Cache Misses (3C's model of cache performance)

Compulsory Misses - Cold start

Capacity Misses - Cache cannot contain all blocks of the program

Conflict Misses - Multiple blocks compete for same set - does not occur in fully associative caches. This is because the LSBs (possibly word level) are used for indexing. e.g. 01**10**00 and 10**10**01 conflict

Kommentar [1]: Tag

Kommentar [2]: Index

Kommentar [3]: Byte addressing

Avoiding Misses

Hardware - increase cache size, replacement policy to avoid **capacity misses**
increase associativity to avoid **conflict misses**

Software - Reduce working set to avoid **capacity misses**
Compilation techniques can make some huge improvements to **capacity misses**

How long does it take to fill a cache?

no. lines = cache size / line size

time = no. lines * time per memory access

Cache Consistency

It is possible for other entities, e.g I/O to write to memory, outdating the cache

Conversely, we could have a problem where new data is in cache, and old data is read from memory by I/O devices

Software Solution

OS knows where and when I/O takes place, and marks these areas as non-cacheable, or clear cache before/after I/O (slow). Cache Flushing.

Hardware Solution

1. Route all I/O through the cache. Slows down cache, and pollutes cache. Evicts other data
2. Snooping - Some "Snoop" logic between the cache and memory. Listens/observes all activity on bus. Verifies data has been requested, and checks cache for hits. Updates cache or memory based on whether read or write, if the address exists in cache

We need a method of coherence with multiple processors. If processor 1 updates a value that exists in the cache of other CPU, and also in RAM, need to keep consistency.

Caches and Virtual Addresses

When we use caches in a system which utilizes virtual memory, we must consider how we will represent the address of data in the cache
CPU addresses - virtual
Memory addresses - physical
Translation Lookaside Buffer - translates between V-to-P
Tag needs to be physical or virtual - is a problem

1) Cache by Physical Address

Place TLB between CPU and Cache
Ideal, however, address translation and cache are in serial, slows cache down.

2) Cache by virtual address

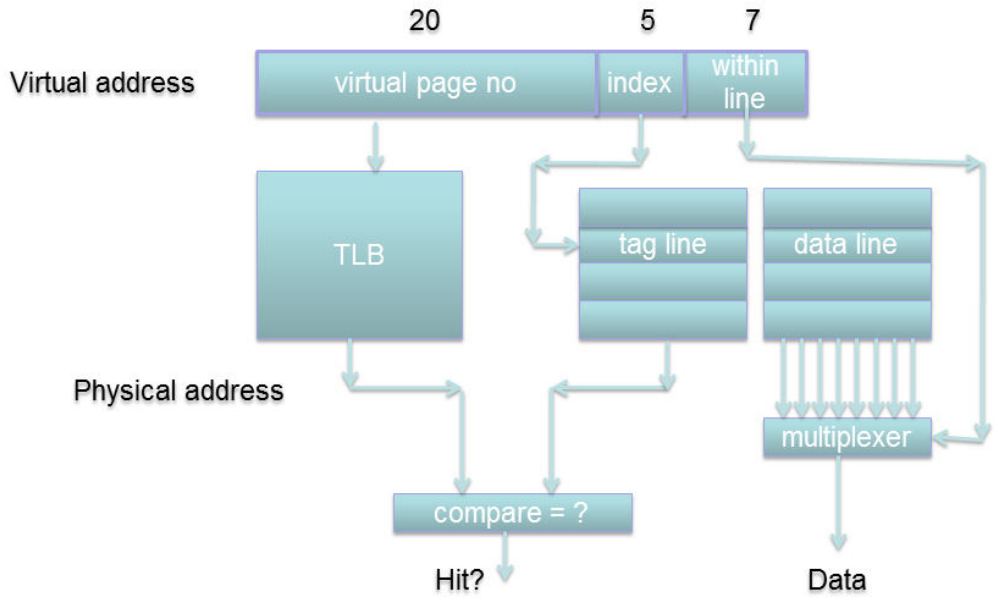
TLB between cache and memory
Faster, but brings more complexity.
Brings a problem with snooping. Snoop is between cache and memory. Snoop will not be able to check the cache, as it only has access to the physical addresses.

Aliasing

Pages shared between processes.
Same physical address, different virtual addresses in each processor

3) Translate in parallel with Cache Lookup

Translation only affects the higher order bits of the address (page number)
Virtual address = virtual page number + page offset
Translation only affects page number, address within page remains unchanged.
Select “index” field of cache address from within low order bits.
Only “Tag” bits changed by translation.



Virtual address = virtual page number + page offset
becomes
Virtual address = virtual page number + (index + within line)

Kommentar [4]: e.g. byte level / word level addressing depending on line size

Cache is necessary to bridge speed gap between CPU and Memory
CPU instruction <1ms Memory >50ms

Kommentar [1]: The so-called 'memory wall'

How do we structure this cache?

CPU gives cache a full (32-bit) address to access some data. Cache stores a small selection of values from main memory. We need to find whether this data already exists in cache.

Fully Associative Cache
Small cache
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 is found - cache hit
Not found - Cache miss, must go to main memory (slow), cache should be updated

How does data get into the cache?
Caches rely on locality. Two types of locality:
Temporal - if something is used, it is likely to be used again
Spatial - things stored together are often used together
Cache Line (64 bytes) - If something is loaded, the next 64 bytes are also loaded into cache. Reduces number of memory accesses. “The next cache line is loaded”. This is a form of exploiting spatial locality.
A “slot” in the cache is of the width of the cache line

Kommentar [2]: line size?

Fraction of cache accesses which are “hit”
Need >98% hit rate to hide a 50:1 ratio of memory speed to instruction speed
Hit rates better for instructions than data, as locality is more common

What happens if we cache miss?
Temporal locality states put recently used data into the cache
If read from memory and cache is full, need a replacement policy

3 Main Cache Replacement Policies
LRU - expensive, a lot of logic. Very useful though
Round Robin - cycle round locations
Random - easy to implement, not as bad as it may seem

Cache Write Strategy
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 is only done to cache (marked as dirty). Only write back to memory when cache entry is evicted to memory. (High performance)

Cache Miss on Write
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.

Fastest strategy for cache miss on write is Write Allocate with Copy Back.
However this can cause cache and RAM to not be coherent (different values), because writing waits. Could be a problem for multi-processors (needing to be accessed multiple times). This could need special handling later.

Associative Cache Data Width
made up of Address and Data
Address may be 32-bits, data is one byte. 1 byte for data is not practical.
Can make more room for data by only accessing a root address, and accessing data linearly from the root address.

e.g. store Word level address - 30 bits, QWord address - 28 bits etc

Real Cache Implementations
We want fully associative, fastest
32-bit address storage is expensive
It is possible to achieve this using standard RAM - RAM not meaning Main Memory, but just simple memory with no additional power hungry hardware.
Few real processor caches are fully associative due to expenses

Kommentar [3]: SRAM means static RAM not standard RAM

Kommentar [4]: This is semi true, but tenuous. Capacitive memory (a la main mem) is more power hungry, but is also slower - that is the main reason for using SRAM which is a simple bistable circuit (pretty much a couple of transistors).

Direct Mapped Cache
Fully associative has a lot of expensive comparison operations, so we want to avoid this
Uses 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, they can only be stored in the same part of the cache.
Not flexible - each memory location has a defined cache location.
A hash table implemented in hardware. Most aspects are identical to a fully associative cache.
Differ on replacement policy.

I believe the less-significant part of the address is used as the index so we can still exploit spatial locality. Otherwise spatially similar elements would be competing for the same slot.

Kommentar [5]: This is right - however it gives rise to cache-trashing wherein having groups of addresses tend to move through blocks of addresses causes lots of conflicts. Increasing associativity helps this.

Direct Mapped Replacement

New incoming address can only be stored in one place (index)
LSBs as index exploits spatial locality, reduces displacing of recently used data.
Lower hit rate than Fully associative due to inflexible replacement / conflicts
Much cheaper than associative

Set Associative Cache

A compromise a small number, N, of DMC's operating in parallel.
If one matches, we have a hit and select appropriate data using a mux.
Replacement strategy more flexible
if 4 DMCs, 4 addresses with the same index but a different tag can be stored
We can then use another replacement policy to choose which entry in which cache to evict.
LRU/Cyclic/Random etc.
Hit rate gets better than single DMC's, as replacement strategy becomes more flexible

Locality is the most important notion in caches
Frequently used functions - instructions used in commonly used functions is stored in cache, reduces memory accesses

Inclusive Caches - Entries in lower level caches also exist in higher level caches
Exclusive Caches - If an entry moves from L2 to L1, it is removed from L2

Kommentar [6]: The notion of functions here is difficult as they imply branching which tends to change the area of memory we are operating within

Kommentar [7]: Does anyone have any info on to which has better characteristics / what drawbacks etc.

