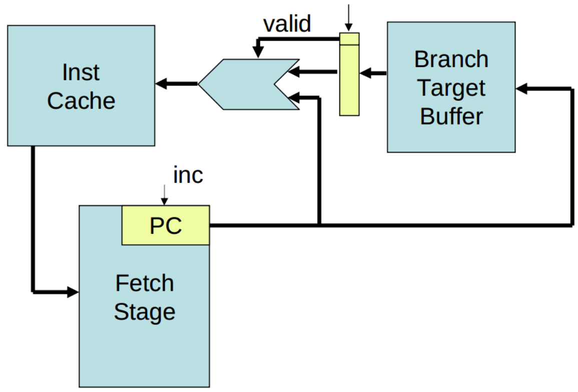


Figure 5: A BTB (Branch Target Buffer) in action

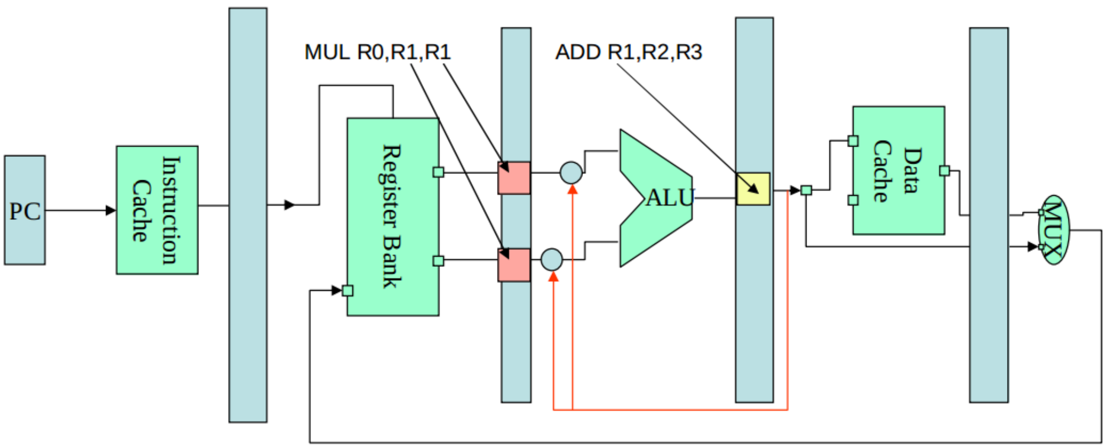


Figure 6: Here, we insert extra paths into the processor so that it can mitigate the `ADD`/`MUL` dependency.

This problem is called a 'data not ready' issue.

It might take two cycles for a value to be written into a register, one for `EX` to complete and push the value into the register bank, and another for the value to be written into the register.

Here, the value in `R0` depends on that of `R1`. This is a problem, because we only know the value of `R1` once the `ADD` instruction has finished the `EX` stage, and the `MUL` instruction will get the values from the registers while this is happening in it's `ID` stage, henceforth we won't read the correct value of `R1`.

Two easy solutions to this problem are to:

- Detect inter-instruction dependencies in hardware and withhold instructions in the decode stage until the data is ready. This creates the bubbles that we've worked so hard to avoid with branch prediction though!
- Have a compiler detect the dependencies, and have it re-order instructions to eliminate them. This is hard to do though, and often results in compilers inserting `NOP` (no operation) instructions that do nothing. These act like bubbles anyway, so we don't gain anything here either.

Since these are unsatisfactory, we could add extra paths to the pipeline between the ALU output and the ALU input. They could be activated if there is an interdependency so that the incorrect result of the `ID` stage could be modified before being processed by the ALU, as shown in Figure 6.

What if there is an interdependency with an instruction that might take a while to execute, such as a memory read (`LDR`)? If we have a program that does:

```
LDR   R1, [R2, R3]
MUL   R0, R1, R1
```

Our path would have to look like in Figure 7.

Adding extra paths to the architecture to pass updated register values back to previous stages of the pipeline is called **forwarding**.

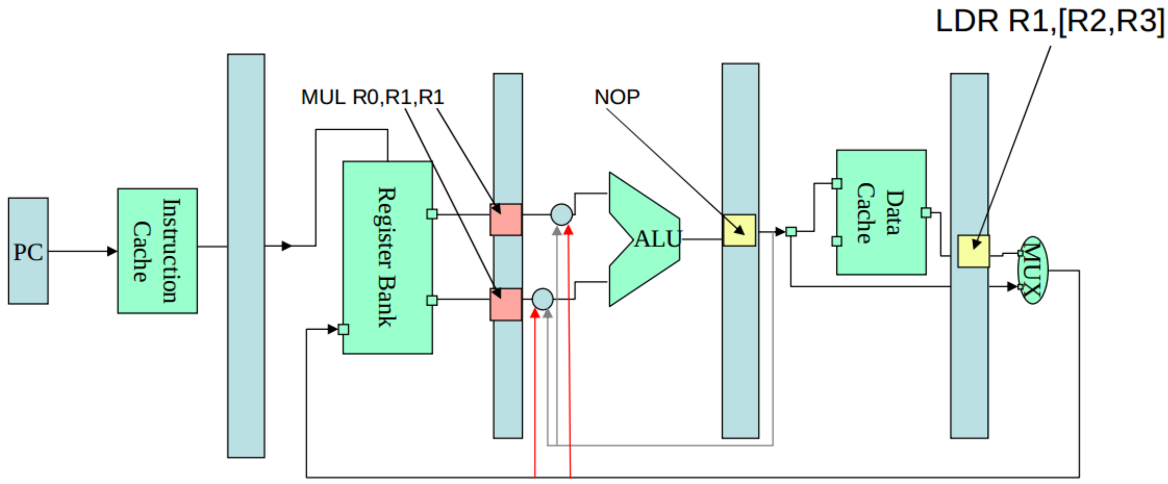


Figure 7: In order to cater for the worst case, where we're waiting on a memory read, we need even more extra paths.

We want longer pipelines, since we want each stage of the pipeline to do as little work as possible, taking as little time as possible, so that we can increase the clock speed. However, as we do that, we will come across more hazards, and we'll need to put more control paths in to mitigate them. Eventually, there will come a point where increasing the length of the pipeline will result in negative returns.

4.3 Instruction Level Parallelism

If there are instructions that do not depend on each other at all, for example:

```
ADD   R0, R2, R3
SUB   R1, R4, R5
```

Then we could run these instructions at the same time, and not experience any side effects. We may run into problems, if there are instructions that depend on these two instructions having completed before they themselves execute, for example:

```
ADD   R0, R2, R3
SUB   R1, R4, R5
MUL   R0, R0, R1
STR   R0, x
```

We can draw a data flow graph to visualise these dependencies, and easily see which instructions can be run in parallel, as in Figure 8.

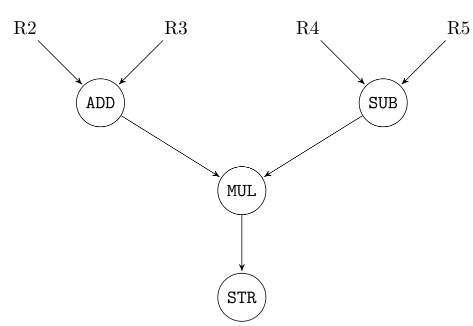


Figure 8: A sample data flow graph.

