L7: Performance

Frans Kaashoek

kaashoek@mit.edu

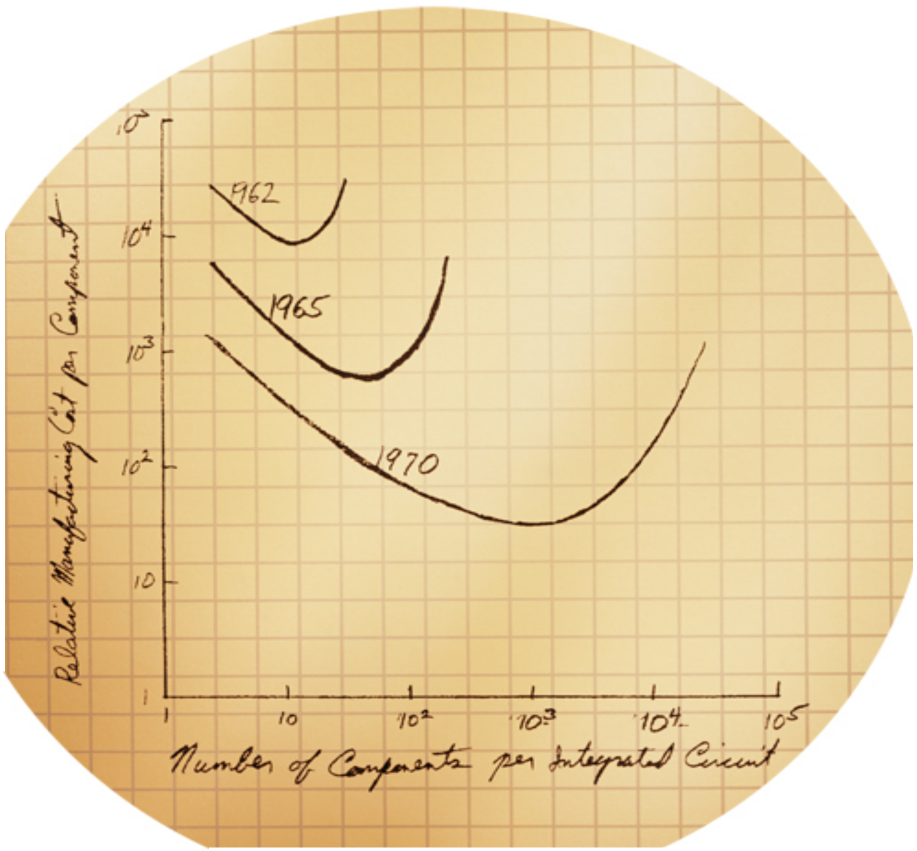
6.033 Spring 2013

Overview

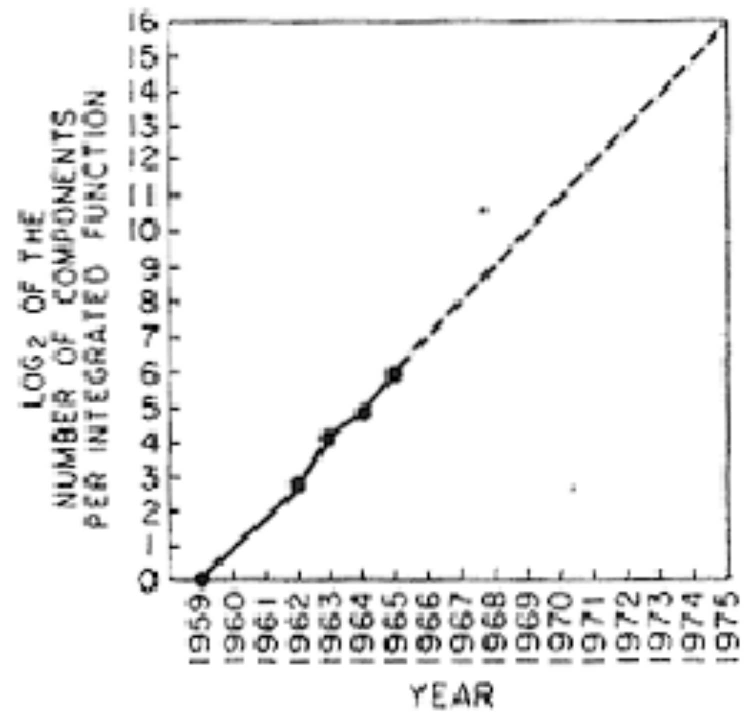
- Technology fixes some performance problems
  - Ride the technology curves if you can
- Some performance requirements require thinking
- An increase in load may need redesign:
  - Batching
  - Caching
  - Scheduling
  - Concurrency
- Important numbers