Last Lecture
Using spatial/temporal locality to maximise cache performance
DMC poor at temporal locality

Cache Control bits
At some point the cache must start empty. We need a "valid" bit for each entry to indicate meaningful data
We need a dirty bit if we are using write back instead of write through

Exploiting Spatial Locality
Storing and comparing address/tag is expensive.
We can use a wider cache line to store more data per address/tag.
Multiple words in the "data" part of DMC

Along with tag and index, we now include an offset in the address, to get the desired word.
The bottom bits of the address are now used to select which word, this can also be used in fully or set associative. We then use a multiplexer, controlled by the word index part of the address to decide which word in the line we want to access.
Data is transferred from RAM in blocks, usually the size of the block is equal to the cache line size. (uses burst mode memory access)

Kommentar [1]: and / or byte / halfword

Spacial locality implies that if we access data, then nearby data will also be accessed. Following this, a larger cache line should yield more hits, but not too big, or we could overwrite useful data in the cache with data that is never used.

The cache itself exploits temporal locality, but if we use a line size of 1 (word), we do not exploit spatial locality.
The graph for miss-rate/line-size is a trough, lowest point around 8.

Separate Instruction & Data Caches
There is an instruction fetch every instruction
Data fetch roughly every 3 instructions
Usually work in separate address spaces
Access patterns are different, so it is better to use separate caches to exploit this. This is called the Harvard Architecture.
Data and instruction caches in parallel between CPU and RAM.

Multi-Level Caches
Bigger caches have lower miss rates, but **always** run **slower**.
L1 Cache should run at CPU speed.
L2 still on-chip, slower and bigger. Usually 16x bigger than L1, 4x Slower than L1.
Vital for modern processors, but introduces complexity in replacement and write policy.

in multi-cores, L1 I+D and L2 usually private for each core. L3 shared.

Kommentar [2]: Leads to coherency problems

Instruction and data cache usually share the L2 Cache

Cache Address Splitting

2-bit align part of address. Redundant, simply used to align.
3-bit word ID, allows to reference 8 words.
18-bit Index
9-bit tag

Kommentar [3]: This is a specific example, not canonical

Kommentar [4]: What if we are specifically selecting bytes

Calculating Perceived Access Time

(hit rate* time taken) + miss rate * (time taken + perceived time at level i+1)

over 1000 accesses
time taken usually cascades. i.e
980*(1) + 18*(1+4) + 2*(1+4+50)
L1 L2 Main

Merge sort - useful to split array into portions the size of the L1 cache and then merge (temporal locality?)

RAID 0 - Striping
Improves performance

RAID 1 - Mirroring
Improves reliability

RAID 2 - Bit-striping + Hamming Code
Striping done at the bit level
Hamming code used for error correction
Extremely high transfer rates possible
No longer used

RAID 3 - Byte Striping + Parity
Striping at the byte level
A1 % A2 % A3 = Ap where % = some operation
Can perform simultaneous reads, as byte positions are synchronized across each disk so we can read several bytes simultaneously.
Extremely high transfer rates possible

Kommentar [1]: Typically XOR, as A1 % A2 % AP = A3 etc

RAID 4 - Block Striping + Parity
Typical usage - Long (stream) accesses
Allows for concurrent reading
Some degradation in short write performance due to having to update the parity

RAID 5
Same as RAID 4, but with distributed parity across all disks, this improves performance

RAID 6 - Double Distributed Parity
Two levels of parity, can tolerate two failures

All RAID forms come with a tradeoff. The idea is to combine multiple RAID configurations to find the most effective method.

Try to keep mirroring as low as possible, to be more tolerant to disk failure.
RAID 0 tends to improve performance

RAID Failure
- RAID 0 - Lose All Data
- RAID 1 - No problem, simply replace failed disk
- RAID 2-6, operate in a degraded mode.

2-6: If a data drive fails, every read must be reconstructed. Parity fail has a low impact on performance. We simply replace the failed drive and reconstruct the array.

Hamming Codes

Think of a venn diagram with 3 circles = parity bits. A1, A0, A2 & A3 are all made via the intersections of each parity bit

Parity

Parity has been around for a long time.
Think of old tape. Parity tracks
Parity can be computed by the XOR of two tracks, and can be reversed to find the value of any track.

WORM - Write Once Read Many (e.g CD-ROM, DVD)
WMRM - Write Many Read Many (Hard Drive & Tape Drive)
WMRM (but not too much) (CD-RW & Flash Drives) - Devices can degrade and fail due to "wear" in components

Hard Disk Drive

- Platters (physical disks) are divided into tracks which are divided into sectors.
- Each platter has it's own Read/Write head.
- All Read/Write heads move in unison as they all operate via a single actuator.
- Moving the read/write head to locate the desired sector is called "seeking"

Performance Attributes

- Seek Time - time taken for the RW head to travel to the required track
- Search Time - Time taken for the target sector to move under the RW head. (also called rotational latency)
- Transfer Rate - Amount of data read/written per unit time.

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

HDD has an internal processor to optimize the schedule.

Search time on average is taken to be the time for half a rotation to complete = (RPM/60)^-1 $(\frac{RPM}{60})^{-1}$

Kommentar [1]: ?
Kommentar [2]: (0.5 * 60) / RPM

File Systems

- Naming Service - Responsible for the naming of files/directories/links
- Storage Service - 'Vector of bytes'?? owners/permissions
- Data/Metadata - Know how a specific file is stored
- Space Allocation - Allocates space in one of several ways. Contiguous/linked list/ indexed
- Recovery Mechanisms

Problems with Disks

- Small
- Slow
- Unreliable

Too Small?

Solved by using multiple disks used either:

- Independently - A/B/C
- Single volume from combined capacity
- Virtualization

Too Slow?
Solved by multiple ways
If the problem is High seek time, use multiple platters
If the problem is High search time, increase Rotation speed
If the problem is low sustained transfer rate, can either increase rotation speed/recording density or apply cache/prefetch principles

Disk Striping (RAID 0)
Split data evenly across all disks to make use of the concurrent RW head movement
Split data into even “stripes”
Creates the illusion of a larger/faster disk.

The downside of this is that reliability scales downwards as disks increase, as losing one disk would result in the loss of all useful data.

Disks are unreliable
- Mechanical components are subject to wear
- We often use helium to reduce resistance
- Partial failure - Losing sectors
- Total failure - lose whole disk

Disk Mirroring (RAID 1)
Write data to every disk
Implements redundancy to increase reliability

Nested RAID (Raid 10)
Stripe across mirrored groups
Failures tolerant as long as we do not lose a whole mirror

Flash Controller

Issues:

- Data Retention of about 10 years
- Wear-out occurs with (Program/Erase Cycles)
- Performance degradation with wearout

Implements:

- Error Correcting Codes
- Block Remapping - Remaps blocks which have worn out
- Wear-Levelling - Avoid wearing out specific blocks by distributing the wear. Remap logical addresses to physical addresses occasionally.

Kommentar [1]: Remap to different logical addresses, to prevent frequently written data being written to the same blocks. You can think of this as trying to make the probability of a physical block being modified follow a uniform distribution.

SSD vs HDD

SSD performance *much* better with random 4KB reads, as there is no seek/search time.
SSD write is usually slower than SSD read due to the Program/Erase cycle.

Storage Virtualization

With a classical file system, there is a FS to HDD mapping which is confined to a single drive.
FS does not span multiple drives.

Logical Volume Management

Virtual mapping between file system code and physical device.
SImilar but not identical to virtual memory addressing.
Implements a “Volume Group” - Pool of drives
The storage space in the volume group is divided into “Physical Extents”, which as a whole make up the “Logical Volume”.

RAID occurs within the LVM layer
We can resize the File System by adding physical extents

Storage Area Networks

Implement LVM features in a separate storage controller
Connect multiple servers to a storage controller
Share disk resources across multiple servers
Rapid migration of disk images.

Storage Area Network (SAN) Controller

Decouple compute servers from storage servers
Connect the two through a network
Must consider bandwidth/latency

SAN Key Features

Key element in System Virtualization
Used in migrating Virtual Machines.

“De-duping” - Share common subsets of file systems e.g. all operating system files / binaries for VMs deployed

Management Features:

- Manage storage separately from server physical resources

ZFS - Volume Aware File System

Lost a file?

- Copy-On-Write
- Simple rollback/recovery.
- Indirect wear-distribution/levelling

Run out of storage?

- Add new storage to live systems
- Self-checking/Self-healing

Hypervisor's Duties

Starting a VM

Gains control (e.g clock tick)
Saves previous VM's CPU registers
Loads next VM's CPU registers
Jumps to next VM's next-PC (in correct privilege state)

Stopping a VM

Saving CPU registers into Hypervisor data area.
Hypervisor starts and stops VM's fairly frequently, to share CPUs and to serialize access to resources

VM state whilst stopped

Memory - (all guest physical memory)
->includes application and OS state
CPU state - registers
Small amount of I/O state. - VM is usually stopped when there is little/no I/O

Freezing a VM

Once suspended, the VM image is self-contained
this can then be copied to a file.

