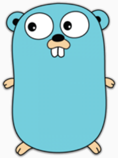
Programming in Go

Matt Holiday
Christmas 2020



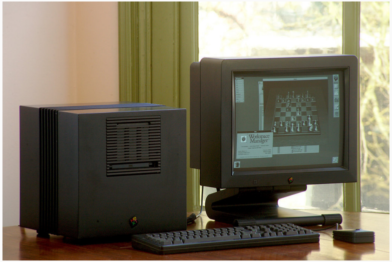
Mechanical Sympathy

Mechanical sympathy

“The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.” — Henry Peteroski

We got similar *perceived* performance 30 years ago with

- 100 times less CPU
- 100 times less memory
- 100 times less disk space



Performance in the cloud

We've made a deliberate choice to accept some overhead

We have to trade off performance against other things:

- choice of architecture
- quality, reliability, scalability
- cost of development & ownership

We need to optimize where we can, given those choices

We still want **simplicity, readability & maintainability of code**

Optimization

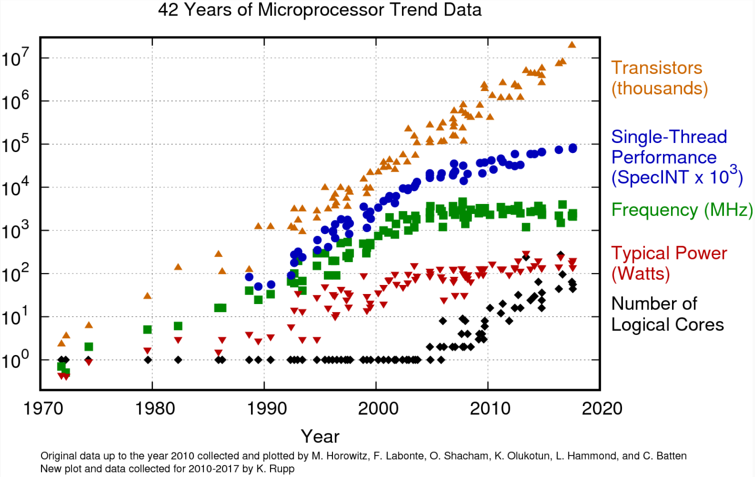
Top-down refinement:

Architecture	latency, cost of communication
Design	algorithms, concurrency, layers
Implementation	programming language, memory use

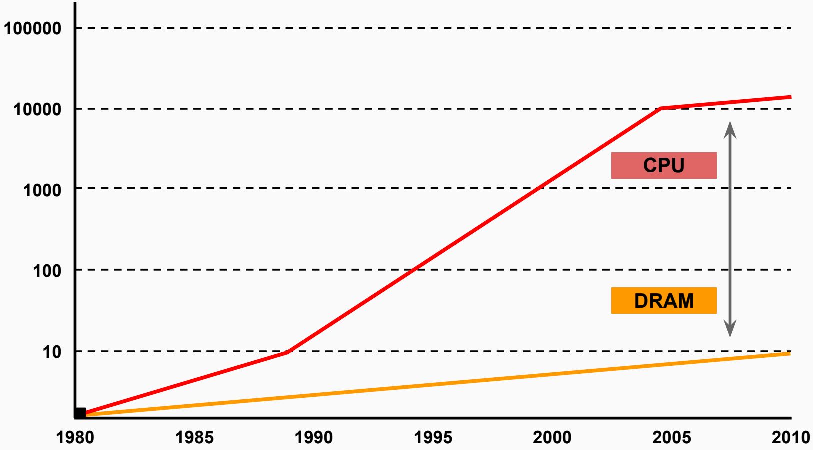
Mechanical sympathy plays a role in our *implementation*