Moore’s law

cost per transistor

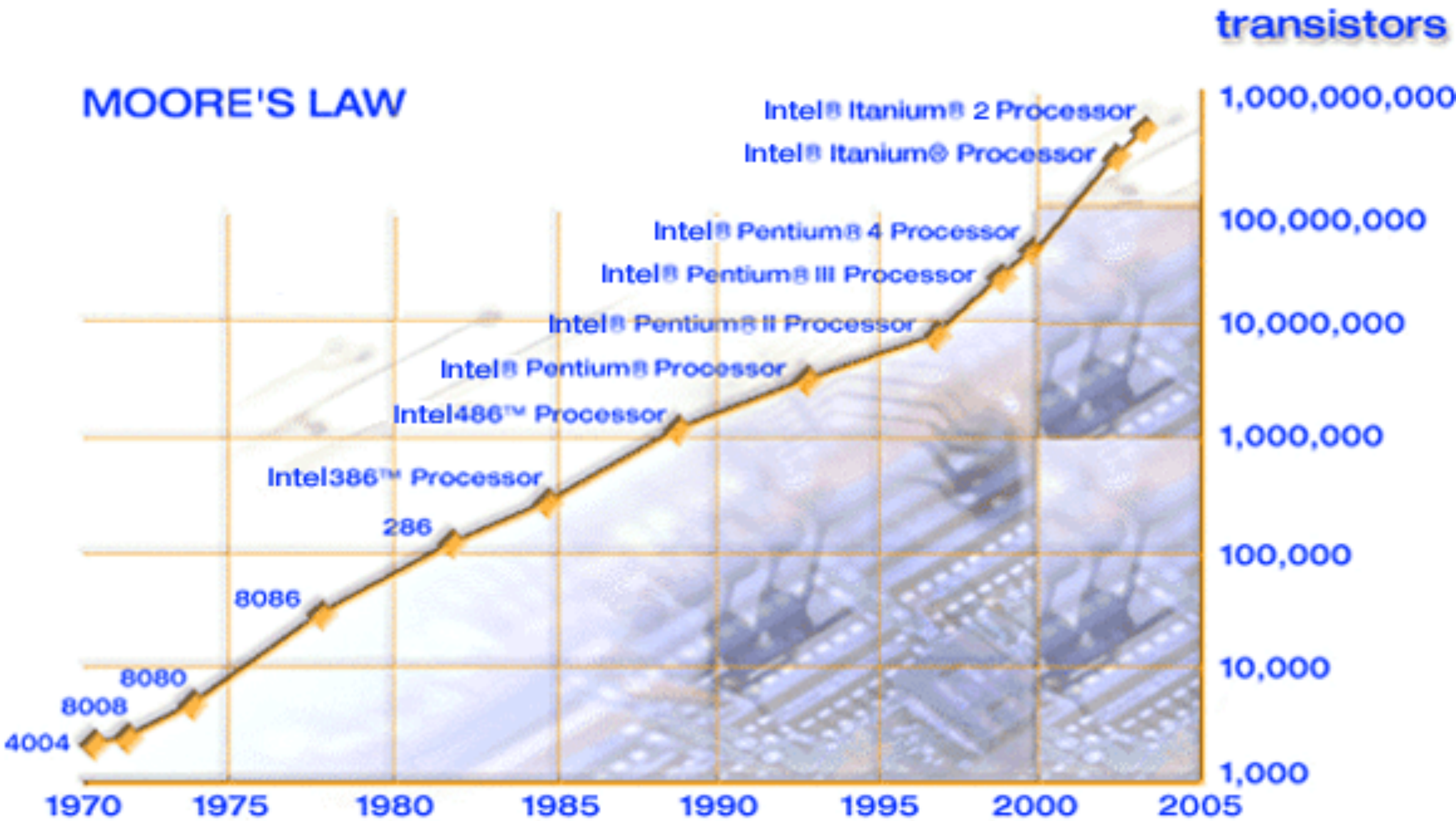


transistors per die



“Cramming More Components Onto Integrated Circuits”, *Electronics*, April 1965

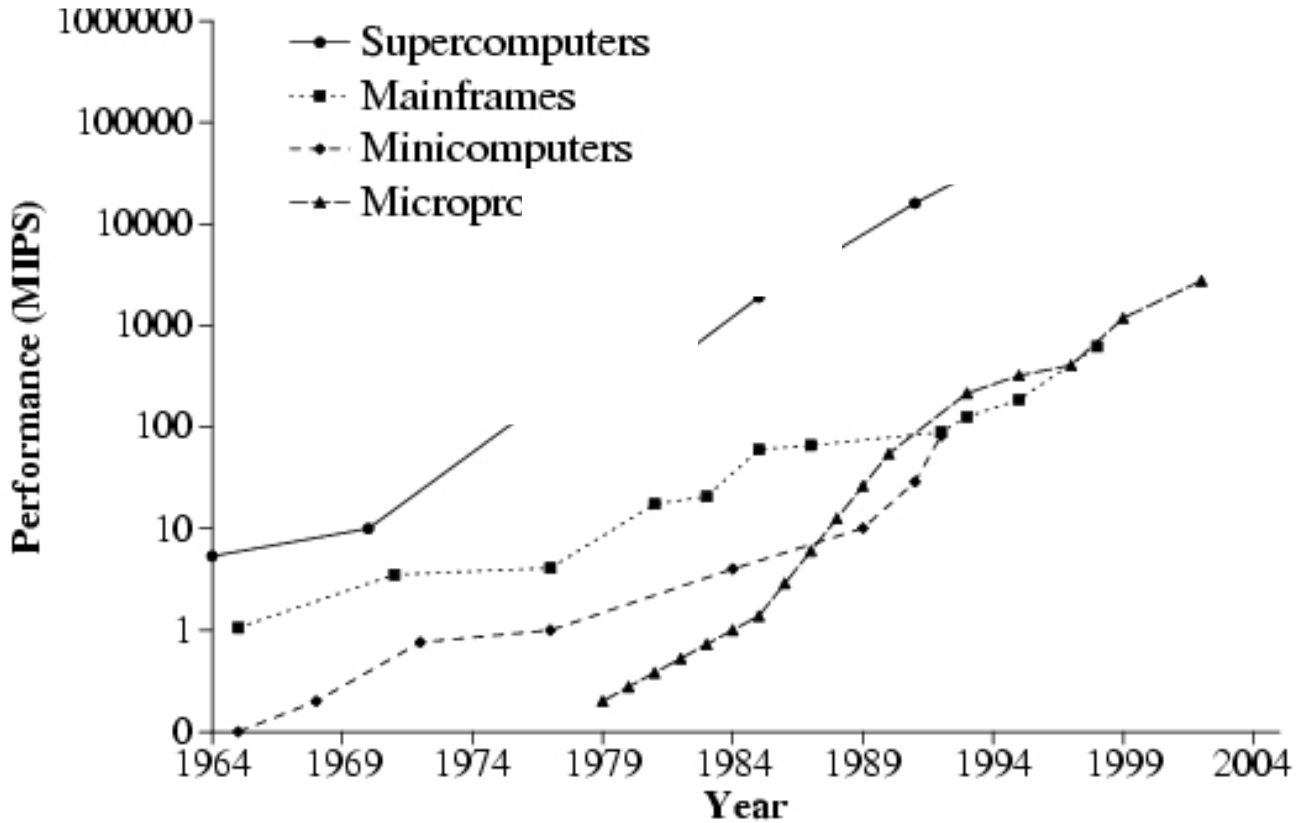
Transistors/die doubles every ~18 months