Moving a VM

This frozen image can then be used to to transfer a VM to another piece of hardware. Very useful for locating processes depending on their resource requirements

Snapshot and Rollback

Can take a snapshot image of a VM, which can be used to rollback to that state in the case of a crash. However this can be quite spatially expensive, so it is common to keep a snapshot at time 0, and then store the differences for each subsequent snapshot.

Kommentar [1]: The terms incremental or delta-based may be used here

Archiving VM's

Store a Frozen VM on disk.
Can be used as a "bundle", storing the state of an entire system (inc libs etc), which significantly reduces the time taken to install an OS on a new machine.
Can also be applied to Virtual Applications

Rapid Provisioning

Rapidly deploying a required service by loading a frozen VM (with the required functionality)

Buffer overflow attacks

Overwriting the PC in Kernel space with the address of a shell program, allowing the attacker to get superuser access to a shell

Deploying Secure Desktops
The user can only see their own desktop, in the state in which is was deployed. The hypervisor implements a layer of security between the virtual OS and the actual system calls in the host OS.

Live Migration
Copying a VM from one hypervisor to another at runtime.
Copy every page from source to destination, resetting the dirty bit in the VMMs page table for every page copied.

Load Balancing
Management software that monitors the load of the host machine, and can force migration from one host to another if the load gets too high. Independent of Application and OS.

High Availability
Keep a copy of a high-demand/critical VM on standby on another hardware system. Keep the standby VM updated with the live one. Activate the standby VM if the active VM crashes/stops responding.

System Virtualization - Guest OS runs on top of a layer of software, isolating it from HW

Kommentar [1]: hypervisor

Resource Underutilization

Cost of hardware and cost of energy are reasons to improve the utilization of a system, as running idle is wasteful.
Time sharing is a way of combatting this (multiprogrammed OS/ Virtualization)

System Virtualization

in a virtualized system, a virtual machine monitor (hypervisor) runs as a layer of software on top of the host hardware to provide a platform for multiple guest OS's.
There is a disadvantage in the fact that to get to the hardware, instructions must traverse more levels (VMM, Host OS), which may impact performance.

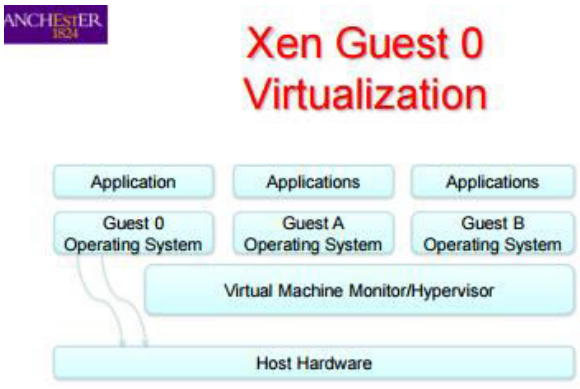
VMM handles the system resources (privileged)
Guest OS is isolated from resources (non-privileged)

Hosted Virtualization

Kommentar [2]: I'm pretty sure this section is wrong. Hosted is where the hypervisor runs as an application on a guest OS, non hosted is where the hypervisor is essentially the OS itself.

Could someone clarify?

Xen is an example.
Hosted virtualization allows for a privileged VM (dom0) to be launched which is the only VM to have access to the host hardware. We then use this privileged domain to manage the hypervisor and launch unprivileged domains



Protection/Privilege

the VMM is in a privileged position, and handles physical resources. The Operating Systems currently running on the VMM are isolated from resources.

What resources are guarded by the VMM?

- Timers - (time slicing mechanism)
- CPU Registers
- Device Control Registers

- Interrupts
- Memory Mapping (page tables)

Virtualized memory access
(guest) OS Page Table and VMM Page Table co-operate, forming a Shadow Page Table.

Interfacing with the VMM

ParaVirtualization - Software Approach (static)
The Guest OS is aware of the virtualization. Calls VMM for all privileged operations, and co-operates with VMM using shared page tables.

Detect and Fix Interfacing - Within VMM (software approach - dynamic?)
We don't want the guest OS fiddling with it's page tables, or performing other unsafe operations (it believes it is in a privileged position). To prevent this we can use Detect and Fix.
Detect - Write-protect guest OS page tables
- Scan the code on the Guest OS

Fix - Use write-error trap to detect guest page-table writes

Detect and Fix Interfacing - In Hardware
Hardware provides application and VMM modes
When virtualization is active, all OS accesses to physical resources trap to VMM

Virtual Memory
Storage Virtualization
Virtual Machines
System Virtualization

Virtualization

Isolates details of hardware from the software that uses it

- VM - Amount of physical memory and layout
- Storage - Position/size/location of virtual disk
- JVM - Instruction set encoding, registers etc
- System - I/O Devices, memory, number CPUs

C++ - Program compiled into Object code (architecture dependent), loaded into memory image
Java - Program compiled into portable/bytecode, loaded by into JVM, which interprets / compiles (JIT) into machine code based on the system's ISA

ISA - Instruction Set Architecture. Instructions + registers that the user can access.

Process and System Virtualization

Process Virtualization - Running a process under the control of a layer of software
System Virtualization - Running the operating system under the control of a layer of software

Virtualization allows to:

- Translate between equivalent facilities (ISA, Libraries, System calls, physical machines)
- Change level of abstraction
- Multiplex/demultiplex resources (hide their physical quantities)

Types of Virtualization

- Multiplexing
- Abstraction - Raise the level of abstraction to avoid looking at low level.
- Translation

Pipeline can cause other problems, not just branches.
ADD R1, R2, R3
MUL R0 R1 R1

New value of R1 will not yet have been committed and updated in the register bank, outdated value. MUL will perform a garbage operation.

One solution is to detect these **dependencies** and insert delays.

Forwarding

Another is forwarding, where we add extra paths for specific cases. Here, we would send the data to be written to the data cache and updated in the register bank, but also create a short route from the output of the ALU, straight back into the ALU. This way, we can use the updated value without skipping a cycle. However, this increases the complexity of the control logic.

Without forwarding we get wasted cycles, as dependencies cause waiting. For example, if an add follows a load, the add gets stuck in the Decode phase whilst the load executes, accesses memory and writes back. 2 wasted cycles.
With forwarding, **the "writeback" stage does not cause blocking**, so the dependent instruction can continue executing after the memory phase, and begin executing during the writeback phase of the independent instruction.

Longer pipelines mean smaller stages, can increase clock frequency, but larger hazards.

Instruction-level Parallelism

Dependencies (blocking waits) are a problem in serial execution.
Method to help approach 1 instruction per cycle. Not possible with serial execution.
If two instructions are **independant** (no dependencies) they can in theory be executed in parallel.

- Need wider instruction fetch, to fetch multiple per cycle
- Multiple decodes per cycle
- Must use common registers - logically the same registers
- Need multiple ALUs for execution
- Must also access common data cache.

Since we now have concurrent operation on the data cache and registry bank, we must increase the access-levels of them both to cope with increased rate of access. We can do this by either duplicating the access circuitry or by duplicating the whole cache structure

This is called a "super-scalar" architecture. 4-way is common

We will need a "dispatch unit" in the fetch stage which uses hardware to examine instructions for dependencies, and only execute in parallel if they are independent.

Instruction Re-order

If two pairs of dependent and dependee, we can re-order to allow for parallel execution. Where i is dependent on j:
Xj, Xi, Yj, Yi
can be re-ordered to
Xj, Yj, Xi, Yi
and execute XjYj and XiYi in parallel, as neither element in a pair is dependent on the other.

Possible exam question is to reorder a set of instructions to exploit superscalar architecture. Firstly draw a dependency diagram, select pairs (triplets or quadruplets depending on the degree of superscalar)

Compiler Optimization

Reordering is often done by the compiler, but may not be able to manage it perfectly
If we can rely completely on the compiler we can get rid of expensive hardware logic.
This is the principle of VLIW (Very Long Instruction Word)
Compiler must add NOPs if necessary, (think parallelism)

There are a few arguments against relying on the compiler

- Legacy binaries - made for a specific hardware
- Code bloat - Useless NOPs

Instead, rely on hardware to reorder instructions if necessary. This is complex but effective

Out Of Order Processors

Uses an instruction buffer - not one by one
Scheduler in charge of sending non-conflicting instructions to execute
Memory and Register accesses need to be delayed until all older instructions are finished to comply with application semantics

Limits of Instruction Level Parallelism

Modern processors are up to 4-way superscalar but rarely reach 4x speed
Hardware complexity
Limited amounts of ILP instructions in real programs

So Far:
Minimize Memory access time impact - **Caches**
Increase Clock Frequency - **Pipelining**
Maximize Pipeline utilization - **Branch prediction/ Forwarding**
Increasing Instructions per clock - **Superscalar**
Maximizing Instruction Issue - **Dynamic Scheduling/OOO Execution**

Software Multithreading - transparent parallelization to the user, but actually just scheduled/interleaved
Parallelism exploitation is limited by the programs
Hardware Multithreading allows different threads to share a processor.

