云计算技术与工程

虚拟化（云计算）数据中心

徐敏贤

副研究员
云计算研究中心
中国科学院深圳先进技术研究院
http://www.minxianxu.info

举头西北浮云，倚天万里须长剑。
——（宋）辛弃疾

内容回顾

- Q1: 请解释虚拟化技术的优点？为什么要采用虚拟化技术？

随着云计算的发展，传统的数据中心逐渐过渡到虚拟化数据中心，即采用虚拟化技术将原来数据中心的物理资源进行抽象整合。

- 实现资源的动态分配和调度，提高现有资源的利用率和服务可靠性
- 提供自动化的服务开通能力，降低运维成本
- 具有有效的安全机制和可靠性机制，满足公众客户和企业客户的安全需求
- 方便系统升级、迁移和改造

内容回顾

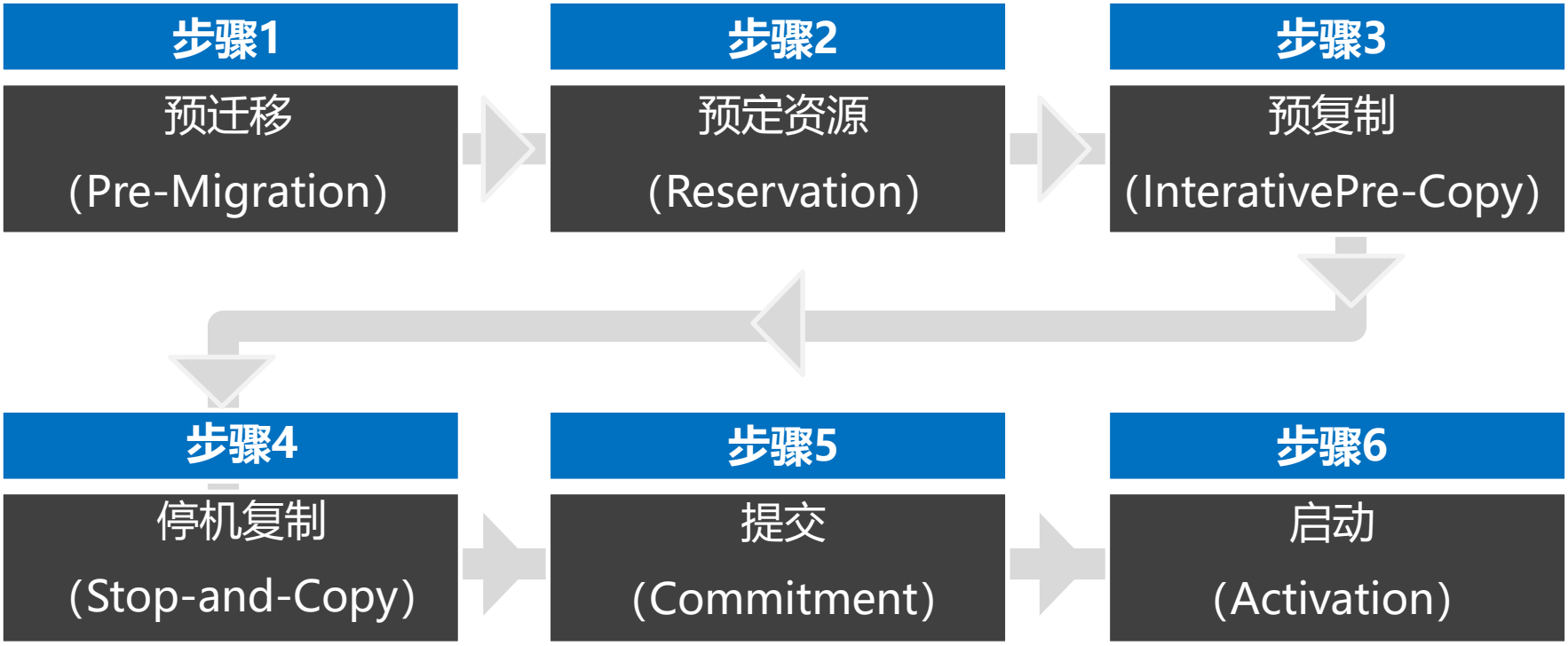
- Q2: 服务器虚拟化有哪些底层实现?

- CPU虚拟化
  - 将物理CPU抽象成虚拟CPU
- 内存虚拟化
  - 内存虚拟化技术把物理内存统一管理，包装成多个虚拟的物理内存提供给若干虚拟机使用，每个虚拟机拥有各自独立的内存空间。
- I/O设备虚拟化
  - I/O设备虚拟化技术把真实的设备统一管理起来，包装成多个虚拟设备给若干个虚拟机使用，响应每个虚拟机的设备访问请求和I/O请求。

内容回顾

- Q3: 为什么要进行虚拟机迁移？虚拟机迁移有哪些主要步骤？

1. 负载动态变化，实现负载均衡
2. 服务器升级维护



内容回顾

- Q4: 存储虚拟化有哪些关键技术?

存储虚拟化将系统中分散的存储资源整合起来

提高了存储资源利用率

降低了单位存储空间的成本

降低了存储管理的负担和复杂性

在虚拟层通过使用数据镜像、数据校验和多路径等技术

提高了数据的可靠性及系统的可用性

利用负载均衡、数据迁移、数据块重组等技术

提升系统的潜在性能

整合和重组底层物理资源

得到多种不同性能和可靠性的新的虚拟设备

满足多种存储应用的需求

目录 Contents

注：本章内容选自《The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines》一书第二版以及Christina Delimitrou教授@MIT

What is a Datacenter (DC)

- The compute infrastructure for internet-scale services & cloud computing
  - x 10k of servers, x 100k hard disks
  - Examples: Google, Facebook, Microsoft, Amazon (+Amazon Web Services, Twitter, Yahoo,…)

- Both consumer and enterprise services
  - Windows Live, Gmail, Hotmail, Dropbox, Bing, Google, Adcenter, GoogleApps, Web apps, Exchange online, salesforce.com, Azure, …

What is a Datacenter (DC)

- A simplistic view
  - Scaled-up/out version of machine rooms for enterprise computing
  - A large collection of commodity components
    - PC-based servers (CPUs, DRAM, disks),
    - Ethernet networking
    - Commodity OS and software stack
    - 10 to 100 of thousands of nodes
  - High-bandwidth networking (10Gbps → 40Gbps → 100Gbps)
  - Power delivery, cooling, UPS (50 – 200MW)
  - System software for DC management (centralized or distributed)
  - Software that implements internet services

Prevalence of Cloud Computing

~500 Datacenters (w/ >10k servers)

$30 billion for AWS (2018)
~ $60 billion by 2021

Microsoft and Google:
> 2,000,000 servers

~3% of US power consumption

A More Complete View of a DC

- Apart from computers & network switches, you need:
  - Power infrastructure: voltage converters and regulators, generators and UPSs,
  - Cooling infrastructure: A/C, cooling towers, heat exchangers, air impellers,…
- Co-designed!

Example: MS Quincy Datacenter

- 470k sq feet (10 football fields)
- Next to a hydro-electric generation plant
  - At up to 40 MegaWatts, $0.02/kWh is better than $0.15/kWh J
  - That's equal to the power consumption of 30,000 homes

