Introduction to Go Profiling

Yuri Nebogatov (Fullscript)

Schedule

- Basic overview of hardware architecture and modern trends
- CPU cache basics
- What is profiling?
- Setup
- Profiling - CPU
- General tips and suggestions

1. Overview of hardware architecture and modern trends

- Basic computer architecture has been around for quite a while.

- CPU has direct access to its registers (very limited amount of memory), as well as, in order of increasing latency, various layers of caches, main memory, and (often memory-mapped) devices.

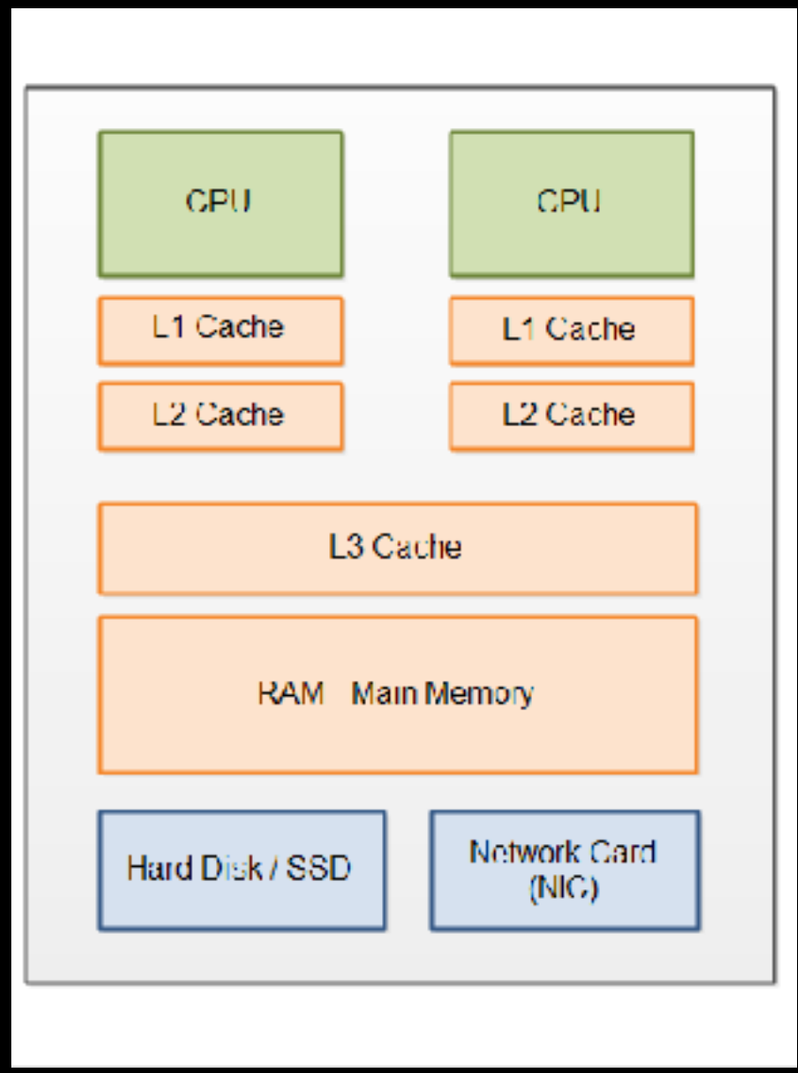

Fig 1. Basic Architecture

1. Overview of hardware architecture and modern trends

- In the past, we often relied on Moore's law - often performance improved "by itself".
- Unfortunately that free ride is mostly over. Single-threaded CPU performance has mostly stagnated, with performance increases between generations in low single digits requiring huge architectural improvements.