Interpreted languages may cost 10x more to operate due to their inefficiency

CPU performance



Memory performance



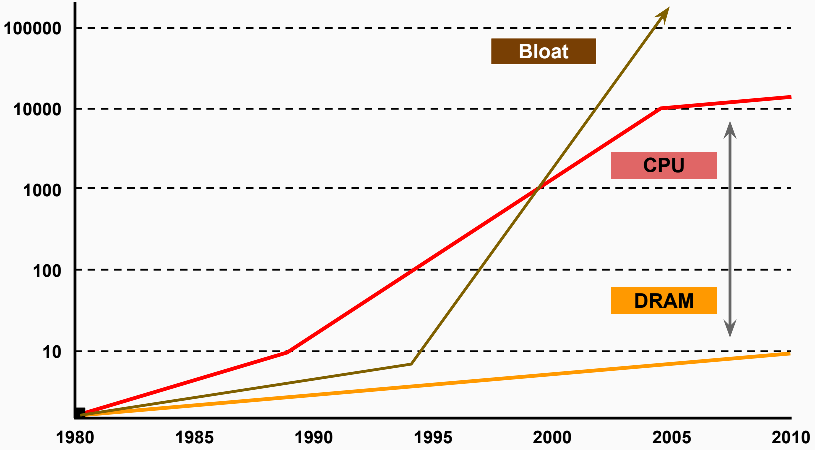
Mechanical sympathy

Some unfortunate realities:

- CPUs aren't getting faster any more
- the gap between memory and CPU isn't shrinking
- software gets slower more quickly than CPUs get faster

Software development costs exceed hardware costs

Software bloatation



Mechanical sympathy

Two competing realities

- maintain or improve the performance of software
- control the cost of developing software

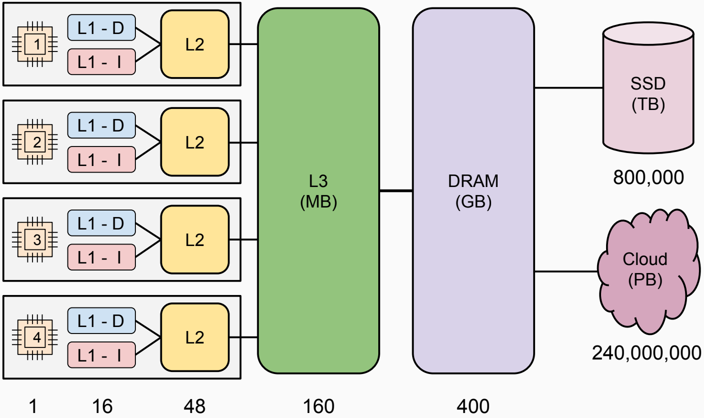
The only way to do that is

- make software simpler
- **make software that works *with* the machine**, not against it

Make software suck less

Memory hierarchy

As memory capacity increases, access latency also increases



Memory caching

"Computational" cost is often dominated by **memory access cost**

Caching takes advantage of access patterns to keep frequently-used code and data "close" to the CPU to reduce access time

Caching imposes some costs of its own

- Memory access by the **cache line**, typically 64 bytes
- **Cache coherency** to manage cache line ownership

Locality

Locality in space:

access to one thing implies access to another nearby

Locality in time:

access implies we're likely to access it again soon

Caching hardware and performance benchmarks usually favor large-scale "number crunching" problems where the software makes optimal use of the cache

Cache efficiency

Caching is effective when we use (and reuse) entire cache lines

Caching is effective when we access memory in predictable patterns
(but only sequential access is predictable)

We get our best performance when we

- **keep things in contiguous memory**
- **access them sequentially**

Cache efficiency

Things that make the cache **less efficient**:

- synchronization between CPUs
- copying blocks of data around in memory
- non-sequential access patterns (calling functions, chasing pointers)

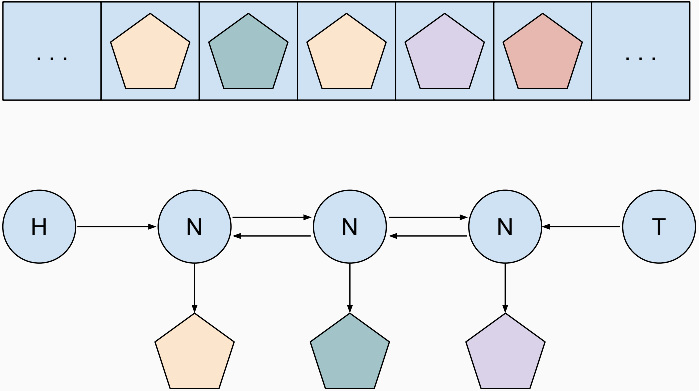
A little copying is better than a lot of pointer chasing!