Example: MS Chicago Datacenter

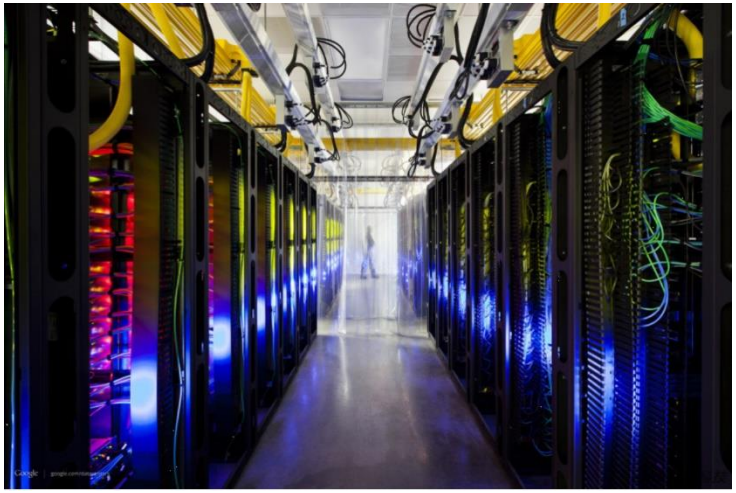
Alibaba Cloud

- 2 millions of nodes
- Investment: RMB 20 billion

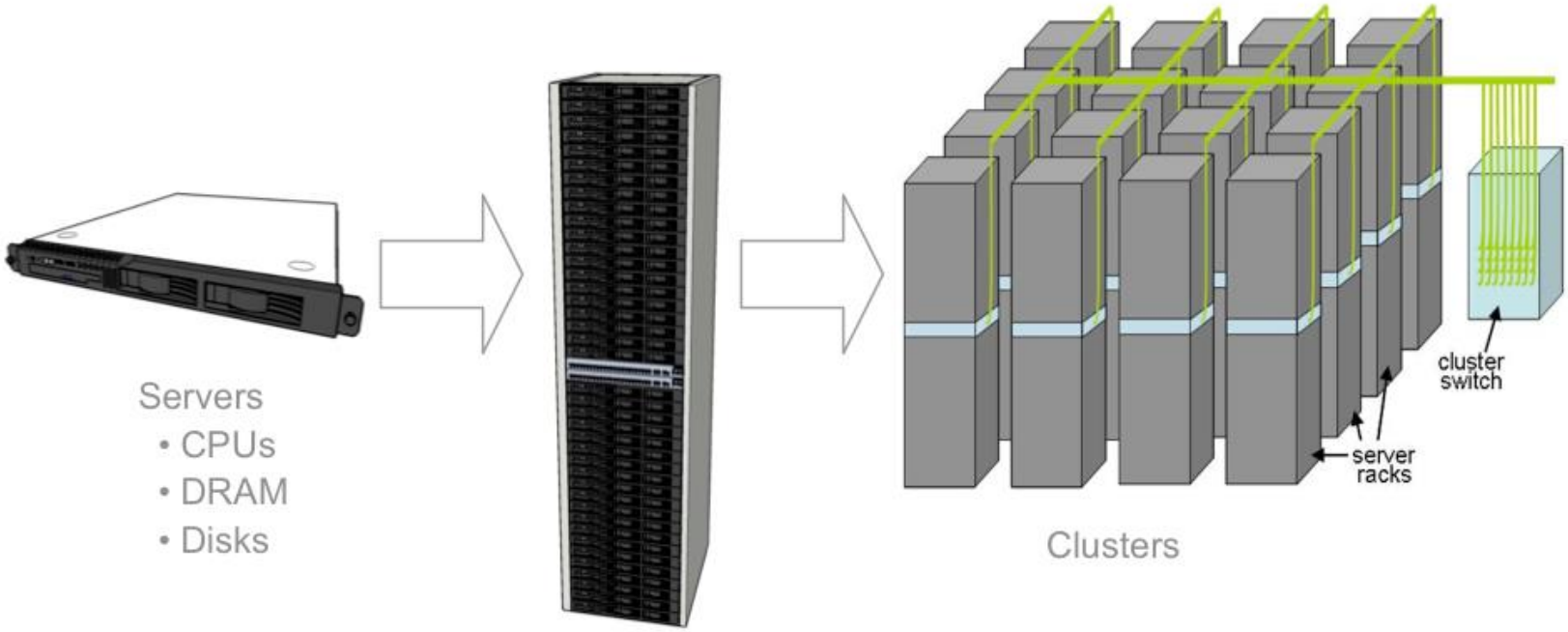
Motivation for Internet-scale Services & Datacenters

- Some applications need big machines
  - Examples: search, language translation, etc
- User experience
  - Ubiquitous access, flexibility, cost-efficiency
  - Ease of management (no backups, no config)
- Vendor benefits (all translate to **lower costs**)
  - Faster application development
  - Ease of (re)deployment for upgrades and fixes
  - Single-system view for storage and other resources
- Lower cost by sharing HW resources across many users
- Lower cost by amortizing HW/storage management costs

DC Hardware Overview



DC Hardware Overview



DC Hardware Overview

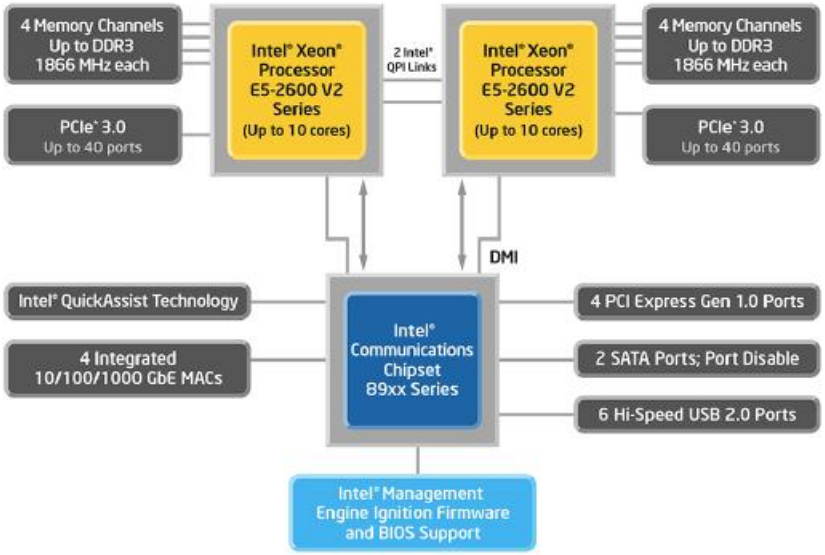
- Compute & Memory
  - A PC-based server (most of them x86)
  - Custom Linux OS (optimized for latency)
  - 1U, 2U, or blade form factors are popular
    - E.g., 2 processor sockets, tens of GB of DRAM, 2 disks,…
- Networking
  - 10Gbps ethernet links to rack switch
    - Typically using TCP/IP as the networking protocol
    - Cost is <$30/Gbps per server within a rack
    - Lower cost than faster networks like Infiniband ($500/port)
  - A few 10Gbps links to cluster level switches
    - Two or three level hierarchy within data centers
    - Oversubscription of uplinks (often 1:10) can be a problem

DC Hardware Overview

- Storage
  - Distributed FS using disks on servers (GFS)
    - Better fault-tolerance across nodes, lower cost, better scalability
  - Network attached storage (NAS) devices
    - Specialized systems with disk arrays that provide FS storage services and connect directly to the networking fabric
    - Better fault tolerance within device, easier management
    - Often set up in RAID configurations

The Hardware

2-socket server



10GbE NIC

Flash Storage

JBOD disk array

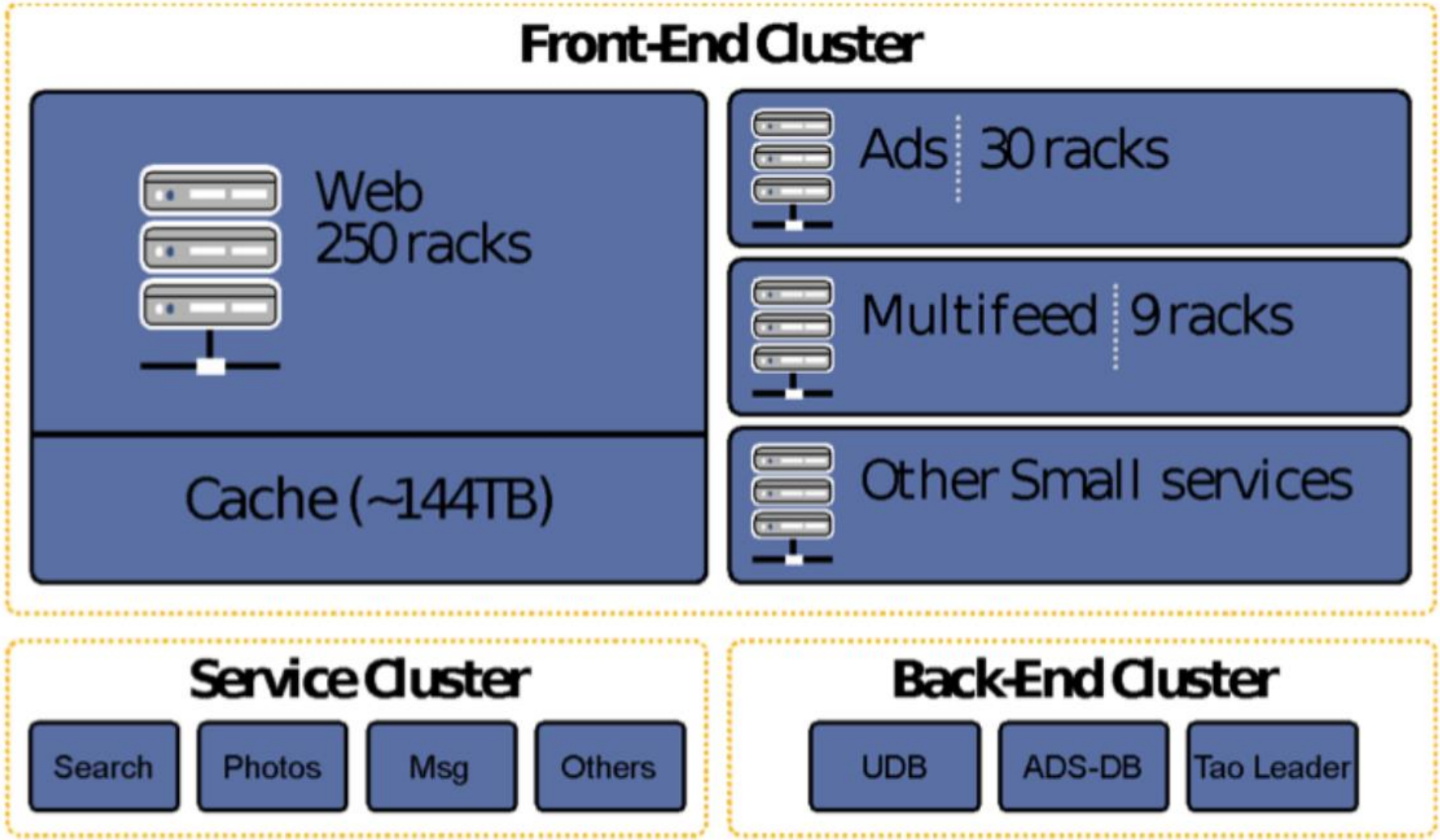
Low-latency 10GbE Switch

Hardware Components

- Custom-design servers
  - Configurations optimized for application needs
  - Few configurations to allow reuse across many applications
  - Roughly constant power budget per volume