Moore's law sets a clear goal

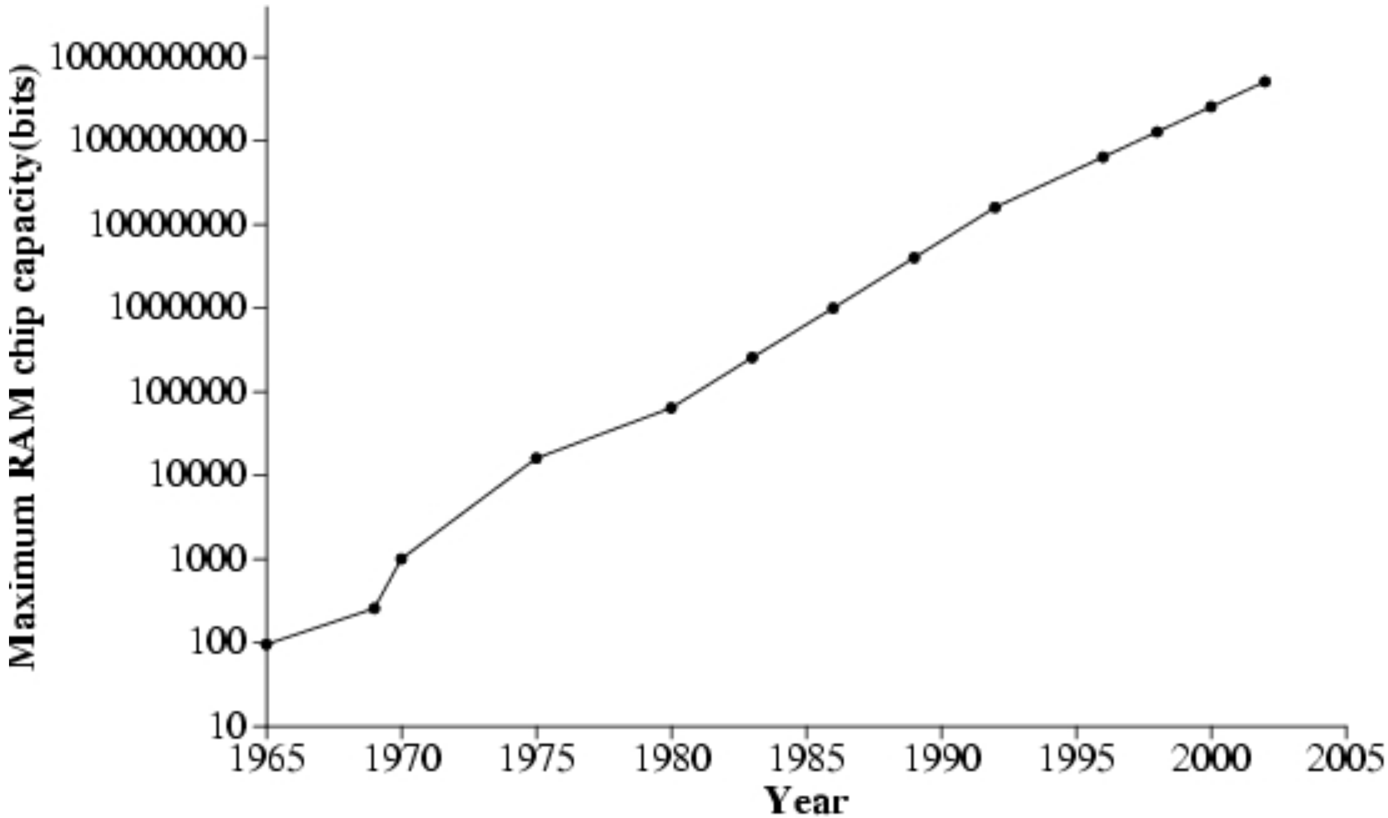
- Tremendous investment in technology
- Technology improvement is proportional to technology
- Example: processors
  - Better processors ⇒
  - Better layout tools ⇒
  - Better processors
- Mathematically: $d(technology)/dt \approx technology$
  - ➢ $technology \approx e^t$