Modern OS's support several processes running concurrently. However this is just transparent scheduling. Not actual concurrency.
Uses context switching to support software multithreading. Save/Load state to switch to another process.

PCB - Process Control Block, stores the information about the state of '**alive**' processes handled by the OS.

Hardware Multithreading
Processor architectural support to exploit instruction-level parallelism
Allows multiple threads to share **a single processor - This is not multi-core/processor**
Requires replicating the independant state of each thread
Virtual Memory can be used to share memory among threads.

HWMT Is usually presented to the OS, by presenting each hardware thread as a virtual processor.
Needs to either share or replicate resources. **Registers are usually replicated**, whereas **caches are usually shared.**
Need to be careful with cache sharing. Each thread use a fraction, but can cause a problem called **cache trashing**.

The problem with trashing is that if two threads are accessing different things with the same index. Results in lots of misses because one is taking the thing out that the other put in. Basically just a problem with the fact that in DMC's, lots of things map to the same area, which has some problems if there are two concurrent processes trying to access the same index.

Coarse-Grain Multithreading

Issue instructions from a single thread at any one time, operating like a simple pipeline.

Switch threads on an “expensive” operation, such as an instruction/data cache miss.
This gives time for the expensive operation to complete, whilst another thread runs.

- Good at compensating for infrequent expensive instructions
- Requires minimal pipeline changes.
- Need to abort all instructions in the shadow of a D-Cache miss
- Short stalls however, such as data/control hazards are not solved.

This approach requires a very fast thread switching mechanism, must be considerably faster than fetching a cache line.

Fine Grain Multithreading

Execution of multiple threads overlap in time.

Typically use Round Robin to cycle among threads in the “ready” state.
This requires instantaneous Thread switching, which can be pretty complex in hardware.

In the event of an Instruction cache miss, we can overcome this transparently, as we can simply move the thread to a “not ready” state, and context switch to a ready process.
Similar with a Data Cache miss. We mark the thread as not ready, and issue only from the other threads

FG Multithreading has a number of advantages.

- Better overall performance, due to increased utilization of pipeline resources
- Impact of short stalls (e.g Data Cache miss) is reduced by executing from other threads

However it requires expensive hardware for instantaneous thread switching

Fine grain multithreading in Out of Order processors

In an OoO processor, we do not need to block a whole thread in the event of a cache miss, as later instructions can be executed (assuming no dependencies)

Simultaneous Multi-Threading

Intended to exploit instruction-level and thread-level parallelism at the same time.
In a superscalar processor, we can issue instructions from different threads at the same time.

Issues

- Asymmetric pipeline stall - If one part of the pipeline stalls, we want the other pipeline to continue
- Overtaking - We want non-stalled threads to make progress.

SMT extracts the most parallelism from instructions/threads. It is only implemented on OoO processors, a only they are able to exploit that much parallelism

However, all this comes at the cost of a significant hardware overhead.

Memory Coherency - Ensure changes are seen everywhere
Memory Consistency - Ensure correct ordering of memory accesses.

- Barrier - synchronizes multiple threads - Must wait for all threads to reach the barrier
- Fence - within a single thread, don't let some instructions run ahead of others
- Lock - bottom-level primitive

Snooping
Each cache listens for activity concerned with it's cached address, as knowing which other cores contain copies of cached data is too expensive.
Normally implemented using a common bus, implements centralization

Write Update
When a core wants to update a value, it broadcasts the address and new data as it updates it's own copy
Other listening cores hear this and update their own local copies.

Write Invalidate
A core wanting to write to an address updates it's own copy and broadcasts that the specified address is now "invalid", and other cores mark the address as invalid. Any shared read in other cores will now miss.

In both above cases, the problem of simultaneous writes is solved by making the bus restricted to only one core at a time.
Invalidate uses significantly less bus bandwidth. Bus bandwidth is a precious resource in multi-core chips, so the invalidate method tends to be preferred.

Issues
If we know that an address is not shared, we can save bandwidth by avoiding broadcasting any messages relevant to this address.

MESI Protocol
Invalidate cache-line state protocol
Stores some extra information about the cache line. We know whether addresses are shared or exclusive, and whether the memory has been updated (dirty bit).

Cache line states
- **M**odified - Dirty & Exclusive
- **E**xclusive - Same copy as memory, but also exclusive
- **S**hared - Same value as memory, but shared between other cores
- **I**nvalid - Line data is not valid

Cache line state changes occur with memory accesses
RWITM message - Write with intention to modify

MESI Local Read Hit

Line must be one of MES
Must be the correct local value (if M, it must have been modified locally)
Simple “if...return”

MESI Local Read Miss

No other copies: Read from memory
One core with E: Memory access is cancelled. Snooping cache puts value on bus, both set to “S”
Several cores with S: Memory access cancelled, any cache puts value up, local cache updates and sets to “S”
One core with M: Memory access is cancelled. Cache with “M” puts its value on the bus, and then writes the value back to memory. Both caches now set to “S”

MESI Local Write Hit

M - No change, as this is the only copy.
E - Set to M.
S - Set to M and broadcast an invalidate on the bus. Each cache with a copy sets their value to ‘I’.

MESI Local Write Miss

Another core has M: we request a copy from it, and the other core takes the opportunity to also copy back to memory.
No other copies - Read from memory and broadcast “RWITM”. Set to M.
Other copies (either S or E) - Broadcast RWITM. All snooping cores set their tags to “i”, and the local copy is updated and tag set to “M”.

MOESI Protocol - Extra state “owned” to avoid the need to copy back to main memory.

However, bus-based protocols do not scale well past a small number of cores.

Directory-based Protocols

Uses distributed hardware. Each cache is equipped with a directory which stores information about what is currently being shared.
Coherence messages are point to point, using an on-chip network.

3 states for cache line - ISM
3 states for directory -

- NC - Not cached (this cache line is not present)
- S - Shared (not modified and is in at least one core)
- M - Modified - Modified and in exactly one core

Directory will contain memory line, a description of whether the memory line is present in any cache (see above) and then a binary string showing which cores have copies of the data.

The directory is still a centralized point. Unless we use distributed directories. Where each core has part of the directory.
This adds a need for a “home” built into the cache.
Distributed caches get rid of the bottleneck caused by having a centralized directory.

False Sharing

A form of trashing caused by two caches accessing different words on the same cache line. If one writes/updates it invalidates the other.

Core - Execution Unit

We have reached the end of the goodtimez. We can no longer extract as much performance from a single-processor architecture.

We're slowing down for several reasons

- Power density increasing - cooling is a problem
- Small transistors are less predictable (quantum mechanics)
- Architectural innovation hitting design complexity problems
- Memory isn't keeping up with processor speeds "The Memory Wall"

Improving Single-core performance at an architectural level

- Caches - Minimize memory access costs
- Pipelines & superscalar - increase amount of potential parallelism
- OOO processing and multithreading increase the amount of independent instructions per cycle
- Branch prediction/forwarding/register renaming

However all of these have limited scalability.
The solution to this is Replication.
Put multiple CPUs on a single chip. Using them in parallel will achieve higher performance.
However it's not all that easy.
Not all programs can run things in parallel. There is also the issue of communication between the chips.

How Do We Connect Them?

Could have independent processor/store pairs with an interconnection network.
This is called distributed memory. Each core has it's own memory, and communication/synchronization must be programmed explicitly

Shared memory approach - all cores share a common memory. If one processor uses memory, all cores can see it. This allows for implicit communication. Shared memory is the common approach

Multiple Cores

Small numbers of cores can be used for separate tasks (run O.S on one core, browser on another etc.)
If we want increased performance on a single application we need to move to parallel processing.
Multi-core programming is an extremely important field and is here to stay. There is no alternative.

In traditional structure, there is a “North Bridge Chip” connecting main memory, Graphics card and CPU/Caches.
However in a more recent multi core architecture, the CPU is directly connected to memory. CPU communicates with externals using an I/O HUB (GFX card connected directly to this via PCIe) which is connected to I/O Controllers, which I assume are used to communicate with all other peripherals.

Data Coherency and Consistency

Processes - Operating System-level processes. Separate applications. Do not share any data and no parallelism required
Threads - Parallel parts of the same application, sharing an address space. This is where the problems lie, with shared memory we gain the problems of memory **coherence** (ensure changes are seen everywhere) and memory **consistency**, ensure correct ordering of memory accesses.

Memory Coherence

If one core updates its cache, it immediately puts all other cache’s out of date. Even if we write-through to memory, we must still ensure that the other cores see the update

Memory Consistency

Multiple copies are consistent if:

- A read operation returns the same value for all copies
- A write operation completes before any other operations take place.