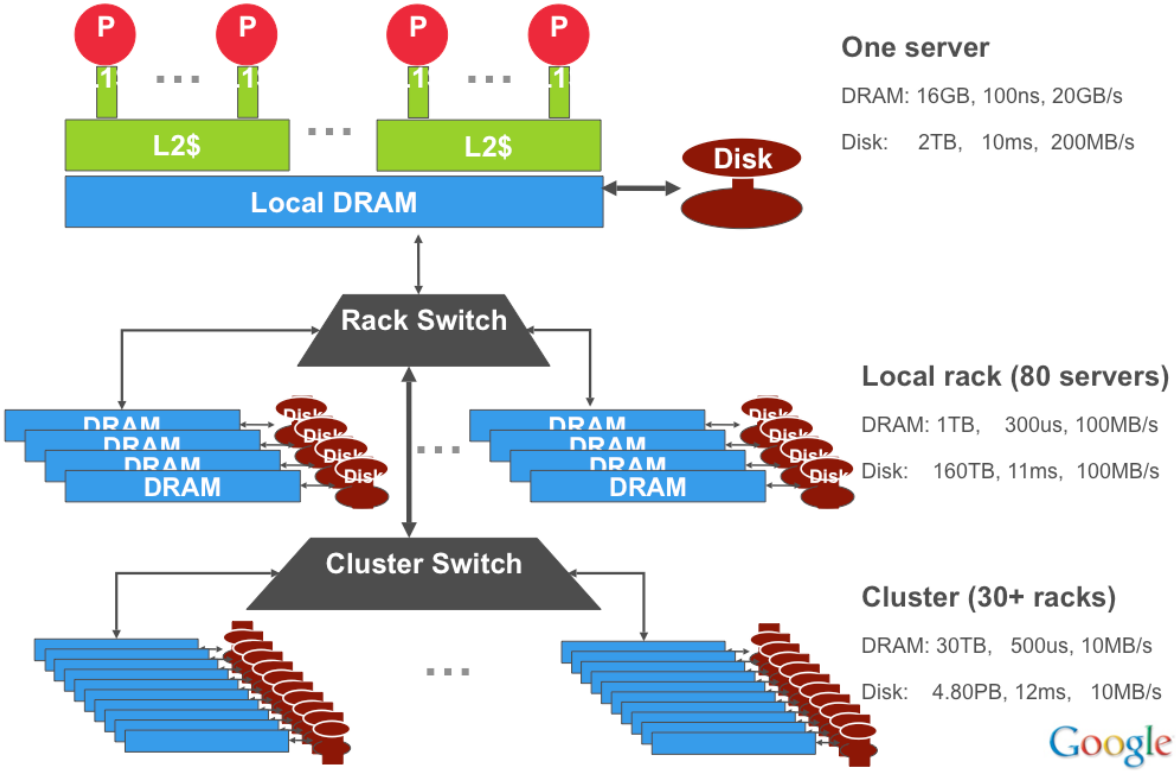
Standard Systems	I Web	III Database	IV Hadoop	V Haystack	VI Feed
CPU	High 2 x E5-2670	High 2 x E5-2660	High 2 x E5-2660	Low 1 x E5-2660	High 2 x E5-2660
Memory	Low 16GB	High 144GB	Medium 64GB	Med-Hi 96GB	High 144GB
Disk	Low 250GB	High IOPS 3.2 TB Flash	High 15 x 4TB SATA	High 30 x 4TB SATA	Medium 2TB SATA + 1.6TB Flash
Services	Web, Chat	Database	Hadoop	Photos, Video	Multifeed, Search, Ads

Application Mapping Example

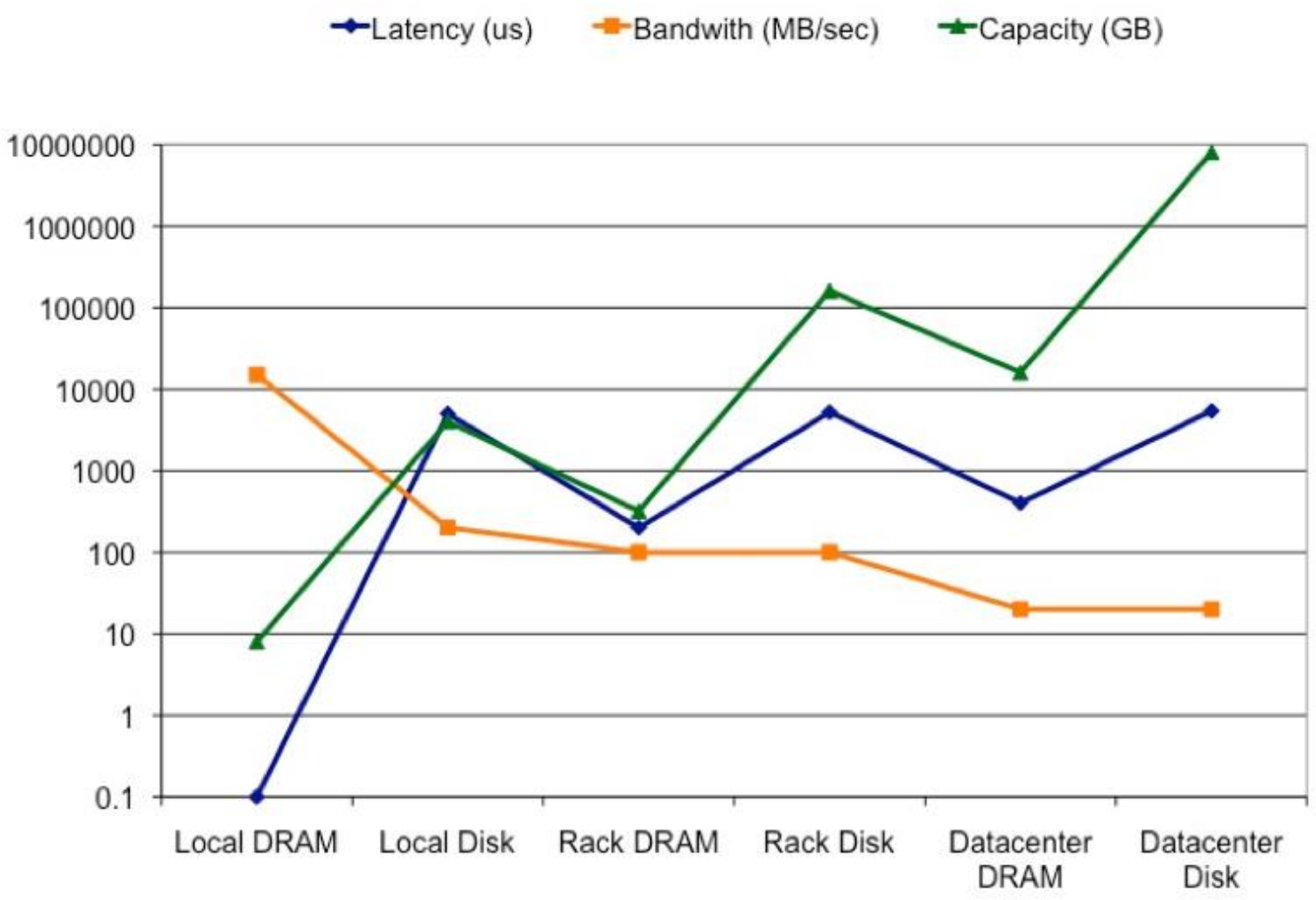


DC Storage Hierarchy: A Programmer's Perspective

- Interesting observations
  - Remote memory is often faster than local disk
  - Bandwidth bottlenecks