CPU performance



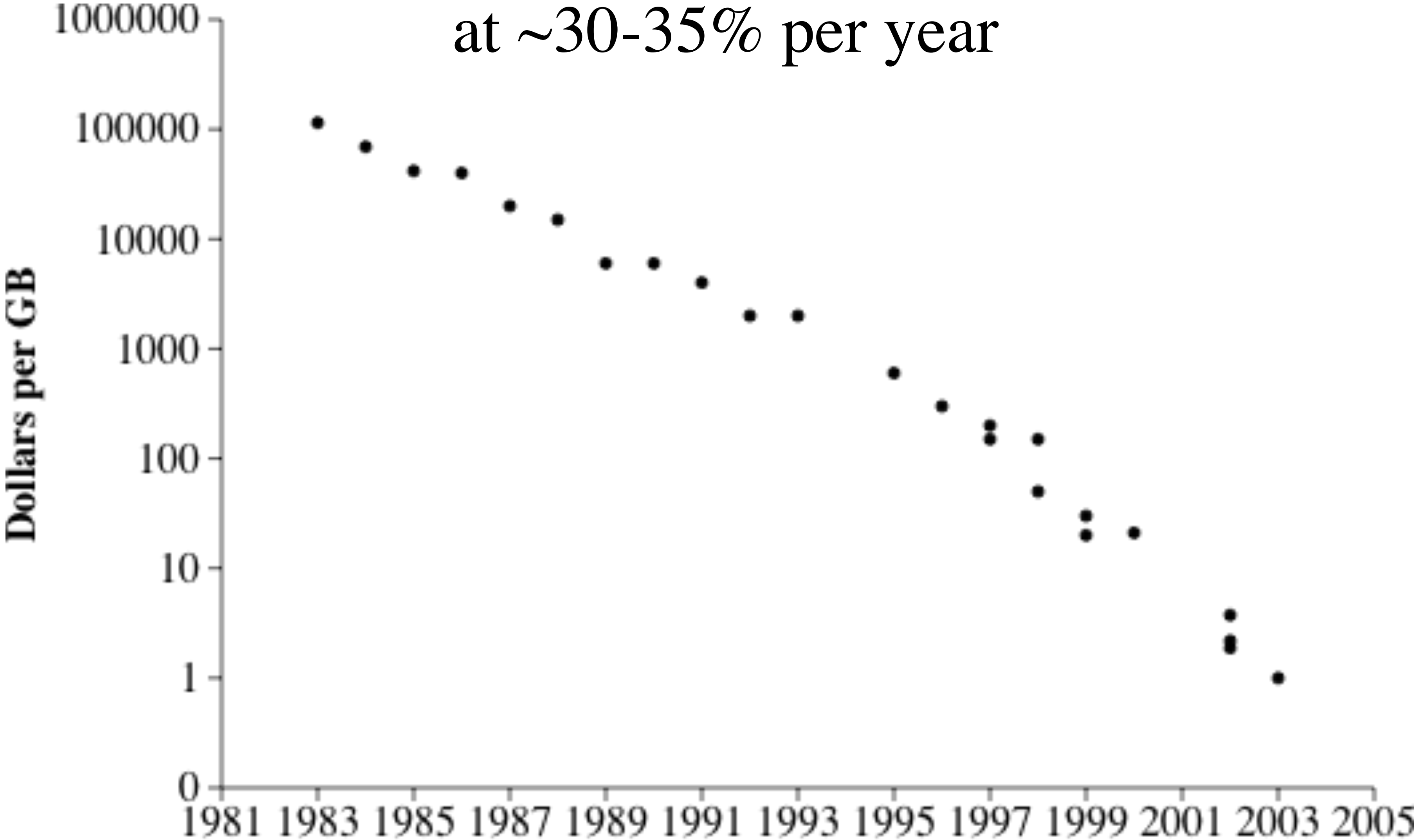
Trends in CPU performance growth, from microprocessors to supercomputers