Sequential Consistency

Operations in place to allow sequential execution
Fence - All memory accesses before the fence need to complete before starting to execute the ones after
Barrier - All threads need to reach the barrier before any of them can continue execution
Lock - Only one thread can proceed to the atomic section. Very important synchronization method, as others can be built on top of it.

Consistency - Synchronization within the core (correct ordering of messages)
Coherence - Synchronization between cores (Changes are seen everywhere)

Independent Process/Store pairs guarantee no coherence at the processor level.

Parallel computing requires some form of message passing to allow synchronization.

Important features of NoC (Network on chip)
Topology - How the cores and network elements are arranged
Routing - How traffic moves through the topology
Switching - How traffic moves from one component to the next

Bus Topology
Common wire connection - Broadcast medium, all cores use it
Only a single component can use the bus at a time, must “Grab a slot”
Timing slots controlled by a clock
Often uses a “split transaction” - Request data in one slot, and memory returns the data in another later slot.

Crossbar Topology
Connects N inputs to N outputs, looks like a grid
Achieves some concurrency
Requires a fairly large spatial area for this, so isn’t too common.

Tree Topology
Switched by hubs, with variable latency between leaves (due to arbitrary number of hubs to travel)
The problem with this topology is the variability and the unreliability of the topology. (Points of failure)
To combat this we instead produce “fat trees” where there are multiple top-level hubs, creating a sort of web. This however, gives variation, as there are now multiple routes between nodes.

Ring Topology
Each node has an integrated router
Simplest topology. Fairly common (used in PS3)
However, this doesn’t scale well at all, as each new node will increase latency.

Mesh/Grid topology
Convenient for spatially large systems
Reasonable bandwidth/latency, although latency is variable.
Also utilizes in-built routers at each node

NoC Routing

- Minimal Routing - Selects shortest distance/route. Packets always move closer to the destination, move closer each clock cycle. Risk of packet-blocking.
- Oblivious Routing - Fixed path. Simple the implement, but prone to **contention,** due to packets competing for links
- Adaptive Routing - Similar to oblivious, but we give some intelligence to the routers, allowing them to adapt packet routes to avoid contention.

NoC Switching

- Packet Switching - Split into packets, add header, allows time multiplexing.
- Store and Forward switching - Packets are not forwarded until all components of the packet have arrived. This allows for live failure detection, but this increases latency
- Wormhole Switching - Packets can be forwarded as soon as the head arrives. This requires nodes keeping information about the routes to send future packets.

Amdahl's Law

$$\text{Speedup} = S + P/ (S + (P/N))$$

S = fraction of code that's serial
P = Fraction of code that is parallel
N = Number of processors

Design Objectives

Execution Speed
Execution bandwidth (most tasks per unit time)
Reliability
Reduce Power

Flynn's Taxonomy Of Processors

SISD - Single Instruction Single Data (Single processor)
SIMD - Single Instruction Multiple Data (vector processor. e.g GPU)
MISD - Multiple Instructions Single Data
MIMD - Multiple Instructions Multiple Data - Multi-core

Types of Processors

RISC

Reduced instruction set. Small number of extremely fast instructions. Complex instructions made from smaller instructions. Fine Granularity

CISC

Complex Instruction Set, Lots of instructions (typically hundreds). Slower execution, but much more work done per instruction.

In a classic 5-stage pipeline, all data follows the same path
In modern processors there are many execution flows (Multiple pipelines) with different functionalities (called **functional units**).
Different functional units can have different lengths.

This introduces a new problem. What if two things want to use the same functional unit? They must wait.
Also, what happens if the first instruction goes down a long path, and the second a shorter path? They may finish out of order

Out Of Order Execution
Original order is not preserved
Instructions are executed as they become available.
This is done by taking advantages of ILP, increases number of instructions per cycle.

Hazards
Cache Misses - Long wait
Structural Hazard - When a Functional Unit cannot be used by more than one instruction. Any new instructions are forced to wait
Data Hazard - Dependencies between instructions

Data Hazards
True Dependency - RAW (Read after Write)
Anti-Dependency - WAR (Write After Read) - Writing into a register that another instruction uses as an operand
Output Dependency - WAW (Write after Write) - Writing to the same register

Dynamic Scheduling
Allow instructions behind a stall to proceed.
Makes use of the multiple execution units by utilizing parallelism

Score Board
Centralized hardware mechanism. Keeps track of all inputs and outputs
We can execute an instruction once all of its operands are available, and there are no hazards
Dynamically constructs dependency graph for a window of instructions
Scoreboard is a data structure.

Out of order execution divides the decode stage

- Issue - Decode instructions and check for structural hazards
- Read Operands - Wait until no data hazards, then read operands

Scoreboards allow an instruction to execute whenever both stages hold.

Scoreboard exists between the register bank and the functional units (pipelines)

Issue
Check if a functional unit for the instruction is free (no structural hazards) and no other instruction has the same register destination
If a structural or WAW hazard exists, execution is stalled until these are cleared

Basically check we are safe to use this FU to write to this register

Read Operands
Wait until no data hazards then read operands.
A source operand is available if no earlier issues active instruction is going to write it

Check for dependencies and ensure there are no RAW hazards

Execution
Fire the instruction through the FU, inform the scoreboard upon completion

Write Back
Once scoreboard is aware that the functional unit has completed execution, it checks for WAR hazards. If none, it writes results to the register bank. If WAR, stalls execution

Scoreboard Contents

1. Instruction status - In which of the above 4 stages is the instruction currently in.
2. Functional Unit status - Indicates state of the FU

Functional Unit Status
Busy - Is FU active?
Operation to perform (+/-)
Fi - Destination Register
Fj, Fk - Source registers
Qj, Qk - Functional units producing the source registers (used to check for dependencies)
Rj, Rk - Flags indicating whether Fj, Fk are ready

3. Register result status - indicates which functional unit will write to the registers

In the FU Status part of the scoreboard, there is also a “timer” for each functional unit, describing the number of cycles left until execution completes

Tomasulo's Algorithm

Control logic for the out of order execution is decentralized
Reservation stations in functional units keep instruction information - similar to scoreboard
In addition to RS, seamlessly rename registers which removes the WAW and WAR hazards of the scoreboard.

Reservation Stations basically act like an internal scoreboard for each Functional Unit

A common data bus. When we finish the execution in one of the functional units, we broadcast the results to the different devices

Distributed control **allows for a larger window of instructions**. This allows for more flexible dynamic scheduling
A problem with this is that due to there only being one bus, only one functional unit can finish at once.

Structural hazards stall the pipeline.
The RS tracks when operands are available and buffers them as soon as they are. (this removes the need to access the register bank)
the impact of RAW dependencies are limited
WAW and WAR dependencies are completely avoided.

Register Renaming

Eliminates WAR and WAW hazards by renaming destination registers. This can be done by the compiler, but the Tomasulo algorithm does it transparently in hardware.
This technique means we become only conscious of true dependencies.

Three stages of Tomasulo

1. Issue - Get instruction from FP Op Queue
2. Execute - operate on operands (EX)
3. Write Result - Finish execution (WP)

Normal data bus = data + destination
Common data bus = data + source

Reservation Station Components

Op = Operation to perform
Vj, Vk = Values of operands
Qj Qk = Reservation stations producing the two source registers
Busy - Indicates whether the FU is busy

Register Result status table used to check for dependencies

Tomasulo Advantages

- Distributed Hazard detection logic - The CDB allows for us to broadcast when operands become available, meaning each FU does not have to personally poll the register bank. This allows for simultaneous dispatch of multiple instructions (assuming they all already have the other operand)
- Avoids stalling due to WAW and WAR hazards, as instructions are always issued in the correct order.

Tomasulo Drawbacks

- Complex Hardware
- Performance is limited by the Common Data Bus. If we only have one CDB, only one functional unit can complete per cycle

Pipelining - vital in modern processors.
Improves performance and efficiency

Fetch Execute cycle - two box model, CPU and Memory. Simple repetitive cycle

Fetch
- Get from memory
- Decode instruction and select registers

Execute
- Perform operation or calculate address
- Access operand in data memory
- write result to a register

Driven by CPU clock. A single operation should take one clock cycle, with each step taking ⅕ of a clock cycle

Kommentar [1]: Assuming the 5 stage pipeline

Fetch-Execute Subdivided stages

Instruction fetch - Done on Instruction Cache
Instruction Decode - Register bank, select correct register to use
Execute - Done in ALU
Memory access - Data cache
Write back

Buffering (pipelining)

Think of black boxes, each does its work then passes onto next
Performing each step in a single cycle is very wasteful
A buffer could be inserted between each box to hold state, updated once per cycle
This is a **pipeline processor**
This allows for us to increase the clock speed by 5x, as one instruction is still being executed per cycle

in a 100 stage processor, with 1GHz clock speed, it takes (1000 + 99)*10^-9s to execute 1000 instructions. 99 cycles per instruction

More stages in pipeline allows a greater clock frequency, as the critical path is shorter between buffers. The main factor in the number of stages a processor can accommodate is the power dissipation. Tends to be limited at ~100 stages.