DC Storage Hierarchy: A Programmer’ s Perspective



Useful Numbers
Courtesy of Jeff Dean, Google

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Snappy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA -> Europe -> CA	150,000,000 ns

Courtesy of Jeff Dean, Google

- Compilers don't warn Jeff Dean. Jeff Dean warns compilers.
- Google search went down for a few hours in 2002, and Jeff Dean started handling queries by hand. Search Quality doubled.
- Jeff Dean builds his code before committing it, but only to check for compiler and linker bugs.
- Jeff Dean writes directly in binary. He then writes the source code as a documentation for other developers.
- All pointers point to Jeff Dean.
- Check more here: https://www.zhihu.com/question/22081653

Useful Back of the Envelop Math

- How long to generate image results page (30 thumbnails)?
- Design 1: Read serially, thumbnail 256K images on the fly
  - 30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
- Design 2: Issue reads in parallel
  - 10 ms/seek + 256K read / 30 MB/s = 18 ms
  - (Ignores variance, so really more like 30-60 ms, probably)
- Lots of other options
  - Caching (single images? whole sets of thumbnails?)
  - Pre-computing thumbnails
  - … Back of the envelope helps identify most promising…

Using Targeted Microbenchmarks

- Use a microbenchmark that stresses a particular aspect of a system to identify inefficiencies, bottlenecks, potential for optimization
- Builds intuition for back-of-the-envelope calculations
- Reduces time needed to test performance improvements
- Examples:
  - Cache intensive microbenchmark to stress a new cache hierarchy
  - Network microbenchmark (small/large packets, fanout, frequency) to test a new topology, switch, protocol, etc.
- Keep in mind for cluster management

Total Cost of Ownership (TCO)

- The key metric in datacenters
  - Scalable capabilities at low TCO (large-scale needs low cost)
  - Key competitive advantage
- TCO components
  - Capital expenses (CAPEX)
    - Facilities, compute, storage, networking
  - Operational expenses (OPEX)
    - Energy, maintenance, employees, …
- Keep in mind
  - Facilities & equipment amortized over a certain period
  - Cost of money (interest rate)

Efficiency in DCs

- PUE (Power Usage Effectiveness) standard DC metric for how much power goes towards useful work

$$PUE = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}}$$

- Ideally PUE=1.0 (no overhead from cooling)
- Used to be ~2.0
- Best today 1.07 (Facebook), 1.12 (Google)

Cost Model: Facilities CAPEX

- Size of facility: 8MW; Cost of facility: $11/W
  - See table next page for typical construction costs
- Total facility CAPEX costs: $88M
  - % facility costs for power and cooling = 82% = $72.16M
  - % other infrastructure = 18% = $15.84M
- US accounting rules to convert CAPEX to OPEX
  - Facilities amortization time = 10-15 years
  - Annual cost of money = 5%
  - OPEX = Pmt(interest_rate, number_payments, principal)
- Monthly opex: Power and cooling: 765K;

other infrastructure: 168K ($0.02/W/month)

Cost Model: Facilities CAPEX

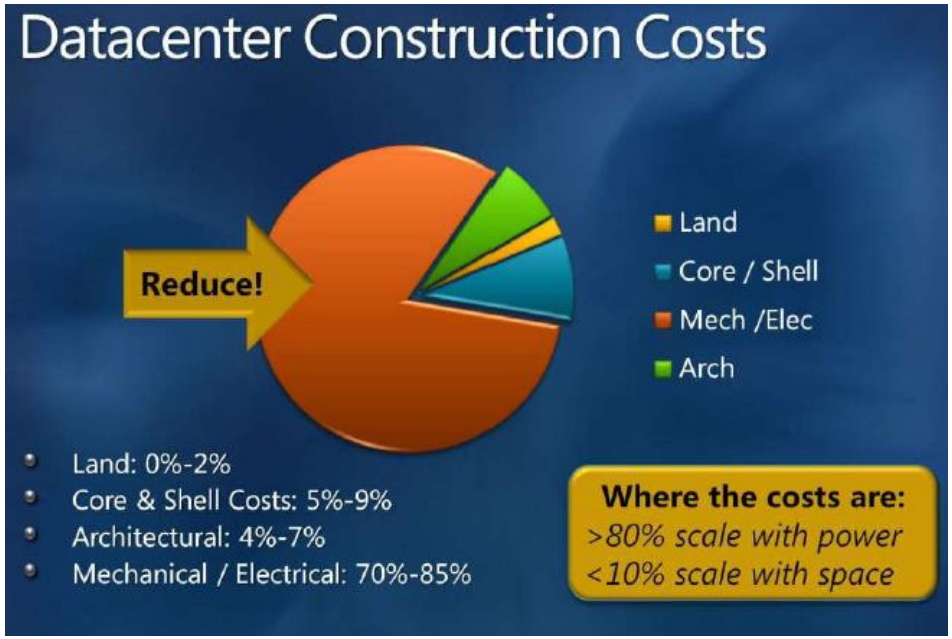


TABLE 6.1: Range of datacenter construction costs expressed in U.S. dollars per watt of critical power.

COST/W	SOURCE
$12–25	Uptime Institute estimates for small- to medium-sized datacenters; the lower value is for "Tier 1" designs that are rarely used in practice [66]
$10+	Microsoft's purchase of two 10-MW datacenters in California for $200 million; this cost excludes the value of land and buildings [72]
$10–14	Dupont Fabros S-1 filing [25, p. 6] contains financial information suggesting the following costs: $102.4 million for the purchase of ACC4, an existing 9.6 MW facility ($10.67/W) $180–300 million for the construction of four 18.2 MW facilities ($10/W—$13.40) The earnings release for Q3 2008 [68] lists the construction cost of a completed 18.2-MW Chicago facility as $190 million, or $10.44/W.

Critical power is defined as the peak power level that can be provisioned to IT equipment.

Cost Model: Systems CAPEX

- Servers
  - 45,978 servers x $1450 per server = $66.7M CAPEX
  - Depreciation: 3 years; cost of money = 5%
  - Monthly CAPEX: $2,000K
- Networking
- Rack switches: 1150 x $4800; Array switches: 22 x $300K; Layer3 switch: 2 x $500K; Border routers: 2 x $144.8K = $13.41M CAPEX
- Depreciation: 4 years; cost of money = 5%
- Monthly CAPEX: $309K

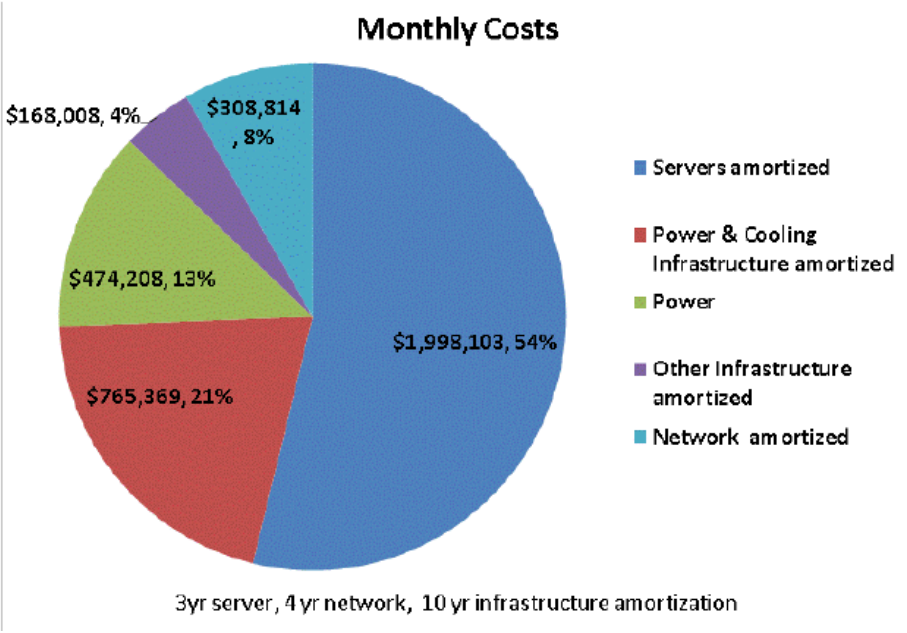
Cost Model: OPEX Costs

- Power
  - =[MegaWattsCriticalLoad*AveragePowerUsage/1000*PUE*PowerCost *24*365/12]
  - 0.07c/KWhr; PUE = 1.45; average power use: 80%
  - $475K OPEX (monthly)
- People
  - Security guards: 3 x 24x365x$20 + Facilities: 1x24x365x$30 ; Benefits multiplier: 1.3
  - $85K OPEX (monthly)
- Network bandwidth costs to internet
  - Varies by application and usage
- Vendor maintenance fees + sysadmins
  - Varies by equipment and negotiations

Cost Model: TCO

- Observations
  - 34% costs related to power (trending up while server costs down)
  - Networking high at 8% of overall costs; 15% of server costs
  - How is this different from traditional enterprise computing?