Though this is a simple example, analysis has shown that it is not uncommon for real programs to have up to four instructions that can be parallelised at some point in the program's running time. Obviously, the amount of times when two or three instructions could be parallelised will be greater than that of four, which increases the argument for ILP.

In order to exploit this parallelism, we can:

- Fetch multiple (in our case two for simplicity) instructions per cycle.
- Have multiple ALU's to execute instructions in parallel.
- Have common registers and caches, since the instructions are operating on the same data.

This could require an architecture looking like that in Figure 9.

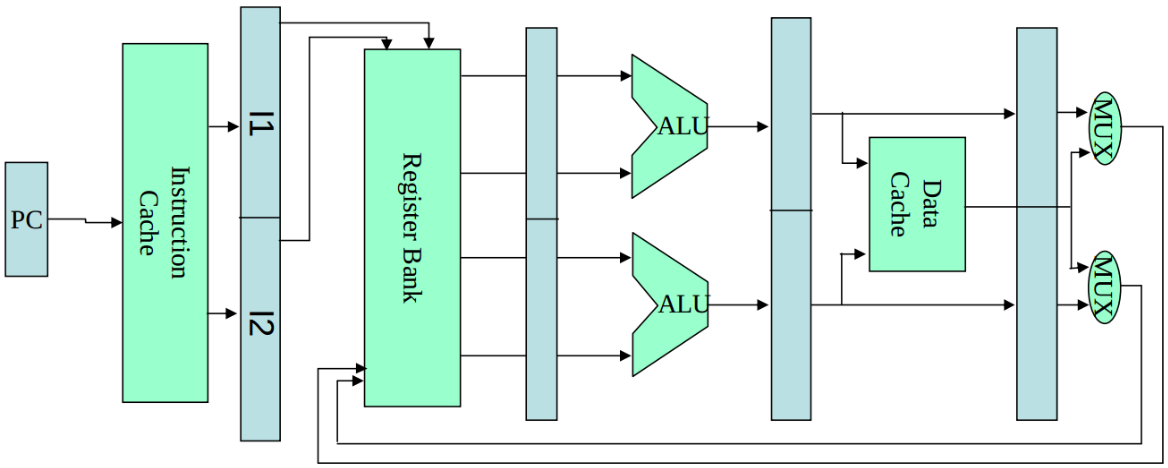


Figure 9: A simple superscalar (implementing ILP) architecture.

If we are able to run multiple instructions at the same time, then our registers and caches are going to have (in this case, which is a low level of parallelism) roughly twice the load that they had previously. In order to allow them to handle this extra load, we can make them **dual ported**, which means that the access circuitry is duplicated so multiple reads and writes can happen simultaneously.

Note, these multiple reads and writes will be occurring in different parts of the register bank and cache, since the parallel instructions are (by nature) independent of each other.

In order to implement ILP, we also need a 'dispatch unit' in hardware which is part of the fetch stage (`IF`). This will fetch multiple instructions if they are independent and can be executed in parallel.

4.3.1 Out of Order Execution

In order to get the maximum number of ILP compatible instruction sequences in a program, the compiler may be able to re-order instructions so that they have a reduced number of interdependencies. One technique that is based on this is the Very Long Instruction Word, where each word will be longer than a normal word (maybe 48, 64 or more bits), and will contain more than one instruction. Since the compiler decides what instructions to execute in parallel by putting them in a VLIW, the processor doesn't need to do so at runtime, therefore the complexity of the instruction scheduling is taken away from the CPU and its complexity can be reduced.

Having the compiler re-order instructions means that it will sometimes add `NOP`'s into the code, which can increase the binary size, and bloat the code. The alternative it to rely on expensive hardware to detect out of order opportunities at runtime.

To implement an out of order processor, you need to have a buffer that instructions are fetched into, a scheduler to choose which (non-conflicting) instructions to execute at what times, and a cache to store memory and register accesses until all the instructions have finished so that the application can execute normally as though all instructions executed in serial. Figure 10 shows how this could all be implemented.

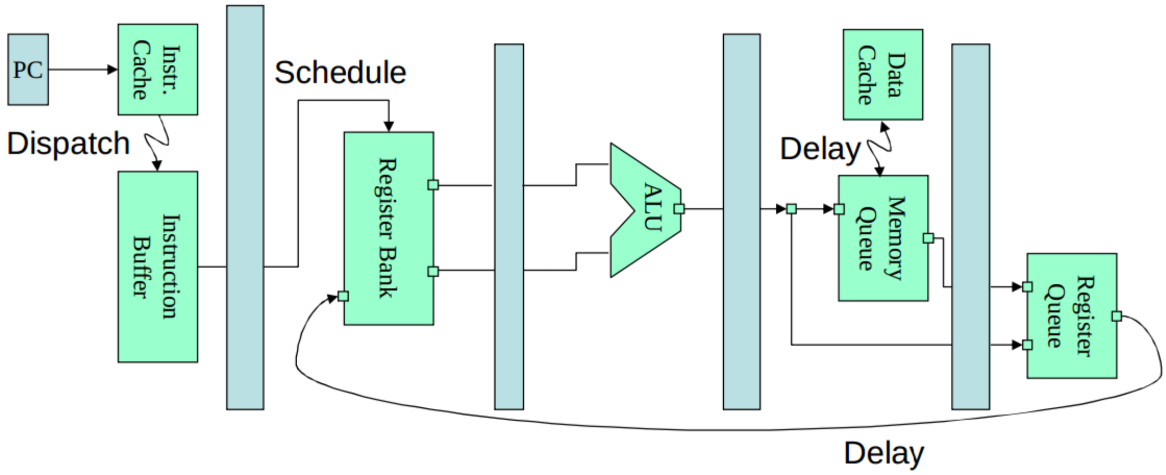


Figure 10: An example of how an out-of-order processor could be implemented.

4.3.2 Programmer assisted ILP

Some processors now support SIMD (Single Instruction Multiple Data) instructions, that allow one instruction to be executed on multiple registers/memory locations etc. This is very useful when doing mathematical computation with stuff like vectors and matrices, since it requires a lot of repetitive operations that lend themselves easily to parallelism.

One example of this is the `UADD8` ARM instruction. It adds two registers together into a third register like a normal add, but it does this four times; once for each eight bit chunk in the registers. For example, `UADD8 R0, R1, R2` will do:

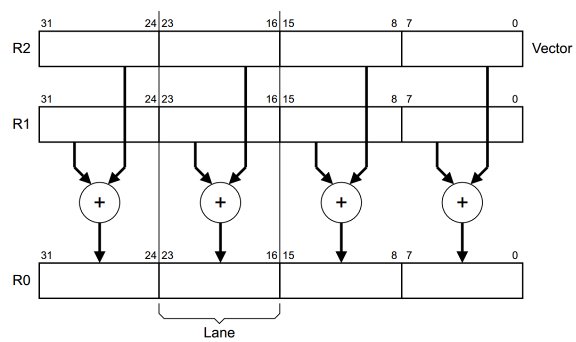


Figure 11: A 4 way 8-bit integer add operation.