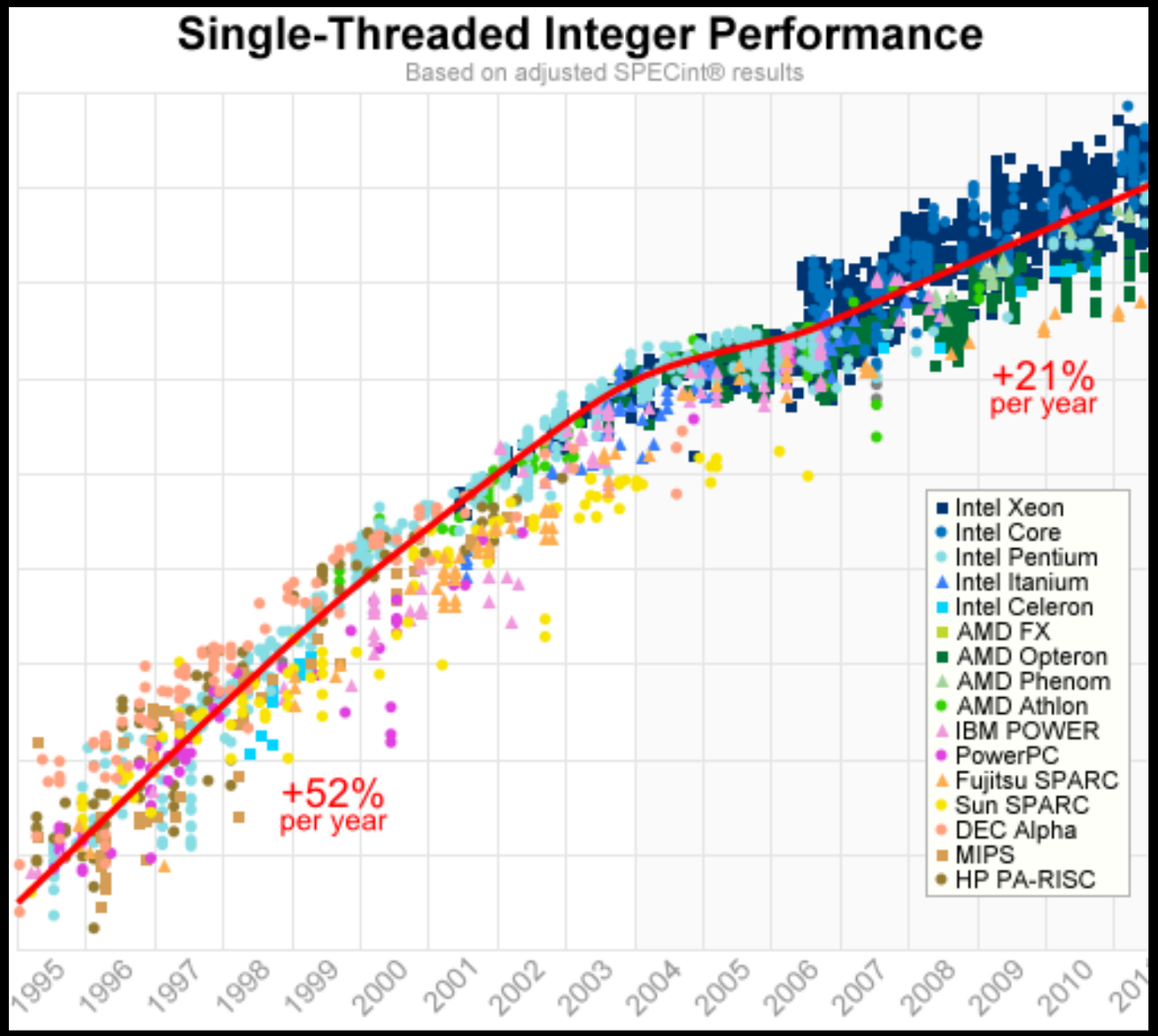

Fig 2. CPU performance growth (SPEC CPU2006)

1. Overview of hardware architecture and modern trends

- The trend is made worse by the fact that memory performance (specifically memory access latency) is increasing at a much slower rate than CPU performance. This creates an ever-growing gap that, if not addressed through proper software design, can starve a modern CPU and drastically lower its performance.