Servers amortized	$1,998,103
Power & Cooling Infrastructure amortized	$765,369
Power	$474,208
Other Infrastructure amortized	$168,008
Network amortized	$308,814
People	$85,410



Enterprise Vs Internet-scale Computing: a Cost Perspective

- Enterprise computing approach
  - Largest cost is people -- scales roughly with servers (~100:1 common)
  - Enterprise interests focus on consolidation & utilization
    - Consolidate workload onto fewer, larger systems
    - Large SANs for storage & large routers for networking
- Internet-scale services approach
  - Largest costs is server H/W
    - Typically followed by cooling, power distribution, power
    - Networking varies from very low to dominant depending upon service
    - People cost under 10% & often under 5% (>1000+:1 server:admin)
    - Services interests centered around work-done-per-$ (or watt)
- Observations
  - People cost shifts from top to nearly irrelevant.
  - **Focus instead on work done /$ & work done/watt**

TCO Discussion

- Anything else missing?
  - Tip: what can go wrong in a data center?

Using the cost analysis

- Cost model powerful tool for design tradeoffs
  - Evaluate "what-if" scenarios
- E.g., can we reduce power cost with different disk?
  - "Burdened cost of a Watt per year"
    - What does this mean?
  - ($765K+$475K)*12/8MW = $1.86/Watt/year
- A 1TB disk uses 10W of power, costs $90. An alternate disk consumes only 5W, but costs $150. If you were the data center architect, what would you do?

Answer

- A 1TB disk uses 10W of power, costs $90. An alternate disk consumes only 5W, but costs $150. If you were the data center architect, what would you do?

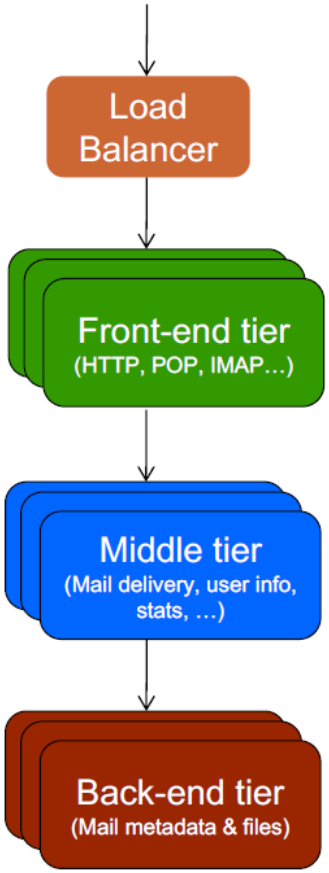
- @ $2/Watt – even if we saved the entire 10W of power for disk, we would save $20 per year. We are paying $60 more for the disk – probably not worth it.
  - What is this analysis missing?

Datacenter Applications

- Batch applications
  - Analytics like Hadoop and Spark
  - E.g., create Netflix recommendations, search index, …
  - Typically operate in the background (not interactive)
  - They care about **throughput**
- Interactive services
  - Search, webmail, maps, webdocs, social networks, etc.
  - Can be single- or multi-tier -> microservices
  - They care about latency (tail latency)
- Native processes, containers, or virtual machines
  - E.g., Amazon EC2 or Microsoft Azure

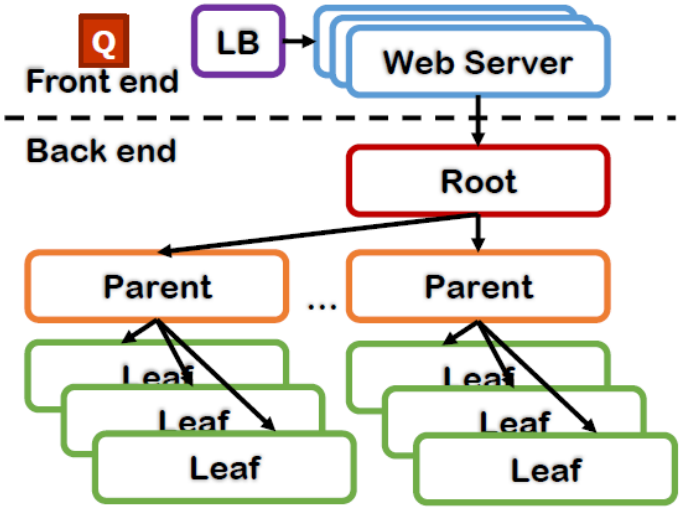
Example 3-tier App: WebMail

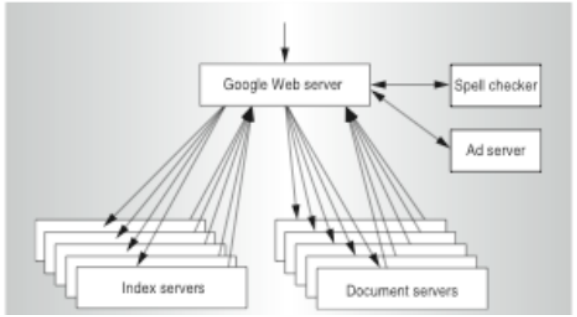
- May include thousands of machines, PetaBytes of data, and billions of users
- 1st tier: protocol processing
  - Typically stateless (some caching)
  - Use a load balancer
- 2nd tier: application logic
  - Often caches state from 3rd tier
- 3rd tier: data storage
  - Heavily stateful
  - Often includes bulk of machines



Example: Search

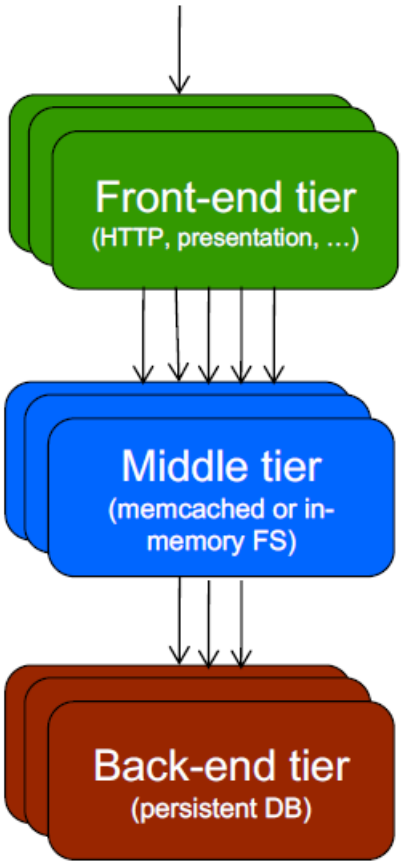
- 1st tier: protocol processing
  - HTTP processing
- 2nd tier: application logic
  - Query caching, query aggregation, …
- 3rd tier: data storage
  - Indexes (mostly in DRAM, multiple types)
  - Document serving





Example: Social Networking

- 3 tier system
  - Web server, fast user data storage, persistent storage
  - 2rd tier: latency critical, large number of servers
- User data storage
  - Using memcached for distributed caching
  - 10s of Tbytes in memory (Facebook 150TB)
  - Sharded and replicated across many servers
  - Read/write (unlike search), bulk is read-dominated
- From in-memory caching to in-memory FS
  - RAMcloud @Stanford, Sinfonia @HP, …



Why Tiered Applications

- Simplifies application development
  - Separate problem of reliable storage from problem of protocol processing
- Simplifies performance tuning
  - Single role for each machine in the system (not always)
  - Scale-out to meet throughput demands
- Simplifies reliable operation
  - Exposes dependencies and communication
    - Minimizes unexpected interactions/bottlenecks
  - Helps create fault-tolerant services
    - Loosely coupled services
    - Can work around failures by accessing replicas or caches
    - Simplifies testing of individual components
- Disadvantages?