Things that make the cache **more efficient**:

- keeping code or data in cache longer
- keeping data together (so all of a cache line is used)
- processing memory in sequential order (code or data)

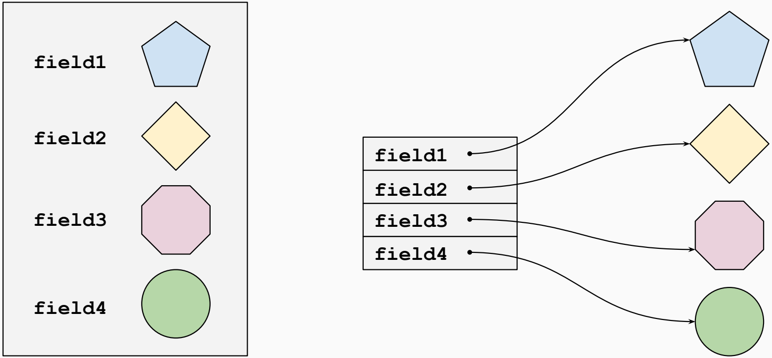
Access patterns

A slice of objects beats a list with pointers



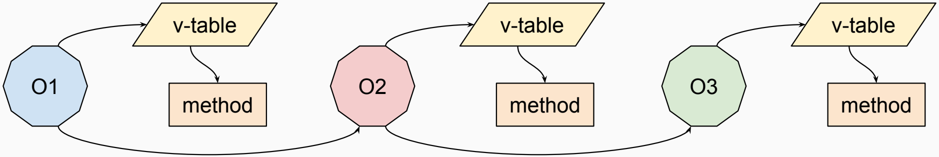
Access patterns

A struct with contiguous fields beats a class with pointers



Access patterns

Calling lots short methods via dynamic dispatch is very expensive



The cost of calling a function should be proportional to the work it does
(short inline functions vs longer methods with late binding)

Synchronization costs

Synchronization has two costs:

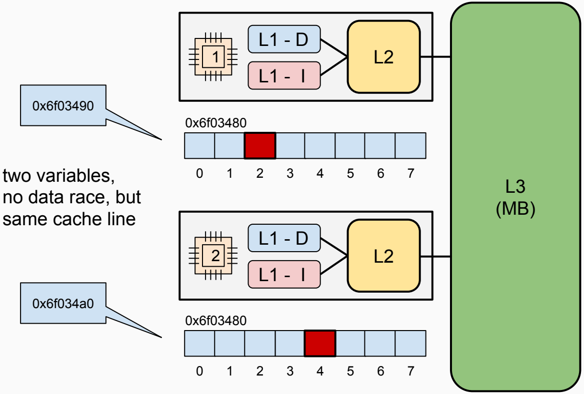
- the actual cost to synchronize (lock & unlock)
- the impact of contention if we create a "hot spot"

In the worst case, synchronization can make the program sequential

Amdahl's Law:
total speedup is limited by the fraction of the program that runs sequentially

Synchronization costs

False sharing: cores fight over a cache line for *different* variables



Other costs

There are other hidden costs:

- disk access
- garbage collection
- virtual memory & its cache
- context switching between processes

The only one you can really control is GC:

- reduce unecessary allocations
- reduce embedded pointers in objects
- paradoxically, you may want a larger heap

Optimization

Go (and the Go philosophy) encourages good design:
you can choose

- to allocate contiguously
- to copy or not copy
- to allocate on the stack or heap (sometimes)
- to be synchronous or asynchronous
- to avoid unnecessary abstraction layers
- to avoid short / fowarding methods

Go doesn't get between you and the machine

Good code in Go doesn't hide the costs involved

Optimization

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about *small* efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

— Don Knuth

Optimization

"There are only three optimizations:

1. Do less
2. Do it less often
3. Do it faster

The largest gains come from 1, but we spend all our time on 3."

— Michael Fromberger