Kommentar [2]: Not that a pipeline this deep is practical

The Control Transfer Problem

Obvious fetch order is serial.
What if we fetch a branch?
In a serial pipeline, we have already fetched it and don't know it is a branch until the second stage, and the next instruction has already been fetched.

Kommentar [3]: An instruction which may modify the flow of execution. Higher order statements like loops, ifs, switches, procedure calls are translated to these (+ extra supporting instructions)

We solve this by “marking” the next instruction as unwanted, so it is ignored when it reaches the execute stage. (called a bubble)
This method wastes a cycle. Bubbles are called “Control Hazards”, they occur when it takes one or more pipeline stages to detect a branch. Larger pipelines usually suffer more from control hazards.

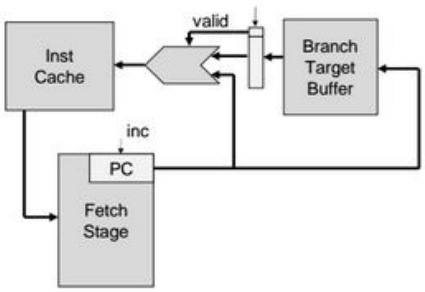
Kommentar [4]: Not all of them surely?

What if the branch is conditional?
We may not know the outcome of the conditional branch until the execute stage. In this case we may have 2 bubbles.

Branch Prediction
A branch is usually executed multiple times, and instructions will usually be at the same virtual address in memory, so if we remember its address, and remember what is executed next, we avoid bubbles.

If we predict a branch and don’t actually branch, we have to **flush** the pipeline.

To avoid this we use a **Branch Target Buffer.** The first time we execute a branch, we create an entry in the BTB, storing both the address of the branch, and the address of the target. The BTB is basically a cache.
So next time we execute the branch, we check for a hit in the cache, if so we can use it to predict the branch. However, this doesn’t work as well for conditional branches, as there are multiple potential target instructions, which does not work in a hash-like structure. Fixed probability of success, **failure results in a bubble**.



Branch Target Buffers are simple to understand but expensive to implement. In practice, branch prediction depends on more history, and also on the context of the branch.

```
while(true)
```

```
      for(i<x)
            do_stuff
      end for
      (conditional branch here, continue for, or loop to while, depending on
i)
end while
```

No Branch Prediction - No prediction possible in main loop, must pay 3 cycles. In the inner loop, when x=2, 50% chance of paying 3 cycles.

Branch Prediction - Only pay 3 cycles on first cycle of main loop, then we have an entry in the BTB as it is unconditional. When x=2 must pay penalty 100% of the time in conditional loop. with x=100, we have only a 2% probability of paying the penalty.

Above shows that for small x, branch prediction does not help much. However, with a value of x similar to 100, we only have to waste 6 cycles (2 bubbles). This is because we gain 3 bubbles to fill the buffer, and then another 3 when the loop exits.

WIth Branch Prediction, bubbles seem to mainly occur at the start and the end of a loop

COMP25212 - System Architecture

Caches

Cache	A small amount of highly fast memory used as temporary storage for frequently used memory locations (instructions and data)
Direct-Mapped Cache	A cache structure in which each memory location is mapped to exactly one location in the cache. Many addresses can potentially map to the same cache line. (Sort of like a Hash Table implemented in memory)
Fully Associative Cache	A cache structure in which a block can be placed in any location in the cache
Set-Associative Cache	A cache that has a fixed number of locations (at least two) where each block can be placed
Tag	A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word
Valid Bit	A field in the tables of a memory hierarchy that indicates that the associated block contains valid data
Temporal Locality	The reuse of specific data and/or resources, within a relatively small time duration (if at one point a particular memory location is references, it is likely that the same location will be referenced again in the near future
Spatial Locality	The use of data elements within relatively close storage locations (if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future)
Cache Miss	A request for data from the cache that cannot be filled because the data is not present
Write-Through	A scheme in which writes always update both the cache and the main memory (or next lower level memory) to ensure consistency
Write Buffer	A queue that holds data while the data is waiting to be written to main memory
Write-Back	A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to main memory (or next lower level memory) when the block is replaced. A caching method in which modifications to data in the cache aren't copied to the cache source until absolutely necessary.
Compulsory Miss	A cache miss caused by the first access to a block that has never been in cache
Capacity Miss	A cache miss that occurs because the cache (even with fully associative) cannot contain all the blocks needed to satisfy the request
Conflict Miss	A cache miss that occurs in a set-associative/direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully

	associative cache of the same size
DMA	Direct memory access, A mechanism that provides a device controller with the ability to or from the memory without involving the processor

The Importance of Caches

- The processor can only make calculations as quickly as it can be fed data - If the processor doesn't have any data to calculate on a cycle, the cycle is missed and it won't occur till the next cycle
- The job of a processor cache is to eliminate as many wasted cycles as possible
- The cache can significantly improve system performance by making sure the CPU has data to process on as many cycles as possible

Direct-Mapped Cache

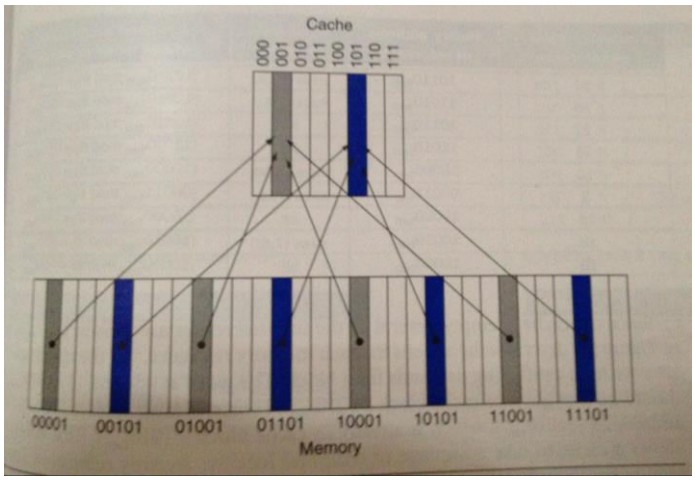
- *We assign a location in cache based on the **address** of each word in memory*
- *Each memory location is mapped directly to exactly one location in cache*
- *The way they are mapped to find a block is typically:*

(block address) % (number of blocks in cache)

Example
The above formula can be calculated using log_2.

An 8-block cache uses the three lowest bits ($8 = 2^3$) of the block address.

Memory locations between 1(00001) and 29(11101) map to locations 1(001) and 5(101) in a direct-mapped cache of eight words.



- *Each cache location can contain the contents of several memory locations, so we have to know whether the data in the cache corresponds to a requested word*
- ***Tags** contain the address information required to identify whether a word in the cache corresponds to the requested word*
    - *It contains the upper portion of the address (so in the above example the upper 2 bits of the 5 total will be the tag)*

- *We also need a way to recognise that a cache block does **not** have valid information*
- *We add a **valid bit** to indicate whether an entry contains a valid address*
    - *If the bit is not set, there is no match for this block*

Fully-Associative Cache

- *For this design, a block can be placed in any location in the cache*
- *A block in memory may be associated with any entry into the cache*
- *To find a given block, all entries in the cache must be searched because a block can be placed in any one*
- *Searching is done in parallel with a comparator associated with each cache entry, which significantly increase hardware cost*
    - *Therefore, FA caches are only practical for caches with small numbers of blocks*

Set Associative Cache

- *This design is somewhat a middle-ground between the two above*
- *There are a **fixed number of locations where each block can be placed***
- *A SA cache with n locations for a block is called an n-way SA cache*
    - *An n-way SA cache consists of a number of sets, each of which consists of n-blocks*
    - *Each block in memory maps to a unique set in the cache given by the index-field, and a block can be placed in any element of that set*
- *So, an SA cache pretty much combines DM and FA placement*
    - *(A block is directly mapped into a set, then all blocks in the set are search for a match)*
- *Formula: (block number) % (number of sets in the cache)*

Accessing a Cache

- *A referenced address is divided into:*
    - ***Tag Field:** used to compare with the value of the tag field of the cache*
    - ***Cache Index:** used to select the block*
- *The index of the block with the tag contents uniquely specifies the memory address of the word contained in the cache block*
- *(Because the index field is used as an address to reference the cache, and because an n-bit field has 2^n values, the total number of entries in a DM-cache will be a power of 2)*

Cache Writes

Write-Through

- *By writing data into the cache, memory will have a different value from the cache, which is said to be **inconsistent***