5 Multi-Threading

We have already seen, there are many different ways of increasing CPU performance:

- Increasing the clock frequency
- Increasing the *ips* (instructions per clock)
- Minimising the impact of accessing memory using a cache
- Minimising bubbles with branch prediction
- Minimising bubbles using out of order execution
- Parallelising instructions (with a superscalar architecture)

And, up to a point

- Lengthening the pipeline

We have said that increasing the parallelism will speed up our pipeline, but in order to do this, we need to find enough instructions to safely parallelise at once. There are multiple ways to do this too:

- When we get a branch instruction, do we pause until we evaluate the outcome of the branch, or keep issuing instructions? If we're using branch prediction, then we can keep issuing instructions.
- After a cache miss, we need to wait for a certain amount of time for the data we want to come from the memory, over the bus and into the CPU. While we're waiting, can we carry on issuing other instructions.
- Process instructions in parallel (e.g. with a superscalar architecture).
- Write to registers while the previous write is pending using a register cache.

These ways of increasing parallelism are good, but assume we have only one source of instructions. However in reality, a CPU will be executing code from multiple processes at once. What about if we run out of instructions to execute in one program, we could just context switch to another!

5.1 Context switches

In a context switch, the Operating System must load and store a lot of data about the switching processes:

- Process ID
- Process state
- Program Counter
- Stack Pointer
- General registers
- Memory management information
- Open file list (and positions)
- Network connections
- CPU time used
- Parent process ID

All of this takes time for the processor, and power too, which is a concern on low power processors.

5.2 Hardware multithreading

We could have two PC's, two sets of registers (GPR's on the diagram), two virtual address mappings etc, and have the CPU support multi threading natively. This would require the OS to be able to handle multiple processors, since the easiest way of making it work is to make the one processor core look like two processor cores (since the inputs and outputs are effectively doubled). See Figure 13 for examples of the three types of hardware multithreading.

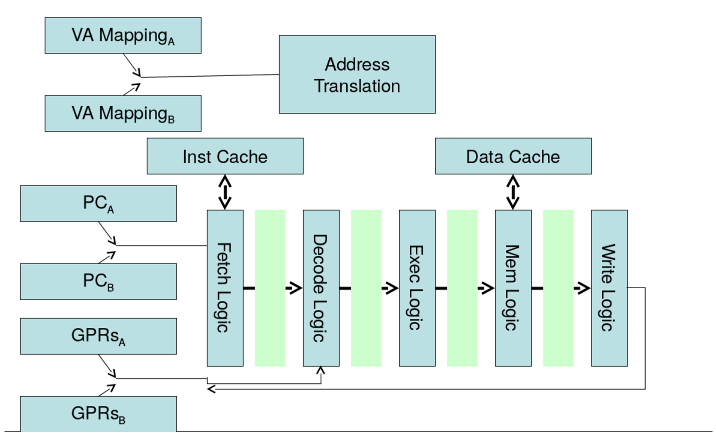


Figure 12: The architecture of a multithreaded CPU

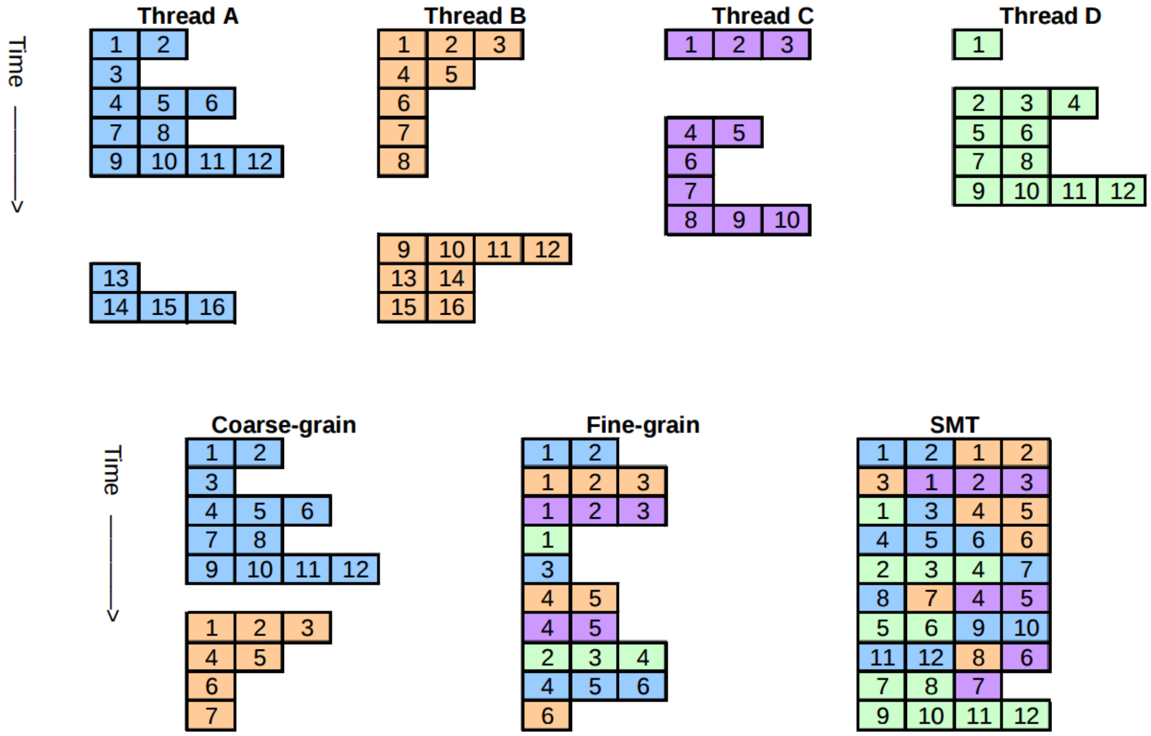


Figure 13: How the three main types of hardware multithreading might execute simultaneous threads.

5.2.1 Coarse grain multithreading

Coarse grain multithreading is when you switch threads whenever the current thread executes an expensive operation. This could be when there is a cache miss for example, or whenever the CPU has to wait to execute more instructions.

This type of multithreading has benefits. If the CPU can execute one instruction per nanosecond ($1GHz$), a cache miss takes 20 nano seconds, and one cache miss occurs on average, every hundred instructions, then the number of instructions per clock cycle is:

Without multithreading

$$\frac{100ns}{100ns + 20ns(\text{one hundred instructions and one cache miss})} = 0.8333$$

With multithreading

$$\frac{100ns}{100ns + 1ns(\text{one hundred instructions and context switch})} = 0.99$$

Coarse grain multithreading is good, because it requires a minimal change in the pipeline; only to abort instructions in the 'shadow' of a cache miss and the ability to resume the instruction stream at the correct point afterwards.

Unfortunately, when you do a context switch, a new program runs on the processor. This is bad, since both the instruction and data caches will suddenly contain an awful lot of entries that are obsolete, and so most of the cache requests will be misses. The term 'trashing the cache' is used to describe another thread (or maybe a procedure in the current thread) ruining the cache hit rate by polluting it with non-relevant data.

5.2.2 Fine grain multithreading

The aim of fine grain multithreading is to be able to switch between CPU threads with as little overhead as possible. This involves interleaving the instructions of several threads. This results in better overall performance, since the impact of short stalls such as accessing memory is reduced by executing instructions from other threads. Each individual thread perceives itself as being executed slower, but the overall performance is better.