Characteristics of Internet-scale Services

- High request level parallelism
  - Without much read-write sharing
- High workload churn
  - New releases of code on a weekly basis
- Platform homogeneity (relative)
  - Simplifies scheduling and management
- Require fault-free operation
  - High availability guarantees

Characteristics of Internet-scale Services

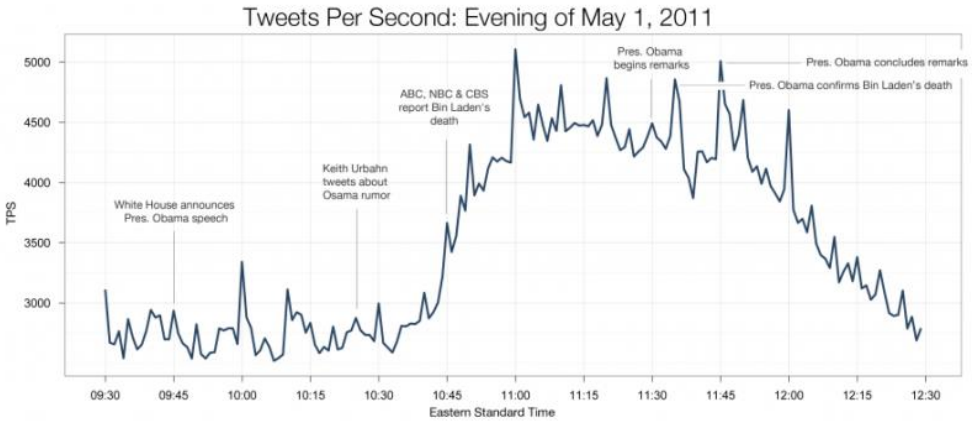
- More requests, bigger problem sizes, complex algorithms
  - Microsoft: Windows Live: 600M IDs,
  - Live Hotmail: 355M Active Accounts,
  - Live Messenger: 303M users,
  - Bing: 4B Queries/month,
  - Xbox Live: 25M users,
  - adCenter: 14B Ads served/month,
  - Exchange Hosted Services: 2 to 4B emails/day
  - “If facebook were a country, second largest in the world (1,500million+ users)”

Characteristics of Internet-scale Services

- A Sheep: 60 million players 5 days after launch (contrast: previous most popular online game: 5 million players)

	Aug-04	Mar-06	Sep-07	Sep-09
Number of MapReduce jobs	29,000	171,000	2,217,000	3,467,000
Average completion time (seconds)	634	874	395	475
Server years used	217	2,002	11,081	25,562
Input data read (terabytes)	3,288	52,254	403,152	544,130
Intermediate data (terabytes)	758	6,743	34,774	90,120
Output data written (terabytes)	193	2,970	14,018	57,520
Average number of servers per job	157	268	394	488

Figure 6.2 Annual MapReduce usage at Google over time. Over five years the number of MapReduce jobs increased by a factor of 100 and the average number of servers per job increased by a factor of 3. In the last two years the increases were factors of 1.6 and 1.2, respectively [Dean 2009]. Figure 6.16 on page 385 estimates that running the 2009 workload on Amazon's Cloud Computing Service EC2 would cost $133M.



Questions about Internet-scale Services

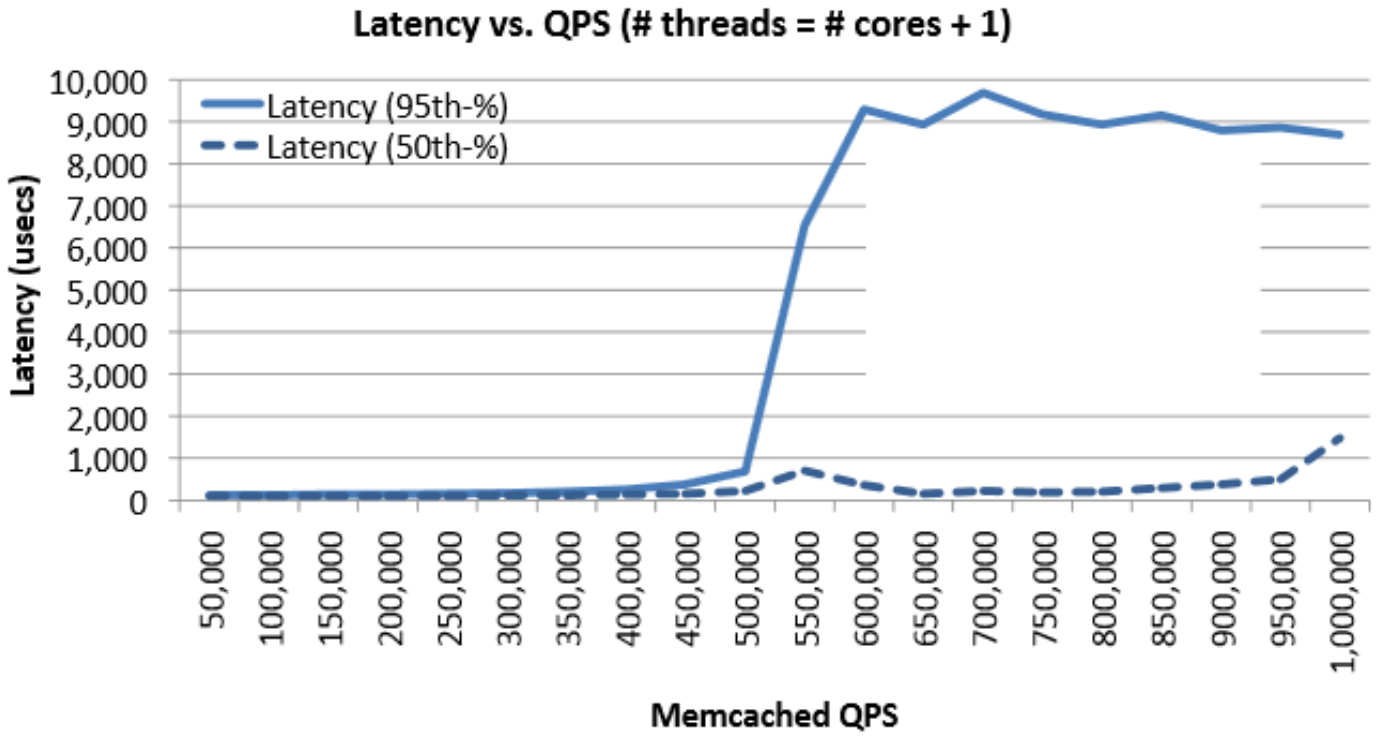
- Do we care about single-thread performance?
- How about locality?

Performance Metrics

- Throughput
  - User requests per second (RPS)
  - Scale-out addresses this (more servers)
- Quality of Service (QoS)
  - Latency of individual requests (90th, 95th, or 99th percentile)
  - Scale-out does not necessarily help
- Interesting notes
    - The distribution matters, not just the averages
    - Optimizing throughput often hurts latency
      - And optimizing latency often hurts power consumption
  - At the end, it is RPS/$ within some QoS constraints

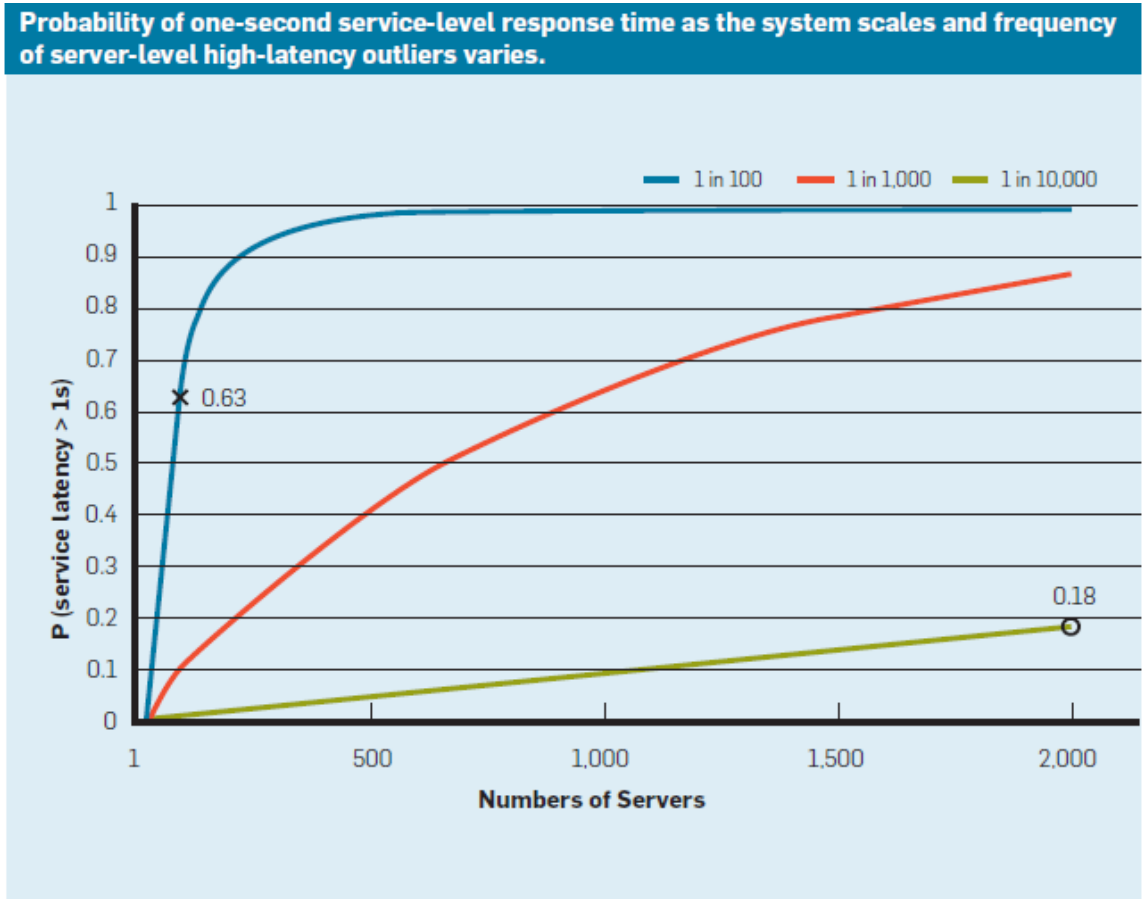
Tail Latency

- Cross-thread interference: avg low, 95th percentile very high → in a single server