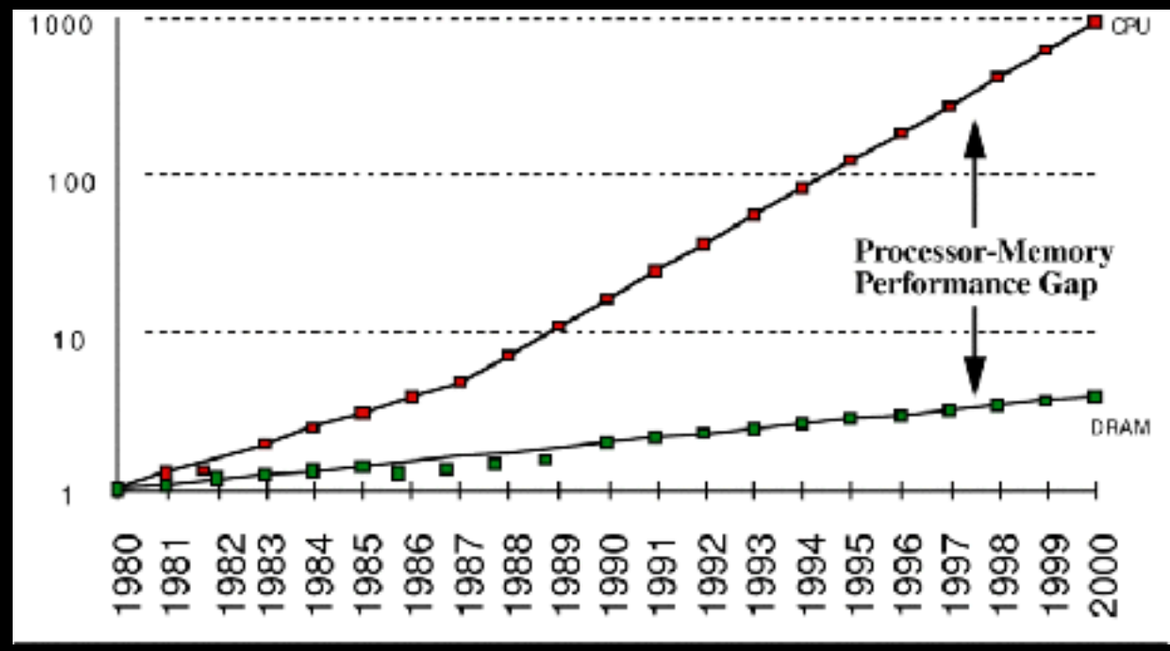

Fig 3. CPU-Memory performance gap

1. Overview of hardware architecture and modern trends

- Multiple layers of cache have been introduced over time to compensate for the slow MM latency.

- Cache (on-chip) is very expensive, thus only a limited amount can be included.

- Optimization is often a game of minimizing cache misses when possible - as the performance hit is gigantic.

L1	0.5 ns
L2	7 ns
Mutex Lock/ Unlock	25 ns
MM	100 ns
Disk	10 000 000 ns

https://gist.github.com/jboner/2841832

2. Cache Basics

- Cache is a way to reduce the memory latency bottleneck and allow the CPU to process instruction at its maximum frequency.

- Typically cache is lazily (opportunistically) loaded upon MM access. The CPU will check its L1, L2, L3 and ultimately talk to the memory controller of the MM for a particular memory location.

- MM is usually accessed in bulk, using so called “cache lines”. This is a range of memory locations that are copied to the L1/L2 cache upon a cache miss.

- On modern CPUs, cache lines are typically 64-128 bytes (usually 64 bytes on consumer CPUs).

- To see the necessity of cache lines, think about sequentially accessing 16 64-bit integers. If the locations aren’t cached, we’d be incurring the MM access (100ns) latency penalty 16 times, instead of once.

2. Cache Basics

- Cache size is typically small. L1 - 32KB, L2 - 1MB, L3 - 16mb.
- Caches typically use the LRU eviction policy to stay up to date and relevant.
- The basic design of cache in modern system lends itself to two basic optimization patterns:
  - Spatial memory locality
  - Temporal memory locality

3. What is profiling? (10 000 feet view)

- Profiling, in essence, is analyzing the various aspects pertaining to the performance of an application.

- This is usually achieved by sampling the various resources (active functions, active memory objects, etc) at specified intervals.

- Typically, these include:
  - CPU (cache)
  - Memory (allocations, heap)
  - Locking (mutex)

4. Setup

- brew install graphviz / apt-get install graphviz
- go get github.com/pkg/profile (Dave Cheney's profiling helper package)
- Go get github.com/yuraneb/intro-to-go-profiling
- That's it …

5. CPU profiling

- Build and run /exercises/cpu1
- Go tool pprof
- go tool pprof —pdf [path] > cpu_profile.pdf
- Go tool pprof -tree [path]
- go tool pprof -http=localhost:8080 [path] <— GREAT

5. CPU profiling

- Run the binary several times with a different number of iterations, why aren’t the results consistent?
- Why aren’t all the function calls visible in the profile?
- What are the biggest offenders?
- Any other surprising observations?

5. CPU profiling

Q: What’s going on under the covers?

A: The process is being profiled by interrupting it (SIGPROF) at a rate of 100Hz (every 10ms) and collecting information about the running goroutines/functions. The sampling rate is considerably low, so the profiling must run for a long time to be accurate - always a chance of aliasing (Nyquist).

Q: Can I modify the sampling rate to minimize the risk of aliasing?

A: SetCPUProfileRate(hz int)

Q: Flat vs Cumulative?

A: Flat reflects only the weight of the code of a particular function, not the weight of the descendants (function calls). Cumulative includes all function calls.

Q: How are recursive functions counted?

A: Only once per chain - as a mercy from the developers of pprof.

5. CPU profiling

Q: What notable profiling options are there?

1) You can specify filters in the form of regex.

pprof -text -show [regex] (or -ignore [regex])

2) You can specify the granularity of the profiles

SetCPUProfileRate(hz int)

5. CPU profiling

- Build and run /exercises/cpu2
- View the profile using the web interface
- Why is there such a discrepancy between the “log-based manual” benchmarks we were collecting, and the data we see in the actual CPU profile?
- What can you suggest to gain full profiling visibility into all aspects of the program (and ultimately - are the missing parts significant in any way)?

6.General tips & suggestions

- Import _ “net/http/pprof” for an easy HTTP api to trigger profiles on demand (albeit not for production)
  This adds profiling endpoints to http.DefaultServeMux

- Comparing profiles: you can actually compare(diff) different profiles using pprof. The intent is to allow you to accurately measure the effect of any code changes you made to address performance issues.
  pprof -text -diff_base=profile_old -normalize profile_new

  Note: the normalize option is your friend, but you should make an attempt to collect both profiles under the same conditions (running time, sampling rate, environment)