DRAM density



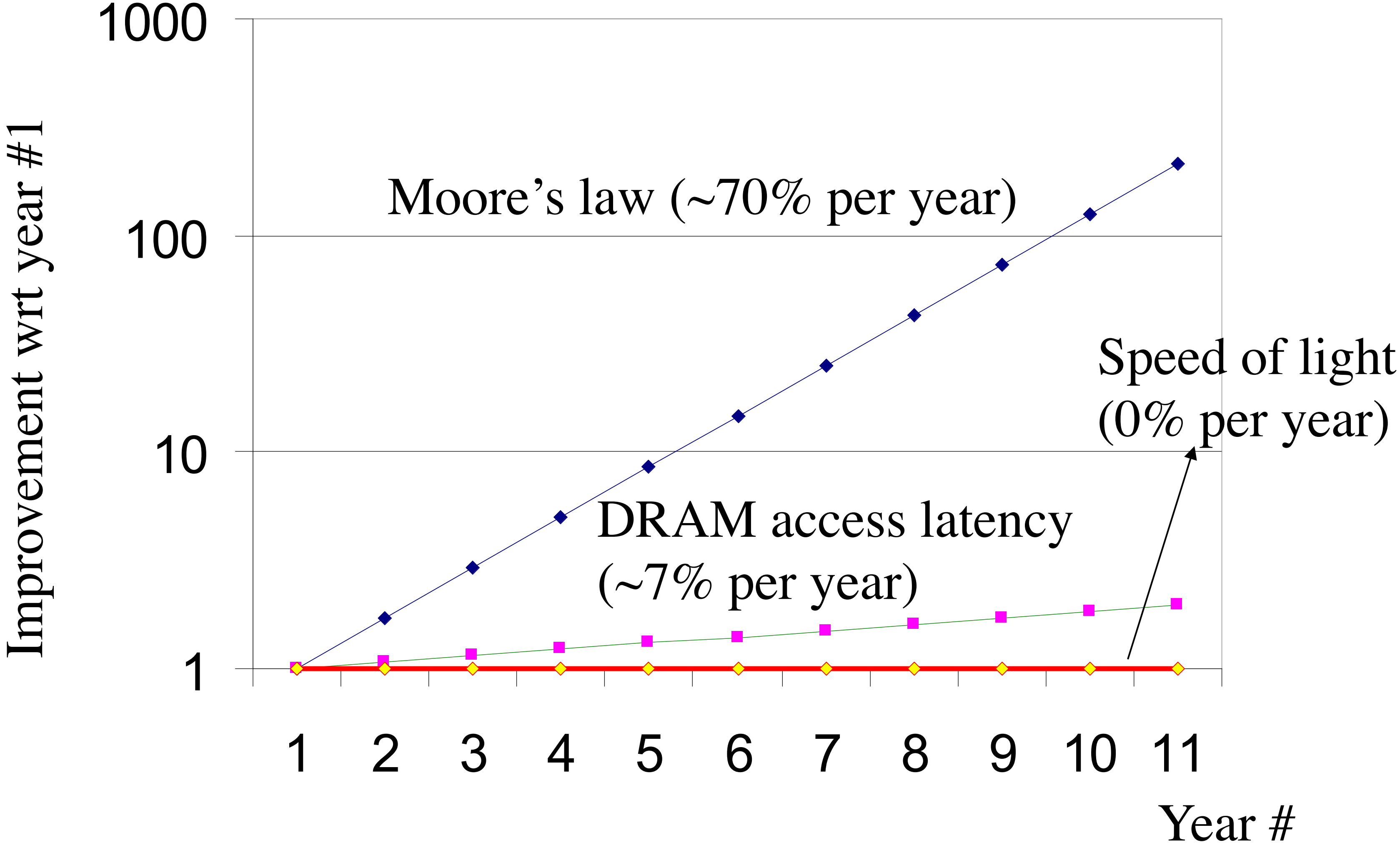
Trends in semiconductor RAM density

Disk: Price per GByte drops at ~30-35% per year



Magnetic Disk Price History

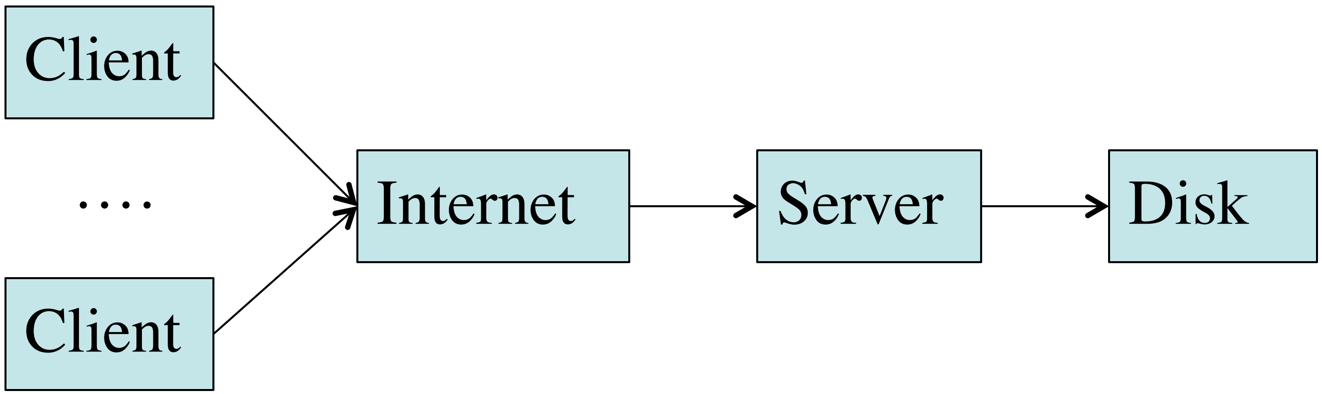
Latency improves slowly



Performance and system design

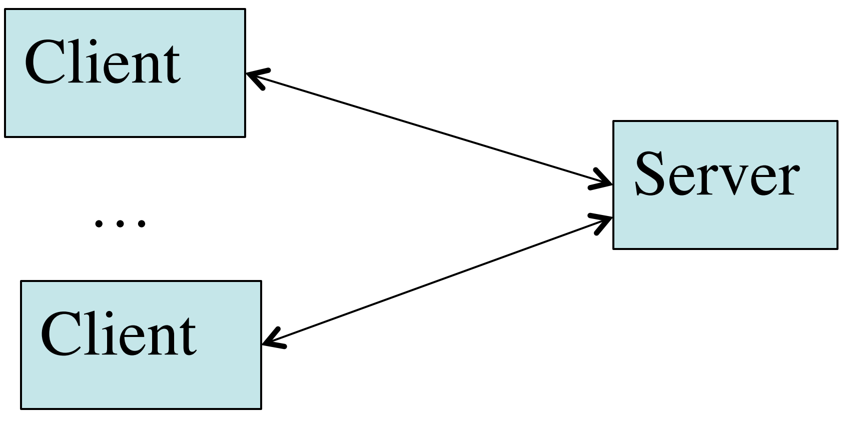
- Improvements in technology can “fix” performance problems
- Some performance problems are intrinsic
  - E.g., design project 1
  - Technology doesn’t fix it; you have think
- Handling increasing load can require re-design
  - Not every aspect of the system improves over time

Approach to performance problems



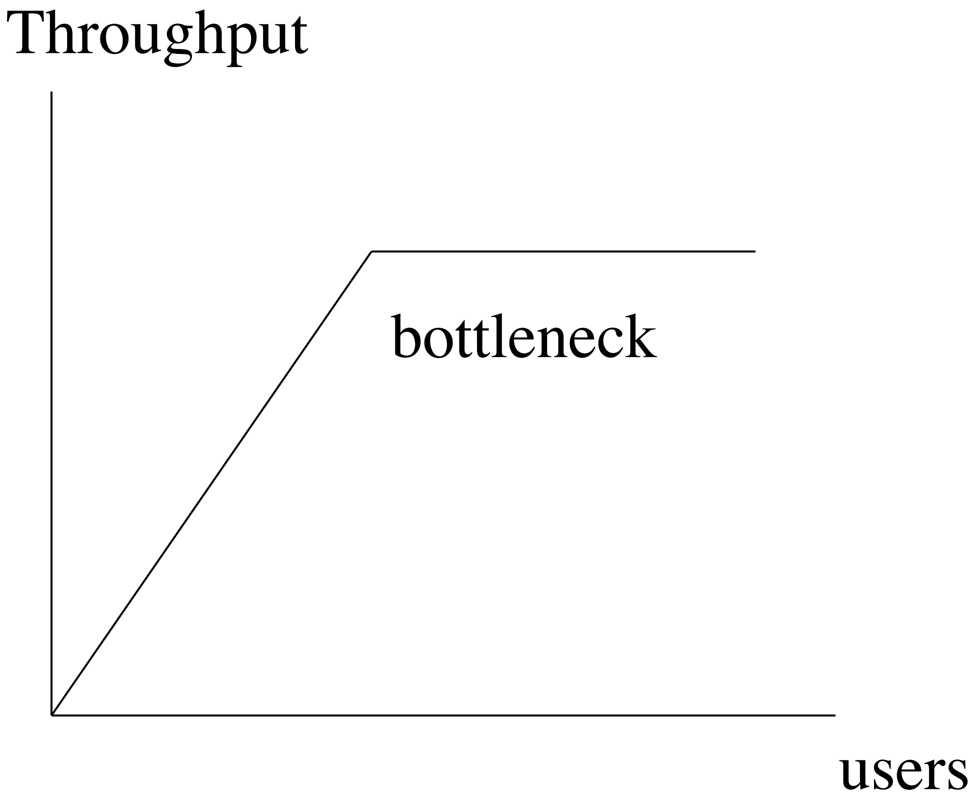
- Users complaint the system is too slow
  – Measure the system to find bottleneck
  – Relax the bottleneck
    - Add hardware, change system design