- *To keep consistency, the data is always written into both memory and cache **(write-through)***
- *If dealing with a cache-miss:*
    - *Fetch the word of the block from memory and place into cache*
    - *Overwrite the word that caused a miss into the cache block (and also write the word to main memory using the full address)*

Write Buffer

- *Because of constant writing to memory, performance would soon be reduced, so to avoid this, we would use a **write-buffer:***
    - *A WB stores the data while it is waiting to be written to memory*
    - *After writing the data into the cache and WB, the processor can continue execution*
    - *When a write to main memory completes, the WB is freed*

Write-Back

- *When a write occurs the new value is written only to the block in the cache*
- *The modified block is written to the lower level of the memory hierarchy when it is replaced*
- *Can improve performance when processors generates writes as fast/faster than the writes can be handled in main-memory (downside of write-through)*

Cache Misses

Conflict Misses

- *Misses occurring in set-associative/direct-mapped caches when multiple blocks compete for the same set*
- *Increasing associativity reduces conflict misses but, however, may slow access time leading to lower overall performance*

Capacity Misses

- *Misses caused when the cache cannot contain all the blocks needed during execution of a program (often occurs when blocks are replaced and then later retrieved)*
- *Can be reduced by enlarging the cache - but when making caches larger, we have to be careful about increasing access time, which could lead to lower overall performance*

Compulsory Misses

- *Misses caused by the first access to a block that has never been in the cache*
- *Because the miss is generated by the first reference to a block, the primary way for the cache system to reduce the number of compulsory misses is to increase the block size*

- *This reduces the number of references required to touch each block of the program once, because the program will consist of fewer cache blocks*
- *However, increasing the block too much will again affect performance because of the increase in the miss penalty*

Cache Coherency with DMA

- Caches hold a copy of sections of memory close to the CPU - the simplest cache strategy assumes that memory is only accessed by the CPU and is always consistent

Problems

1. If we consider a read from disk that the DMA unit places directly into memory - if some of the location into which the DMA writes are in the cache, the processor will receive the old value when it does a read.

2. If the cache is write-back, the DMA may read a value directly from memory when a newer value is in the cache and the value has not been written back (which is the coherence problem).

Solutions

1. *A snooping device may be placed on the memory bus to check the DMA operations and update the memory and cache as needed*
2. *A second approach is to have the OS selectively invalidate the cache for an I/O read or force write-backs to occur for an I/O write (this is called flushing)*

Storage

Seek Time	The time taken for a hard disk controller's head to locate a specific piece of stored data (the target track)
Search Time / Rotational Latency	The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time
Transfer Time	The time taken to transfer a block of bits; it is a function of the sector size, the rotation speed and the recording density of the track

RAID	An organisation of disks that uses an array of small inexpensive disks so as to increase both performance and reliability.

Disk Read Time

Formula: *Av. seek time* + **Av. rotational latency* + *Transfer timeu*

**Av. rotational latency = 0.5 rotation *60 / RPM*

Example of Disk Access Time

Seek time: 8.5ms

Search time: 7200RPM → $\frac{0.5\ rotations \cdot 60\ secs/min}{7200RPM}$= 4.16ms

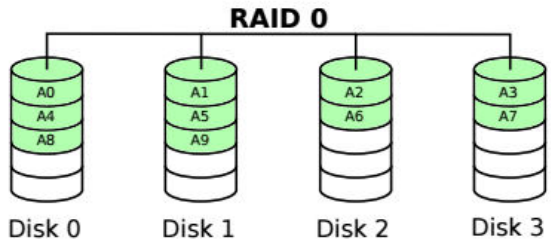
Transfer time: $\frac{512B}{177 \cdot 10^6 bytes/sec}$= 2.89$\mu s$

Access time = 8.5 + 4.16 + $2.89 \cdot 10^{-3}$ = 12.66ms

RAID

No Redundancy (RAID 0)

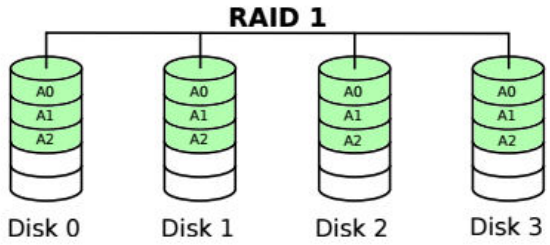
- Spreading data over multiple disks (striping) automatically forces accesses to several disks
- Striping across a set of disks makes the collection appear to software as a single large disk, simplifying storage management
- Improves performance for large accesses since many disks can operate at once



lik

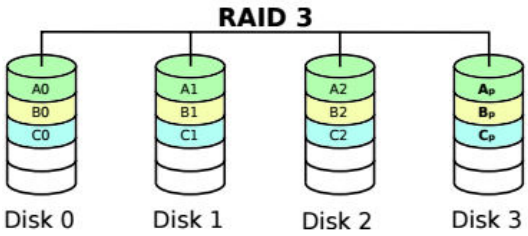
Mirroring (RAID 1)

- A scheme for tolerating disk failure - uses twice as many disks as RAID 0
- Whenever data is written to once disk, that data is also written to a redundant disk (always making 2 copies available)
- If a disk fails, the system goes to the mirror
- Mirroring is the most expensive RAID solution (as it requires the most disks)



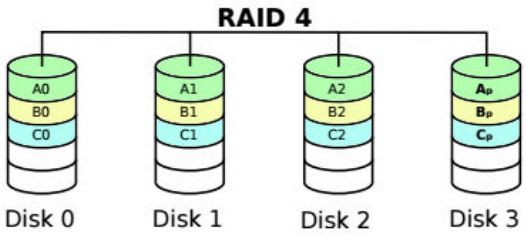
Bit-Interleaved Parity (RAID 3)

- The cost of availability can be reduced to 1/n (where n is the number of disks in a protection group*)
- Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure
- Reads/writes go to all disks in the group, with one extra disk to hold the check information in case of failure
- The redundant disks have a sum of all data in the other disks - when a disk fails you can subtract all the data in the good disks from the parity disk; the remaining information will be the missing information (Parity = sum % 2)



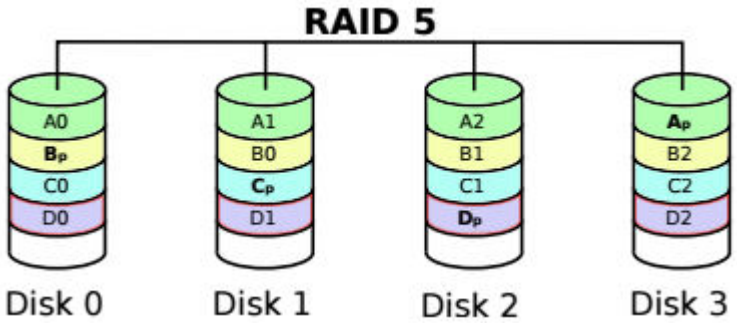
Block-Interleaved Parity (RAID 4)

- Uses same ratio of disks/check disks as 3, but access data differently:
    - Parity is stored as blocks and associated with a set of data blocks
- In 3, every access went to all disks - some applications prefer smaller accesses, allowing independent accesses to occur in parallel
- Error detection information in each sector is checked on reads to see if the data is correct - “small reads” to each disk can occur independently as long as the minimum access is one sector
- A small access goes to just one disk in a protection group* while a large access goes to all the disks in a protection group
- The parity is a sum of information - by watching which bits change when we write the new information we need only change the corresponding bits on the parity disk



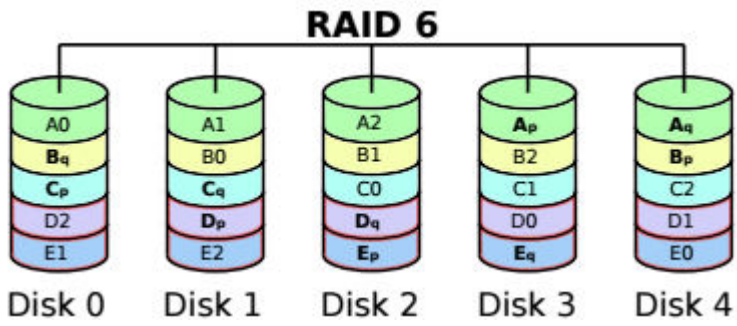
Distributed Block-Interleaved Parity (RAID 5)

- In 4, the parity disk must be updated every write, so there is a bottleneck for back-to-back writes
    - *To fix this bottleneck, the parity information can be spread throughout all disks so that there is no single bottleneck for writes*



P + Q Redundancy (RAID 6)

- Parity based schemes protect against a single self-identifying failure - when a single failure correction is not sufficient, parity can be generalised to have a second calculation over the data and another check disk of information
- The second check block allows recovery from a second failure - the storage overhead is twice of RAID 5