5.2.3 Simultaneous multithreading (SMT)

The idea behind SMT is to exploit both instruction level parallelism and thread level parallelism at the same time. In a superscalar processor (i.e. one that can execute more than one instruction per clock cycle), we can issue instructions from different threads in the same cycle.

SMT requires a significant overhead, and is only really feasible to processors with an out of order execution capability.

5.2.4 Disadvantages of hardware multithreading

Threads can 'trash' the cache for other threads, since they will probably have completely different access patterns, so the overall cache performance may decrease. This will not occur if different threads are accessing the same areas of memory, but this is a relatively rare case (since processes can only access disjoint areas of memory) unless a process has multiple threads processing.

Furthermore, there is a significant increase in the complexity of the hardware required. The thread state, priorities, OS-level information etc must all be maintained by the processor in hardware to facilitate (fast) context switches.

5.2.5 Other techniques

There are more techniques we can apply to thread aware processors in order to eke out the very best performance from our precious silicon. Here are a few:

Slipstreaming:
: This is when an application is split into two parts; the critical path, and all the rest. The critical path will run ahead, and pass the result of expensive operations back to the 'main thread'

Memory prefetching:
: We could compile the application into two parts here too, except one will get all the memory accesses, and the other will do everything else. This means that the data will always be in the cache when it is needed in the second thread.

Speculative execution:
: When we get to a conditional branch in the program, we could spawn two threads; one for each path. When we know which is the correct path, we can kill the thread that wasn't executing that path. In this way, we can reduce the impact of control hazards.

6 Multi-Core

Since Moore's law is slowly coming to an end, chip designers are having to look to other techniques to increase performance. Cooling is a serious problem now, since there are so many transistors on chips, that the power density (watts per unit area) is becoming unsustainable. Smaller transistors have very unpredictable characteristics, and the architecture of processors is becoming so complex that it's hard to reason about.

Older problems are becoming worse too; memory is still not getting faster at the same rate as processors are, and as a result even when the clock speed and the instructions per second increases on the processor, lots of the time it may be sat idle.

Lots of solutions have been tried and implemented to increase single core performance:

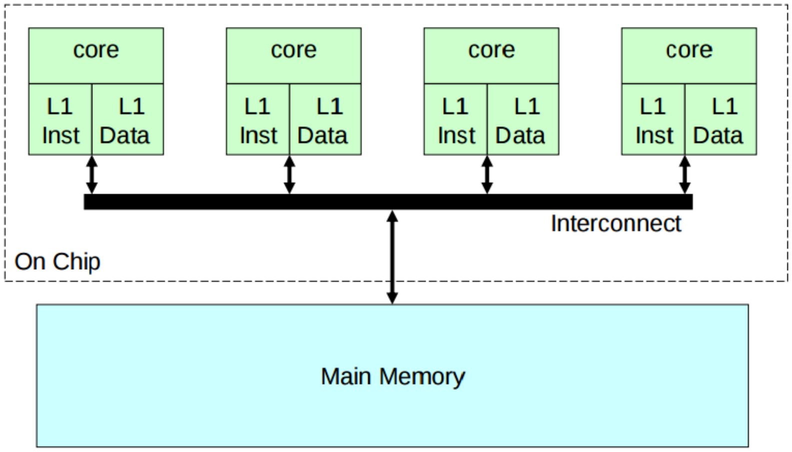


Figure 14: A simplified multicore architecture

- Caching
- Pipelines
- Superscalar processors
- Out of order processing
- Multithreading
- Branch prediction
- Forwarding
- Register renaming

However, these all have a limited scalability; the hardware costs increase in a roughly quadratic manner but the performance increase is sublinear.

Another way to speed up execution is to have multiple CPU cores on one chip. Multiple simple cores may achieve a higher performance than one complex core whilst being easier to design.

There are different opinions on how to connect the processors; should they all access the same memory, or each have their own? From a software point of view, it is easier to have processors that can access shared memory (since synchronization is implicit) than having separate memories. Unfortunately, after a few cores, shared memory becomes harder to attain.

Having more cores does not speed up programs without the programs making use of the cores explicitly. Threads must be spawned by a program if it wants to execute on more than one core, however, different programs can run concurrently on different cores. For example, one core could run an antivirus scan, while another could run a web browser.

6.1 The structure of a multi core processor

In a multicore processor, some components will be shared between processors, and some won't. Usually, each processor has its own an L1 cache, and sometimes an L2 cache too. You can see a generalisation in Figure 14.

6.2 Data coherency and consistency

Different CPU cores trying to access the same areas of memory can be an issue, since the caches may not be *coherent*. Coherence is a desired property of multiple caches all pointing to the same shared resource; each cache should provide the same view of the resource at all times (at least from the point of view of the users of the cache).

This shouldn't be a problem, since after all, processes have disjoint memory spaces, so each process will be running on one core and accessing different areas of memory. However, threads *do* share memory, and they can run on multiple cores (and other stuff can update memory, such as DMA etc) so assuming disjoint memory access across cores is a bad idea.

When we talk about CPU cache coherency, the L1 cache is what we mainly focus on. Each core will have its own L1 cache, and if a value is updated in one cache that the others are also holding, then the other caches will be immediately out of date. Even if we wrote back to memory on every cache write, the other cores' caches wouldn't be updated. We need to ensure that every core has an up-to-date cache at all times.

You can think of consistency as representing the model of the machine as presented to the programmer. If one thread updates a value in memory, it is expected that all the other threads will also be able to see that update.

Sometimes, we want to read and update a memory location as a single atomic operation, and make sure that no other threads can update the same location at the same time.

Sequential consistency is the idea that memory operations should appear to execute one at a time, even though they may not do in practice. Leslie Lamport describes it as:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. (Leslie Lamport)

Since lower down the technology stack of a running program, instructions may be executed out of order (for example in Java), the compiler must insert special instructions such as the following to ensure that consistency is maintained:

Fence:
: A fence will make sure each memory access before the fence completes before new ones are started.

Barrier:
: All threads in the program must reach the barrier before any of them can continue executing.

Lock:
: Only one thread can enter a section of the program that is protected by a lock at any one time. That section may be referred to as atomic.

  Hardware support is usually required to implement locks, since normal memory reads and writes cannot be guaranteed to be atomic without, guess what... locks!

There is some Instruction Set Architecture support for synchronization, such as atomic 'compare and swap' instructions conditional load and stores (based on if the memory location has mutated since it was last accessed), and transactional memory.

Transactional memory allows you to read and write to memory with no restrictions, however once the operation is completed, the transaction is checked to see if it conflicted with any other transaction, and it is rolled back if it did (and started again).

6.2.1 Coherence protocols

A scheme where each core would know exactly when other cores are caching the same data is infeasible as it is would be far too complex (and probably slow). As an alternative to this, each cache *snoops* on the other caches for activity related to its own cache. This is most simply implemented by routing all data through a bus so every cache can see the activity of other caches.

There are two very simple snooping protocols in use:

Write Update:
: When a core writes a value to memory, it is updated in its local cache. The cache then broadcasts the address and new data to the bus and all the snooping caches update their own copy.

Write Invalidate:
: A core that wants to write to an address writes to its own cached copy, but also sends a 'write invalidate' message that will tell the other cores to invalidate the cache line that they have stored. Any read to that address will now miss.