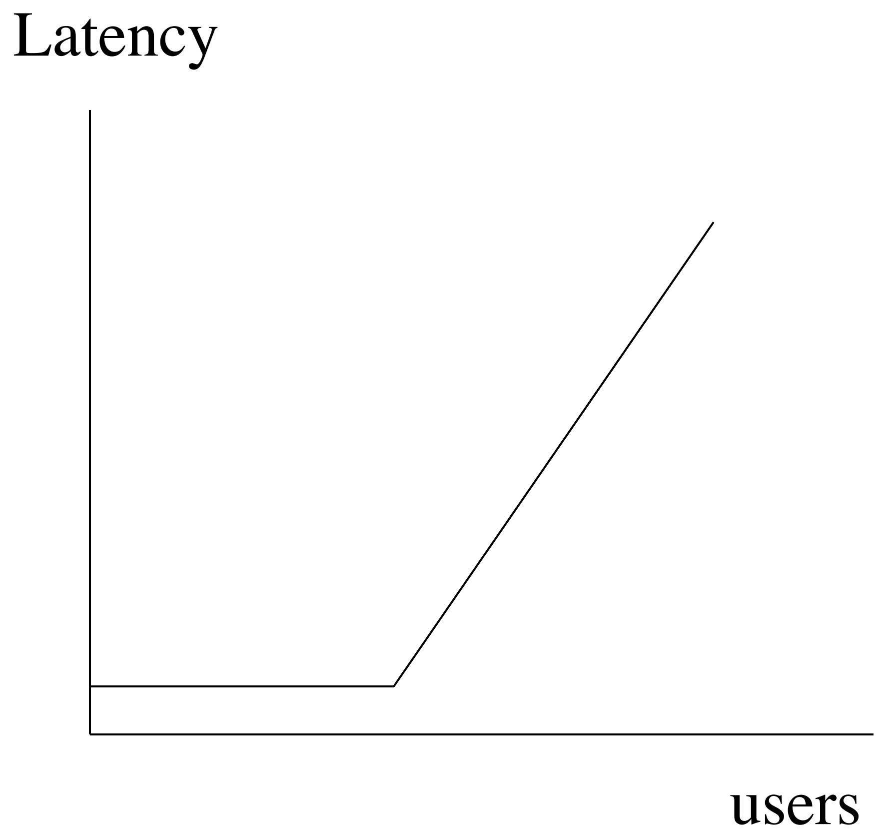
Performance metrics



- Performance metrics:
  - **Throughput**: request/time for many requests
  - **Latency**: time / request for single request
- Latency = 1/throughput?
  - Often not; e.g., server may have two CPUs

