Evolution of cloud computing

1. Virtualization (own servers)
2. Hardware on Demand (AWS)
3. "Serverless computing"

Datacenter

- Large collection of commodity components
- High Bandwidth networking gear
- Distributed computing software

Why 2-socket CPU

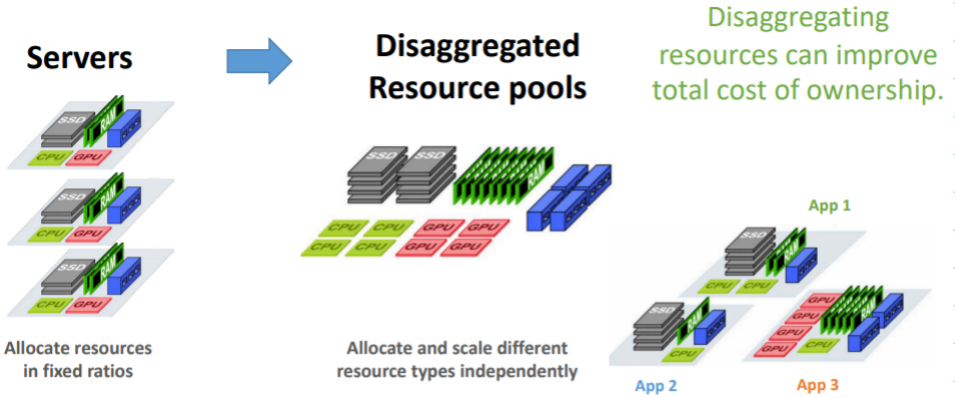
- Amortize cost of other hardware components
- Increase RAM
- Cores not that closely integrated
- No uniform memory access

Performance at large scale

- Total response time: Latency + Transfer Time
- Speedup: $\frac{\text{Latency old}}{\text{Latency New}}$
- Amadahl's Law: $\frac{1}{1-p+\frac{p}{s}}$ ← % parallelizable; ← speedup on parallelizable part
  - Speedup limitid by parallel part
  - Constand size
- Gustafson's Law: $1-p+s\cdot p$
  - Constant computing power
- Brawny CPU (Standard: 2 Brawny CPUs)
  - High per-core performance
  - Good when a great serial part
  - Limited by serial part (Amadahl's Law)
- Wimpy CPU
  - Lower power
  - Good when highly parallelizable
  - Systems still consume significant power at low load

- Bisection Bandwidth: Available bandwidth between any two nodes
- Scaling bisection bandwidth
  - For hierarchical network topologies
  - Best switches at the top
  - Over-subscription

Disaggregated Resource Pool



Total Cost of Ownership

(amortize over time but dominates)

- CAPEX: capital expenses (Facilities, hardware, ...)
- OPEX: operative expenses (Power, employees, ...)
- Efficiency = $\frac{\text{computation}}{\text{total power}}$
- Power Usage Effectiveness (= PUE) = $\frac{\text{Total facility Power}}{\text{IT Equipment Power}}$

Tiered Software Architecture

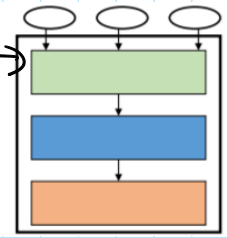
- Presentation layer: client, outside world
- Application logic: Process
- Data layer: Database
- The more boxes, the modular the system is, but more connections need to be kept alive and more context-switches

Cloud (work/watt) provider

Enterprise (work/$) data centers

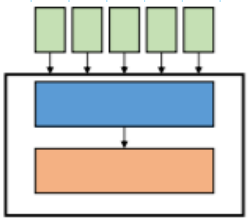
One Tier architecture

screen scraping →



- Fully centralized
- No processing, just a terminal
- Everything in one box
- Beginning of computing, no networking yet
- Virtual one tier today: webservers

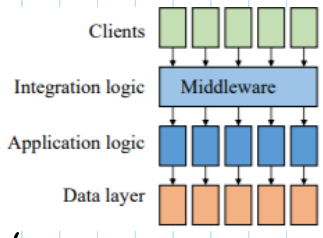
Two Tier architecture



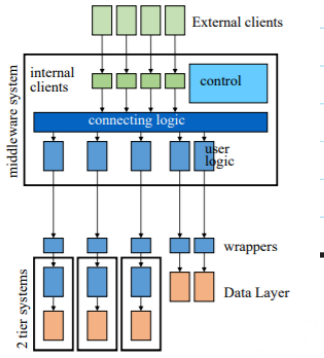
- Only a few clients
- Representation layer moved to client
- Introduces API
- Off-loed work to client
- Server has to deal with all possible connections from client
- No load balancing
- Client would need representation layer for each server
- Client needs to know where things are