Tail At Scale

- Larger clusters → more prone to high tail latency



Reasons for Variability

- Interference:
  - Different apps or different requests from the same app are sharing resources (compute, memory, network, etc.)
  - Background tasks: profiling/monitoring daemons, garbage collection, log compaction, data reconstruction, etc.
- Heterogeneity:
  - DC does not have a single platform (different generations, configs)
  - Performance varies across platforms
- Power limits & energy management:
  - Machines have to throttle when running over their Thermal Design Power
  - Machines go to low-power modes when underutilized/unused
- Input load:
  - Diurnal input load (high during day, low during night)
  - Unexpected spikes (need to provision for them)
  - Varying sizes of input dataset for analytics jobs (e.g., Hadoop)
- **Often hurt avg performance as well**

Reducing Variability

- Differentiate service classes:
  - Production → high-priority requests/apps
  - Typically will prioritize requests from interactive apps (where a user is waiting for the result) over batch requests e.g., background analytics
- Overprovisioning: allocate more resources than needed
- Reduce head-of-line blocking:
  - Some requests are much more expensive than others (e.g., complex search queries) → can increase the tail latency from blocking the simpler requests from executing
  - Time slicing: break the long requests into smaller components to allow shorter requests to interleave with them

Reducing Variability

- Manage background tasks:
  - Disallow events such as log compaction or garbage collection during periods of peak user load → differ for periods when load is low, e.g., night
  - Synchronize background tasks across machines:
    - Slows down all machines for a very brief period of time → interactive queries serviced during that period will suffer, e.g., with garbage collection
    - Alternative would always slow down some of the machines → tail latency would always be bad

Reducing Tail Latency

- Accept service time variability and work around it
- Replicate requests:
  - Send the same request to two servers
  - Either immediately or with some small delay (e.g., 90th percentile)
  - Wait until the first completes the request before you cancel the second (wastes some computation) or
  - Cancel the second request when the first arrives at the head of the local server queue (response time variability is lower post-queueing)
  - “Power of two choices” on steroids!

Reducing Tail Latency

- Address tail-latency through long-term techniques:
  - Data in DCs is distributed across the machines
  - Some data "hotter" than others → load imbalance
  - Divide data in smaller, micro-partitions (e.g., 20 partitions per machine) → easier shuffling, better load balancing, easier recovery
  - Detect (and sometimes predict) the partitions that will cause imbalance (very popular data) and replicate them more (typical replication factor: 3, for popular data: > 10) → selective replication
  - Put slow machines on "probation" → use other machines until performance recovers

Reliability & Availability

- Common goal for services: 99.99% availability
  - 1 hour of down-time per year
- But with thousands of nodes, things will crash
  - Example: with 10K servers rated at 30 years of MTBF, you should expect to have 1 failure per day

Reliability & Availability

- Typical yearly flakiness metrics:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hrs to come back)

~1 rack-move (plenty of warning, ~50-100 machines powered down, ~6 hrs)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (800-1600 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (200-400 machines see 50% packet loss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vIPs for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures (2-4% failure rate, machines crash at least twice)

~thousands of hard drive failures (1-5% of all disks will die)

Robustness to Failures

- Failover to other replicas/datacenters
- Bad backend detection:
  - Stop using for live requests until behavior gets better
- More aggressive load balancing when imbalance is more severe
- Make your apps do something reasonable even if not all is right
  - Better to give users limited functionality than an error page

Consistency

- Multiple datacenters implies dealing with consistency issues
  - Disconnected/partitioned operation relatively common, e.g., datacenter down for maintenance
  - Insisting on strong consistency likely undesirable
  - "We have your data but can't show it to you because one of the replicas is unavailable“
  - Most products with mutable state gravitating towards "eventual consistency" model
  - A bit harder to think about, but better from an availability standpoint

Performance/Availability Techniques in DCs

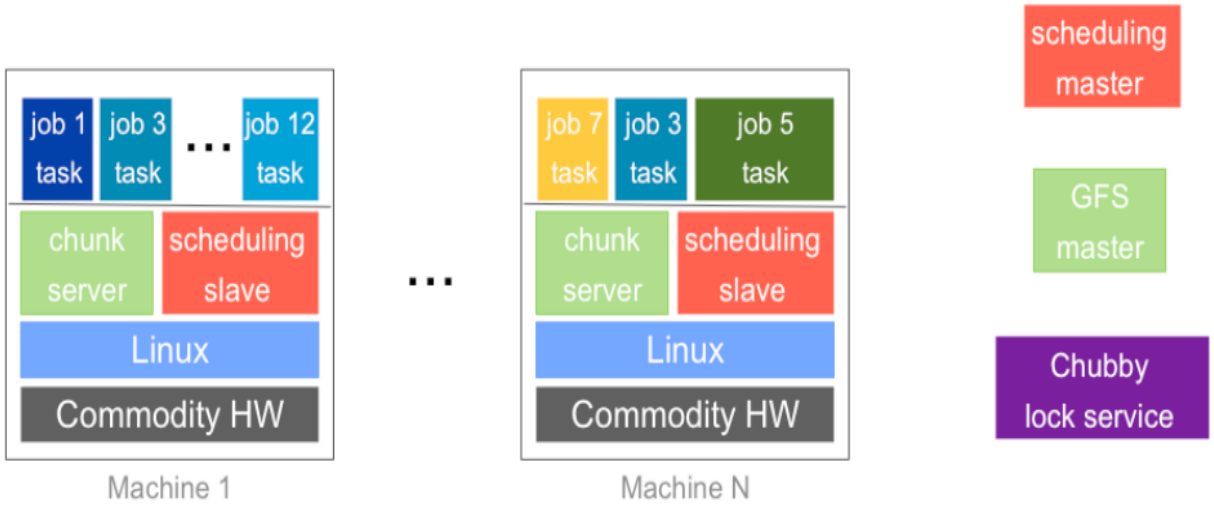
Technique	Performance	Availability
Replication	✔	✔
Partitioning (sharding)	✔	✔
Load-balancing	✔	~
Watchdog timers		✔
Integrity checks		✔
App-specific compression	✔	
Eventual consistency	✔	✔

What Else is Running in a DC

- Platform-level software
  - Firmware, operating system, key libraries
- Cluster-level infrastructure
  - Distributed file system, cluster schedulers, distributed programming models (e.g., MapReduce), software for monitoring and deployment management (e.g., Autopilot)…
- Application services
  - Easier to develop given cluster-level infrastructure…

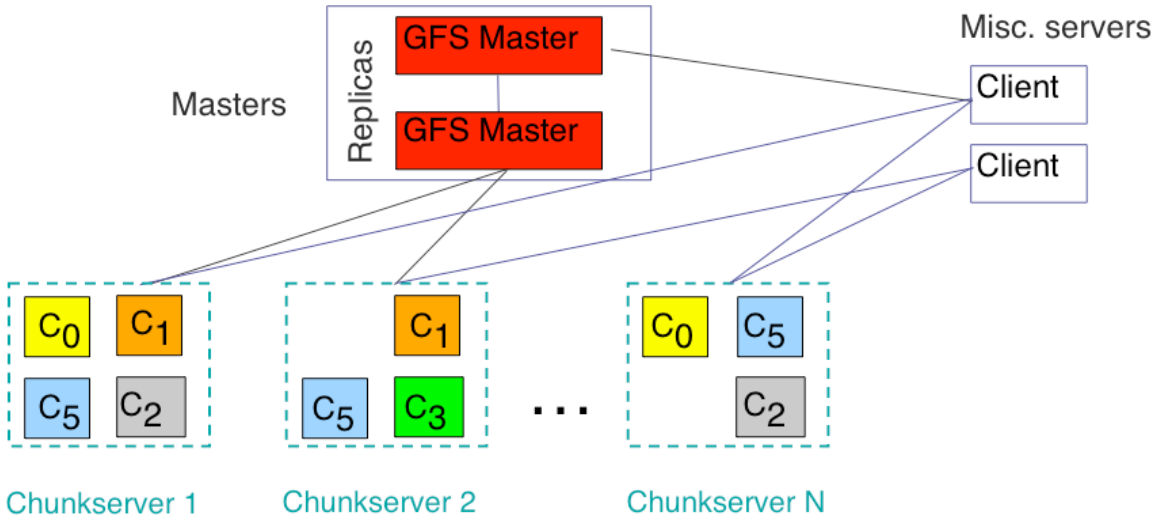
Example: Google Cluster Environment

- 1000s of machines, typically in few configurations
- File system (GFS) + Cluster scheduling system are core services
- Typically 100s to 1000s of active jobs
  - Some with 1 task, some with 1000s
  - Mix of batch and low-latency, user-facing production jobs