Virtualization

Process Virtualization	Running a process under the control of a layer of software, e.g. JVM
System Virtualization	Running an OS under the control of a layer of software, e.g. VMWare
Rapid Provisioning	Provides a method for deploying new virtual machines to storage arrays without the requirement for copying virtual machines over the network
Live Migration	The movement of a VM from one physical host to another whilst still being powered up - the process takes place without any noticeable effect for the end user

Process Virtualization

- Running a process under the control of a layer of software, e.g. JVM
- A process Virtual Machine is capable of supporting an individual process
- The virtualizing software is placed in an ABI (application binary interface), on top of the OS/HW combination
    - The software emulates both user-level instructions and OS system calls
    - The VS (or 'runtime') is created to support a guest process and runs on top of an OS
    - The VS supports the guest process as long as the guest process executes/terminates support when the guest terminates

System Virtualization

- A system Virtual Machine provides a complete system environment, which can support an OS along with its potentially many user processes
- It provides a guest OS with access to underlying HW resources such as: networking, I/O, display, GUI etc.
- The VM supports the OS as long as the system environment is still alive
- Virtualizing software is placed between the underlying HW machine and software

Live Migration

- The movement of a VM from one physical location to another while still being powered up (minimal noticeable effect from user's point of view)
- You copy pages from 'source' to 'destination' machine and reset the dirty bit in the VMM's page table for every copied page
- You repeat this stage until there is a minimal 'subset' of pages left for transfer
- Finally, suspend the source VM, copy the remaining pages to destination and resume the VM at the destination machine

Rapid Provisioning

- You can instantiate a new VM from a stored, pre-configured image

- You store a VM image with an installed OS and applications that have already been configured - so only need to be deployed on a VMM and are ready to run
- Example: You can configure a web server once in a VM, then freeze the VM and copy the image, then restart the VM on any VMM to provide the service

High Availability

- For critical applications, keep standby VM available on a different HW system
- Regularly copy archive VM image to standby VM (but don't activate)
- Activate if active VM stops responding

Checkpoint and Restore

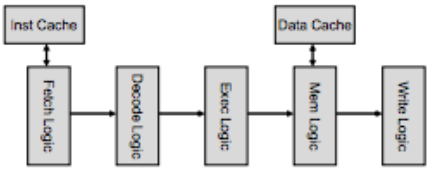
- The ability to take a 'snapshot' of the state of an application and later restore the application to a running state (possibly on different VM's)

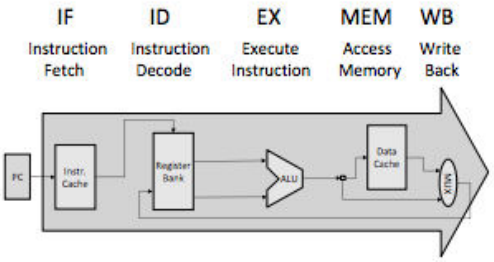
Pipelines

Control Hazard	When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed - the flow of instruction addresses is not what the pipeline expected
Data Hazard	When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available
Structural Hazard	When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute
Superscalar	An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution

Benefits to Pipelining

- It provides a higher throughput of a classic system - when there are many instructions to be executed, the throughput decreases the total time to complete the work





The Classic 5-Stage Pipeline

Fetch Logic (IF): *Fetch instruction from memory*
Decode Logic (ID): *Read registers while decoding the instruction*
Execute Logic (EX): *Execute the operation or calculate an address*
Memory Logic (MEM): *Access an operand in data memory*
Write Logic (WB): *Write the result into a register*

Control Hazards

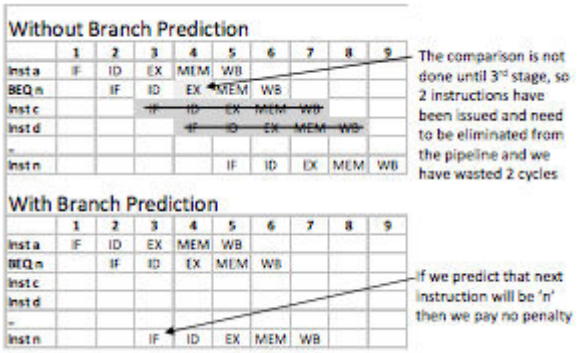
<u>Problems</u>

- If fetching instructions in serial program order (incrementing the PC) and fetch a branch, we only know it's a branch when we decode it in the 2nd stage - by that time we're already fetching the next instruction in serial order.
- If we have a conditional branch, we might not be able to determine the branch outcome until the 3rd stage - we would have '2 bubbles' (this can be avoided by reading registers during the decode stage)
- Bubbles due to branches are usually called **Control Hazards**
    - Occurs when it takes one or more pipeline stages to detect the branch
    - The more stages, the less each does and more likely to take multiples stages

- Longer pipelines usually suffer more degradation from control hazards

Branch Prediction

- Most programs execute many branch instructions (and instructions will be the same virtual address in memory)
- When a branch is executed we could:
  - Remember its address
  - Remember the address that was fetched next



Data Hazards

- Occurs when the pipeline must be stalled because one step must wait for another to complete
  - *ADD R0, R1, R2*
  - *SUB R3, R0, R4*
  - *The ADD instruction doesn't write its result until the 5th stage, meaning we would have to waste three clock cycles in the pipeline*
- **Solution:** We don't have to wait for the instruction to complete before trying to resolve the data hazard - (For the code sequence above), as soon as the ALU creates the sum for the ADD, **we can supply it as an input for the subtract**
  - *(Adding extra HW to retrieve the missing item early is **forwarding**)*

True dependency

r1 <- r2 op r3
r4 <- r1 op r5

Read-after-write
RAW

Anti-dependency

r1 <- r2 op r3
r2 <- r4 op r5

Write-after-read
WAR

Output dependency

r1 <- r2 op r3
r1 <- r4 op r5

Write-after-write
WAW

Data Dependencies

- **Key Idea: Allow instructions behind stall to proceed. => Instructions executing in parallel. There are multiple execution units, so use them**

DIVD F0, F2, F4
ADDD F10, F0, F8
SUBD F12, F8, F14

Even though ADDD stalls, the SUBD has no dependencies and can run.

– Enables out-of-order execution => out-of-order completion

Dynamic pipeline scheduling overcomes the limitations of in-order pipelined execution by allowing out-of-order instruction execution

Stages of a Scoreboard Pipeline

1. **Issue:** ***Decode instructions & check for structural / WAW hazards***
    a. If a functional unit for instruction is free *(no structural hazards)* and no other active instruction has the same destination register *(no WAW),* the scoreboard issues the instruction to the FU and updates its internal data structure
    b. If a structural / WAW hazard exists, the instruction issue stalls, no further instructions will issue until these hazards are cleared

2. **Read Operands:** ***Wait until no data hazards, then read operands***
    a. A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit (no RAW)
    b. When the source operands are available, the scoreboard tells the FU to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order

3. **Execution -** ***Operate On Operands***
    a. The functional unit begins execution upon receiving operands; when the result is ready, it notifies the scoreboard that it has completed execution
4. **Write Result -** ***Finish Execution***
    a. Write result —finish execution. Once the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction

Multithreading

Multithreading	Increasing utilization of a processor by switching to another thread when one thread is stalled
Fine-grained Multithreading	A version of multithreading that suggests switching between threads after every instruction
Coarse-grained Multithreading	A version of multithreading that suggests switching between threads only after significant events, such as a cache miss
Simultaneous Multithreading	A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture

Multithreading

- Allows multiple threads to share the FU of a single processor in an overlapping fashion - processor must duplicate the independent state of each thread
- Each thread would have a separate copy of the register file and PC - the memory itself is shared through the VM mechanisms *(also shares a fraction of the cache)*
- The hardware must support the ability to change to a different thread quickly

Fine-grained Multithreading

- Switches between threads on each instruction, resulting in interleaved execution of multiple threads - this is often done in a round-robin fashion, skipping any threads that are stalled at that time *(must be able to switch threads on every clock-cycle)*
- +ve: It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls
- -ve: It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by other threads

Coarse-grained Multithreading

- Switches threads only on costly stalls, such as I/D-cache misses
- +ve: This change relieves the need to have thread switching be essentially free and is much less likely to slow down the execution of a thread, since instructions from others will only be issued when a thread encounters a costly stall
- -ve: It is limited in its ability to overcome throughput losses, especially from shorter stalls - this is because of pipeline start-up costs of CGM
    - Because a processor with a CGM issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen

Simultaneous Multithreading

- Uses the resources of a multiple-issue, dynamically scheduled processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism
- Multiple-issue processors often have more FU parallelism available than a single-thread can effectively use

Multicore

IPC Instruction per cycle

MESI:
Modified - This core has the only copy in cache and it has been modified
Exclusive - This core has the only copy in cache and it is consistent with memory
Shared - Multiple cores have a copy in cache and they are consistent with memory
Invalid - This core does not have a valid copy in cache

RWITM **R**ead **W**ith **I**ntent **T**o **M**odify