↳ solution: Web browsers standardized representation layer

Three Tier architecture



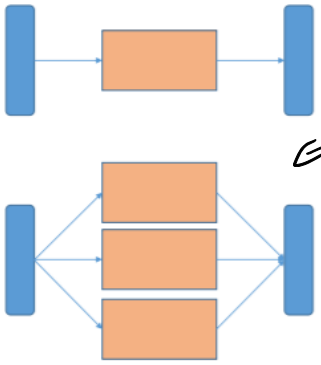
- Completly modular
- Needs to introduce a middleware logic
- Middleware: provides single interface to clients



Performance

- Requests completed per unit of time

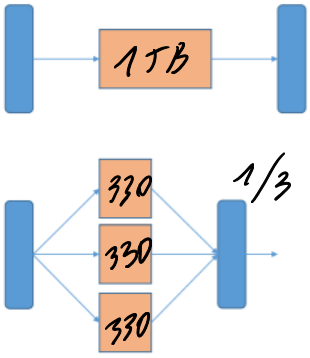
- Throughput: Can be optimized when parallel



Add more parallel machines

- Cost
- Bottleneck
- Independance

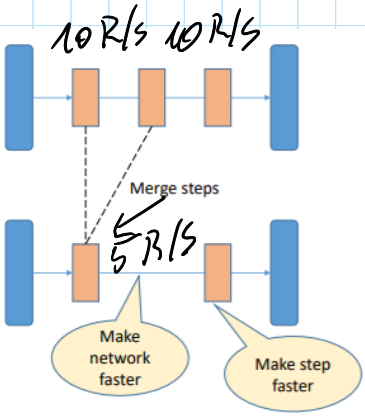
- Latency: - Typically a system measurement
  - If tasks take long => Parallelize them



1/3 latency

- Perform tasks in smaller chunks
- Overhead for communication

- Reduce length of critical path (consolidation)
  - Less hops / communication



Latency >< Throughput

- Cache it for faster access

- Respond time: percieved by client/user (inverse of throughput)
- The more tiers/components the better replication can be done

Open System

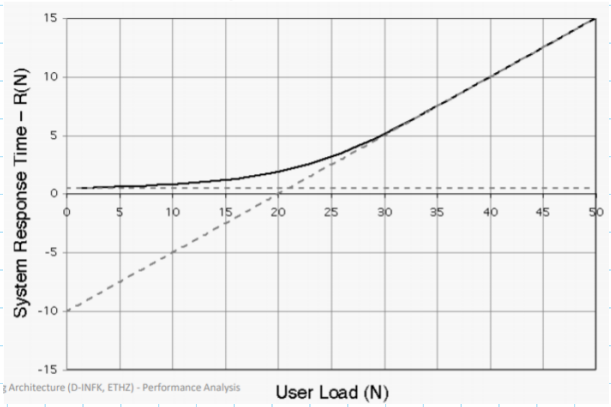
- Load comes from internet
- Not self-adjusting (if server down, more requests)
- e.g. Webserver ↳ Can use flow control
  ↳ Drop packets

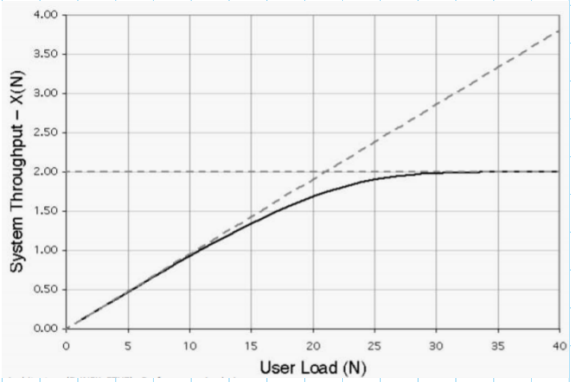
Closed System

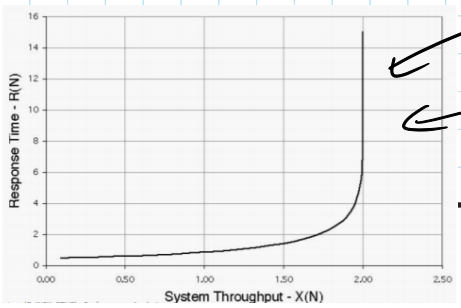
- Load from limited set of clients
- Client wait for response before sending next request
- e.g. database with local clients

Interactive Response time law

- Response time: R
- Wait for next request: Z
- Total cycle time: $R+Z$
- Number of clients: N
- Latency $X = \frac{\text{Jobs}}{\text{Time}} = \frac{N \cdot \frac{T}{R+Z}}{T} = \frac{N}{R+Z}$
- $R = \frac{N}{X} - Z$







Becomes irresponsible

More throughput, higher response time

Why is speed-up sublinear?

