Performance optimization: Pros & Cons

Aliaksandr Valialkin, VictoriaMetrics

About me

- I’m @valyala, the author of fasthttp, fastjson and other fast* packages
- I like Go and performance optimization
- I like fast code only if it is clear
- I work at VictoriaMetrics - the best long-term remote storage for Prometheus

What is a performance optimization?

- Code modifications aimed towards better performance
- There are many performance optimization types exist:
  - Memory usage optimization
  - CPU cache usage optimization
  - GC optimization
  - ….
  - Others. We’ll talk about them later

When performance optimization is needed?

- When your program runs slower than you expect
- When your program eats a lot of memory
- When your program overloads external db or microservices
- When your program eats a lot of network bandwidth
- When your program requires a lot of persistent storage on expensive SSD instead of HDD
- **When your program is expensive to run in the Cloud**
- ???
- Your case here

So, doesn’t everyone needs performance optimizations?

No

When performance optimization isn’t needed

- When your program satisfies performance requirements
- Programs without performance requirements don’t need performance optimizations
- Many programs have no performance requirements

What is a performance requirement?

- If your users suffer from a slow program, then you have a performance requirement - make users happy by speeding up the program
- If your hosting expenses exceed budgets, then you have a performance requirement - reduce hosting expenses by optimizing the program
- If your boss says “this program must be the fastest on the market” :)

What isn’t a performance requirement?

- “I see a clear slow algorithm in the code. Let’s substitute it with a faster uglier one!”
- “The program may run faster by 1%. Let’s optimize it!”
- “I’ve just read a blog post about performance optimization. Let’s optimize our program!”
- “Our service handles 100 qps in production and all the clients are happy. Let’s optimize it to 100K qps!”

Performance optimization types

Performance optimization types

- CPU usage optimization
- CPU cache optimization
- Memory usage optimization
- Minimizing GC overhead
- Network usage optimization
- Disk usage optimization
- External service (db, microservice) usage optimization
- Scalability optimization
- Latency optimization

CPU usage optimization

- Reduce CPU usage, so a CPU may perform more work
- Good tool: CPU profiler. Go has built-in CPU profiler that can run in production - see https://blog.golang.org/profiling-go-programs . It quickly discovers CPU hogs in your program.
- Bad tool: “I think this code eats a lot of CPU. Let’s optimize it!”

CPU cache optimization

- Cram the most frequently accessed data into CPU caches, since CPU cache is 10-100x times faster than RAM access
- Google for “Latency numbers every programmer should know”
- Good tools:
  - Cache-friendly and cache-oblivious data structures + algorithms
  - Sequential memory access instead of random memory access
  - Pointer-free structures, data embedding
  - Compact data structures
- Bad tools:
  - Pointer chasing
  - Bloated data structures

Memory usage optimization

- Reduce memory usage, so more programs can be crammed on an expensive Kubernetes node
- Good tool: memory profiler. Go has multiple memory profilers that run in production:
  - --alloc_space - discovers a code that allocates big chunks of memory. Mostly useless.
  - --alloc_objects - discovers a code that allocates a lot of objects. Useful for reducing GC overhead.
  - **--inuse_space - discovers a code that allocates the biggest memory chunks currently in use. Useful for reducing memory usage and for detecting memory leaks.**
  - --inuse_objects - discovers a code that allocates many objects that aren't freed yet. May be used for reducing GC overhead.
- Bad tool: "I guess this code eats a lot of memory. Let's optimize it!"

Minimizing GC overhead

- GC isn’t free - it spends CPU and memory bandwidth on:
  - memory allocations
  - garbage detection
  - garbage removal
- GC overhead is proportional to the number of:
  - memory allocations - more memory allocations means higher GC overhead
  - active objects in use - more active objects means higher GC overhead during garbage detection
- Good article - further dangers on large heaps in Go
- **--alloc_objects** memory profile detects the code with many allocations
- **--inuse_objects** memory profile detects the code that generates many active objects

Network usage optimization

- Reduce network bandwidth usage and/or the number of network packets generated by the program in order to reduce network expenses
- Good tools:
  - dstat, ss, ifstat, iftop - for detecting network hogs
  - bufio.Reader and bufio.Writer for reducing the number of syscalls and the number of network packets
  - compress/* packages for reducing network bandwidth usage
  - Optimized binary marshalling protocols such as protobufs for reducing network bandwidth usage
  - RPC systems with minimized network overhead
- Bad tools:
  - SOAP, XML, JSON for data transfer. They eat a lot of network bandwidth and CPU

Disk usage optimization

- Reduce IOPS, disk bandwidth usage and disk space usage, so more data could be crammed into a persistent storage at max speed
- Good tools:
  - dstat, iostat, iotop - for detecting disk hogs
  - bufio.Reader and bufio.Writer for minimizing the number of syscalls and, probably, the number of IOPS when reading/writing to disk
  - Various compression algorithms (gzip, snappy, zstd, etc.) for minimizing disk space usage and the time required to store / load the data
  - Persistent data layout optimized for sequential writes/reads. This reduces IOPS and increases the speed for data writing / reading
- Bad tools:
  - XML and JSON as a storage format. It takes a lot of disk space.

External service usage optimization

- Reduce load on external service, so it could serve more requests with lower latencies
- Good tools:
  - Distributed tracing for detecting slow external services
  - Requests’ batching - group multiple requests into a single batch
  - Query optimizations - for instance, SQL optimizations for external DB
  - Response caching
- Bad tools:
  - ORMs, since they can generate N+1 requests instead of 1 or 2 requests
  - Deep abstractions, since they can generate a lot of useless requests under the hood

Scalability optimization

- Make the program run up to N times faster on N CPU cores comparing to a single CPU core
- Good tools:
  - Mutex profiler - see https://rakyll.org/mutexprofile/ . It detects contended mutexes in the code
  - Share nothing architecture
  - Immutable shared state
  - sync/atomic package
  - Parallel algorithms
- Bad tools:
  - Mutable shared state and coarse mutexes in hot paths
  - Sequential algorithms

Latency optimization

- Minimize response times
- Latency optimization != CPU optimization
- Latency optimization depends on all the previous optimization types
- Latency vs throughput: choose one
- Good tools:
  - Tracing for detecting the code that increases latency the most
- Bad tools:
  - “I guess this code increases latency the most. Let’s optimize it!”

Performance optimization pros

- Faster programs
- More qps
- Faster response times
- Lower hosting expenses
- Happier customers

So why performance optimization is the root of all evil?

Performance optimization cons

- It complicates the code
- It slows down the development
- It introduces hard-to-catch bugs
- It makes APIs uglier and easier to shoot in the foot - see fasthttp, fastjson and other fast* projects from https://github.com/valyala/ ;)

General optimization rules

- Don’t optimize the program unless optimization requirements exist
- Don’t guess where the bottleneck is - use Go profilers for detecting the code to be optimized
- Do not complicate the code for a small performance win

General optimization rules

- Write a clear code first, optimize later
- Prefer simpler algorithms and data structures over more complex ones
- Don’t focus on CPU usage optimization only - there are other optimization types exist
- Try switching to new data structure and algorithm if the previous one doesn’t give the expected performance
- Don’t chase for big O notation. Simpler algorithms and data structures are usually faster on small inputs

Questions?