Example: Google File System

- Distributed file system using server disks
  - Master provides a naming service
  - Clients access data directly
- Replication support for availability & throughput
  - E.g., replicate across racks to survive node/switch failures



Example: MapReduce

- Programming model & runtime for processing large data-sets
  - E.g., Google's search algorithms
  - Goal: make it easy to use 1000s of CPUs and TBs of data
- Inspiration: functional programming languages
  - Programmer specifies only "what"
  - System determines "how"
    - Schedule parallelism, locality, communication, …
- Ingredients:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

Google's MapReduce Initial Implementation

- Runs on Google clusters:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory, local-based storage (GFS), limited bandwidth
  - Implemented as a C++ library linked to user programs (on top of RPC)
  - Beat the best Terasort score (2004) with commodity computing
- Scheduling/runtime system (aka master)
  - Assign tasks to machines: typically map tasks > machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Dynamic load balancing
- Other MapReduce implementations
  - Hadoop: an open-source, Java-based MapReduce framework
  - Phoenix: an open-source MapReduce framework for multi-core

MapReduce Programming Model

- Similar to functional languages (e.g., LISP) but with C syntax
  - High-level abstraction, hides most of the details, but scales well
  - Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
  - map (in_key, in_value) -> list(out_key, intermediate_value)
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - reduce (out_key, list(intermediate_value)) -> list(out_value)
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)
  - User also specifies I/O locations and tuning parameters

Fault Tolerance & Optimizations

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Not dealing with master failures initially
- Optimization for fault-tolerance and load-balancing
  - Slow workers significantly lengthen completion time (stragglers)
    - Due to other jobs on machines, disk with errors, caching issues, …
    - Other jobs consuming resources on machine
  - Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"

Fault Tolerance & Optimizations

- Scheduling for locality
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks schedule to node with copy of input data
    - Effect: Thousands of machines read input at local disk speed
- Skipping “bad” records in input
  - Map/Reduce functions sometimes fail for particular inputs
    - Best solution is to debug & fix, but not always possible
  - Solution
    - On seg fault, notify master about the record that fails
    - If master sees 2 failures for same record, it notifies workers to skip
  - Effect: Can work around bugs in third-party libraries
- Other
  - Compression of intermediate data

Cluster Manager

- Software system that orchestrates where/when applications are scheduled and how many resources they receive:
  - High per-application/request performance
  - High cluster utilization
  - Low scheduling overheads (good scalability)
- Different designs & tradeoffs:
  - Centralized (more sophisticated but slow)
  - Two-level (central coordinator divides resources in groups, individual frameworks manage their own resources)
  - Distributed, shared-state (distributed transactional decisions, but full visibility in cluster state)
- Has to account for interference, heterogeneity, input load, etc.

Monitoring Tools

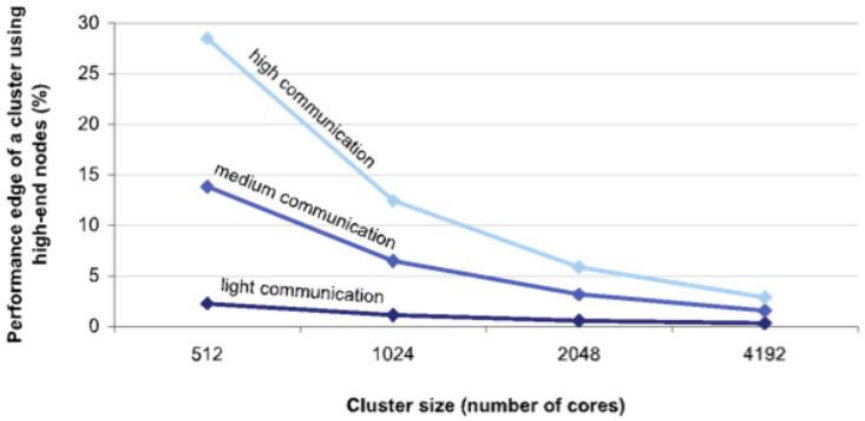
- Monitor performance of production apps
  - Time or rate of work
  - Compare to expected performance + identify bottlenecks
  - Gooda @Google: processes perf.data obtained using “perf record”
    - Count cycles of execution, cache misses, branch mispredictions, etc.
    - Profile where events occur vs. source code
    - Identify stalled cycles → recommendations for optimizations
- Dapper@Google (or Zipkin@Twitter) samples full request round trips and decomposes request latency to individual RPCs

What Server Should We Use in a Datacenter?

- Cluster of Large vs Small Servers
  - Simple performance model
    - Parallel execution time + penalty for communication (within/across servers)

$$\text{Execution time} = 1\ \text{ms} + f * [100\ \text{ns}/\#\ \text{nodes} + 100\ \mu\text{s} * (1 - 1/\#\ \text{nodes})]$$

  - Advantage of cluster with 64-socket server vs 1-socket server
  - Primary issue: DC apps are often too large even for largest server
  - Cluster of smaller server is more cost effective



Performance Scaling of Internet Scale Applications

- Scaling analysis for Search & MapReduce at Microsoft
- Any observations?

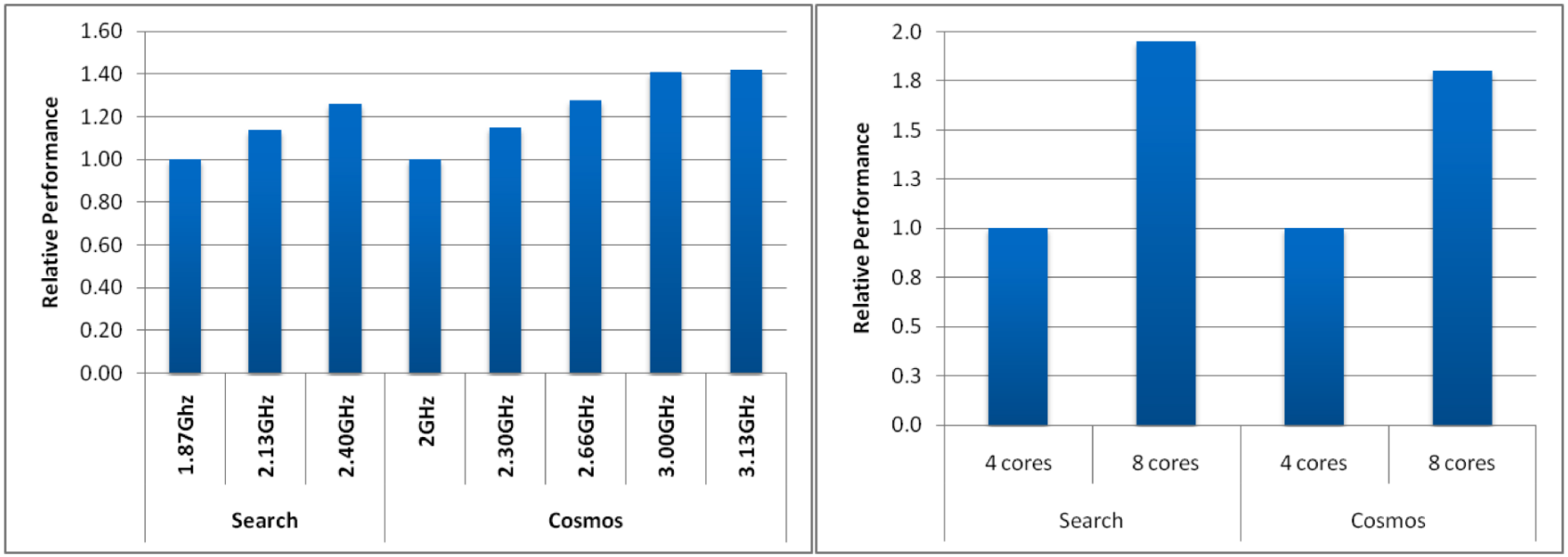


Figure 0. Performance scaling as a function of processor frequency and number of cores for Bing and Cosmos.

IEEE Micro'11

Assignment Option 1

- For the teams who want to do the assignment (option 1) with CloudSim:
- Read paper: "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms"
- Tutorial: https://www.cloudsimtutorials.online/cloudsim/
- Codes: you can use version 3.0.3
  - https://github.com/Cloudslab/cloudsim/releases/tag/cloudsim-3.0.3
- Demo