- Cost for split/merge operations
- Synchronize servers
- Not splitted into equal chunks

Experiments

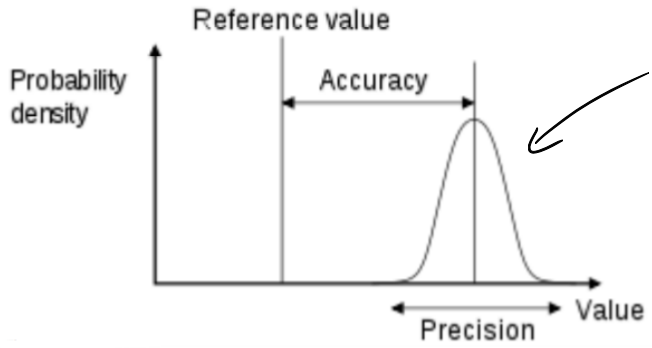
- Done on the machine itself (expensive)
- Benchmark it

Modeling

- Create a model of the system
- Calculate results with model

Simulation

- Implement system that behaves like real one



similarity of experiments

Repeat experiment for higher precision → ∞ ↳ #experiments

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

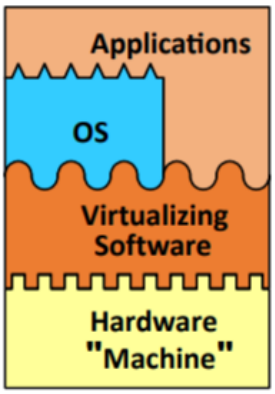
↑ mean

↑ std. deviation (68% of all values)

Virtual Machines

- Security
- No overhead
- Doesn't impact results

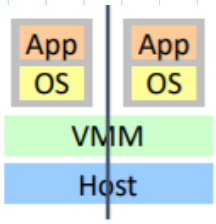
} Goal



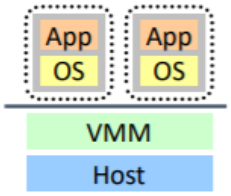
} Guest

} Virtual Machine Monitor / Hypervisor

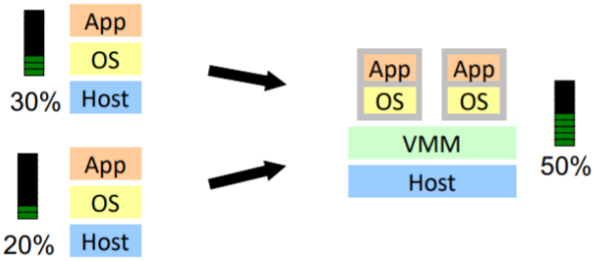
} Host



} Partitioning
- Resource sharing
- Context switch

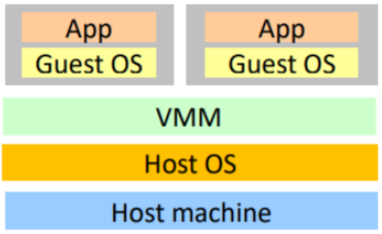


} Encapsulation
- Security

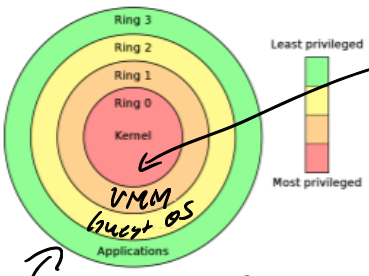


} Consolidation to improve utilization

- Hosted interpretation:
  - e.g. Virtual box
  - VMM sits on top of OS
  - VMM emulates hardware
  - Complete isolation
  - Emulation is difficult/slow



- Direct execution with trap and emulate:
  - When VM wants to execute privileged instruction => Trap and switch to VMM



Flush caches, I/O operations, ...

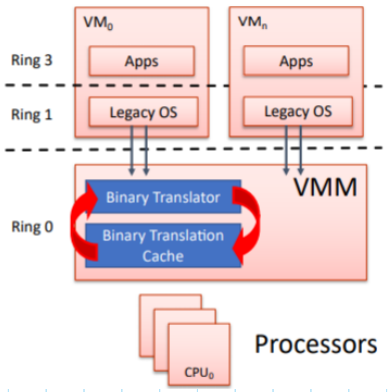
- No host OS is run
- Faster than with host OS

Trap for context switch

- Trap: context switch
- System call: priviledged, switch to (kernel)
- Exception: switch to kernel because of event } synchronous
- Interrupt: exterior event, e.g. keyboard input
- Sensitive Instructions: change hardware configurations
- Priviledged Instructions: Cause a trap