In both schemes, the bus makes sure that only one core can use the bus at one time, so that simultaneous writes do not occur. Though the first option (write update) might look fastest, it isn't always, since it will update the cache when it might not be needed (e.g. two writes to the same cache in a row would require two updates to the other caches, but only one invalidate using write invalidate). This can also happen when you're writing to different words in the same block of a multi word cache (most caches are multi word remember!).

Unfortunately, this can happen often due to spatial and temporal locality (the principles that make CPU caches work), and bus bandwidth is a precious commodity, especially in shared memory multi-core chips. Invalidate protocols have been modelled to use less bandwidth, and so these are more commonly used.

In both of the above schemes, merely knowing if other CPU caches hold the value is enough to know whether to send a message on the bus in the first place.

If we use the invalidation scheme, then if the writes don't also get written through to memory, then when other caches read from the memory, they will read old values, so we need a protocol to handle this. . .

MESI Protocol

This is a practical multi-core invalidation protocol which tries to minimise the bus usage. It also implements a 'copy back' scheme where the main memory or L2 cache is not updated until a 'dirty' cache line is displaced.

As a result of this, cache lines have two state bits, meaning they can be in the following states:

Modified:
: The cache line has been modified and is different from the main memory (i.e. it's a dirty copy).

Exclusive:
: The cache line is exactly the same as main memory and it's the only copy.

Shared:
: It's the same value as the main memory, but other copies may be in other caches which may differ.

Invalid:
: The cache line data is not valid.

Another protocol called `MOESI` has another state:

Owned:
: The cache line has been modified and is different from memory. There are other copies in other caches.

Owned means that we don't have to copy back to main memory on writes.

The changes of state of the cache line are dependent on the memory access events, which can be either from local core activity or snooping from other cores. The cache line state will only be effective if its address matches the address of the memory event.

The state changes are as shown in Figure 15.

'RWITM' means Read With Intent To Modify.

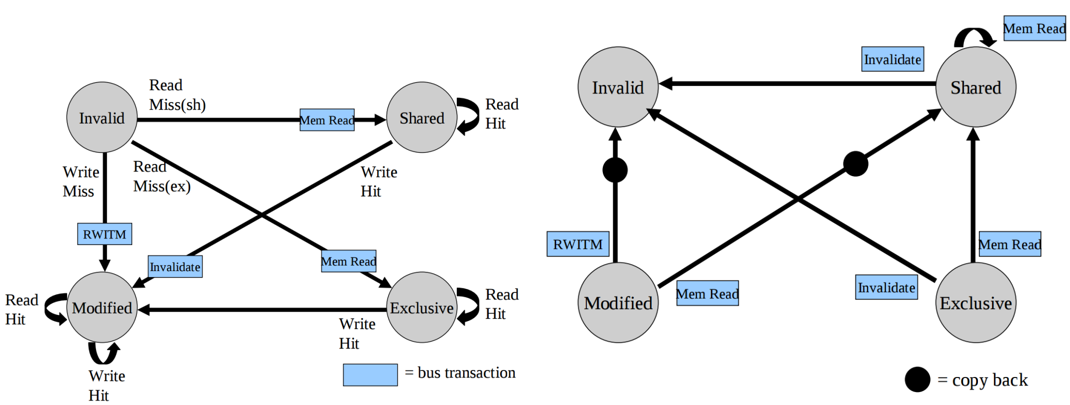


Figure 15: MESI - local cache view (left) and snooping cache view (right)

The consensus is that snooping protocols don't scale well beyond around 16 cores, since the bus connecting the cores becomes saturated. We could have a hierarchy of buses, but this is complicated.

Instead, we can create directory based protocols. If we use distributed hardware, we can have a directory listing what data other caches have, and caches can talk to each other in a point-to-point manner.

Since there is more traffic required for this between the cores, the bus is often replaced with a network on a chip.

The cache lines can be in three states that are like those of MESI:

- *Invalid*
- *Shared* (coherent with the main memory)
- *Modified* (value is changed and no other copies are anywhere)

The directory can have three states too:

- *NC* (Not Cached; the cache line isn't present anywhere)
- *Shared* (the cache line is present in at least one core)
- *Modified* (the cache line is modified and is in one other core)

The directory also stores a *sharing vector* describing what core holds the same cache line, which can be seen in Figure 16.

If we have a central directory, then it might become the bottleneck we were trying to avoid with a bus. To stop this, we could use a distributed directory, where each core has its own directory, this requires an extra flag 'home' which indicates the data is stored locally.

To sum it up, directory protocols are:

- More scalable (better for CPU's with more cores).
- Use p2p messaging to simplify interconnection and allow parallel communication.
- Each core doesn't have a global view of the caches, we need to store the state separately in the directory.
- Control is more complex, but performance is better overall.

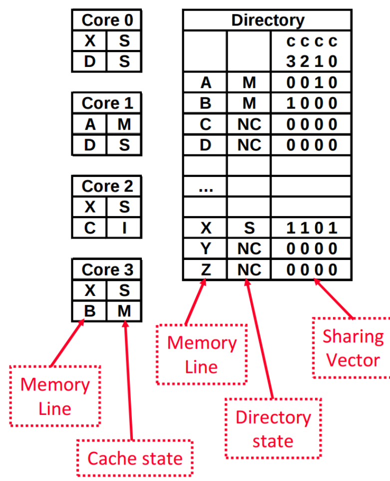


Figure 16: How data might be organised in a directory protocol

6.3 On-chip interconnects

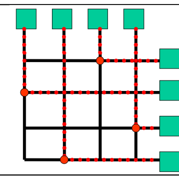
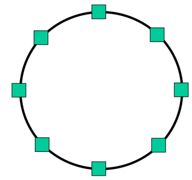
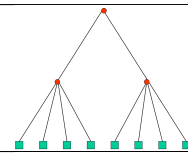
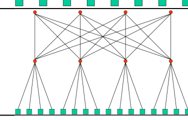
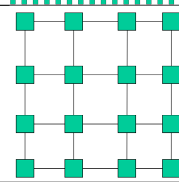
So far, we've only really considered buses as a way of communicating between cores, however we could use a network on a chip (NoC) to communicate. Things we need to consider about networks on a chip are:

- Bandwidth
- Latency
- Congestion
- Fault tolerance
- Area (mm^2)
- Power dissipation

There are three important features of a NoC; topology (how the cores and network infrastructure is organised), routing (how the traffic moves around) and switching (how the traffic moves from once component to another).

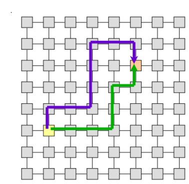
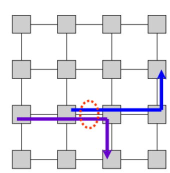
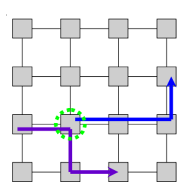
The medium through which data is transmitted is a bus. Buses are single usage at any one time and are controlled by a clock that divides its use into time slots. Transactions are often split, since a message will be sent in one slot, and th reply received in another later slot (slots can be used in between these two events).