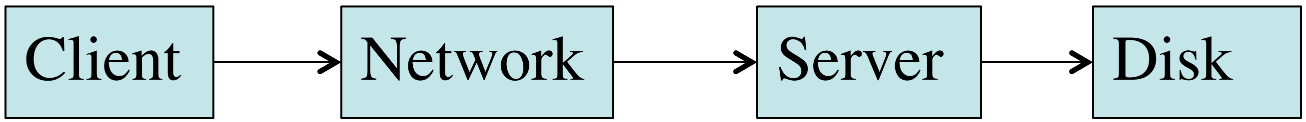
Heavily-loaded systems





- Once system busy, requests queue up

Approaches to finding bottleneck

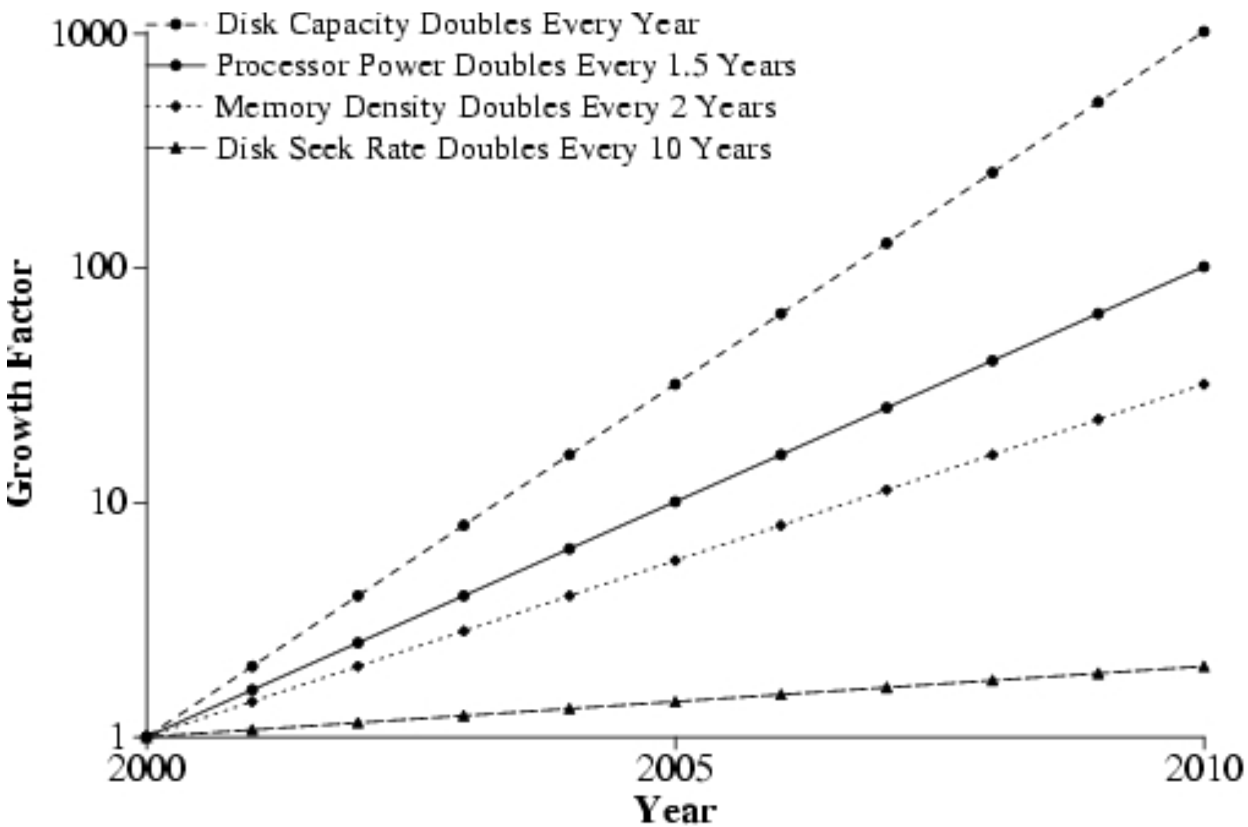


- Measure utilization of each resource
  - CPU is 100% busy, disk is 20% busy
  - CPU is 50% busy, disk is 50% busy, alternating
- Model performance of your system
  - What performance do you expect?
  - Say net takes 10ms, CPU 50 ms, disk 10ms
- Guess, check, and iterate

Fixing a bottleneck

- Get faster hardware
- Fix application
  - Better algorithm, fewer features
  - 6.033 cannot help you here
- General system techniques:
  - Batching
  - Caching
  - Concurrency
  - Scheduling