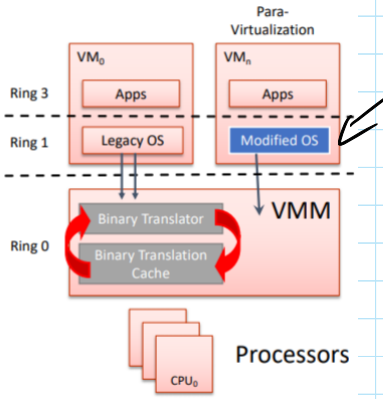
↳ CPU is virtualizable when also the sensitive instructions trap

- Direct execution with binary translation:



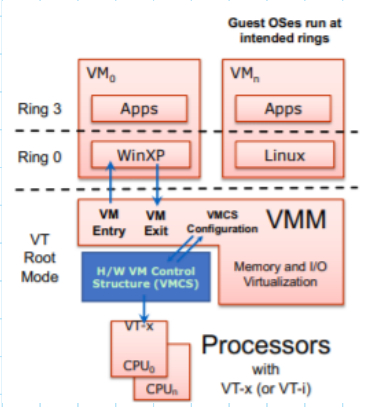
- Can run unmodified guest OS
- Most instructions run directly
- Need to translate virtual memory
- VMM rewrites non-virtualizable instructions

- Para-Virtualization



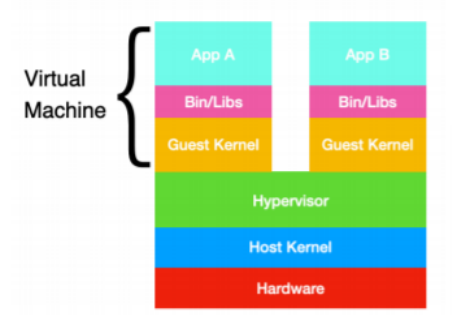
Modify guest-OS to remove sensitive but unpriviledged instructions

- Direct execution with hardware support



- Direct execution of VM on processor until priviledged instruction is executed
- Commonly used today

- Virtualizing Memory: - Shadow page-table for OS
  - VMM intercepts paging operations
  - Virtual Page
    - VMM { → Physical page } OS
    - → Machine page



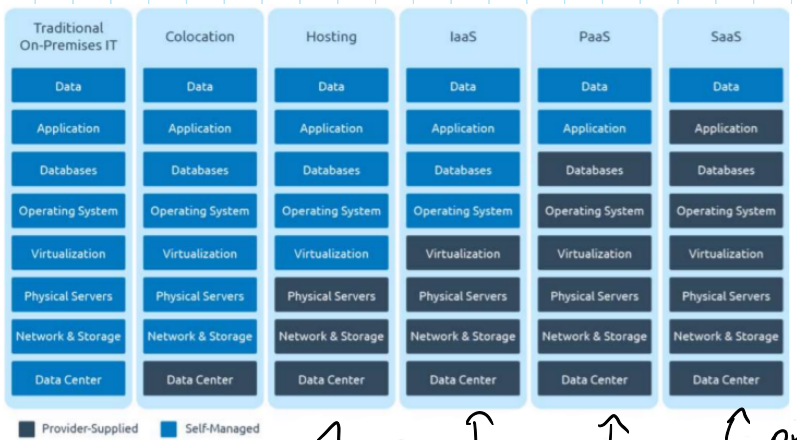
- Less secure
- No VMM
- Share Host-OS
- Light weight
- Faster start-up
- Bare-metal performance
- Less isolation

- Docker: - Used for packaging
  - Builds on top of Linux containers:
    - Image: - creates container
      - read only
    - Resource isolation
    - Manage CPU cores, etc.
- Linux kernel namespaces: - global system resources
  - Restrict what container can see
  - eg. net: network devices

- Linux control groups: · Limit resources
  · Groups tell limits
- Why are containers less secure?
  · OS is providing isolation
  · Container run directly on kernel, which is much bigger => more bugs
  · VMM has a smaller codebase
- Kubernetes
  · Runs and manages containers
  · Run across a cluster
  - Coordinates containers
  - Pod: · bunch of containers with same task
    · Used to run a common task together
  - Service: group of pods that work together

Cloud Deployments

- On-premise: · "enterprise-data center"
  · Often underutilized
- Hybrid: · Parts local, parts in the cloud
- cloud: · data center is run by a provider
  · Accessed remotly



Root server

Rent a VM

Rent common use case applications

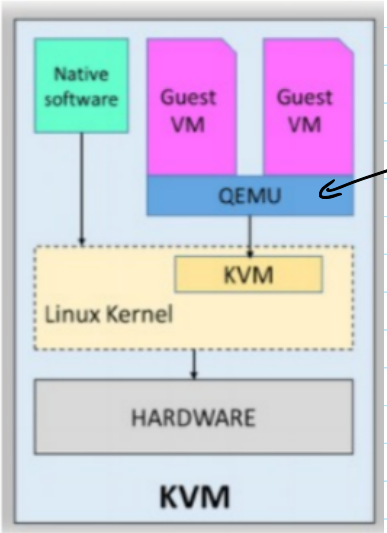
Only add data (Office 365)