Networks on a chip can take many different forms though:

Crossbar	You can connect n inputs to n outputs	
Ring	Simple, but low bandwidth and variable latency	
Tree	Variable bandwidth and latency (depth etc), may be unreliable	
Fat tree	Faster, more reliable, but uses more resources than a tree	
Mesh	Okay bandwidth, variable latency, but good for large systems because of the layout	

6.3.1 Routing

There are three types of routing we're going to look at:

Minimal	• Always selects the shortest path • Packets move closer at every step • But are more likely to be blocked	
Oblivious	• Unaware of network state • Packets take a fixed path • Very simple and deadlock free • Prone to contention	
Adaptive	• Aware of network state (moves packets to avoid contention) • Higher performance • More area and power required • Deadlock prone (more hardware needed) • Rarely used in NoC's	

6.3.2 Switching

There are two types of packet switching we'll look at, store and forward and wormhole switching. The premise is that data is split into small packets, some extra information is added to enable them to get to their destination and they are sent into the network. This allows time-multiplexing of network resources and is good for performance (especially in short messages). Packets are also split into flits, which are teeny packets (see Figure 17).

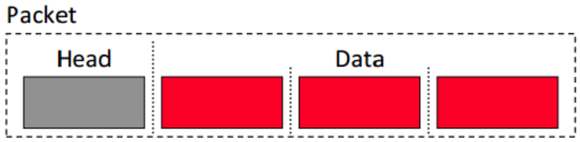
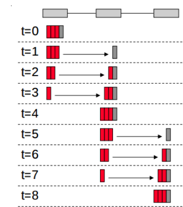
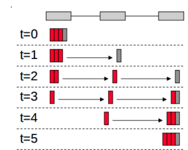


Figure 17: How a packet is split up into flits.

Store and forward	• A packet is not forwarded until all its phits arrive to each node • Failure detection happens on the fly • But performance is low • And large buffers are required	
Wormhole	• A packet is forward as soon as its head arrives • Performance is better than store and forward • But fault detection is only possible at the destination • Less hardware	

7 Virtualization

Virtualization isolates the details of the hardware from the software that uses it. You can break virtualization 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 a whole OS under the control of a layer of software, e.g. VMware.

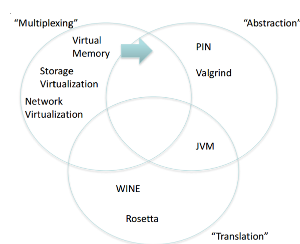


Figure 18: The different types of virtualization. PIN is a tool to annotate binary executables if you hadn't heard of it.

Here, a "layer of software" can mean a lot of things, but generally one or more processes.

This is useful, since virtualization can translate between technologies (different instruction sets, system calls etc), change the level of abstraction (providing garbage collection, debugging etc) and even make the system resources look different (emulate CD drives, reduce the amount of RAM for a virtual machine).

For example, the JVM interprets byte code, and maps it onto the host OS's API. Rosetta translates PowerPC binaries to x86 ones on the fly.

There's a really cool debugging technique called Reverse Debugging that is made possible my virtualization. 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. Different granularities can be implemented; the debugger could save resources by just implementing a function reset feature that discards the current stack frame and lets you run the function again while looking at exactly what's happening.

7.1 The details of virtualization

A virtualised OS runs on top of a Virtual Machine Monitor (VMM) or Hypervisor. This often (but not always, since it is sometimes one itself) sits on top of the *actual* operating system running on the machine. The VMM will handle physical resource access for the guest OS since it runs in a privileged mode, and it makes sure the guest OS is isolated from other resources. These resources include:

- Timers
- CPU registers
- CPU flags (e.g. interrupt enable)
- Device control registers (DMA, interrupt IO etc)
- Memory mapping (page table etc)

Often, 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. Since the VMM does the privileged operations, the guest OS can be unprivileged.

This can be tricky sometimes though; some instructions behave differently according to what mode they are in, so the VMM must be able to handle that.

If the guest OS is virtualization aware, then it is much easier to build a VMM, since the guest OS can call the VMM specifically for privileged operations, cooperate with the VMM over shared page tables and call the VMM for IO.

7.2 Operations on VM's

When you start a VM, the hypervisor will save the current registers, load the VM's initial registers and jump to the new VM's PC address. Likewise, when a VM is stopped, the hypervisor will save its registers into its own memory space. It is important to note that VM's are stopped and started all the time so that the CPU (and other resources) can be shared.

When the VM is stopped, its memory and IO state is also retained as well as its CPU registers. It is best to stop a VM when its IO is *quiescent*, so that we don't have to save loads of IO buffers etc.

Quiescent means to be 'in a state or period of inactivity or dormancy'

Once we've stopped a VM, we can freeze it by saving all its state into a file. This file tends to be large, since it contains the whole operating system, its installed applications and the state such as the registers etc.

Because virtual machines can be stored as files, there are a number of things we could do with them:

- Move a VM onto a different machine
- Snapshot the state of a VM and roll it back later
- Archive a VM into a database or secondary storage
- Quickly start an archived VM (rapid provisioning)
- Live migration (we'll get on to this)
- Load balancing (and this)

7.2.1 Live migration

Wouldn't it be good to be able to move a VM from one machine to another without pausing execution of the VM? That means we could change the physical machine a VM is running on (maybe to repair or upgrade the old one) with minimal interruption for the end user.

I'm going to use Wikipedia's explanation for this, since it's really good:

> **Warm-up phase**
>
> In pre-copy memory migration, the Hypervisor typically copies all the memory pages from source to destination while the VM is still running on the source. If some memory pages change (become 'dirty') during this process, they will be re-copied until the rate of re-copied pages is not less than page dirtying rate.
>
> **Stop-and-copy phase**
>
> After the warm-up phase, the VM will be stopped on the original host, the remaining dirty pages will be copied to the destination, and the VM will be resumed on the destination host. The time between stopping the VM on the original host and resuming it on destination is called 'down-time', and ranges from a few milliseconds to seconds according to the size of memory and applications running on the VM. There are some techniques to reduce live

migration down-time, such as using probability density function of memory change.
(http://en.wikipedia.org/wiki/Live_migration)

7.2.2 Load balancing

Say we had two physical machines and six VM's. The six VM's were distributed equally between two people, one of which did parallel text mining, and the other ran a twitter bot. Since we wanted to be fair, we assigned them one physical box each, with their three VM's on it, however, the text mining guy was always running at 100% CPU utilisation, while the other guy barely got over 2%.

This is where load balancing can be helpful; we could have the Hypervisor move VM's between physical machines based on the relative load on the physical machines. This would work independently of the application of the running VMs. This way, our text mining guy can (in theory) have 98% of the CPU power of the second physical machine as well as near 100% of the first one too!

We can also use a similar technique to keep services highly available, if we regularly copy the state of one VM onto another physical machine, then the secondary one is effectively a standby backup in case the first one crashes.

8 Permanent Storage

Broadly, there are three main categories of permanent storage media:

Write Once, Read Many (WORM):
: This includes CD-ROM, DVD-ROM, and some Blu-ray Disks. Once you've written, you can't write over it.

Write Many, Read Many:
: This includes hard drives, tape drives etc. The writes are fully reversible for the purposes of the computer.

Of course, the writes aren't literally fully reversible because of wear etc, but they are near enough.

Write (not too) Many, Read Many:
: Rewritable CD/DVD's have hundreds to thousands of write cycles, and flash memory has more, from the thousands to the low millions. These devices will wear slightly on each write, making them less effective.