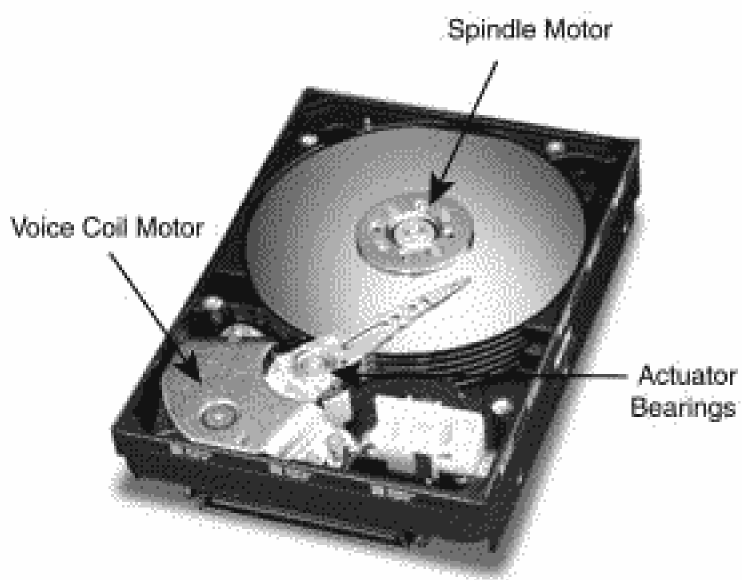
Case study: I/O bottleneck



Hypothetical Effects of Dissimilar Doubling Rates Over a Decade

Hitachi 7K400

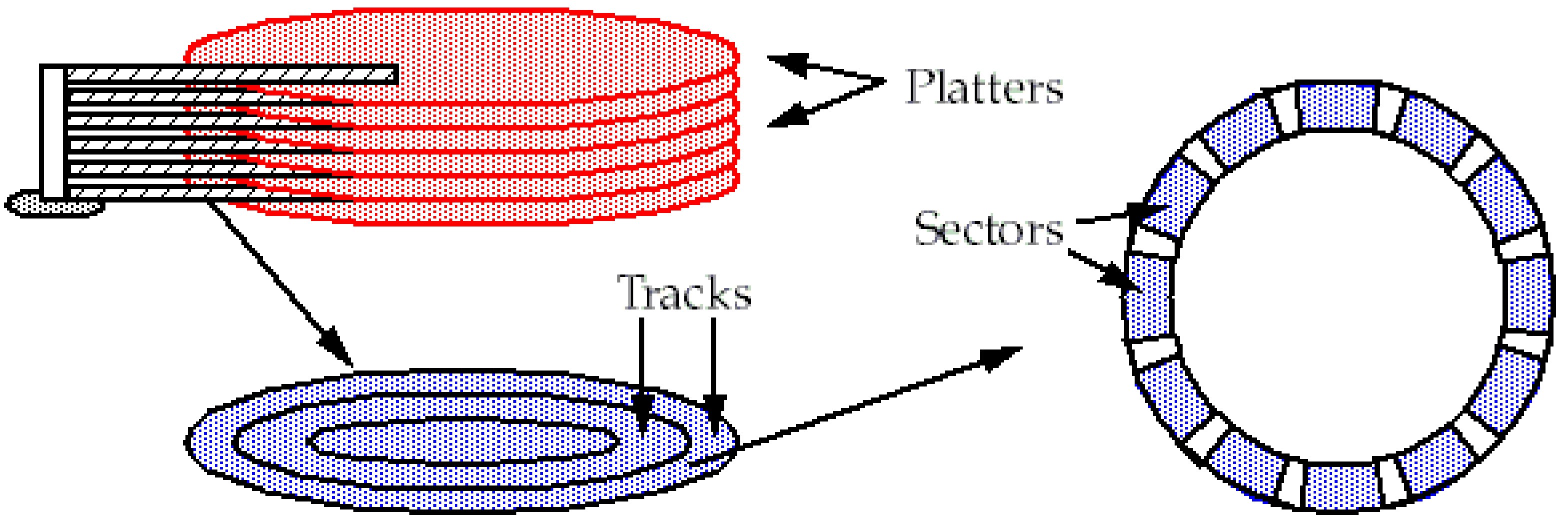
Inside a disk



7200 rpm

8.3 ms per rotation

Top view



88,283 tracks per platter

576 to 1170 sectors per track

Performance of reading a sector

- Latency = seek + rotation + reading/writing:
  - Seek time: 1-15 ms

    avg 8.2msec for read, avg 9.2ms for write
  - Rotation time: 0-8.3 ms
  - Read/writing bits: 35-62MB/s (inner to outer)
- Read(4KB):
  - Latency: 8.2msec+4.1msec+~0.1ms = 12.4ms
  - Throughput: 4KB/12.4 msec = 322 KB/s
- 99% of time spent moving disk; 1% reading!

Batching

- Batch into reads/writes into large sequential transfers (e.g., a track)
- Time for a track (1,000×512 bytes):
  - 0.8 msec to seek to next track
  - 8.3 msec to read track
- Throughput: 512KB/9.1 msec = 55MB/s
- As fast as LAN; less likely to be a bottleneck

System design implications

- If system reads/writes large files:
  - Lay them out contiguously on disk
- If system reads/writes many small files:
  - Group them together into a single track

- Modern Unix: put dir + inodes + data together

Caching

- Use DRAM to remember recently-read sectors
  - Most operating systems use much DRAM for caching
  - DRAM latency and throughput orders of magnitude better
- Challenge: what to cache?
  - DRAM is often much smaller than disk

Performance model

- Average time to read/write sector with cache:

  hit_time × hit_rate + miss_time × miss_rate

– Example: 100 sectors, 90% hit 10 sectors
  - Without cache: 10 ms for each sector
  - With cache: 0ms * 0.9 + 10ms*0.1 = 1ms

– Hit rate must be high to make cache work well!

Replacement policy

- Many caches have bounded size
- Goal: evict cache entry that’s unlikely to be used soon
- Approach: predict future behavior on past behavior
- Extend with hints from application

Least-Recently Used (LRU) policy

- If used recently, likely to be used again
- Easy to implement in software
- Works well for popular data (e.g., "/")

Is LRU always the best policy?

- LRU fails for sequential access of file larger than cache
  - LRU would evict all useful data

- Better policy for this work load:
  - Most-recently Used (MRU)

When do caches work?

1. All data fits in cache
2. Access pattern has:
   – Temporal locality
      • E.g., home directories of users currently logged in
   – Spatial locality
      • E.g., all inodes in a directory (“ls –l”)

- Not all patterns have locality
  – E.g., reading large files

Simple server

```
while (true) {
    wait for request
    data = read(name)
    response = compute(data)
    send response
}
```

Caching often cannot help writes

- Writes often must to go disk for durability
  - After power failure, new data must be available
- Result: disk performance dominates write-intensive workloads
- Worst case: many random small writes
  - Mail server storing each message in a separate file
- Logging can help
  - Writing data to sequential log (see later in semester)

I/O concurrency motivation

- Simple server alternates between waiting for disk and computing:

CPU: --- A ---   --- B ---

Disk: --- A ---   --- B ---

Use several threads to overlap I/O

- Thread 1 works on A and Thread 2 works on B, keeping both CPU and disk busy:

```
CPU:  --- A --- --- B ---  --- C --- ….
Disk:               --- A --- --- B --- …
```

- Other benefit: fast requests can get ahead of slow requests
- Downside: need locks, etc.!

Scheduling

- Suppose many threads issuing disk requests:

  71, 10, 92, 45, 29
- Naïve algorithm: random reads (8-15ms seek)
- Better: Shortest-seek first (1 ms seek):

  10, 29, 45, 71, 92

  High load -> smaller seeks -> higher throughput

  Downside: unfair, risk of starvation
- Elevator algorithm avoids starvation

Parallel hardware

- Use several disks to increase performance:
  - Many small requests: group files on disks
    - Minimizes seeks
  - Many large requests: strip files across disks
    - Increase throughout
- Use many computers:
  - Balance work across computers?
  - What if one computer fails?
  - How to program? MapReduce?

Solid State Disk (SSD)

- Faster storage technology than disk
  - Flash memory that exports disk interface
  - No moving parts
- OCZ Vertex 3: 256GB SSD
  - Sequential read: 400 MB/s
  - Sequential write: 200-300 MB/s
  - Random 4KB read: 5700/s (23 MB/s)
  - Random 4KB write: 2200/s (9 MB/s)

SSDs and writes

- Writes performance is slower:
  – Flash can erase only large units (e.g, 512 KB)
- Writing a small block:
  1. Read 512 KB
  2. Update 4KB of 512 KB
  3. Write 512 KB

- Controllers try to avoid this using logging

SSD versus Disk

- Disk: ~$100 for 2 TB
  - $0.05 per GB
- SSD: ~$300 for 256 GB
  - $1.00 per GB

- Many performance issues still the same:
  - Both SSD and Disks much slower than RAM
  - Avoid random small writes using batching

Important numbers

- Latency:
  - 0.00000001 ms: instruction time (1 ns)
  - 0.0001 ms: DRAM load (100 ns)
  - 0.1 ms: LAN network
  - 10 ms: random disk I/O
  - 25 ms: Internet east -> west coast
- Throughput:
  - 10,000 MB/s: DRAM
  - 1,000 MB/s: LAN (or100 MB/s)
  - 100 MB/s: sequential disk (or 500 MB/s)
  - 1 MB/s: random disk I/O

Summary

- Technology fixes some performance problems
- If performance problem is intrinsic:
  - Batching
  - Caching
  - Concurrency
  - Scheduling
- Important numbers