Faas

↳ · Serverless / Stateless
  · Small applications
  · Executed rarely
  · Charged by ms
  · Short-lived

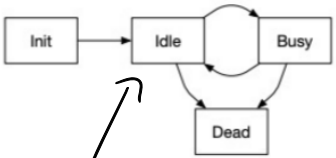
Firecracker

- VMM monitor developed by Amazon
- Uses Linux Kernel-based virtual machine
- Used for lightweight microVMs
- Used for serverless functions
- Best of both worlds: secure & fast
- serverless functions only need certain features in the VMM
  - ↳ No USB device support for example
  - ↳ only need networking support
- Linux Kernel VM: · Virtualization module
  · Linux as VMM



emulates the hardware
↳ Replaced by Firecracker

- One Firecracker process per microVM
- Written in Rust: · Provides memory safety
  · Provides thread safety
  · Fast
- AWS Lambda: · Slots for predefined functions



only ensures memory

Resource Assignment

- Private: · static set of resources
  · high priority workloads
  · allows hardware specification
- Shared: · Needs scheduler
  · Work conserving: unallocated resources get assigned to work

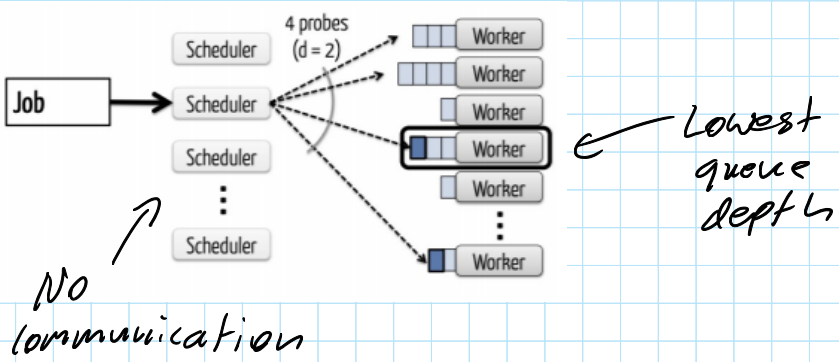
Scheduler

- Single Resource scheduler: · n users
  · give 1/n of resource
- Max-Min-Fairness: · If user needs more/less
  · min (actual work, max. resource)
  · min (act. work, weight · max resource)
  ↳ Weighted Max-Min
  · each user gets at least 1/n
  · Gets less when demand is less
  · Problem: only single resource
- Dominant Resource Fairness: · Apply Max-Min to dominant share
  · 9 CPU, 18 GB
  · 1 CPU, 4 GB ⇒ 1/9 < (4/18)
  ↳ x CPU, 4x GB
  · 3 CPU, 1 GB ⇒ (3/9) > 1/18
  ↳ 3y CPU, y GB
  · maximize x/y subject to
    - CPU constraint: $x + 3y \leq 9$
    - Memory constraint: $4x + y \leq 18$
    - Equality: $4x/18 = 3y/9$

$\Leftrightarrow 2x/9 = y/3$

Cluster Manager

- Centralized: One single manager responsible for entire cluster
- Two level: One master and multiple frameworks
  - One master: makes resource offers
  - multiple frameworks: selects resources and provides tasks
- Distributed:
  - Multiple scheduling agents
  - Better scalability



- Problem: 70% of tasks use more resources than they actually need
  - ↳ Overprovisioning

Quasar

- Observation: resource utilization is low
- Idea: let user specify performance goals
- Dynamically adjust based on profiles of service
- Performance can be affected by:
  - Scale up
  - Scale out
  - HW heterogeneity
  - Interference

1. Short profiling runs
2. Collaborative filtering to fill missing data
3. Greedy scheduler finds number and type of resources that maximizes performance/utilization

Machine Learning

- Training:
  - Forward pass (loss)
  - Backward pass (gradients)
- Inference:
  - forward pass on small batch of data
- Computation graph:
  - Static: better optimization
  - Dynamic: more intuitive, better for debugging
- Distributed training
  - Data parallelism:
    - Partition data
    - Train multiple copies of model
    - Synchronize weight updates
  - Model parallelism:
    - More complicated
    - split model across nodes

RPC (= Remote procedure calls)

- Invented when going from 1-Tier → 2-Tier
- Function call is converted into remote procedure
  ↳ Not seen by program
- Synchronous

1. Call remote function
2. Bind: Find remote server with directory service
3. Marshalling: Convert to a packet

- Tied to TCP/IP
- Tied to programming languages
- No standardization

REST (= Representational State Transfer)