8.1 Hard drives

Hard drives consist of multiple magnetic disks (platters) laid on top of each other that spin around and can be written to or read from by a 'head'. Each platter can store around 2TB at the moment, and hard drives often have around four of them, meaning that the largest hard drives can store about 8TB of data. The platters rotate at one of four speeds, $5400RPM, 7200RPM, 10000RPM$ or $15000RPM$.

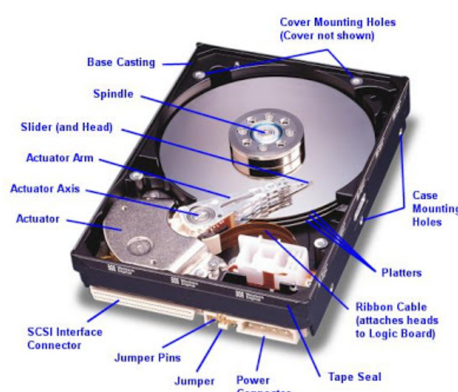


Figure 19: The structure of a 'traditional' spinning disk Hard Drive

Hard drives are power hungry devices compared to lots of other components in the computer since they need power for spinning the platters, moving the heads, reading and writing, the chips on the hard drive that control the IO etc.

Hard drives are very slow in terms of data bandwidth and latency compared to other forms of memory storage such as RAM. The following terms are used to quantify hard drive performance characteristics (refer to Figure 20 if you've not heard of some of the terminology before):

Seek Time:
: This is the time it takes for the head to reach the target track on the platter.

Search Time:
: This is the time for the target sector to arrive under the head.

Transfer Rate:

This is the amount of data that can be read per unit time. It depends on a lot of factors such as where data is on the disk and access patterns of the data, though the transfer rate is usually given as a 'sustained transfer rate' in MB/s.

Disk Access Time:

The total time it takes to access data on the disk:

$$\text{Disk access time} = \text{Seek time} + \text{Search time} + \text{Transfer time}$$

It is sometimes important to compute the average access time of a hard drive given the seek time, rotation speed, transfer speed and sector size. For example, if a sector is 512B, the seek time is 8.5ms, the disk rotates at $7200RPM$ and the transfer speed is 177MB/s, then we can compute the access time to be:

This assumes that the hard drive will have to do 0.5 rotations to reach the correct sector (this is the average number of rotations when you think about it).

$$\begin{aligned}
\text{Search Time (ms)} &= \frac{0.5 \text{ rotations} \times 60}{7200RPM} \\
&= 4.16\text{ms} \\
\text{Transfer Time (ms)} &= \frac{512\text{B}}{177 \times 10^6\text{B/s}} \\
&= 2.89\mu\text{s} \\
\text{Disk access time} &= \text{Seek time} + \text{Search time} + \text{Transfer time} \\
&= 8.5\text{ms} + 4.16\text{ms} + 2.89\mu\text{s} \\
&= 12.66\text{ms}
\end{aligned}$$

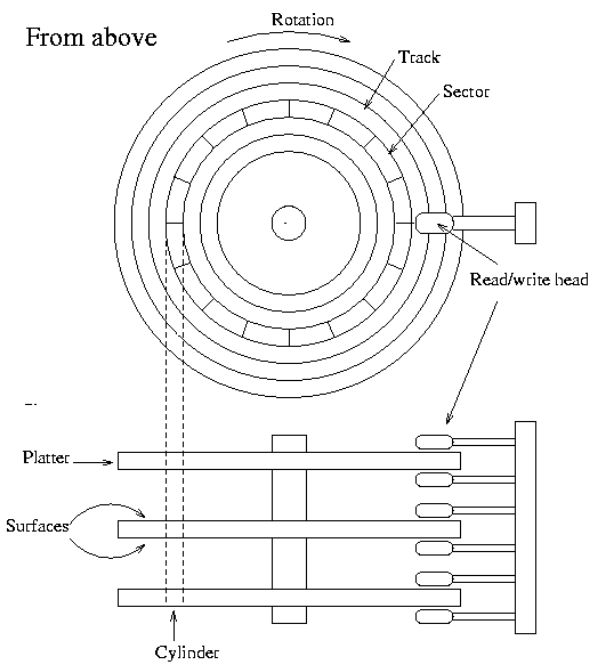


Figure 20: The internal structure of a hard drive and the layout of its disk.

See slides 9-14 on the first of the storage lecture notes for a pictorial example.

Sometimes, when the Operating Systems wants to read a file off the hard disk, the file will be split over multiple sectors. In order to read the file, the hard drive will need to move to the head to each sector to read the whole file. If the sectors are physically distant from each other (say one was on the very inner track of the disk, and one was on the outermost track), then the hard drive may have to 'waste' rotations in order to move the head to the correct track. 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.

8.2 RAID

RAID stands for a Redundant Array of Independent Disks, which is a type of storage virtualization.

Hard disks are often too slow because of a high seek time, a high search time and a low sustained transfer rate. We can combat the first two problems by having multiple platters on the disk so there are more sectors per cylinder (so the head has to move less), and increasing the speed of rotation will decrease the search time.

A low sustained transfer rate cannot be easily increased though. One solution is to *stripe* a file over multiple disks. This is called `RAID 0`. If a file can be split up into nine sections, we could put it on four disks like so, and speed up our sustained transfer rate by around four times:

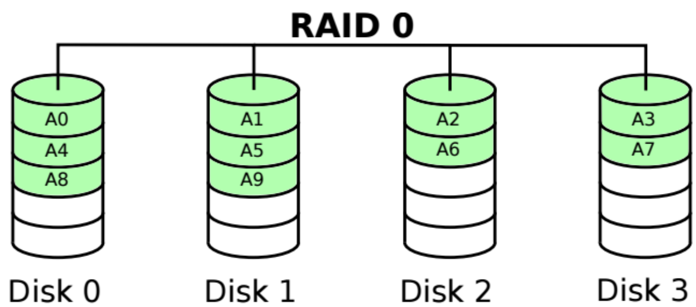


Figure 21: RAID 0 gives higher speed, but lower reliability.

If a disk did break in this situation, then it'd be bad since whole swathes of the file would disappear.

To avoid this, we could use `RAID 1`, which mirrors the data onto another disk. This means we would have four copies of the same data, but *all* of our hard drives would have to fail before we lost any data:

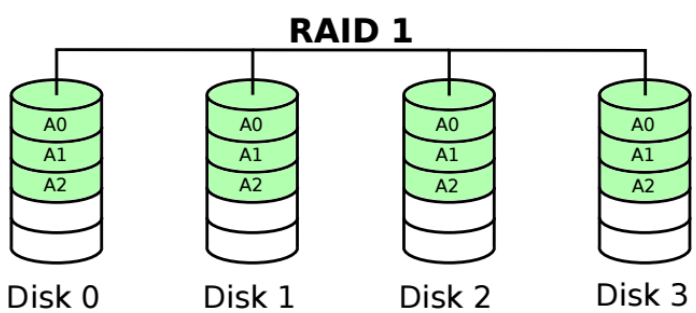


Figure 22: RAID 1 doesn't increase speed, but it does increase reliability

A half-way-house solution between these two, would be `RAID 10`. This is a nested configuration, which still stripes the data, but keeps copies of the stripes. It can tolerate disk failure as long as no `RAID 1` mirror loses all of its drives:

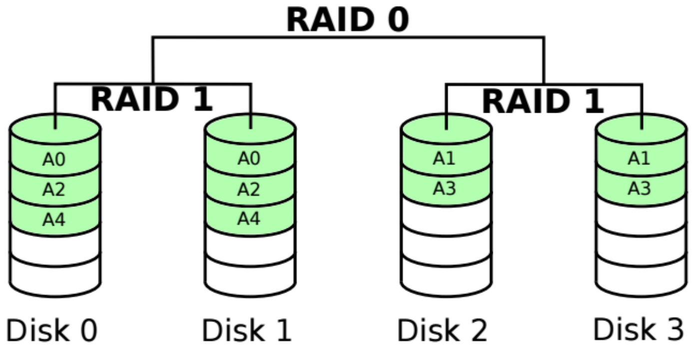
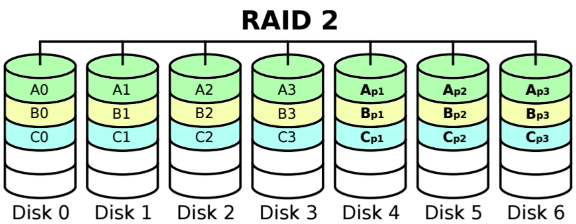


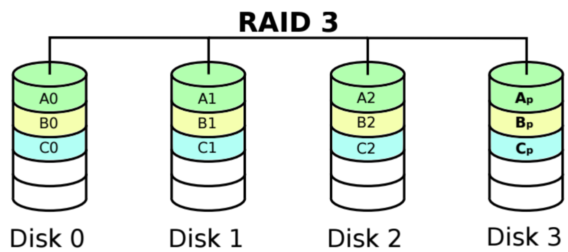
Figure 23: RAID 10 gives higher speed, and higher reliability.

See my `COMP28512` notes for more information on hamming codes and parity bits.

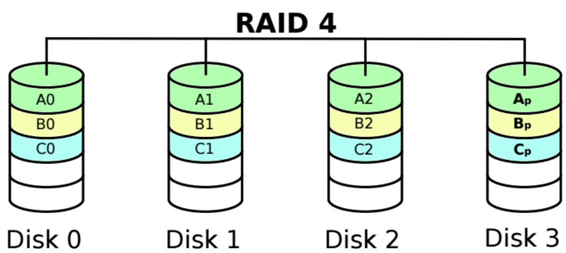
We can use error correction (e.g. hamming codes and parity bits) to provide redundancy *and* striping as done using `RAID 2,3` and `4`. `RAID 5` and `6` both distribute their parity bits over other drives.



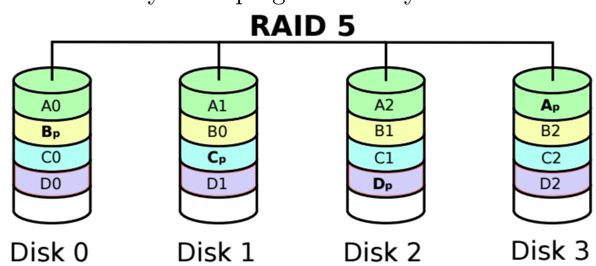
Bit Striping and Hamming Codes



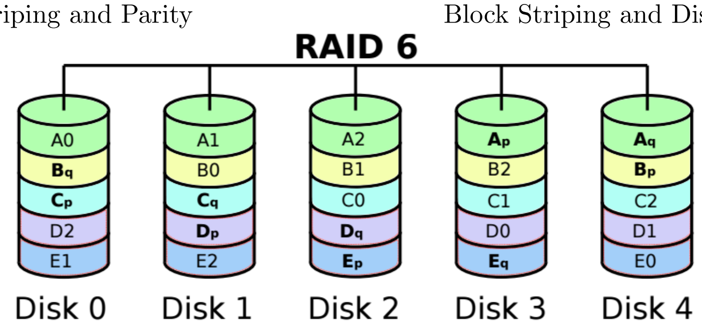
Byte Striping and Parity Bits



Block Striping and Parity



Block Striping and Distributed Parity



Double Distributed Parity

Figure 24: RAID 2-6

Popular other combinations of `RAID` include `RAID 50` and `RAID 160`. Remember that `RAID 160` is `RAID 1` inside `RAID 6` inside `RAID 0`.

What happens if a disk fails in the following situations:

RAID 0 You lose all your data (and hope there's another RAID layer).

RAID 1 You just hot-swap the failed disk.

RAID 2-6 Operate in a 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).

Most operating systems support some kind of `RAID`, or you can buy a dedicated hardware controller.

8.3 Solid State Drives

Solid State Drives (SSD) are made of flash memory. They have a Floating Gate Field Effect Transistor that can store 0's and 1's. It is possible that they will become multi level (i.e. store two bits) in the future.

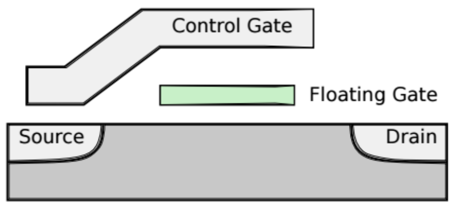


Figure 25: A Floating Gate Field Effect Transistor

Solid state storage has a data retention time of around ten years, but will also wear out with write cycles (and so the performance decreases too). X-rays can affect the data on the drive (so don't take it through airport security).

Error correcting codes, block remapping and wear-levelling are all used to mitigate these flaws though. Wear levelling is when logical block addresses are mapped to physical addresses differently over time so that specific blocks aren't worn out.

SSD's are much faster than hard drives though, since they don't have moving parts, so the data access is much faster (the latency is around 0.1ms or less. Their sustained read and write rates are much faster too. You pay more per gigabyte though, around 14 times more at current rates.

8.4 Storage Virtualization

Classically, we have a file system that maps from our logical thinking of how files are ordered onto the disk. File systems do not naturally span onto multiple drives though. RAID enables us to break this mapping by striping and mirroring. It's kind of similar to virtual memory in the sense that the file path doesn't have much of a relation to where the data is actually stored.

A *volume group* is a set of drives in a pool, and storage space in such a group is divided into *physical extents*, which are usually of the same size. A *logical volume* will be made of some of the physical extents.

These abstractions let us add more drives, extend partitions within drives, take a snapshot of a file system etc. This greater control can be used to great advantage. For example, with Linux:

`/` Mostly read, we want fast read seeks and transfer rates.

`swap` Reads and writes, but we want a high bandwidth. This can tolerate data loss better than any other part of the hard drive (since its only RAM).

`/opt` Infrequent access

`/var` Infrequent access

This means we could mirror `/`, stripe the `swap` and allocate the spare space to `/opt` and `/var`.

8.4.1 Storage Area Networks (SAN)

Storage Area Networks are used to decouple compute servers from storage. They allow for storage specific functions to be localised to and optimised for them. This means we can share disk resources across servers, rapidly migrate disk images, share common subsets of file systems (e.g. virtual machine images) and more. One downside is the decreased bandwidth and increased latency, but with modern network protocols, this isn't as much of an issue.

ZFS is a volume aware file system. It protects against losing files, running out of space, corruption of data etc by being very flexible and having lots of ECC. These are implemented by copy-on-write, simple rollback and recovery, wear leveling, self checking and healing, sumchecking and more.