- Implemented in HTTP
- Stateless interaction
- Still synchronous
- Get, Put, Delete, Post
- Materialized view: Cache rest-call
- Service aggregation: Make all calls from a centralized place

Data Replication

- Why replication: · Performance (near me)
  · Fault Tolerance
- Synchronous replication: used for transactions (DBs)
- Asynchronous replication: eventual consitency
- Update everywhere: changes can be done on every replication set
- Primary copy: · only one copy which can be updated
  · Databases

	Primary copy	Update everywhere
Synchronous	Advantages: Updates not coordinated No inconsistencies Disadvantages: Longest response time Only useful with few updates Local copies are can only be read	Advantages: No inconsistencies Elegant (symmetrical solution) Disadvantages: Long response times Updates need to be coordinated
Asynchronous	Advantages: No coordination necessary Short response times Disadvantages: Local copies are not up to date Inconsistencies	Advantages: No centralized coordination Shortest response times Disadvantages: Inconsistencies Updates can be lost (reconciliation)

Storage systems

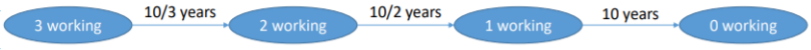
- Block Storage: virtual raw disk
- Object storage: block blobs / append blobs / page blobs
- Key-Value-Store: distributed hash table in ring
  ↳ used as cache e.g. hash database queries
- Databases
- File storage: on top of block storage (=NAS)
  · millions of PB files
  - append only

Reliability

- Errors: operational but produces wrong results
- Faults: operational except some parts
- Failures: system no longer operational
- Performance issues: not usable because too slow
- Frequency of failures: F/T
- Mean Time To Fail: T/F (=MTTF)
- MTTF multiple components: $MTTF_1/N$
- Failure Rate: $1/MTTF$

↳ When we have a lot of components, the system is more likely to fail

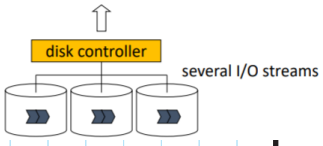
- Mean time between fails: $MTTF + MTTR$ (MTTR: mean time to repair)
- Availability: $\frac{MTTF}{(MTTF + MTTR)}$
- Availability of 99% ≙ 87,6 hours downtime
- Redundancy:



- Countermeasures:
  - Replication
  - Error correcting codes
  - sharding

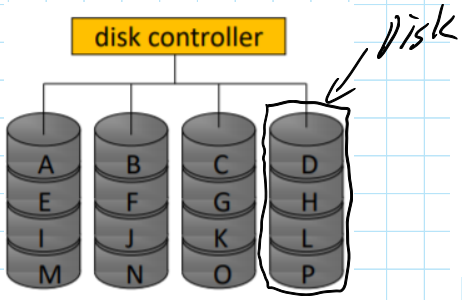
Data Striping

- Distribute data among several disks for faster read/writes and backup



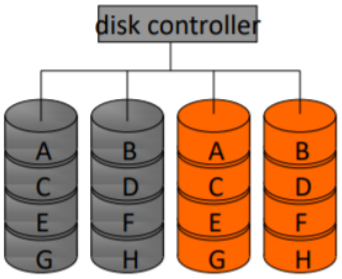
- Fine grained:
  - all requests use all disks
  - maximize disk bandwidth
  - only one I/O request serviced at a time } e.g. multimedia
- Coarse grained:
  - small requests can be serviced in parallel } e.g. DB
  - large requests still benefit from high transfer rates
- Striping + Parity (detect and correct errors)
  - ↳ RAID

- Raid 0



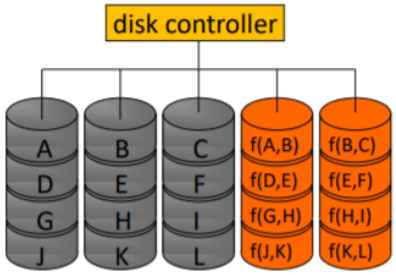
- No redundancy
- Nx faster

- Raid 1



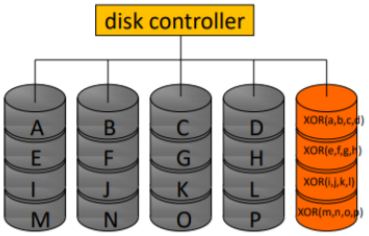
- 50% backup

- Raid 2



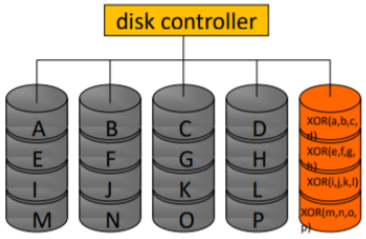
- Hamming code

- Raid 3



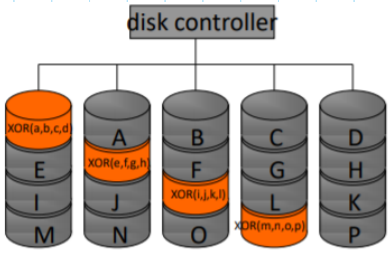
- One disk failure
- parity

- Raid 4

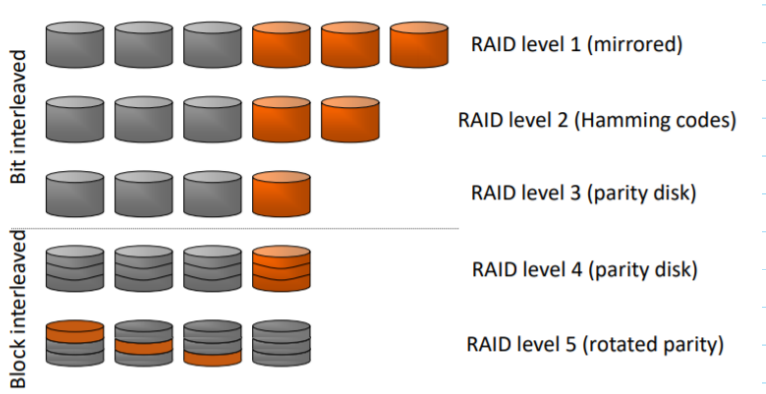


- Blocks instead of bits

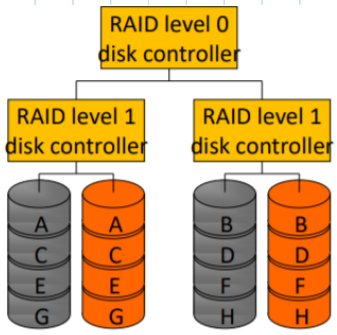
- Raid 5



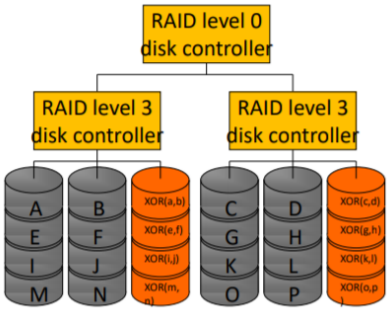
- Read/Write in parallel



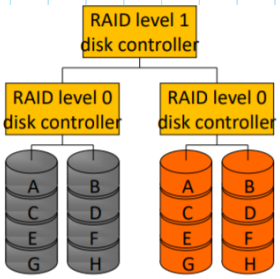
- Raid 10



- Raid 53



- Raid 0+1

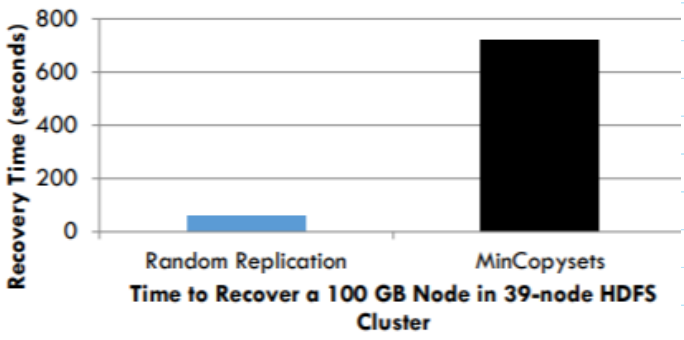


- Level 1 controller has less disks to manage

- Sharding
  - split data-sets into smaller pieces
  - Place them across multiple machines
  - copySet: set of machines which contain all replicas of one data-block
  - If you loose copyset, you loose data
  - Random replication eventually leads to $Pr(\text{dataloss}) = \frac{\#\text{current copysets}}{\#\text{max. copysets}}$
  - When $n \to \infty$: $Pr(\text{dataloss}) = 1$, so if any 3 nodes fail, we have dataloss

- MinCopyset: - Minimize #copysets
  1. Randomly place first chunk
  2. Place two other replicas accordingly to already existing copyset

	MinCopysets	Random Replication
Mean Time to Failure	625 years	1 year
Amount of Data Lost	1 TB	5.5 GB



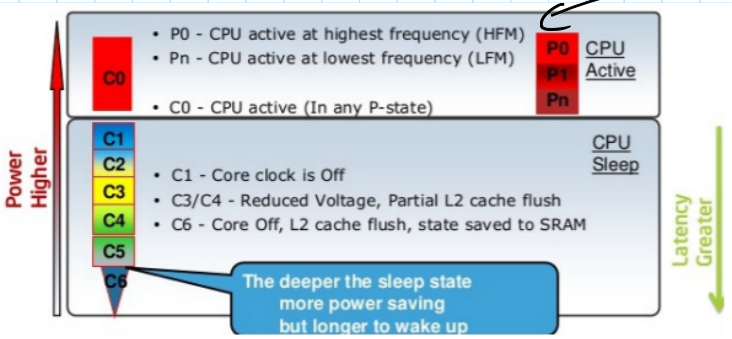
CPU Core isolation (=Noisy neighbours)

- Intel Cache Allocation Technology: Partition last level cache which is shared

CPU Power Management

- p-states: performance while code being executed
- C-states: CPU is in sleep

Dynamic Voltage/Frequency scaling



- SRAM
  - Static Random Access Memory
  - Used for caches => Fast
  - No refreshing => keeps state
  - Bigger than DRAM
  - High cost

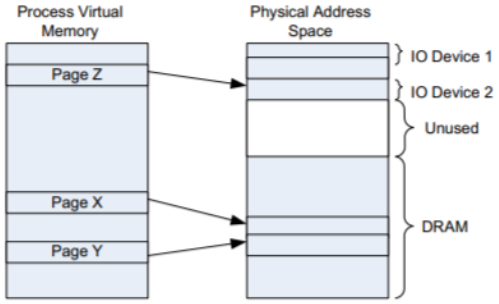
- DRAM
  - Dynamic Random Access Memory
  - cell loses charge when read
  - cell loses charge over time

  } Must periodically refresh

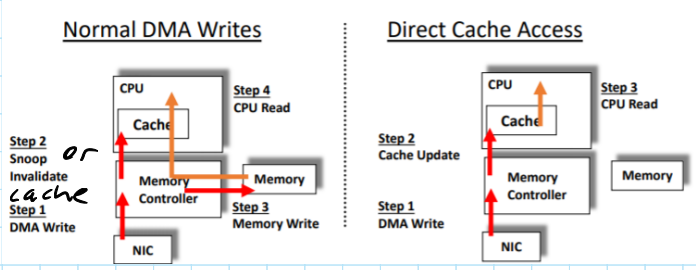
- Flash Storage
  - Non-volatile
  - Can only process one operation at a time
    - → Flash translation layer (wear count)

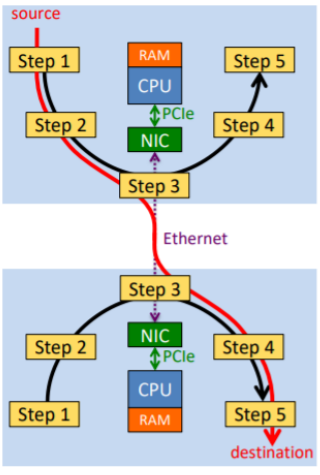
Networking

- Memory mapped I/O
  - use load/store to read/write locations on device
  - slow
  - often used for device configuration
  - CPU always involved



- Direct Memory Access
  - I/O device to memory transfer
  - without involving CPU
  - DMA completes transfer by sending interrupt to CPU
  - Inconsistency with CPU caches
    - Map to non-cachable area
    - Cache can snoop DMA bus-line
    - Direct cache access





Source (TX)

1. Allocate DMA-able buffer with packet data
2. Write DMA descriptor to NIC
3. NIC DMA engine copies data for transmission
4. TX completion notification (interrupt)
5. Deallocate buffer

Destination (RX)

1. Allocate buffer for received packet
2. Write DMA descriptor to NIC
3. NIC DMA engine copies data to host
4. RX notification to host (interrupt)
5. Forward received packet to user application

- NIC acceleration techniques
  - Checksum
  - Large packets to smaller ones
  - Process multiple packets before interrupt
- Interrupt steering
  - To one core
  - spread across all cores
  - To running app core

Specialization

- CPU: low latency, low parallelization
- GPU: high latency, high parallelization
  - ↳ single instruction multiple threads
- TPU: matrix multiplication with reduced accuracy
  - Driven by CPU (accelerator)

Accelerator

Cloud security

- Goals: - Confidentiality: protect data → encrypt it
  - Integrity: Only authorized users can modify
  - Availability: prevent DDOS
  - Access control: Who and what can user do/is
  - Privacy
- Secure boot: check signature of bios/os
- Trusted execution environment (= Enclaves)
  - Run code/memory isolated from system
  - Only trust CPU, all other is encrypted (DRAM)
  - CPU prevents cache access from outside
  - Can only run unprivileged code
  - e.g. for private keys

- Side channels
  - Communication between components
  - Memory access patterns
  - State of branch predictor

  } Examples

- Covert Channel
  - Hidden channel to get information out
  - Extract patterns from system on shared resources

To send a bit $b \in \{0,1\}$ malware does:

- b= 1: at 1:00am do CPU intensive calculation
- b= 0: at 1:00am do nothing

At 1:00am listener does CPU intensive calc. and measures completion time

b = 1 $\Rightarrow$ completion-time > threshold